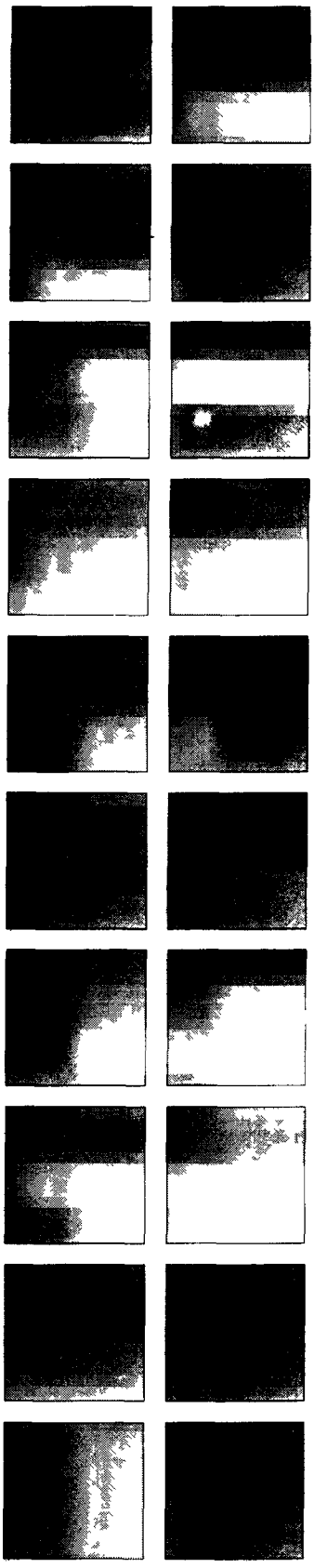
32.973.2-018
С24



Освоение Turbo Assembler®

Второе издание

Том Сван

540310

БІБЛІОТЕКА
ЧЕРКАСЬКОГО ДЕРЖАВНОГО
УНІВЕРСИТЕТУ
ІМЕНІ БОГДАНА ХМЕЛЬНИЦЬКОГО

ДИАЛЕКТИКА
Киев • Москва • Санкт-Петербург
1996

ББК 32.973
С24

Издательство "Диалектика"

Зав. редакцией канд. физ.-мат. наук *А.А. Минько*

Перевод с английского *Ю.И. Корниенко*, канд. физ.-мат. наук *А.М. Цветкова*
Редакция *К.Э. Островского*

По общим вопросам обращайтесь в издательство "Диалектика" в Киеве
по телефонам: 261-5936, 261-5643, факс 227-1456,

Internet: info@dcp.kiev.ua, http://www.dcp.kiev.ua

Для оптовых закупок и заказов по почте: diamail@dcp.kiev.ua
либо
в России: Москва-419, 117419, а/я 10, тел./факс 955-5288, 955-5397
в Украине: Киев-6, 252006, а/я 100, тел./факс 268-6027, 230-2667, 261-9287

Сван, Том.
С24 Освоение Turbo Assembler : Пер. с англ. — 2-е изд. —К.; М.; СПб. : Диалектика, 1996. — 544 с.: ил — Парал. тит. англ.
ISBN 0-672-30526-7 (англ.)
ISBN 5-7707-9254-X (рус.)

Цель данной книги — помочь читателю разобраться в программировании на языке ассемблера. Написанная одним из самых читаемых авторов — Томом Сваном, она является полным введением в программирование на языке ассемблера, но в то же время затрагивает вопросы средней и повышенной сложности. Сотни работающих примеров кода, резюме в конце глав, упражнения и задания — все это, безусловно, поможет читателю безболезненно научиться создавать программы на языке ассемблера и овладеть всеми особенностями Borland Turbo Assembler.

Книга рассчитана на программистов с различным уровнем подготовки.

ББК 32.973

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства **Sams Publishing.**

Authorized translation from the English language edition published by Sams Publishing. Copyright © 1995.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Dialektika Computer Publishing. Copyright © 1996.

Научно-популярное издание
Сван Том
Освоение Turbo Assembler
2-е издание

Литературный редактор *Г.В. Яроцкая*
Художественные редакторы *Е.П. Дынник, С.А. Чернокозинский*
Технический редактор *А.Н. Полинчик*
Корректоры *О.В. Мишутина, Л.В. Пустовойтова, З.В. Александрова*

Оригинал макет подготовлен в отделе компьютерной верстки издательства "Диалектика".

Подписана к печати 26.01.96. Формат 84x108/16. Бумага офсетная. Гарнитура Укр.Петербург. Печать офсетная. Усл.печ.л. 57,12.
Заказ № 0218063

ISBN 0-672-30526-7 (англ.)
ISBN 5-7707-9254-X (рус.)

© Tom Swan, 1995
© Издательство "Диалектика", 1996

Предисловие

Программисты всегда спорят о том, какой язык программирования лучше. Попробуйте склонить программистов, привыкших работать на C, к работе с Pascal, и они пошлют вас подальше. Попробуйте заставить специалистов по Pascal обратить хоть какое-то внимание на Basic, и они обязательно объяснят, куда вам пойти. И даже не думайте доказывать фанам FORTH, что это — совершенно непонятный язык и не приспособлен для какой-либо серьезной работы: в лучшем случае вас просто прогонят.

Я стараюсь избегать всех подобных аргументов в спорах. Они кажутся мне просто смешными. Представьте, что это не программист, а шеф-повар доказывает, что вкуснее суфле, рецепт которого написан на английском языке, а не на французском или на испанском. Конечно же, это — глупость: мы получаем одно и то же блюдо, независимо от того, на каком языке обозначены его ингредиенты. Цветок есть цветок, не правда ли?

То же самое относится и к программированию. Все языки высокого уровня для работы таких процессоров, как 8086, 80386 или 80486, должны транслировать свои команды в машинный код. Учитывая это, легко понять, что в сущности все языки программирования— это один и тот же язык.

Итак, не имеет значения, какой из языков высокого уровня вы предпочитаете: необходимо изучить язык ассемблера, который позволяет разговаривать с компьютером на его родном диалекте. В последующих главах я постараюсь более подробно разъяснить, как писать целые программы на языке ассемблера, особое внимание будет уделено развитию повторно используемых библиотечных модулей. В книге есть главы, где описывается, как соединять язык ассемблера с Pascal, C, C++. Это новое издание также включает главы по объектно-ориентированному программированию на Turbo Assembler и по программированию для Windows.

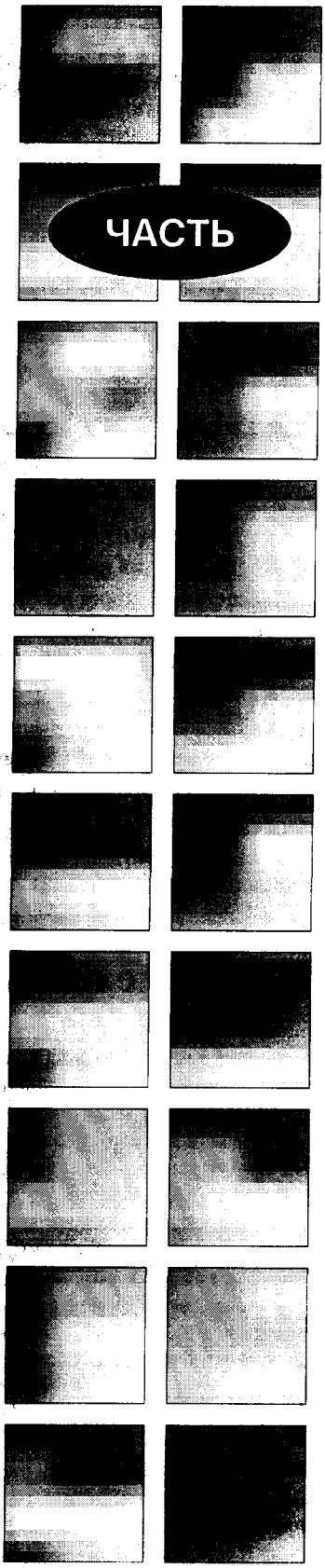
Для начинающих я хотел бы добавить: если кто-то вам говорил, что ассемблер очень сложен, не верьте.

Том Сван

Благодарность

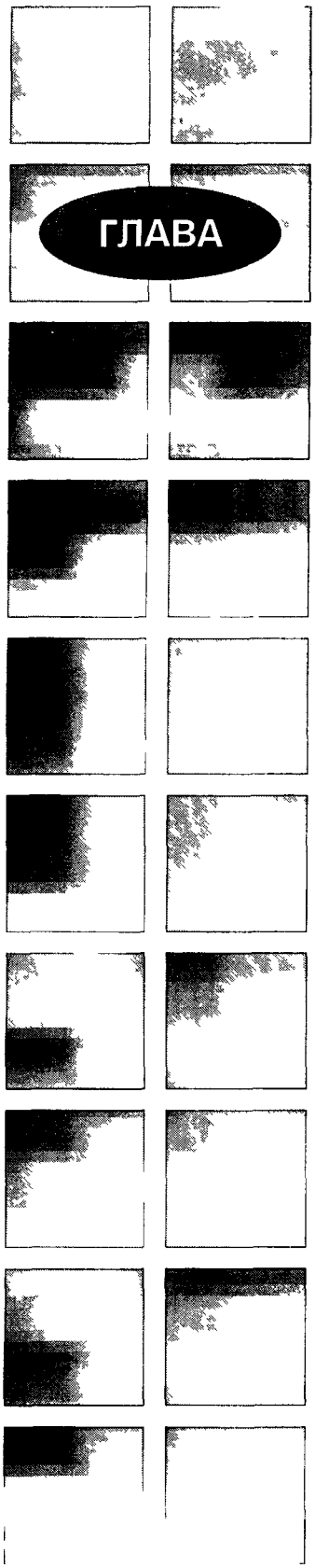
Говорят, что писать — это занятие для одиноких. К счастью, при написании этой книги, ее первого и второго изданий, я не был одинок. Среди тех, кто уделил много внимания написанию этой книги — сотрудники издательства *Sams*, Грег Крой (Greg Croy), Ричард Сведлей (Richard Swadley), Фран Хаттон (Fran Hatton), мои агенты из *Waterside Production* Билл Гледстон (Bill Gladstone) и Матт Вагнер (Matt Wagner), Нан Борресон (Nan Borreson) из *Borland International*. Спасибо также моим родителям Рейер (Reyer) и Мэри Сван (Mary Swan), присматривавшим за домом и разбиравшим мою почту.

Я хотел бы выразить личную и особую признательность Ричарду Дею (Richard Day), которому и посвящаю эту книгу — за дружбу и понимание. Большое спасибо Фреду Мак-Гихену (Fred McGeehan) за обсуждение книги и великолепный кофе, а также Анни, которая смогла выдержать меня, один Бог знает — как.



ЧАСТЬ I

Программирование на языке ассемблера



ГЛАВА 1

Введение

- Изучение языка ассемблера
- Преимущества языка ассемблера
- Недостатки языка ассемблера
- Требования к аппаратному обеспечению
- Требования к программному обеспечению
- Пользователям Microsoft Windows
- Как пользоваться этой книгой
- Как подготовить ваши диски
- Ввод исходных текстов программ
- Резюме
- Упражнения

Изучение языка ассемблера

Я помню, как открыл для себя язык ассемблера. Такое же восхищение я испытывал в тот день, когда впервые балансировал на двухколесном велосипеде, виляя по нашей улице. Большой палец моего отца больше не поддерживал меня за пояс, а мой страх перед падением таял, как пчелиный воск на солнце.

Нечто похожее ощущают многие программисты при изучении языка ассемблера. Вы хотите узнать — почему? Потому что ассемблер — единственный компьютерный язык, который позволяет общаться с компьютером на его собственном языке и непосредственно управлять аппаратными средствами. Если вы отвечаете за свои действия, любите управлять событиями и вникать в детали, тогда ассемблер — это ваш язык.

Цель данной книги — помочь вам разобраться в программировании на языке ассемблера В настоящей главе вы найдете информацию о том, как подготовить ваш диск и как лучше использовать каждый раздел книги. Вводите примеры программ из этой книги, разбирайтесь в упражнениях и заданиях, которые находятся в конце каждой главы, и не бойтесь экспериментировать. Успеха вам!

Я выбираю программирование на низком уровне

На первый взгляд, кажется, что компьютер "понимает" языки высокого уровня Basic, Pascal или C. На самом деле программы исполняются на машинном языке, представляющем собой определенный набор чисел, составляющих код, который и

управляет центральным процессором компьютера (CPU). Поэтому лучшим термином для определения низкоуровневых языков программирования является машинный код — единственный язык, понятный процессору. Поскольку процессор не может непосредственно исполнять операторы C или Pascal, программы на этих и других языках высокого уровня должны предварительно переводиться в машинный код (компилироваться). Аналогично программы, написанные на интерпретаторах Basic или LISP, переводятся в машинный код, хотя в этих случаях перевод (компилирование) происходит обычно пооператорно и незаметно — по ходу исполнения программы.

Программы на языке ассемблера также переводятся в машинный код с помощью программы-транслятора, называемой ассемблером. Вопреки этому сходству с другими языками, ассемблер не является языком ни высокого, ни низкого уровня — он занимает некоторое промежуточное положение. Основное отличие между языком ассемблера и языками высокого уровня состоит в том, что операторы C или Pascal обычно переводятся в целые наборы машинных кодов, а команды ассемблера непосредственно преобразуются в соответствующий машинный код. Существуют свои достоинства у каждого языка, однако только на языке ассемблера можно писать программы, напрямую используя множество команд процессора.

Замечание

У опытных программистов на языке C могут возникнуть возражения, так как некоторые операторы C и Turbo Pascal тоже преобразуются в отдельные машинные коды.

Они также могут утверждать, что язык FORTH обеспечивает непосредственный низкоуровневый доступ. Но даже несмотря на то, что C и другие языки программирования обеспечивают выполнение некоторых низкоуровневых операций, эти языки все равно не могут сравниться с языком ассемблера в возможностях доступа к процессору.

Кстати, чтобы обратить ваше внимание на определенные моменты при изучении отдельных глав, в книге будут использоваться фрагменты текста, выделенные подобно этому замечанию.

Почему нельзя разрабатывать программы непосредственно в машинных кодах, если язык ассемблера и машинный код однозначно соответствуют один другому? Ответ прост: машинный код слишком громоздкий. Хотя первые программы для компьютеров действительно писались в машинном коде, сегодня по веским причинам это практически не делается. Например, многие машинные коды зависят от относительного положения в памяти. Кроме того, в чисто машинном коде не используются именованные переменные и нет возможности указать программе фиксированные адреса, по которым содержатся различные значения и подпрограммы. Это означает, что если вы измените одну команду в 10000-байтовой программе на машинном коде, то, возможно, вам придется модифицировать кроме этого еще 9000 других кодов!

Ясно, что такая тяжелая работа не привлекает даже преданных своему делу программистов, поскольку, несмотря на распространенное мнение, их мозг не оперирует битами и не построен из логических схем.

Программирование непосредственно в машинных кодах — очень трудоемкая работа. Программируя на языке ассемблера, вы пользуетесь одновременно преимуществами низкоуровневого доступа и характерными для языков высокого уровня возможностями определения числовых выражений и переменных. Используя язык ассемблера, можно изменить одну команду, загрузить измененный текст в Turbo Assembler и получить в результате соответствующий машинный код. Некоторые считают, что язык ассемблера стоит только на один шаг выше машинного кода. Это действительно так, но это большой шаг.

Лучше знать устройство компьютера — значит лучше программировать на ассемблере

Команды языка ассемблера непосредственно преобразуются в машинный код процессора, поэтому для того, чтобы стать первоклассным программистом, необходимо хорошо представлять себе основные принципы работы компьютера. Чем больше вы знаете об устройстве компьютера, чем лучше вы разбираетесь в функциях DOS и ROM BIOS, тем полнее вы сможете использовать возможности ассемблера при написании компьютерных программ.

В следующих главах подробно объясняется, как управлять различными частями аппаратного обеспечения компьютера. Например, в одной главе рассматриваются клавиатура и дисплей, в другой объясняется работа с последовательными портами. Цель этой главы — помочь вам разобраться в том, что действительно происходит внутри компьютера, и показать, как с помощью команд ассемблера можно управлять оборудованием компьютера.

Замечание

Как вы, вероятно, знаете, MS DOS и PC DOS почти как две капли воды похожи одна на другую. Для простоты в этой книге обе дисковые операционные системы обозначаются как DOS. Для обозначения стандартных программ, хранящихся в машинных кодах в постоянном запоминающем устройстве, используется термин BIOS (базовая система ввода-вывода). Вообще говоря, BIOS управляет аппаратным обеспечением компьютера, в то время как DOS обеспечивает стандартный интерфейс (способ взаимодействия), который для разных систем может быть различным.

Отладка

Иногда сложно найти соответствие между операторами программы и собственно результатами ее выполнения в основном из-за зашифрованных ассемблерных команд `mov ax,bx` или `xor cx,cx`, которые, на первый взгляд, не имеют ничего общего с появляющимися на экране символами, распечаткой текста или связью через модем с удаленной системой. Попытка понять программу с помощью мысленного выполнения малосодержательных команд языка ассемблера может вывести из себя даже самого стойкого программиста. Но это не должно портить ваше настроение, так как это сложно постичь любому.

Одним из способов, возможно самым лучшим, для понимания работы программ на языке ассемблера является использование программы-отладчика Turbo Debugger, поставляемой с большинством версий Borland C++ и Pascal. Многие считают, что отладчик полезен только при исправлении ошибочных программ. Однако он может не только лечить, но и предотвращать "заболевания". С помощью Turbo Debugger можно по ходу выполнения программы просматривать память, наблюдать за изменениями в регистрах процессора, видеть значения отдельных байтов памяти и пошагово исполнять программу. Кроме того, вы можете просматривать команды языка ассемблера вместе с соответствующими им машинными кодами, видя при этом, во что Turbo Assembler преобразовал исходный текст вашей программы.

Изучая программу, исполняемую с помощью Turbo Debugger, вы сможете лучше разобраться в специфических командах языка ассемблера. В следующих главах я буду часто предлагать вам воспользоваться Turbo Debugger для проверки значений регистров и флагов, изучения областей памяти, исполнения программ до определенного места. Он позволяет в удобном для вас режиме наблюдать за тем, что делает программа на каждом шаге.

Замечание

В предыдущем разделе я использовал два новых термина — регистр и флаг. Регистр — это небольшой фрагмент памяти внутри центрального процессора, не сохраняемой при выключении питания. Как вы увидите, целый ряд машинных команд непосредственно оперируют регистрами Флаг — это однобитовый переключатель, также расположенный внутри процессора и используемый определенными машинными командами. Подробнее на этом мы остановимся позже.

Доводы в пользу режима Ideal

Turbo Assembler в действительности содержит в себе два ассемблера, точнее, два синтаксических стандарта языка ассемблера — *MASM* и *Ideal*. Обычно Turbo Assembler обрабатывает программы, написанные в популярном стандарте, называемом *MASM* (сокращенная запись для Microsoft Assembler). Он используется для ассемблирования программ, заимствованных из систем с разделением времени или набранных из книг по MASM.

В примерах этой книги используется режим *Ideal*, так как я считаю, что он превосходит стандарт MASM, особенно при написании чисто ассемблерных программ. В режиме Ideal программы ассемблируются быстрее, вы будете совершать в них меньше ошибок, чем в режиме MASM с его многими известными странностями и упрощениями в синтаксисе. (Отличия между командами режимов MASM и Ideal описаны в Руководстве пользователя по языку ассемблера.)

Кроме большей скорости и отсутствия странностей поведения, режим Ideal обладает и другими преимуществами. Структуры (аналогичные записям в Pascal и структурам в C) могут использовать одинаковые имена полей. Директивы ассемблера в этом случае легче запоминаются и используются. Всегда можно определить значения любых выражений и равенств. Кроме того, различные методы адресации памяти должны соответствовать общепринятым стандартам программирования. Если вы до сих пор не осознали важности некоторых пунктов, то поверьте моему мнению — программистам на языке ассемблера еще долгое время будет необходим режим Ideal.

Однако не надо считать, что, изучая режим Ideal, вы отказываетесь от использования большого количества свободно распространяемых MASM-программ. После изучения режима Ideal вы сможете, приложив лишь минимальные усилия, читать и понимать программы, написанные в режиме MASM. Между двумя режимами существуют тонкие различия: незначительные изменения в написании операторов могут принципиально менять их назначение. Я без особого труда постоянно работаю с программами, написанными в обоих стандартах, но при написании новых предпочитаю использовать режим Ideal.

Преимущества языка ассемблера

Преимущества и недостатки программирования на языке ассемблера описаны во многих книгах. Очевидны и наиболее известны такие преимущества, как низкоуровневый доступ к компьютеру, достижение максимальной скорости за счет возможности полностью управлять процессором. Программы на языках высокого уровня исполняются медленнее ассемблерных, так как компиляторы C и Pascal используют стандартные методы для записи и чтения переменных, вызова процедур и т.п. В языке ассемблера, при желании, вы можете разместить переменную в доступном для чтения регистре процессора. Однако, несмотря на подобные утверждения, сама по себе скорость не гарантируется при программировании на языке ассемблера. Опытные программисты на C и Pascal могут создавать программы, работающие быстрее, чем неумело написанные программы на ассемблере. Ассемблер предоставляет вам только возможность создавать максимально эффективные программы — достойная цель, требующая на практике много времени и терпения. И если скорость — это ваша цель, то вы можете быть уверены в том, что выбрали правильный путь.

Недостатки языка ассемблера

Основные недостатки программирования на языке ассемблера часто связываются с повышенным риском совершения ошибок, слабой переносимостью на другие системы и отсутствием библиотеки подпрограмм для выполнения операций вывода строки или чтения данных из файла на диске. Давайте рассмотрим их в отдельности.

Повышенный риск совершения ошибок. Я не согласен с этим мнением. Ошибки — результат невнимательности, а не следствие некоторых особенностей языка программирования. На любом языке можно писать неправильные программы, но это не мешает создавать программы без ошибок на языке ассемблера. Действительно, простые ошибки в программе на ассемблере могут часто приводить к более серьезным последствиям, чем ошибки в C и Pascal. Ассемблер позволяет вам полностью управлять процессором, поэтому единственный ошибочный оператор значительно чаще может приводить к зависанию системы, чем в языках высокого уровня, в которых машинный код для вас создается компилятором. Одним из путей решения этой проблемы является исполнение программ под управлением Turbo Debugger, который позволит вам уменьшить вероятность сбоев системы.

Замечание

При написании этой книги я понял, почему для многих программирующих на ассемблере зависание системы с последующей перезагрузкой или выключением компьютера стали обычными процедурами. По мере изучения Turbo Debugger я практически избавился от сбоев системы. Теперь перед запуском новой программы я всегда предварительно проверяю ее с помощью Turbo Debugger.

Слабая переносимость. Естественно, что язык ассемблера привязан к процессору, под который разрабатывается программа. Команды ассемблера преобразуются непосредственно в машинные коды и поэтому могут исполняться только на компьютерах с совместимыми процессорами. Для переноса ассемблерной программы на компьютер с отличным процессором приходится многое переделывать. Я согласен с этим. Платой за преимущества языка ассемблера является невозможность простого переноса программ на другие системы. Нельзя получить все сразу.

Отсутствие библиотечных подпрограмм. Во всех языках высокого уровня имеются команды для выполнения таких распространенных операций, как вывод строк, печать текста, обработка дисковых файлов. Кроме того, языки высокого уровня позволяют вам использовать математические выражения, например (x * 2 + 8) В ассемблере отсутствуют такие возможности, и вам необходимо самим писать код для подобных операций. Хотя это действительно так, но не надо забывать, что при использовании библиотек в языках высокого уровня теряется возможность полного управления ресурсами компьютера и, следовательно, написания максимально эффективных и быстрых программ. Более того, существует большое количество ассемблерных библиотек, позволяющих выполнять характерные операции высокого уровня. Хотя на ассемблере многие действия сложнее, но в языках высокого уровня не существует ничего такого, чего нельзя сделать на ассемблере. Кроме того, вы можете использовать определенные возможности в C, C++ или Pascal, комбинируя языки высокого уровня с языком ассемблера (об этом подробнее в главах 12 и 13).

Требования к аппаратному обеспечению

Для полноценного использования этой книги вам необходимо иметь, как минимум, следующее оборудование:

- IBM PC, XT, AT, PS/2 или совместимый с ними компьютер
- 384 Кбайт памяти (256 Кбайт, если не использовать Turbo Debugger)
- Один или два дисковода
- Монохромный или цветной дисплей

Для простоты базовую систему, подходящую для ввода и исполнения большинства примеров из этой книги, я буду называть PC. Вам будет проще, если в вашем распоряжении будет следующее необязательное оборудование:

- Принтер
- Жесткий диск
- Дополнительная память

Почти все программы из этой книги работают на любом IBM-совместимом компьютере с процессором 8086, 8088, 80286, 80386, 80486 или Pentium. Некоторые программы, однако, требуют процессоров 80386 или 80486 (либо эквивалентных). Для Windows-программ необходим жесткий диск, но только потому, что он требуется самой Windows.

Замечание

Я буду часто употреблять словосочетание „процессор 8086" и обсуждать методы „программирования на 8086". Если не оговорено иное, то все сказанное будет относиться и к процессорам 8088, 80286, 80386, 80486 и Pentium, которые применяют тот же набор команд, что и процессор 8086. В некоторых книгах, руководствах и статьях для описания семейства процессоров Intel используются термины 80×86 и iAPX-86. В этой книге используется более простое обозначение — *8086*.

Требования к программному обеспечению

Кроме аппаратного обеспечения, перечисленного в предыдущем разделе, вам будет необходимо следующее программное обеспечение:

- Turbo Assembler 4.0 и Turbo Debugger 4.0
- DOS 4.01 или выше
- Необязательно: Microsoft Windows 3.1 или следующая версия (для программ из главы 15)

Вероятно, вы сможете использовать большинство приемов программирования из этой книги с Turbo Assembler 3.2 и Turbo Debugger 3.2, входящими в состав Borland Pascal 7.0. Однако я проверял все программы в Turbo Assembler 4.0.

Для ввода текстов программ вам также потребуется текстовый редактор, так как он не входит в пакет Turbo Assembler. Вы можете использовать один из следующих редакторов:

- Редактор Borland Pascal или C++
- Brief
- VEdit Plus
- Edit (из MS DOS)
- Epsilon
- WordStar (в недокументном режиме)
- SideKick или SideKick Plus
- MS (MicroStar)
- ME (MultiEditor)

При наличии у вас одного из языков фирмы Borland используйте редактор, встроенный в интегрированную среду. Вы можете также использовать любой экранный ASCII-редактор, но не пользуйтесь текстовыми процессорами типа WordPerfect, которые добавляют к тексту специальные символы форматирования.

После ввода программы или просмотра соответствующего файла на диске примените для возврата в DOS команду "временный выход в DOS" вашего редактора, затем следуйте рекомендациям, предшествующим каждому примеру программы. После ассемблирования и исследования программы для возврата к редактированию наберите EXIT и нажмите <Enter>. Если в вашем текстовом редакторе отсутствует команда временного выхода в DOS, то необходимо выйти из редактора, ассемблировать программу, затем снова загрузить его для ввода следующего примера. Некоторые редакторы, например Brief, могут непосредственно вызывать Turbo Assembler, но для запуска на исполнение результирующих программ все равно необходимо выйти в DOS.

Пользователям Microsoft Windows

Если вы работаете в Microsoft Windows, то для редактирования и ассемблирования примеров программ из этой книги откройте окно DOS. За исключением Windows-программ из главы 15, вам необязательно ассемблировать и исполнять их непосредственно как приложения Windows.

Однако некоторые программы из этой книги в окне DOS могут исполняться некорректно, так как Windows берет на себя исполнение функций DOS и прерываний ROM BIOS. Я буду предупреждать вас о любых подобных случаях. Если у вас возникнут проблемы с исполнением некоторых программ, выйдите из Microsoft Windows и попытайтесь запустить их из DOS.

Как пользоваться этой книгой

Начинающим необходимо читать эту книгу от начала до конца. Тексты примеров программ специально подобраны таким образом, чтобы избежать использования еще не введенных терминов. Если вы читаете главы не по порядку, то имейте в виду, что многие примеры программ используют ранее описанные модули. Например, вы можете не разобраться в программах из главы 9, если не прочитаете в предыдущих главах о модулях, используемых в этих программах. Для поиска определенных тем обратитесь к содержанию.

О главах

Для того чтобы вы смогли использовать книгу как учебник и как справочник, все главы этой книги организованы одинаково. На первой странице каждой главы приводятся названия основных разделов. Затем следует текст главы, заканчивающийся резюме, списком упражнений для проверки ваших знаний и, за исключением этой главы, предлагаемыми заданиями.

Книга разделена на три части. Часть I, "Программирование на языке ассемблера", является руководством по языку ассемблера для процессора 8086. Часть II, "Прикладное программирование", описывает, как комбинировать язык ассемблера с Pascal, C, C++, как использовать объектно-ориентированные черты Turbo Assembler и как создавать с помощью языка ассемблера Windows-приложения. Часть III, "Справочник", содержит описания команд процессора и Turbo Assembler. Ниже приведено краткое описание каждой главы.

- В главе 1 вводятся основные понятия программирования на языке ассемблера, объясняется, как использовать эту книгу, даются другие рекомендации, о чем вы, несомненно, знаете, если дочитали до этого места!
- В главе 2 описывается структура программы на языке ассемблера, рассказывается, как пользоваться командами Turbo Assembler и Turbo Debugger, объясняется, как создавать EXE- и COM-файлы.
- В главе 3 рассматриваются основы двоичной системы счисления, в особенности, необходимые для программирования на языке ассемблера. Начинающие, не пропускайте эту главу! Специалисты, просмотрите материал, чтобы освежить его в памяти.
- В главе 4 исследуются различные аспекты сегментации памяти и приводится большинство команд процессора 8086.
- В главе 5 объясняются режимы адресации и показывается, как выделять память под переменные. Кроме того, вы научитесь пользоваться утилитой TLIB, с помощью которой строится библиотечный файл, содержащий модули из этой книги и необходимый для примеров из следующих глав.
- В главе 6 расширяются темы, введенные в главе 5; показано, как создавать структуры с большим количеством полей, объединения, массивы и упакованные записи с битовыми полями.
- В главе 7 содержатся рекомендации для чтения с клавиатуры и записи текста в стандартный выходной файл (обычно вывод на дисплей). Для этого в некоторых примерах вызываются стандартные программы DOS и ROM BIOS. Другие примеры показывают, как, используя запись непосредственно в видеобуфер, можно увеличить производительность дисплея.
- В главе 8 показано, как объединять в макросы повторяющиеся команды, добавляя тем самым новые удобные команды в Turbo Assembler. Кроме того, обсуждаются методы условного ассемблирования, используемые при написании широкопрофильных программ, которые в соответствии с требованиями могут ассемблироваться по-разному.
- В главе 9 рассматриваются методы создания, чтения и записи файлов данных, хранимых на диске, а также объясняется, как с помощью языка ассемблера читать дисковые каталоги.
- В главе 10 рассматриваются запутанные и часто неправильно понимаемые случаи написания служебных программ обработки прерываний на примере таймера и доступа к параллельным портам ввода-вывода.
- В главе 11 обсуждаются некоторые реже употребляемые (и, возможно, плохо понимаемые) возможности Turbo Assembler.
- В главе 12 открываются хитрые секреты взаимодействия между языком ассемблера и Turbo Pascal, позволяющие оптимизировать качество программ.
- В главе 13 показано, как организуется взаимодействие между языком ассемблера и Borland C++; основное внимание, как и в главе 12, уделяется вопросам оптимизации.
- В главе 14 объясняется, как использовать в Turbo Assembler возможности объектно-ориентированного программирования (ООП), а также описываются преимущества и недостатки использования технологии ООП в языке ассемблера.
- В главе 15 содержится руководство по написанию на языке ассемблера приложений для Windows. (Для программ этой главы требуется Microsoft Windows 3.1 или совместимые более поздние версии.)
- В главе 16 содержится алфавитный справочник по наборам команд процессоров 80×86 (включая неиспользуемые в прикладном программировании команды защищенного режима).
- В главе 17 содержится список заранее определенных символов, операторов, эквивалентов для режимов MASM, Ideal и директив Turbo Assembler.

О модулях

Многие приведенные в этой книге программы разработаны в виде отдельных модулей, которые вы можете ассемблировать и записывать в библиотечный файл для применения другими программами. В книге рекомендуется использовать библиотечный файл с именем MTA.LIB, но, при желании, вы можете записывать модули и в другой файл.

По мере изучения книги в библиотеку будут включены следующие модули:

- STRING.ASM — пакет подпрограмм для работы с ASCII-строками
- STRIO.ASM — программы для чтения и записи ASCII-строк
- BINASC.ASM — утилиты для числовых и строковых преобразований
- SCREEN.ASM — процедуры для работы с видеопамятью

- KEYBOARD.ASM — программы для обработки нажатых клавиш, включая функциональные
- DOSMACS.ASM — макросы для вызова функций DOS
- DISKERR.ASM — программы для расшифровки ошибок диска
- PARAMS.ASM — программы для чтения параметров командной строки DOS
- ASYNCH.ASM — программы для управления последовательными портами ввода-вывода

Как подготовить ваши диски

Жесткие диски

Со времени первого издания этой книги жесткие диски получили более широкое распространение. Если у вас в распоряжении нет жесткого диска, то для помощи при установке ассемблера на компьютере с гибкими дисками прочтите следующий раздел — "Использование гибких дисков".

Порядок установки Turbo Assembler зависит от используемой вами версии. Некоторые версии устанавливаются автоматически в составе других программных продуктов Borland, например в Pascal 7.0. Другие должны устанавливаться в уже существующих каталогах. (Turbo Assembler 4.0, например, обычно устанавливается в C:\BC4\BIN — "бинарном" каталоге Borland C++.)

Установите Turbo Assembler в соответствии с Руководством пользователя, прилагаемым к вашей версии языка. Чтобы проверить правильность установки, выйдите в DOS (откройте окно DOS, если вы используете Microsoft Windows) и введите команду tasm. В результате на экране должны появиться строки

```
Turbo Assembler Version 4 0 Copyright (c) 1988, 1993 Borland International
Syntax  TASM [options] source [,object] [,listing] [,xref]
```

и соответствующий список опций командной строки. Если вам не удалось запустить ассемблер, то, возможно, неправильно определен оператор PATH. Проверьте, чтобы в файле AUTOEXEC.BAT содержалась строка, подобная следующей:

```
PATH=C \WINDOWS,C \DOS,C \BC4\BIN
```

Пользователям Borland Pascal 7.0 необходимо изменить C:\BC4\BIN на C:\BP\BIN (или на каталог, в котором установлены исполняемые файлы Pascal).

Некоторые версии Turbo Assembler, в частности используемые с Borland Application Frameworks, устанавливают Turbo Assembler и Turbo Debugger в отдельных каталогах. При этом вы должны установить PATH таким образом:

```
PATH=C \WINDOWS,C \DOS,C \TASM,C \TD
```

Использование гибких дисков

Если вы работаете на компьютере, на котором нет жесткого диска, то можете запускать Turbo Assembler и большинство программ, приведенных в этой книге, с дискет. Хотя вы и не сможете выполнить некоторые сложные примеры, в частности, требующие Microsoft Windows, однако на компьютере с двумя дисководами — A: и B:, можно будет все же научиться основам программирования на языке ассемблера. Поэтому, чтобы начать программировать, достаточно даже такой дешевой конфигурации компьютера.

Создайте загрузочную дискету с файлами операционной системы, COMMAND.COM (программа DOS, позволяющая задавать команды операционной системы и запускать другие программы из командной строки DOS), вашим текстовым редактором и Turbo Assembler. Для создания этого диска загрузите свой компьютер с загрузочной дискеты DOS с дисковода A:. Вставьте чистую дискету в B: и введите следующую команду (опция /s переносит на диск системные файлы):

```
format b  /s
```

Скопируйте также другие необходимые вам файлы. К примеру, чтобы использовать для ввода и просмотра исходных текстов программу DOS EDIT, скопируйте ее на свой диск (полная команда зависит от места расположения файла EDIT.EXE):

```
copy a \edit exe b
```

И наконец, скопируйте на дискету исполняемый файл Turbo Assembler (TASM.EXE):

```
copy tasm exe b
```

Полный вид команды опять-таки зависит от вашей версии Turbo Assembler. Некоторые версии могут устанавливаться непосредственно на гибкий диск. Для дополнительных инструкций по установке обращайтесь к Руководству пользователя, которое поставляется вместе с ассемблером или компилятором.

После создания дискеты с Turbo Assembler отредактируйте или создайте файл AUTOEXEC.BAT, включив в него строку вида

```
PATH=A \;B.\
```

Это сделает возможным запуск программ с обоих дисководов, независимо от текущего. Единственным недостатком такого подхода является то, что в обоих дисководах все время должны находиться отформатированные дискеты, иначе вы можете получить сообщение об ошибке чтения дискеты ("not ready"). В этом случае, вставив дискету, нажмите клавишу <R> для повторного чтения дискеты. Теперь перезагрузите компьютер и работайте с дискетами.

Предыдущие версии Turbo Assembler

Для большинства программ из этой книги вы сможете использовать предыдущие версии Turbo Assembler. Однако, в зависимости от вашей версии, вы не сможете либо использовать объектно-ориентированные возможности, либо создавать приложения для Microsoft Windows. Лучше, если вы будете использовать версию Turbo Assembler 4.0, так как все программы из этой книги проверялись на ней, хотя можно использовать и версию 3.0.

Ввод исходных текстов программ

Если вы набираете исходные тексты программ в своем любимом текстовом редакторе, вводите их точно так, как они напечатаны в книге, за исключением номеров строк, расположенных слева (если они имеются). *Эти номера используются исключительно для ссылок в книге — не набирайте их.* Постарайтесь делать одинаковый отступ при наборе команд. Если вы будете выравнивать колонки по аналогии с примерами в книге, вы сможете лучше понимать команды языка ассемблера. Для экономии времени используйте в редакторе клавишу табуляции.

В главах применяется сплошная нумерация примеров программ (1.5, 4.3 и т.д.), после номера следует имя файла. Не изменяйте эти имена, они иногда используются другими программами, а в случае их изменения у вас могут в дальнейшем возникнуть сложности при запуске этих программ.

Резюме

Цель этой книги — ввести вас в достаточно сложный мир программирования на языке ассемблера для IBM PC и совместимых компьютеров в среде DOS и Windows. Освоение языка ассемблера намного проще, чем об этом говорят. Множество примеров из этой книги поможет вам приобрести опыт программирования, недостаток которого иногда испытывают даже профессиональные программисты. Приведенные программы хорошо спроектированы и оттестированы, и вы можете свободно использовать их в своей работе.

Когда появились первые компьютеры, программисты пользовались непосредственно низкоуровневыми машинными кодами, однако при появлении языка ассемблера, каждая команда которого соответствует определенному машинному коду, стало гораздо удобнее писать программы, управляющие низкоуровневыми процессами компьютера. Ассемблер стоит только на одну ступень выше машинного кода, в то время как C, Pascal, Basic, PROLOG и другие являются языками высокого уровня. Поскольку ассемблер максимально приближен к машинному коду процессора, для его изучения очень полезно представлять себе основные принципы работы компьютера. Для понимания работы ассемблерных программ можно использовать Turbo Debugger.

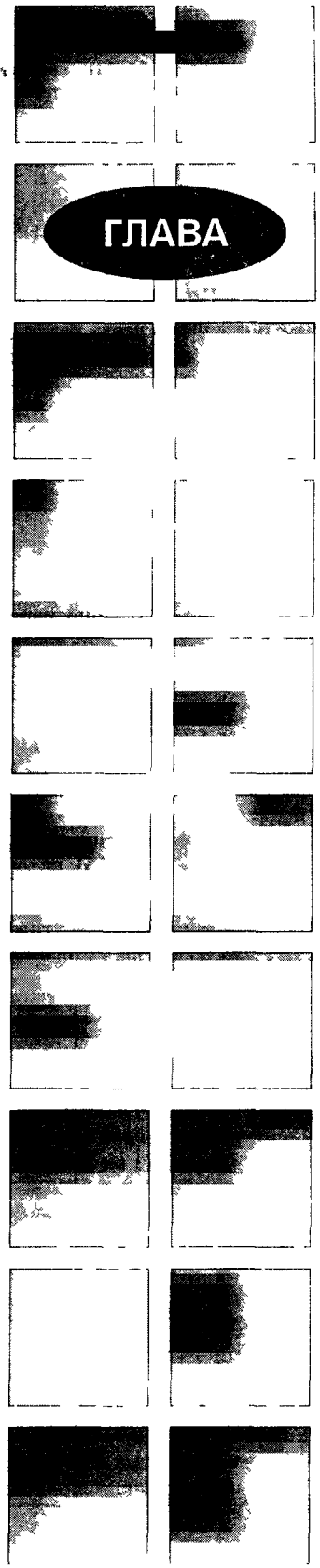
Turbo Assembler запускается в двух режимах: MASM и Ideal. Все программы этой книги написаны для режима Ideal.

Язык ассемблера, подобно другим языкам, имеет свои преимущества и недостатки. Основным преимуществом является максимальная скорость выполнения программы и возможность непосредственного программирования процессора. Основной недостаток связан с тем, что ассемблерная программа может исполняться только на том процессоре, для которого она была написана.

Номера строк в приводимых примерах программ используются только для удобства ссылок в книге. При вводе программ набирать номера и точки не следует. Для достижения лучших результатов рекомендуется пользоваться Turbo Assembler 4.0.

Упражнения

1. Почему термин "машинный язык" не совсем правильный?
2. Что означают термины "высокий уровень" и "низкий уровень" при описании компьютерных языков?
3. В чем заключаются основные различия между ассемблером и языком высокого уровня?
4. Почему программисты больше не разрабатывают программное обеспечение непосредственно в машинных кодах?
5. Каким образом отладчик может помочь при изучении языка ассемблера?
6. Что означает регистр?
7. Что означает флаг?
8. Какими преимуществами обладает режим Ideal в Turbo Assembler?
9. Какие основные преимущества программирования на языке ассемблера?
10. Какие основные недостатки программирования на языке ассемблера?



ГЛАВА 2

Первые шаги

- Основные части программы на ассемблере
- Ассемблирование программы
- Понимание объектного кода
- Опции командной строки
- Обработка ошибок
- Введение в Turbo Debugger
- Написание COM- и EXE-программ
- Распечатка листингов программ
- Резюме
- Упражнения
- Задания

Основные части программы на ассемблере

Язык ассемблера — необычный компьютерный язык. Почти весь текст исходной программы состоит из непроизносимых слов `cli`, `movsb` и `sbb`, которые, на первый взгляд, следуют в полном беспорядке и никак не связаны одни с другими. Можно долго удивляться комментариям — тексту, расположенному после точки с запятой в конце большинства ассемблерных строк; их содержание часто кажется не связанным с командами программы.

Одной из причин этого кажущегося (но обманчивого) беспорядка в языке ассемблера является недостаток команд, составляющих структуру программы. Отсутствуют операторы для группирования повторяющихся действий (REPEAT-UNTIL или WHILE), нет операторов для принятия решений (IF-THEN-ELSE или CASE), нет и символа присваивания для инициализации переменных. Для выполнения таких высокоуровневых действий используется множество низкоуровневых машинных команд, что делает тексты на языке ассемблера чрезвычайно похожими, затемняя тем самым содержательную сторону программы. Кроме того, ассемблер является строчным языком, а не операторным, как C, Pascal или BASIC. Таким образом, выполнение даже таких простых операций, как сложение чисел или инициализация переменных, требует написания большого количества строк на языке ассемблера.

Однако в этом кажущемся хаосе существует определенный порядок. Большинство программ на языке ассемблера можно разделить на пять основных частей: *заголовок, макроопределения, данные, тело и заключение*. (Эти термины ввел я, так как не существует общепринятых названий для отдельных частей ассемблерных

БІБЛІОТЕКА
ПЕРВОГО ДЕРЖАВНОГО
УНІВЕРСИТЕТУ
БОГДАНА ХМЕЛЬНИЦЬКОГО
540310

программ). *Заголовок* содержит установочную информацию. В *макроопределениях* определяются переменные, которым вы присваиваете значения различных выражений и констант. В *данных* определяются переменные, которые будут храниться в памяти. Тело содержит собственно код программы. *Заключение* отмечает конец исходного текста.

Рассмотрим подробнее каждую из этих частей.

Заголовок

Программа на языке ассемблера начинается с заголовка. В нем содержатся команды и директивы, не приводящие к созданию машинного кода при трансляции. Они указывают ассемблеру, как выполнить определенные действия, генерируя исполняемый файл.

На рис. 2.1 представлен простой заголовок, характерный для большинства примеров программ в этой книге. (Это фрагмент программы, так что не пытайтесь его транслировать.)

Необязательная строка %TITLE описывает назначение программы, и при распечатке ее в Turbo Assembler текст, находящийся в кавычках, будет располагаться в начале каждой страницы исходного текста. Директива IDEAL переводит Turbo Assembler в режим Ideal.

Если программа написана на Microsoft Macro Assembler (MASM), директиву Ideal нужно опустить.

```
%TITLE 'Тестовый заголовок--Не ассемблируйте"
IDEAL
MODEL small
STACK 256
```

Рис 2 1. Типичный заголовок ассемблерной программы

Далее следует директива MODEL, выбирающая одну из нескольких моделей памяти (табл. 2.1), многие из которых используются при написании комбинированных ассемблерных программ с языками C или Pascal. Наиболее удачный выбором при программировании на языке ассемблера является small (малая) модель памяти. Однако пусть вас не смущает такое название. Малая модель памяти выделяет вам 64 Кбайт — под исполняемый код и 64 Кбайт под данные и позволяет создавать программы размером до 128 Кбайт, что практически является пределом эффективности в мире машинных кодов.

Директива STACK на рис. 2 1 резервирует пространство для стека программы — области памяти, в которой содержатся два типа данных: промежуточные значения, записываемые подпрограммами или передаваемые им, а также адреса, на которые подпрограммы возвращают управление. (В главе 10 изучается использование стека при обработке прерываний.) Работа со стеком является важной частью технологии программирования на ассемблере и будет детальнее рассматриваться в других главах. Значение, следующее за директивой STACK на рис. 2.1, определяет количество байтов (256), которое будет зарезервировано Turbo Assembler под сегмент стека. В основном программы используют небольшой стек, и даже самые большие из них редко требуют более 8 Кбайт.

Таблица 2 1 **Модели памяти**

Название	Код	Данные	Определение	Описание
tiny	near	near	cs = dgroup ds = ss = dgroup	Код, данные и стек содержатся в одном сегменте 64 Кбайт. Используется только для COM-программ
small	near	near	cs = _text ds = ss = dgroup	Код и данные содержатся в различных сегментах, размером до 64 Кбайт. Используется для небольших и средних EXE-программ. Наилучшим образом подходит для большинства чисто ассемблерных программ
medium	far	near	cs = <module>_text ds = ss = dgroup	Неограниченный размер кода. Под данные отводится один сегмент 64 Кбайт. Используется для написания больших программ с небольшим объемом данных
compact	near	far	cs = _text ds = ss = dgroup	Размер кода ограничен одним сегментом 64 Кбайт. Размер данных неограничен. Используется при написании малых и средних по размеру программ с большим количеством переменных
large	far	far	cs = <module>_text ds = ss = dgroup	Размер кода и данных неограничен. Используется в больших программах. Размер переменной не может превышать 64 Кбайт

Окончание таблицы 2.1

Название	Код	Данные	Определение	Описание
huge	far	far	cs = <module>_text ds = ss = dgroup	Размер кода и данных неограничен. Аналогична большой модели памяти. (Введена для совместимости с языками высокого уровня)
tchuge	far	far	cs = <module>_text ds = nothing ss = nothing	Совпадает с большой моделью памяти, но по-другому определяет регистры. В основном используется при программировании на языках Turbo C и Borland C++
tpascal	near	far	cs = code ds = data ss = nothing	Обеспечивает поддержку ранних версий Turbo Pascal. В Borland Pascal не применяется
flat	near	near	cs = _text ds = ss = flat	Используется только в среде OS/2; в остальном аналогична малой модели памяти

Макроопределения

После заголовка программы следуют различные описания констант и переменных. В языке ассемблера константы часто называют *макроопределениями*, использующими директиву EQU, которая связывает значение с идентификатором (определенным именем MaxValue или PortAdress, вместо которого в программе Turbo Assembler подставит указанное значение). Исключительно для числовых значений, помимо директивы EQU, можно применять знак равенства (=).

Замечание

Макроопределения могут располагаться в любом месте программы, но чтобы ваша программа легко читалась, размещайте макроопределения сразу после заголовка программы.

Использование идентификаторов макроопределений вместо "магических" чисел 0100h или 0B800h позволяет вам обращаться по именам к выражениям, строкам и другим величинам, что упрощает программу для чтения и отладки. Ниже приводится несколько примеров макроопределений, которые могут следовать за заголовком, представленным на рис. 2.1:

```
Count       EQU     10
Element     EQU     5
Size        =       Count * Element
MyBoat      EQU     "Gypsy Venus"
Size        =       0
```

Несмотря на то что большинство имен макроопределений просто заменяется соответствующими им значениями и выражениями — аналогично использованию констант в языках Pascal или C, существует несколько строгих правил, которые надо помнить при создании и использовании макроопределений в языке ассемблера.

- После описания имени константы с помощью директивы EQU вы не можете изменять его значение. Переопределение этой константы в равенстве не допускается (например, изменение значения Count на 11).
- Это ограничение не распространяется на имена-идентификаторы, описанные с помощью знака равенства (=), — вы можете свободно изменять их значения. Обратите внимание, как в рассмотренном выше примере значение Size изменяется с 50 на 0. Можно осуществлять это в любом месте программы, а не только в секции равенств.
- EQU может описывать все типы равенств, включая числа, выражения и символьные строки. Знак равенства может описывать только числовые равенства, состоящие из чисел 10 или 0Fh либо выражений Count*Size или Address+2.
- Идентификаторы макроопределений равенств не являются переменными — ни они, ни их значения не содержатся в сегменте данных программы. Команды ассемблера не могут изменить значения идентификатора в макроопределениях, независимо от того, были они описаны с помощью директивы EQU или знака =.
- Хотя вы можете располагать макроопределения в любом месте программы, но для лучшей видимости предпочтительнее размещать их в начале текста. Макроопределения, расположенные глубоко в тексте программы, могут стать источником труднонаходимых ошибок.

- Выражения, описанные с помощью EQU, вычисляются, когда соответствующее имя-идентификатор используется программой. Выражения, описанные через знак равенства (=), вычисляются непосредственно в месте определения. Ассемблер сохраняет *текст* EQU-выражения, а для выражения, описанного с помощью знака равенства, — только его *значение*.

Для лучшего понимания последнего правила рассмотрим несколько примеров. Предположим, что есть три макроопределения:

```
LinesPerPage    =  66
NumPages        =  100
TotalLines      =  LinesPerPage * NumPages
```

Очевидно, что TotalLines равно произведению LinesPerPage и NumPages, или 6600. (В большинстве языков программирования звездочка "*" означает умножение.) Поскольку TotalLines описано с помощью знака равенства (=) — подразумевается числовое значение, выражение вычисляется сразу и результат ставится в соответствие TotalLines.

Если где-либо в программе вы измените значение NumPages, значение TotalLines останется прежним. Ситуация изменится, если вы опишете TotalLines с помощью EQU:

```
TotalLines    EQU   LinePerPage * NumPages
```

В этом случае Turbo Assembler сохранит не вычисляемое значение, а действительный текст выражения, следующего за директивой EQU (в нашем случае — текст выражения LinePerPage * NumPages). Позже в программе, когда вы будете использовать TotalLines, ассемблер вставит этот текст так, как будто вы набирали его в этом месте исходного текста, затем выражение будет вычислено и заменено конечным значением. Если перед этим вы изменили значение одного или обоих идентификаторов, используемых в выражении (NumPages или LinesPerPage), полученный результат также соответственно изменится.

Это влияние на значение выражения макроопределения может в ряде случаев оказаться полезным. Вы можете создать один программный модуль, на значения выражений макроопределений которого будут оказывать влияние макроопределения другого модуля. Нужно понимать тонкую разницу между макроопределениями, определенными с помощью знака равенства и директивы EQU. Их небрежное использование также может приводить к возникновению ошибок.

Сегмент данных

Сегмент данных программы обычно располагается между макроопределениями и командами программы. Можно, хотя это редко бывает полезным, определять сегмент данных в произвольном месте программы или иметь несколько сегментов данных в разных частях программного текста. Несмотря на эти возможности, ваша ассемблерная программа будет проще для понимания и модификации, если вы последуете простым рекомендациям, предложенным ниже, определяя все свои переменные между макроопределениями и исполняемым кодом.

Сегмент данных вашей программы должен начинаться с директивы DATASEG. Она дает указание ассемблеру разместить в памяти переменные, указанные в сегменте данных программы, который может достигать 64 Кбайт для малой модели памяти. Сегмент данных может содержать два типа переменных: *инициализированные* и *неинициализированные*. При исполнении программы инициализированные переменные имеют определенные значения, которые вы определили в тексте программы, и содержатся в файле программного кода на диске. Эти переменные автоматически загружаются в память и доступны для чтения при исполнении программы. Неинициализированные переменные аналогичны инициализированным, за исключением того, что они не занимают пространства в исполняемом файле и, следовательно, имеют неопределенные значения при исполнении программы. Вследствие этого определение большой неинициализированной переменной для резервирования в памяти большого буфера, заполняемого из дискового файла, не приводит к увеличению размера исполняемого файла программы.

Замечание

Для того чтобы неинициализированные переменные не содержались в объектном коде, они должны описываться после последней инициализированной переменной в исходном тексте программы. Неинициализированные переменные, описанные между другими инициализированными переменными, будут занимать место в объектном коде, тем самым увеличивая размер исполняемого файла на диске.

Резервирование пространства под переменные

Правила описания переменных в сегменте данных программы подробно рассматриваются в главе 5 на нескольких простых примерах; приводятся важные понятия, необходимые для дальнейшего понимания материала. Рассмотрим типичный сегмент данных, который может следовать после заголовка программы и макроопределений:

```
DATASEG
numRows       DB       25
```

```
numColums    DB      80
videoBase    DW      0B00h
```

Первой идет директива DATASEG, информирующая ассемблер о необходимости выделения пространства в памяти под сегмент данных программы. Затем определены три переменные: *numRows*, *numColumns* и *videoBase*. Как правило, я предпочитаю записывать константы (*Count*, *NumPages* и т.п.) с прописной, а переменные — со строчной буквы. Это необязательное соглашение, вы можете набирать символы по своему желанию либо прописными, либо строчными буквами. Некоторые программисты используют символ подчеркивания для улучшения читаемости названий переменных, состоящих из нескольких слов, например, записывая *num_rows* и *video_base* вместо слитного написания, используемого здесь.

DB (определить байт) и DW (определить слово) — две чаще всего используемые директивы, которые применяются для выделения пространства под соответствующие переменные программы. В отличие от языков высокого уровня, в которых расположение переменных в памяти обычно не является важным, в языке ассемблера необходимо выделять пространство в памяти под переменные и, в случае инициализированных переменных, приписывать значения этому пространству. Почувствуйте, насколько это отличается от идентификаторов макроопределений, которые ассоциируются с переменными исключительно в исходном тексте программы. Для них, в отличие от переменных, не выделяется пространство в сегменте данных памяти.

Имена переменных (numRows, numColumns и videoBase) являются метками, которые указывают на выделяемые в памяти области — в данном случае, на пространство, резервируемое под значения переменных. Программа может обращаться к этому пространству, используя метку как указатель на соответствующее значение в памяти. В ассемблерных программах метки преобразуются в адреса памяти, по которым содержатся переменные, что позволяет вам обращаться к памяти по именам, а не по численным адресам.

Замечание

При программировании непосредственно в машинных кодах вы должны вместо меток определять действительные адреса. Использование символьных меток для определения положения в памяти является одним из основных преимуществ языка ассемблера.

Переменные располагаются внутри сегмента данных непосредственно одна за другой. Зная это, можно проделывать различные "штуки". К примеру, кажется, что определения

```
DATASEG
aTOm         DB      "ABCDEFGHIJKLM"
nTOz         DB      "NOPQRSTUVWXYZ"
```

создают две символьные строки с метками aTOm и nTOz. Однако в памяти символы от A до Z хранятся последовательно, создавая одну строку букв алфавита. Метка nTOz просто указывает на середину строки, при этом в памяти не создаются два отдельных фрагмента.

Внимательные читатели могут спросить: "Подождите! Если DB означает 'определить байт', каким же образом он может определить строку символов?" Это хороший вопрос. DB имеет специальную возможность определять многобайтовые значения от одного до необходимого вам количества байтов. Строка состоит из отдельных ASCII-символов, каждый из которых занимает один байт, следовательно, директива DB является в ассемблере простым инструментом для определения строк символов, которые представляют собой просто записанный последовательно в памяти набор ASCII-символов. Вы можете использовать DB для определения отдельных символов и байтов, разделяя их запятыми:

```
DATASEG
perfectTen   DB      1, 2, 3, 4, 5, 6, 7, 8, 9, 10
theTime      DB      9, 0              ; т е. 9 00
theDate      DB      12, 15, 98        ; т е  12/15/1998
```

Кроме того, можно комбинировать символьные и байтовые значения, создавая двухстроковые переменные с ASCII-символами возврата каретки (13) и конца строки (10). Следующий пример показывает, что символьные строки могут заключаться как в одинарные, так и в двойные кавычки:

```
combo        DB  'Line #1', 13,  10, "Line #2"
```

Некоторые языки, в частности Pascal, различают одинарные символы и строки символов. В ассемблере символы и строки отличаются исключительно длиной. В ассемблерных строках отсутствуют вспомогательные значения вроде байта длины или символа конца строки, если, конечно, вы сами не разместили их там.

Строки будут рассматриваться подробнее при изучении команд ассемблера, специально созданных для работы с ними. Сейчас только запомните, что, в отличие от языков высокого уровня, строки в ассемблере являются просто набором последовательных значений в памяти, созданных директивой DB.

Тело программы

После сегмента данных располагается тело программы, известное под названием кодового сегмента — раздела памяти, который содержит исполняемый код вашей ассемблерной программы. Внутри этой области можно выделить четыре колонки текста: *метки*, *мнемонику*, *операнды и комментарии*. Каждая колонка имеет свою важную функцию, которую лучше рассмотреть на примере. Количество пробелов между колонками в тексте программы произвольное. Чаще всего колонки выравнивают с помощью одно- или двукратного нажатия клавиши табуляции в редакторе.

Замечание

Если ваш редактор позволяет выбирать между вставкой символов табуляции и вставкой пробелов, используйте табуляцию, задав ее в каждой восьмой колонке, что является стандартной установкой для большинства редакторов. Вставка символов табуляции позволяет легко выравнивать колонки. При этом вы сможете повторно редактировать текст каждой колонки, не смещая текст в других колонках. Конечно, при желании, вы можете вставлять и пробелы между колонками — ассемблер никаких различий при этом не делает.

Хотя вы еще не знакомы с командами языка ассемблера, тем не менее рассмотрим пример сегментов данных и кода на рис. 2.2 и попытаемся выделить четыре колонки (это незаконченная программа, так что не пытайтесь ее транслировать). Он содержит основные элементы кодового сегмента языка ассемблера. В сегменте данных для дальнейшего использования определяется однобайтовая переменная с именем exCode со значением 0.

Замечание

В первом издании во многих исходных текстах программ использовался идентификатор exitCode, который в новом издании заменен exCode. После выхода первого издания в Turbo Assembler 3.0 и следующих версиях слово EXITCODE зарезервировано под директиву. (Подробности — в главе 17.)

После директивы CODESEG на рис. 2.2 представлено несколько строк, разделенных на метки, мнемонику, операнды и колонки комментариев. В первой колонке находятся две метки — Start: и Exit:. Метки помечают места в программе, на которые могут ссылаться другие команды и директивы. Для строк без меток эта колонка не заполняется. В кодовом сегменте метка всегда заканчивается двоеточием (:), в сегменте данных двоеточие не используется (например, в метке exCode). Просто запомните это правило, за которым не кроется какой-либо логический смысл.

Во второй колонке содержатся мнемоники. Под каждой мнемонической формулировкой в этой колонке скрывается одна машинная команда: mov — для Move (пересылка данных), jmp — для Jump (безусловный переход), int — для Interrupt (прерывание). Некоторые мнемоники запоминаются легко: dec — для Decrement (декремент), shl — для Shift Left (сдвиг влево), ror — для Rotate Right (циклический сдвиг вправо). Другие кажутся делом рук сумасшедшей машинистки: jcxz — для Jump if cx is Zero (переход, если CX равно 0) и rcr — Rotate through Carry Right (циклический сдвиг вправо через перенос). В некоторых редких случаях мнемокод полностью совпадает с командой: out — для Out (вывод байта или слова в порт), push — для Push (занесение слова в стек), pop — для Pop (извлечение слова из стека).

Итак, как вы убедились, в языке ассемблера максимально используются разные сокращения. Для изучения названия и назначения каждого мнемокода вам потребуется много времени и терпения. Полный список всех мнемокодов процессора 8086 вы найдете в главе 4. В главе 16 содержится список всех мнемокодов с описанием связанных с ними команд. Обращайтесь к этой главе и старайтесь запомнить, по возможности, большее количество мнемокодов. При просмотре программы всегда произносите полные мнемонические имена. Через некоторое время это сделает для вас язык ассемблера если не легкочитаемым, то по крайней мере более понятным.

Третья колонка на рис 2.2 содержит операнды, которые обрабатываются мнемоническими командами. Некоторые команды не требуют операндов, в этом случае третья колонка остается пустой. Многие команды требуют двух операндов, другие — только одного. Ни одна из команд процессора 8086 не требует больше двух операндов. Первый оператор обычно называется *назначением*, второй — *источником*. Операнды могут иметь разные представления и будут изучаться при рассмотрении каждой конкретной мнемонической команды.

Четвертая и последняя колонка являются необязательными, и если включаются в программу, то должны начинаться с точки с запятой (;). Turbo Assembler игнорирует все символы от точки с запятой до конца строки, предоставляя вам место для размещения короткого комментария, описывающего выполняемые в данной строке действия. Почти все строки в каждом примере этой книги заканчиваются комментариями, которые вы можете пропустить для экономии времени при наборе программы вручную. При самостоятельной работе обязательно добавляйте четкие комментарии, которые полностью описывают вашу программу. Именно так надо начинать, в особенности если язык ассемблера является новым для вас, поскольку программы, написанные на нем, весьма сложны для чтения.

```
Метка       Мнемоника       Операнд          Комментарий

            DATASEG
exCode      DB              0                ;Байтовая переменная
            CODESEG
Start.      mov             ax, @data        ;Пересылка в DS адреса
            mov             ds, ax           ;сегмента данных
            jmp             Exit             ,Переход на метку Exit
            mov             cx, 10           ;Эта строка пропускается!
Exit
            mov             ah, 04Ch         ;Функция DOS: выход из программы
            mov             al, [exCode]     ,Возвращает код выхода программы
            int             21h              ,Вызов DOS  Останов программы
            END             Start            ,Конец программы/точка входа
```

Рис. 2.2. Четыре колонки в программе на языке ассемблера

Несколько замечаний по поводу комментариев

Иногда вы можете увидеть строку в ассемблерной программе, которая начинается с точки с запятой сразу в первой колонке. Такие, более длинные комментарии часто используются программистами для описания различных частей программы. Как и комментарии в конце строк, вы можете их опустить для экономии времени при наборе примеров программ из этой книги. Многие программисты начинают свои программы многострочным пояснительным комментарием:

```
; ---------------------------------------------------------------------
; Назначение  Предсказание выигрышных номеров лотереи
; Система     IBM PC / Turbo Assembler Ideal Mode
, Автор       Алексей Калязин
; ---------------------------------------------------------------------
```

В режиме MASM существует другой тип комментариев, но, к сожалению, он не поддерживается в режиме Ideal. В режиме MASM вы можете начинать большие комментарии с директивы COMMENT, за которой следует символ, называемый *разделителем комментария*, и непосредственно текст комментария, заканчивающийся аналогичным разделителем. Для того чтобы использовать такой комментарий в режиме Ideal, необходимо на время переключиться в режим MASM:

```
MASM

COMMENT /* Комментарий, который может располагаться на нескольких строках и который можно легко переформатировать с помощью команд используемого вами редактора */

IDEAL
```

После перехода в режим MASM многостроковый комментарий начинается с директивы COMMENT, косая черта используется в качестве разделительного символа. Завершается комментарий также косой чертой. (Звездочка используется для лучшего выделения комментария в тексте, а также для обозначения сходства с комментариями на языке C.) И наконец, директива Ideal возвращает Turbo Assembler в режим Ideal. Пустые строки после MASM и перед Ideal позволяют форматировать весь блок комментария, используя соответствующие возможности редактора, и упрощают редактирование длинных строк в тексте программы. Применяйте этот прием, если ваш редактор имеет аналогичные возможности.

Заключение

Последней частью программы на языке ассемблера является заключение — одиночная строка, информирующая Turbo Assembler о достижении конца программы. В заключении используется единственная директива END. Повторяя последнюю строку рис. 2.2, рассмотрим типичное заключение:

```
END         Start               ,Конец программы/точка входа
```

Директива END помечает конец текста исходной программы. Ассемблер игнорирует любой текст ниже этой строки и, между прочим, здесь можно разместить дополнительные комментарии. Справа от директивы END вы должны определить метку, с которой вы хотите начать выполнение программы. Обычно эта метка совпадает с меткой, указывающей на первую команду, следующую за директивой CODESEG.

Вы можете начинать исполнение программы с любой точки, хотя для этого вряд ли существуют особые причины.

Ассемблирование программы

В предыдущем разделе вы познакомились со структурой программы на языке ассемблера, узнали об основных частях любой ассемблерной программы. Следующим шагом является ассемблирование текста программы для получения исполняемого файла на диске. Рассмотрим эту процедуру на примере. Наберите текст программы FF.ASM, *приведенный ниже, используя свой текстовый редактор. Постарайтесь выровнять четыре колонки в соответствии с* напечатанным текстом. Не старайтесь скопировать программу один к одному, для экономии времени можно не набирать комментарии. Выйдите из своего редактора (или временно перейдите в DOS, если редактор имеет такую возможность) и наберите следующие строки:

```
tasm ff
tlink ff
```

Команда tasm запускает Turbo Assembler, который считывает файл FF.ASM и, если программа введена правильно, создает файл FF.OBJ, содержащий объектный код, еще не готовый для непосредственного исполнения. Если обнаружились ошибки, исправьте их и повторите те же действия. Команда tlink запускает Turbo Linker, который считывает файл FF.OBJ и создает исполняемый файл FF.EXE. Заметьте, что ни одна команда не требует, чтобы вы набирали расширения файлов (ASM или OBJ). Вы можете, при желании, набирать эти расширения, но зачем усложнять себе работу?

Теперь включите свой принтер. (Если его у вас нет, вы не сможете использовать эту программу. Простите!) Наберите FF в командной строке DOS и нажмите <Enter>. Этим вы запустили на исполнение файл FF.EXE, который посылает на принтер команду прокрутки страницы. Скопируйте FF.EXE в каталог с другими утилитами и запускайте эту программу вместо нажатия соответствующей кнопки на принтере. (Мой принтер стоит в другом конце комнаты, и я давно написал FF, чтобы не вставать со стула для прокрутки бумаги. Можете назвать меня ленивым.)

Программа 2 1 FF.ASM

```
1    %TITLE  Посылка на принтер команды прогона строки -- by Tom Swan
2
3:           IDEAL
4:
5:           MODEL   small
6.           STACK   256
7
8    ,----- Макроопределения
9
10   ASCIIcr EQU      13          ASCII-символ возврата каретки
11   ASCIIff EQU      12          ASCII-код прогона страницы
12
13           CODESEG
14
15   Start
16           mov     ax, @data  , Установить в DS адрес
17.          mov     ds, ax     ,  сегмента данных
18:
19:          mov     dl, ASCIIcr         , Переслать код возврата каретки в dl
20:          mov     ah, 05h             , Функция DOS   Вывод на принтер
21:          int     21h                 , Вызов DOS - возврат каретки
22:
23·          mov     dl, ASCIIff         , Переслать код прогона строки в dl
24.          mov     ah, 05h             , Функция DOS   Вывод на принтер
25·          int     21h                 , Вызов DOS - прогон страницы
26
27   Exit
28·          mov     ax, 04C00h          , Функция DOS   выход из программы
29:          int     21h                 , Вызов DOS  Останов программы
30:
31:          END     Start               , Конец программы/точка входа
```

Замечание

Некоторые читатели сообщили мне, что FF.EXE не работает на их принтерах. Программа работает нормально на совместимых с Epson FX-1050 матричных принтерах и по ряду причин может не работать на других системах. Например, команда прокрутки бумаги может быть запрещена на вашем принтере специальной установкой DIP-переключателей, либо у вас, возможно, не матричный, а лазерный или струиный принтер, которые для экономии бумаги не позволяют прокручивать пустые страницы. FF.EXE не является аппаратно-независимой программой — она просто показывает, как посылается байт на порт вывода, в нашем случае — код ASCII 12, на принтер. Если ваш принтер не понимает такой команды, значит, FF.EXE вам не пригодится. Такова сущность языка программирования ассемблера.

Понимание объектного кода

На предыдущем примере с программой FF.ASM вы увидели, что для преобразования ассемблерной программы из текстовой формы в исполняемую программу требуется выполнение двух шагов — ассемблирования и компоновки. Turbo Assembler никогда сразу не создает готовую к исполнению программу, а генерирует вместо этого промежуточный файл, содержащий ассемблерную программу в виде так называемого *объектного кода*. Перед запуском программы на исполнение вы должны обработать этот объектный код с помощью компоновщика, создав исполняемый EXE-файл на диске.

Такая последовательность шагов для простой программы может показаться слишком сложной, однако существуют убедительные причины для разделения процесса на ассемблирование и компоновку.

Как вы поймете в следующих главах, Turbo Linker (так же как и другие компоновщики) может объединять множество файлов объектного кода для получения одной исполняемой программы. Эта возможность позволяет разделить большую программу на небольшие фрагменты, ассемблировать их, создавая отдельные объектные файлы, а затем скомпоновать с помощью одной команды. Отдельные фрагменты, или модули, могут иметь общие данные и вызывать подпрограммы, описанные в других модулях. Большинство программистов предпочитают строить библиотеки из объектных модулей, собирая в них свои любимые и хорошо тестированные программные фрагменты для использования их при написании новых программ. По некоторым странным причинам во многих языках программирования высокого уровня написание программ, состоящих из разных частей, вызывает затруднения, требует использования необычных команд и непонятных действий. К счастью, в Turbo Assembler компоновка отдельных объектных модулей осуществляется очень просто.

В файле объектного кода содержатся машинные команды, полученные на основе трансляции вашего ассемблерного текста. Кроме этого, в объектном коде содержится текстовая информация, используемая другими модулями, плюс необязательная информация для Turbo Debugger. Нет необходимости разбираться во всем, что находится в объектном файле. Просто помните, что Turbo Assembler создает такой файл с расширением OBJ и никогда не генерирует окончательный исполняемый файл. Это делает исключительно Turbo Linker. Вообще, объектные файлы Turbo Assembler соответствуют стандарту OBJ, и вы можете компоновать их, используя другие компоновщики (например, компоновщик, поставляемый с некоторыми ранними версиями DOS) с объектными файлами, полученными в языках других производителей (например, в Microsoft C). Естественно, вы можете компоновать объектные файлы Turbo Assembler с объектными файлами из других Turbo-языков. Для этих целей всегда используйте Turbo Linker.

Замечание

Внимательно читаите в Руководстве пользователя о совместимости между Turbo Linker и другими компоновщиками. Формат объектных файлов постоянно модифицируется, и все, о чем здесь говорится, через шесть месяцев может не соответствовать действительности.

Опции командной строки

В процессе ассемблирования или компоновки вы можете выбирать различные характеристики выполнения процесса, которые задаются опциями в командной строке Turbo Assembler и Turbo Linker. Для вывода списка опций командной строки Turbo Assembler наберите `tasm` и нажмите <Enter>. Для вывода списка опций командной строки Turbo Linker наберите `tlink` и нажмите <Enter>.

Опции описываются одной или несколькими буквами, за некоторыми из которых может следовать дополнительная информация. Для задания опции наберите дефис и соответствующую букву между командой `tasm` или `tlink` и именем программы, которую вы ассемблируете либо компонуете. Например, для ассемблирования программы 2.1 и получения файла с *листингом* (файл, в котором содержится описание процесса ассемблирования данной программы) используйте команду

```
tasm -l ff
```

Опции разрешается набирать как прописными, так и строчными буквами. Кроме того, вместо дефиса, по желанию, можно использовать косую черту. Опция -l информирует Turbo Assembler о необходимости, помимо ассемб-

лирования программы, создать файл с листингом FF.LST на диске. Исполните эту команду и затем рассмотрите файл FF.LST, используя ваш текстовый редактор. Вы найдете в нем полный листинг программы, при этом каждая строка начинается с номера, затем следуют байты объектного кода и, наконец, собственно текст программы. При желании распечатайте этот файл в качестве примера.

Замечание

Не создавайте файл листинга при каждом ассемблировании программы — это может сильно замедлить весь процесс. Большинство программистов создают и распечатывают файл листинга только после завершения разработки программы, либо когда возникают проблемы, для разрешения которых требуется изучение объектного кода, создаваемого ассемблером.

При ассемблировании программы вы можете использовать в одной командной строке несколько опций, разделяя их пробелами, что не является обязательным. Ниже приводится несколько примеров:

```
tasm /h
tasm -l-c ff
tasm /l /c ff
tasm -zi ff
tasm -l -iC \INCLUDES ff
```

Проверьте эти команды на своем компьютере. Вместо ассемблирования первая команда просто выводит список опций командной строки. Чтобы распечатать его, используйте команду tasm /h > prn. Вторая команда создает файл листинга с перекрестными номерами строк (#10, #25 и т.п.) в конце. Третья команда делает то же самое и просто показывает, что при определении опций можно использовать косую черту вместо короткой черты. Четвертая команда добавляет в FF.OBJ информацию для Turbo Debugger. Последняя команда создает файл листинга и определяет путь включаемых (дополнительно используемых) файлов. (Включаемые файлы —это отдельные текстовые файлы, которые по вашему желанию Turbo Assembler вставляет в программу. Программа 2.1 не использует включаемые файлы, следовательно, эта команда не приведет к каким-либо практическим результатам.)

Turbo Linker также имеет различные опции командной строки, определяемые аналогичным образом, за исключением того, что некоторые ранние версии TLINK требуют использования опций с косой чертой (/m), а не дефиса (-m). Некоторые версии компоновщиков допускают использование как косой черты, так и дефиса, но при наборе нескольких опций с косой чертой в одной строке они должны разделяться пробелами. Ниже приводится несколько примеров применения опций в Turbo Linker (я проверял их в Turbo Linker 6.00; если вы используете другую версию, проверьте эти команды для определения тех опций, которые работают на вашей версии):

```
tlink -v ff
tlink /v ff
tlink -m -l ff
tlink /m/l ff
tlink -x ff
tlink /x ff
```

Опции -v или /v в первых двух строках подготавливают FF.EXE для использования его с Turbo Debugger. Следующие строки определяют две опции, которые приводят к созданию расширенного файла отображений (FF.MAP) и добавляют в этот файл вспомогательную информацию о номерах строк (/l). После проверки этих команд изучите файл FF.MAP, используя свой текстовый редактор. Опции /x и -x запрещают Turbo Linker создавать файл отображений, что приводит к небольшой экономии дискового пространства и времени компоновки. Применяйте эти команды, если вам не нужен файл отображений, который показывает организацию памяти программы и обычно используется отладчиками либо как часть программной документации.

Обработка ошибок

Человеку свойственно ошибаться, и программисты не являются исключением. Не имеет значения, насколько вы внимательны и прилежны, все мы допускаем много ошибок в своей повседневной работе. Но вы не сможете обмануть Turbo Assembler. По крайней мере, вы можете задать ассемблеру неправильную команду. Если вы это сделаете, намеренно или нет, то получите сообщение об ошибке или предупреждение, либо и то, и другое. Между ошибками и предупреждениями существует важное отличие:

- Ошибки являются критичными для программы. Даже если будет создан объектный код, он не будет компоноваться и исполняться, т.е. TLINK не скомпонует его в исполняемый файл.
- Предупреждения не являются критичными. Результирующий объектный код, вероятно, будет компоноваться, но исполняться он может некорректно.

Сделайте намеренно несколько ошибок, чтобы потом уметь разбираться в своих собственных ошибках. Если вы ассемблируете программу из командной строки DOS, загрузите программу в редактор, исправьте ошибки, вернитесь в DOS и снова повторите те же действия.

Если Turbo Assembler находит ошибку, он выводит сообщение об этом после соответствующего номера строки, взятого в скобки. Некоторые программисты сохраняют эти сообщения в файле на диске или выводят их на принтер для более подробного рассмотрения, используя следующие команды:

```
tasm ff>err txt      (ошибки сохраняются в файле err txt)
tasm ff>prn          (ошибки распечатываются на принтере)
```

Если не используется символ перенаправления вывода (>) и имя файла, то сообщение выводится на экран. Если ошибки помещаются на дисплее, вы можете распечатать копию экрана одновременным нажатием клавиш Shift и PrtScr. Для экспериментирования с ошибками скопируйте FF.ASM (см. программу 2.1) в новый файл FF2.ASM Удалите в строке 3 букву L в слове IDEAL. Для ассемблирования наберите tasm ff2 в командной строке DOS. Поскольку Turbo Assembler не понимает слова IDEA, в результате появится сообщение:

```
Assembling file   ff2 ASM
**Error** ff2 ASM(3) Illegal instruction
Error messages       1
Warning messages     None
Passes               1
Remaining memory     375k
```

Сообщение после "Assembling file..." подсказывает вам, в каком файле произошла ошибка, показывает номер строки в скобках и дает краткое описание ошибки. Для получения более подробной информации смотрите алфавитный список сообщений об ошибках в конце Справочного Руководства по Turbo Assembler. Исправьте IDEA на IDEAL и внесите другую ошибку, удалив двоеточие после метки Start в строке 15. Ассемблирование этого файла приведет к сообщению такого вида:

```
Assembling file    ff2 ASM
**Error**   ff2 ASM(15)  Illegal instruction
**Error**   ff2 ASM(31)  Undefined symbol   START
Error messages       2
Warning messages     None
Passes               1
Remaining memory     375k
```

Несмотря на то что вы сделали только одну ошибку, Turbo Assembler выдает сообщение о двух ошибках, одной — в строке 15 из-за пропуска двоеточия, и другой — в строке 31, где находится ссылка на метку Start. В результате первой ошибки метка Start остается нераспознанной, поэтому помните, что метки в сегменте данных должны заканчиваться двоеточием — обращение к ней также приводит к ошибке. Это пример *распространения ошибки*: одна ошибка может привести к появлению других. В длинных программах небольшие ошибки могут иногда распространяться по всему пространству. Если так случилось, особенно если ошибки вдруг стали возникать в местах программы, которые ранее ассемблировались нормально, то попробуйте исправить только пару первых ошибок и снова ассемблируйте программу. Во многих случаях остальные ошибки при этом просто исчезают.

Возвращаясь к нашему примеру с ошибкой, верните двоеточие в конец строки 15 и затем добавьте в строку 14 слова PROC DUMMY. Не задумывайтесь о смысле этих слов. Я просто хочу нечто показать вам. В результате ассемблирования выдается сообщение

```
Assembling file     ff2 ASM
*Warning* ff2 ASM(31) Open procedure  DUMMY
Error messages       None
Warning messages     1
Passes               1
Remaining memory     375k
```

Так же как и сообщения об ошибках, предупреждения указывают на неверно написанные строки. Заметим, что в нашем случае отмечается строка с номером 31, а не 14, как это можно было ожидать. Директива PROC определяет начало *подпрограммы*, т.е. группы команд, которые вашей программой трактуются как завершенная программная единица.

Каждой директиве PROC в Turbo Assembler должна соответствовать директива конца подпрограммы ENDP. Эта директива не была найдена во всем тексте программы, поэтому ассемблер предупреждает вас, что подпрограмма осталась незавершенной.

Поскольку это является предупреждением, а не ошибкой, вы можете скомпоновать и запустить результирующую программу на исполнение. В этом случае открытая несуществующая подпрограмма не принесет никакого вреда. Фактически, она не влияет на результирующий код. Однако это не всегда так, и игнорировать предупреждения

Turbo Assembler достаточно рискованно. Например, пропуск ENDP мог быть следствием удаления строк в вашем текстовом редакторе или вы случайно оставили подпрограмму незавершенной. Turbo Assembler весьма лоялен к таким неточностям, позволяя вам в ряде случаев совершать серьезные ошибки, но это и является платой за низкоуровневый доступ и повышенную скорость выполнения программы, предоставляемые исключительно языком ассемблера. Ассемблер только предупреждает вас о возможных неприятностях, и лишь глубокое знание своей программы позволит вам определить, является ли предупреждение существенным либо его можно безболезненно пропустить. Поэтому, получив предупреждение, постарайтесь на него отреагировать.

Введение в Turbo Debugger

В предыдущем разделе вы научились находить синтаксические ошибки, просматривая соответствующие сообщения Turbo Assembler. Но обнаружить логические ошибки не так просто. Turbo Assembler умеет ассемблировать синтаксически правильные программы, но не понимает, что, собственно, эта программа должна делать. Часто программы работают не так, как, по вашему мнению, должны были бы работать. В такой ситуации вам может помочь Turbo Debugger — программа, специально разработанная для упрощения поиска и исправления логических ошибок.

Подобно всем отладчикам, Turbo Debugger работает в режиме супервизора, беря на себя управление программой и позволяя вам пошагово исполнять код программы и проверять значения переменных в памяти. Вы можете потребовать, чтобы Turbo Debugger исполнял программу до некоторой точки или до появления определенной ситуации. Можно изменять значения в памяти, временно вставлять новые команды, а также менять значения регистров и флагов. Кроме того, вы можете использовать Turbo Debugger для программ в машинных кодах, что иногда бывает полезно, если количество команд не слишком велико.

Такая универсальная программа чрезвычайно необходима при программировании на языке ассемблера, когда из текста программы трудно уловить ее логику. Turbo Debugger также помогает находить ошибки в программах на C и Pascal, хотя здесь мы в основном будем заниматься отладкой ассемблерных текстов. Как уже упоминалось в главе 1, Turbo Debugger является замечательным учителем, давая вам возможность изучать свои программы и наблюдать за работой различных команд. Самый лучший способ изучить отдельные мнемокоды — написать небольшую тестовую программу, загрузить ее в Turbo Debugger и наблюдать за результатами их работы в пошаговом режиме. Если вы будете проделывать это каждый раз при возникновении вопросов о работе тех или иных команд, то удивитесь количеству информации, которое можно получить, просто наблюдая за работой команды.

Отладка на процессорах 80386 и выше

Если ваша система снабжена процессорами 80386, 80486 или Pentium, то вы можете пользоваться преимуществами специального режима в Turbo Debugger. Если в вашей системе есть процессор 8086, 8088 или 80286, эта возможность для вас недоступна. Однако, несмотря на это, Turbo Debugger является мощной программой, позволяющей с помощью своих команд отлаживать программы на любом PC. Если ваша система работает на 80386 или старшем CPU, вставьте в файл CONFIG.SYS, расположенный в корневом каталоге, следующую команду, определяющую правильный путь к файлу драйвера TDH386.SYS:

```
DEVISE=\TDEBUG\TDH386 SYS
```

Это позволяет Turbo Debugger использовать специальные отладочные регистры, доступные только в процессоре 80386. С помощью этих регистров Turbo Debugger может останавливать программу при изменении любого байта в указанной области памяти или даже если эти байты просто просматриваются отлаживаемой программой. Кроме того, вы можете запустить программу в виртуальной памяти, моделируя исполнение программы как автономного (самостоятельного) приложения DOS. Без процессора 80386 ваша программа обязательно использует общую с отладчиком память. В результате некоторые ошибки, в особенности те, которые зависят от положения программы в памяти, могут не проявляться под управлением отладчика, а затем снова возникать при обычном исполнении программы. Эту хитрую ситуацию бывает весьма сложно зафиксировать.

С инсталированным драйвером можно использовать версию Turbo Debugger для виртуальной памяти TD386.EXE вместо стандартной версии TD.EXE. (Вы можете, однако, использовать и стандартную версию.) Всякий раз, когда в книге вам предлагается набирать TD, набирайте вместо этого TD386.

Замечание

Для Turbo Assembler 4.0 драйвер TDH386.SYS и отладчик TD386.EXE не используются. Команды из этой главы применяются только к более ранним версиям Turbo Assembler и Turbo Debugger.

Turbo Debugger в качестве учителя

Чтобы показать, как использовать Turbo Debugger при изучении языка ассемблера, исследуем программу 2.1 под управлением отладчика. Сначала скопируйте FF.ASM в LF.ASM и загрузите эту копию в ваш текстовый редактор. Затем измените три строки следующим образом:

```
  1:   %TITLE   "Посылка на принтер команды прогонки строки"
 11:  ASCIIlf   EQU     10     ;ASCII-код, управляющий прогонкой строки
 23:     mov    dl, ASCIIlf    ;пересылка в dl кода прогонки строки
```

Эти модификации превращают программу прогонки страницы в программу прогонки строки, которая за один раз продвигает ваш принтер на одну строку. Это не столь полезная программа, чтобы сохранять ее в качестве утилиты, однако эти изменения будут экономить бумагу при ее многократном исполнении.

Сохранив LF.ASM, ассемблируйте и скомпонуйте программу с опциями, которые добавляют отладочную информацию в OBJ- и EXE-файлы. В результате Turbo Debugger получит информацию о программных символах, расположении переменных, организации сегментов и т.д. Для подготовки программы к отладке введите следующие команды:

```
tasm /zi lf
tlink /v lf
```

Без этих опций Turbo Debugger все равно загрузит вашу программу, однако будет показывать только дизассемблированные машинные коды. С опциями командной строки отладчик может показывать метки, структуры переменных, строки исходной программы и другую информацию. В дальнейшем, при рассмотрении примеров программ, используйте эти опции при ассемблировании и компоновке всякий раз, когда я рекомендую исследовать программу с помощью отладчика.

Замечание

Использование опций /zi и /v может значительно увеличивать размеры OBJ- и EXE-файлов на диске. По окончании отладки снова выполните ассемблирование и компоновку, но уже без этих опций, для придания дисковым файлам нормальных размеров.

После ассемблирования и компоновки с опциями /zi и /v убедитесь, что на диске находятся файлы LF.ASM и LF.EXE, затем загрузите программу в отладчик с помощью команды

```
td lf
```

Помните: если вы установили драйвер устройства TDH386.SYS и ваша система работает на процессоре 80386, тогда вы можете использовать версию Turbo Debugger для виртуальной памяти, вызвав его командой

```
td386 lf
```

Через некоторое время вы увидите экран Turbo Debugger с исходным текстом программы. (Если Turbo Debugger не найдет файл с расширением ASM, он не сможет показать окно с исходным текстом программы.) Просмотрите программу, используя клавиши управления курсором, передвигая его вверх и вниз по тексту. Кроме того, для передвижения внутри окна с исходным текстом можно использовать клавиши <PgUp>, <PgDn>, <Home>, <End>. Вам позволено только просматривать этот текст, для исправления обнаруженных ошибок необходимо выйти из Turbo Debugger и воспользоваться текстовым редактором.

Замечание

Для получения дополнительной справочной информации нажмите <F1> (клавишу справки) и прочитайте содержимое окна, которое появится на экране. При использовании Turbo Debugger вы можете в любой момент с помощью нажатия <F1> получить справку о текущем окне.

Для получения другого представления программы последовательно нажмите комбинацию клавиш <Alt+V+C>, выбирающую команду View⇨CPU⇨Window (просмотр окна центрального процессора). Нажмите <F5> для распахивания окна на весь экран. В CPU-окне содержится в сокращенном виде исходный текст вашей программы, действительные машинные коды, находящиеся в памяти, значения регистров и флагов, а также дамп байтов памяти. *Кроме чисто внешних деталей, между этим и предыдущим окном существует принципиальное различие.* В окне исходного кода, также называемом окном просмотра модуля, вы видите копию текста программы. В CPU-окне вы находитесь непосредственно в памяти, видите действительные значения расположенных там байтов. CPU-окно обеспечивает вам фантастическое путешествие в стиле новелл Айзека Азимова, уменьшая и отправляя вас в память компьютера. Естественно, оперируя байтами памяти, вы должны быть очень внимательны, чтобы не "убить пациента". Turbo Debugger помогает вам избежать подобных катастроф, однако неаккуратные действия все равно могут привести к неприятностям.

Для передвижения выделенного прямоугольного курсора по различным командам используйте клавиши <Up> и <Down>. Инструкции, принадлежащие вашим программам, отмечаются ромбами. Заметим, что, в отличие от окна исходного кода, здесь кроме отмеченных строк можно увидеть и другие области. Для передвижения курсора из од-

ной секции окна в другую нажимайте клавишу <Tab>. Переходя между секциями, можно изменять значения регистров и модифицировать байты памяти. (В данный момент не вносите никаких изменений.)

Нажимайте клавишу <Tab> до тех пор, пока прямоугольный курсор снова не проявится в большей секции. Для изменения вида этого окна нажмите <Alt+F10> или выберите команду Mixed (для этого нажмите клавишу <M> или переместите курсор на команду Mixed и нажмите <Enter>). Эту же команду можно выбрать, просто нажав <Ctrl+M>. Она имеет три установки: No, Yes и Both, которые позволяют изменять тип просмотра вашей программы:

- No показывает дизассемблированные машинные коды, аналогичные командам языка ассемблера. Это удобно для просмотра кода в случае, если у вас нет соответствующего ASM-файла. Многие программисты предпочитают использовать это представление, так как по сравнению с другими оно наиболее практично.
- Yes показывает исходный текст вашей программы вместе с дизассемблированным машинным кодом. Используется для отображения строк в языках программирования высокого уровня вместе с соответствующим им откомпилированным машинным кодом. Как правило, мы не пользуемся этой установкой при просмотре ассемблерных программ.
- Both устанавливается по умолчанию и, возможно, является наилучшим способом просмотра в CPU-окне, показывая в левой колонке байты машинного кода вместе с создающими их строками исходного текста. Пустые строки при этом не отображаются.

Turbo Debugger, кроме различных способов просмотра вашей программы и памяти, может по-разному исполнять ваш код. Рассмотрим это на примере. Включите принтер и выполните следующую последовательность действий для выполнения программы под управлением Turbo Debugger.

1. Для запуска программы нажмите <F9>. При этом лист бумаги в принтере должен протянуться на одну строку. Выполните эту команду и исследуйте состояние памяти, регистров и флагов по завершении программы.
2. После исполнения программы нажмите <Ctrl+F2> для возврата в начальное состояние. При этом программа снова загрузится с диска и Turbo Debugger восстановит свои исходные опции. (Если вы не сделаете этого, а снова нажмете <F9>, то увидите сообщение с вопросом о необходимости перезагрузки программы.)
3. Дважды нажмите <F6> для возврата в окно исходного кода.
4. Нажмите <Alt+V+R> для выбора команды просмотра регистров. При необходимости нажмите <Ctrl+F5> (перемещение текущего окна) и воспользуйтесь клавишами со стрелами для перемещения этого окна вправо либо выберите и перетащите его мышкой, открывая при этом команды программы. Нажмите <Esc> для фиксации нового положения окна. Окно регистров показывает значения регистров и флагов процессора вашего компьютера. Оно чрезвычайно полезно для исследования результатов выполнения различных машинных команд, большинство из которых изменяют значения одного или нескольких регистров.
5. Маленькая стрелка слева от первой команды программы mov ax, @data в окне исходного текста показывает, что она является следующей исполняемой командой. Для выполнения этой команды нажмите <F8>. Стрелка перейдет на следующую команду и изменится значение регистра ax в окне регистров. Эта команда "занесла" значение в регистр — назовем ее успешной. Передвижение по отдельным командам с помощью клавиши <F8> позволяет вам исполнять программу постепенно, выполняя за одно нажатие одну команду, и останавливаться для просмотра результатов работы каждой команды.
6. Снова нажмите <F8> для выполнения следующей команды — mov ds, ax. Обратите внимание на окно регистров — вы должны заметить, что значение в регистре ds стало таким же, как и значение в регистре ax. Команда mov переместила значение ax в ds. (Сейчас не нужно задумываться над тем, для чего это делается.)
7. Понажимайте <F6>, пока курсор снова не появится в окне исходного кода, которое при этом закроет окно регистров. (<F6> позволяет переключаться между всеми открытыми окнами, но вы можете также нажимать комбинацию <Alt+#>, где # — номер окна от 1 до 9.)
8. Передвиньте курсор вниз на строку mov dl, ASCIIlf — при этом три команды, расположенные выше, будут отмечены стрелками. Нажмите <F4> и программа выполнится от текущей команды до курсора. Используйте этот прием для выполнения небольших фрагментов программы, если нет необходимости останавливаться после каждой команды.
9. Несколько раз нажмите <F6>, пока снова не появится окно регистров. Затем дважды нажмите <F8>, выполнив две следующие команды. При этом наблюдайте за значением регистра ds — оно должно частично измениться.
10. Теперь стрелка указывает на команду Int 21h (строка 25 в программе 2.1). Эта команда вызывает функцию DOS, активизируя одну из многих возможностей операционной системы, в нашем случае, посылая символ на принтер. Для выполнения команды нажмите <F8>. Если принтер включен, лист бумаги в нем должен продвинуться на одну строку.
11. Нет необходимости исполнять эту программу дальше, так как оставшиеся команды просто возвращают управление DOS или, в нашем случае, — Turbo Debugger. Нажмите <Alt+X> для завершения работы с отладчиком.

Turbo Debugger имеет множество других возможностей, которые позволяют проверять, исполнять и модифицировать программы. Однако описанные выше шаги — это практически все, что вам нужно знать для выполнения большинства примеров ассемблерных программ этой книги и исследования результатов работы различных команд. В дальнейшем я расскажу, как использовать другие команды Turbo Debugger. Вы убедитесь, что Turbo Debugger, как ничто другое, может помочь вам при исследовании программ. Помогая вам изучать язык ассемблера, Turbo Debugger действительно является великим учителем.

Написание COM- и EXE-программ

Вы, вероятно, знаете, что в DOS существуют два типа исполняемых файлов, один из которых заканчивается расширением COM, другой — расширением EXE. С помощью ассемблера вы можете создавать программы обоих типов. Хотя большинство примеров программ в этой книге рассчитаны на EXE-файлы, иногда вы можете захотеть получить также COM-файл.

Замечание

Компания Microsoft объявила о своем намерении отказаться от использования формата COM-файла, хотя эти попытки пока безуспешны. Если вы будете писать программы в этом формате, то должны понимать, что в будущем вам, вероятно, придется многое переделывать, если Microsoft удастся стереть COM-файлы „с лица Земли".

Вместо того чтобы разрабатывать новую программу с нуля, полезно рассмотреть шаблоны, содержащие только самое необходимое для написания COM- и EXE-программ. На примере программы 2.2 показана оболочка для COM-программ. Пример программы 2.3 содержит соответствующую EXE-оболочку. Вы можете использовать эту EXE-оболочку для экономии времени при вводе примеров программ данной главы. Каждый шаблон содержит комментарии, показывающие, где необходимо расположить макроопределения, переменные и другие разделы, с некоторыми из которых вы еще не знакомы. Вы можете убрать эти комментарии, начиная на основе шаблона разрабатывать новую программу.

Программа 2.2. COMSHELL.ASM

```
%TITLE "Оболочка для COM-файлов -- by Tom Swan"

          IDEAL

          MODEL  tiny

;-----  Вставьте здесь директивы INCLUDE "filename

;-----  Вставьте здесь макроопределения EQU и =

          DATASEG

;-----    Если программа будет прервана по ошибке, запишите
;         соответствующий код ошибки в exCode и выполните команду
;         JMP Exit

exCode    DB   0

;-----  Здесь опишите другие переменные с помощью DB, DW и т.п.

          CODESEG

          ORG   100h        ; Стандартный адрес начала COM-программы

Start:

;-----  Здесь располагается программа, вызовы подпрограмм и т.п.

Exit:
          mov   ah, 04Ch    ; Функция DOS : выход из программы
          mov   al, [exCode]        ; Возврат значения кода выхода
          int   21h                 ; Вызов DOS. Останов программы

          END   Start               ; Конец программы/ точка входа
```

Программа 2.3. EXESHELL.ASM

```
1:  %TITLE "Оболочка для EXE-файлов -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL   small
6:          STACK   256
7:
8:  ;----- Вставьте здесь директивы INCLUDE "filename"
9:
10: ;----- Вставьте здесь макроопределения EQU и =
11:
12:         DATASEG
13:
14: ;-----  Если произойдет ошибка и программа вынуждена будет прерваться,
15: ;       запишите соответствующий код ошибки в exCode и выполните команду JMP Exit
16: ;       Чтобы это можно было сделать из подмодуля, опишите метку Exit
17: ;       с помощью директивы EXTRN
18:
19: exCode  DB   0
20:
21: ;-----  Здесь опишите другие переменные с помощью  DB, DW и т.п.
22:
23: ;-----  Здесь опишите все переменные типа EXTRN
24:
25:         CODESEG
26:
27: ;-----  Здесь определите все подпрограммы типа EXTRN
28:
29: Start:
30:         mov   ax, @data          ; Установка в DS адреса
31:         mov   ds, ax             ; сегмента данных
32:         mov   es, ax             ; Установка es=ds
33:
34: ;-----  Здесь располагается программа, вызовы подпрограмм и т.п.
35:
36: Exit:
37:         mov   ah, 04Ch           ; Функция DOS : выход из программы
38:         mov   al, [exCode]       ; Возврат значения кода выхода
39:         int   21h                ; Вызов DOS. Останов программы
40:
41:         END   Start              ; Конец программы/ точка входа
```

Написание COM-программ

Программа 2.2 представляет правильный формат файла для написания COM-программ в режиме Ideal. Строка 5 выбирает сверхмалую модель памяти (tiny), которая объединяет переменные программы, код и стек в одном 64 Кбайт сегменте памяти. Вследствие этого COM-программа всегда занимает в памяти 64 Кбайт, независимо от ее размера на диске. Этот факт является одной из причин предпочтительного использования EXE-программ. Хотя EXE-файл может занимать больше дискового пространства (так как в него включена дополнительная информация об организации программы), небольшие EXE-программы занимают при исполнении в памяти значительно меньше места, чем эквивалентные им COM-программы.

Строка 23 показывает другую характеристику COM-программы. Директива ORG указывает компилятору Turbo Assembler, что первая команда программы должна загружаться по адресу 100h (буква h используется для шестнадцатеричных чисел) относительно начала *кодового сегмента* программы. По такому же адресу загружаются программы в операционной системе CP/M, на которой во многом основана DOS и которая обычно используется на компьютерах, имеющих 64 Кбайт *общей* памяти. Под управлением DOS COM-программы работают в режиме псевдо-CP/M адресного пространства, несмотря на то, что большинство современных персональных компьютеров имеет в десять и более раз больший размер памяти. Таким образом, сегодня уже нет разумных причин для использования этого устаревшего формата файла.

В главе 4 вы познакомитесь с большинством 8086 мнемокодов, поэтому здесь я не буду объяснять назначение строк 30–32 в программе 2.2. В результате их выполнения управление передается DOS и программа завершается. Этими командами (или их эквивалентами) должны заканчиваться все COM-программы.

Ассемблирование COM-программ

Для ассемблирования COM-программ используются другие, отличные от описанных выше, команды. Вы должны указать Turbo Linker опцию /t, которая определяет сверхмалую модель памяти. Для практики ассемблируйте и скомпонуйте программу 2.2 с помощью следующих команд:

```
tasm comshell
tlink /t comshell
```

Это не конец. Это переход к чему-то другому...

Наконец пришло время для введения самого важного постулата: все программы на языке ассемблера должны передавать управление либо другим программам, либо DOS, используя для этих целей специально предназначенные команды. Это часто смущает программистов, работающих на других языках программирования вроде C, Pascal или Basic, в которых программы просто заканчиваются. В языке ассемблера программы никогда не заканчиваются в обычном смысле — они только приостанавливаются, передавая управление другим исполняющимся программам.

Вы поймете причину этого, если вспомните, что процессор компьютера работает без остановок. Как только вы включили компьютер и до тех пор, пока вы его не выключили, непрерывно выполняются те или иные вычисления. Даже если DOS просто ожидает вашей следующей команды, процессор не останавливается, выполняя биллионы циклов, постоянно обрабатывая инструкции, обеспечивающие режим ожидания компьютера. Ничего не делать — это для компьютера совсем непросто!

Поскольку процессор работает непрерывно, программа не может просто закончиться, она должна передать управление другой программе. Игнорирование этого действия почти всегда приводит к разрушительным результатам.

Если вы не передали управление другой программе, процессор продолжает обрабатывать содержимое памяти за физическим концом вашей программы. При этом в памяти может содержаться все, что угодно — фрагменты кода и данных других программ или просто случайные значения, появляющиеся там после включения компьютера. Обработка этой неизвестной информации обычно приводит к эффектному краху системы, появлению "мусора" на экране или, в наихудшем случае, к частичному разрушению данных на диске. Поэтому используйте шаблоны из программ 2.2 и 2.3, содержащие необходимые команды для передачи управления DOS, и вы никогда не пропустите этот важный шаг.

При завершении большинства программ исполняется команда DOS, перезагружающая программу COMMAND.COM, расположенную на вашей системной дискете либо в корневом каталоге жесткого диска, чаще всего C:\. Программа COMMAND.COM — это обычная программа специального назначения, позволяющая вам давать команды DOS. При запуске программы из DOS COMMAND.COM загружает код вашей программы в память и передает управление ее командам. При окончании программа должна возвращать управление COMMAND.COM для появления обычного приглашения DOS. Обязательно прочувствуйте этот процесс — это жизненно необходимо при написании программ на языке ассемблера.

Написание EXE-программ

Написание программ в EXE-формате требует несколько большей работы, чем при создании COM-программ, однако результат обычно оправдывает усилия. EXE-формат занимает в памяти только место, необходимое непосредственно для исполнения программы, оставляя больше свободного пространства для записи данных, создания больших массивов, а также для других EXE-программ в многозадачных операционных системах. (DOS не обеспечивает многозадачности, т.е. одновременного исполнения двух или более программ, хотя вы можете добавить эту возможность к DOS, запустив Microsoft Windows. Написание программ в EXE-формате позволяет им более эффективно использовать память.)

При использовании малой модели памяти переменные, стек и машинный код содержатся в различных сегментах памяти, размером до 128 Кбайт, поэтому написание EXE-программ требует дополнительных усилий. (Малая модель памяти объединяет сегмент стека и данных, другие модели допускают большие объемы кода и данных.) В программе 2.3 размер стека определяется директивой STACK (строка 6). Размер сегмента данных определяется суммарным размером переменных программы. Размер кодового сегмента зависит от количества команд в вашей программе.

Поскольку в EXE-программах переменные хранятся отдельно от кода программы, прежде всего необходима инициализация сегмента данных ds. Это выполняется строками 30–31 в программе 2.3 путем записи в регистр ax встроенного символа @data (строка 30), т.е. записи в регистр ax значения сегмента данных, и затем пересылки значения регистра ax в ds (строка 31). Это выполняется за два шага, так как в соответствии с синтаксисом ассемблера нельзя переслать значение вида @data непосредственно в сегментный регистр, запись значений в который допускается только через другие регистры общего назначения типа ax.

Как показывают строки 37–39, EXE-программы заканчиваются аналогично COM-программам. Опять-таки, пока не задумывайтесь над тем, что делают эти команды, просто помните, что их назначение состоит в передаче управления программе COMMAND.COM с использованием специальной функции DOS. Для ассемблирования и компоновки программы 2.3 воспользуйтесь следующими командами:

```
tasm exeshell
```

```
tlink exeshell
```

Распечатка листингов программ

Вы уже знаете, как вводить, ассемблировать и компоновать программы, и у вас может возникнуть желание распечатать листинги программ из этой главы. Поскольку строки в ассемблерных листингах могут превышать стандартную для большинства принтеров длину в 80 символов, сначала надо написать программу, задающую вашему принтеру сжатый режим печати, расширив этот предел до 132 символов или даже более для некоторых из них.

Ниже представлена простая программа 2.4 (PR132.ASM), написанная в EXE-формате и позволяющая на большинстве Epson-совместимых принтеров выводить строки длиной до 132 символов. Ассемблируйте и скомпонуйте эту программу с помощью следующих команд:

```
tasm pr132
tlink pr132
```

Программа 2 4 PR132.ASM

```
1     %TITLE  Выбор режима печати 132 символов в строке -- by Tom Swan
2:
3:            IDEAL
4:
5             MODEL  small
6.            STACK  256
7:
8             DATASEG
9
10      Вставьте здесь код  переводящий ваш принтер в режим печати 132 символа (или
11:   ; более) в одной строке, иногда называемый уплотненным режимом
12.   ; Значение, представленное ниже, должно работать на всех Epson-совместимых
13    , принтерах  Последнее число должно быть  0!
14
15    prCodes DB    27, 15, 0             ; Последовательность должна оканчиваться 0!
16
17            CODESEG
18
19    Start
20            mov   ax, @data             , Установка в DS
21            mov   ds, ax                ;  адреса сегмента данных
22:
23            cld                         , Очистка df - автоувеличение si
24            mov   si, offset prCodes    , si указывает на prCodes
25    Next
26:           lodsb                       ; Загрузка в al следующего кода
27:           or    al, al                ; al = 0?
28:           jz    Exit                  ; Если да, переход на exit
29:           mov   dl, al                ;  иначе, переслать al в dl
30:           mov   ah, 05h               , Функция DOS печати символа
31:           int   21h                   , Вызов DOS   Печать символа
32:           jmp   Next                  ; Обработать следующий код
33:   Exit
34.           mov   ax, 04C00h            , Функция DOS   Выход из программы
35:           int   21h                   , Вызов DOS. Останов программы
36:
37:           END   Start                 ; Конец программы / точка входа
```

После ассемблирования PR132.ASM проведите эксперимент. Включите ваш принтер и наберите DIR>PRN для распечатки в обычном режиме на принтере содержимого текущего каталога. Наберите PR132 и нажмите <Enter>. Затем снова наберите DIR>PRN, после чего каталог должен распечататься в сжатом виде. Если этого не происходит, тогда вам, вероятно, необходимо изменить строку 15. Найдите в руководстве по вашему принтеру правильные для нее значения. После директивы DB вы должны определить десятичные, шестнадцатеричные (начинающиеся с 0 и заканчивающиеся h) или символьные (один или несколько символов, взятых в одинарные или двойные кавычки) коды. В некоторых руководствах по принтерам шестнадцатеричные коды описываются с помощью знака доллара, например $1F. По правилам ассемблера этот код записывается в 01Fh. К примеру, если ваш принтер определяет последовательность Escape-C, $1F, вы можете использовать одну из следующих строк:

```
prCodes       DB       27  67, 31, 0              , десятичные
```

```
prCodes     DB      01Bh, 043h, 01Fh, 0     ; шестнадцатеричные
prCodes     DB      27, 'C', 01Fh, 0        ; десятичные, символьные,
                                            ; шестнадцатеричные
```

Последнее значение должно равняться 0, отмечая конец последовательности. Этот формат — список байтов, заканчивающийся 0, — типичная конструкция для программы на языке ассемблера, позволяющая создавать списки с произвольным числом элементов, отличных от 0.

Если вы раньше не программировали на ассемблере, команды в PR132.ASM могут остаться для вас непонятными. Это не столь важно. Цель этой главы состоит в том, чтобы только научить вас вводить, ассемблировать и компоновать программы, т.е. приобрести самые необходимые навыки на будущее. Однако, прочитав комментарии, вы по меньшей мере должны понимать основную идею этой программы. Она проста: поочередно брать байты из prCodes и посылать их значения на принтер до достижения 0, отмечающего конец списка. Затем вернуть управление DOS.

Распечатка листинга программы PR132

После ввода файла PR132.ASM, его ассемблирования, компоновки и проверки все готово для распечатки листинга. Включите ваш принтер и наберите **PR132** для задания сжатого режима печати. Затем снова ассемблируйте программу, но уже используя команду

```
tasm /l PR132
```

Для включения в конец листинга перекрестных символов используйте команду

```
tasm /l/c PR132
```

Каждая из этих команд приводит к созданию PR132.LST, называемого *файлом листинга*, готового к распечатке. Для вывода на печать файла листинга наберите команду

```
type pr132 lst>prn
```

Файл листинга содержит управляющие печатью символы, вследствие чего вы, возможно, не сможете распечатать его, используя текстовые редакторы, так как обычно эти программы обрабатывают страницы автоматически. Кроме того, вы можете направить листинг на спуллер печати (промежуточный буфер при печати файлов), что позволяет одновременно с печатью исполнять другие программы. Если вы не находитесь в сети, то наберите для печати файла листинга команду DOS

```
print pr132 lst
```

Если вы впервые используете команду печати, вас может удивить отсутствие имени выходного файла. Обычно при нажатии <Enter> по умолчанию выбирается файл с именем PRN. Вы можете вывести на печать несколько листингов, разделив их в командной строке пробелами, — это сохранит ваше время, если необходимо распечатать несколько листингов и одновременно продолжать редактировать и ассемблировать другие программы. Так же вы можете распечатать и несколько файлов, разделив их имена пробелами или использовать отдельные команды печати. Листинги ассемблера, как правило, намного объемнее листингов, получаемых в языках высокого уровня, поэтому программисты в этом случае обычно используют печать через спуллер.

Многие сохраняют файлы листингов для будущей работы с ними. Если же вы испытываете недостаток дискового пространства, то можете удалить файлы с расширением LST.

Замечание

Поскольку строка с директивой %TITLE не включается в файл листинга, нумерация строк текста программы в книге и листинге не совпадают. Третья строка в книге соответствует второй строке листинга и т.д. Для нахождения соответствия между нумерацией строк в книге и в распечатанном листинге вычитайте единицу из номера строки в книге. (Другими словами, если я говорю „смотрите строку 20", то находите в файле листинга строку 19.)

Резюме

Программу на языке ассемблера можно разделить на пять частей: заголовок, макроопределения, данные, тело и заключение. Тело состоит из четырех колонок: метки, мнемонические команды, операнды и комментарии. Метки ссылаются на переменные и команды, представленные мнемоническими кодами. Большинству ассемблерных команд требуются операнды в качестве данных для обработки. Необязательные комментарии позволяют вам запомнить назначение различных команд.

Ассемблерная программа создает объектный код, который должен быть скомпонован для получения исполняемого файла с расширением EXE или COM. Вы можете использовать специальные опции для выбора различных режимов работы Turbo Assembler и Turbo Linker. Turbo Assembler выдает на экран сообщения об ошибках и предупреждения в процессе ассемблирования.

Turbo Debugger может постепенно исполнять ассемблерную программу и позволяет вам заглянуть в память и увидеть реальные байты, из которых состоит ваша программа и данные. Вы можете использовать Turbo Debugger для поиска и устранения ошибок, а также в качестве персонального учителя при изучения языка ассемблера, который может исполнять тестовые программы и позволяет вам наблюдать за результатами выполнения отдельных машинных команд. COM-формат файла наследуется из операционной системы CP/M. Полезный в некоторых случаях, этот формат уже не рекомендуется использовать для PC-программ. Код, данные и стек в COM-программах занимают один сегмент памяти размером в 64 Кбайт. EXE-формат эффективнее, несмотря на то, что программы могут занимать несколько больше места на диске. В памяти EXE-программы занимают минимальное пространство. Написание EXE-программ требует дополнительных усилий, так как вы отвечаете за определение сегментов данных, кода и стека программы.

Ассемблерные программы не заканчиваются — они передают управление другой программе, чаще всего COMMAND.COM. Пропуск этого шага может привести к серьезным проблемам в результате выполнения случайных команд, находящихся в памяти после физического конца вашей программы.

Файл листинга документирует программу. Большинство программистов распечатывают листинги своих законченных программ для использования их в будущем. Вы можете применить спуллер печати DOS для распечатки длинных листингов, продолжая одновременно работать.

Упражнения

2.1. Рассмотрев текст программы 2.3, укажите номера строк, соответствующие заголовку, макроопределениям, данным, телу и заключению.

2.2. Назовите имена переменных в программе 2.4.

2.3. Сколько комментариев содержится в тексте программы 2.1?

2.4. Какой символ предшествует буквам опций для Turbo Assembler и Turbo Linker?

2.5. Предположим, что в файле BUGABOO.ASM имеется некоторый текст программы. Какие требуются шаги по ассемблированию и компоновке, чтобы создать необходимые файлы для отладки BUGABOO с помощью Turbo Debugger?

2.6. Какую программу, Turbo Assembler или Turbo Linker, вы используете для создания объектного кода? Какую программу необходимо использовать для создания исполняемого кода? Для чего создается объектный код?

2.7. В чем состоит различие между ошибками и предупреждениями? Что вы будете делать при получении ошибки или предупреждения?

2.8. В чем состоит отличие между COM- и EXE-файлами?

2.9. Предположим, что у вас есть программа LISTME.ASM. Какие шаги требуются для ассемблирования и распечатки файла листинга этой программы?

2.10. Как правильно заканчивать программу на языке ассемблера?

2.11. Какие действия выполняет директива DB? Какие типы данных вы можете создать с помощью директивы DB?

Задания

2.1. Выведите на печать список опций Turbo Assembler.

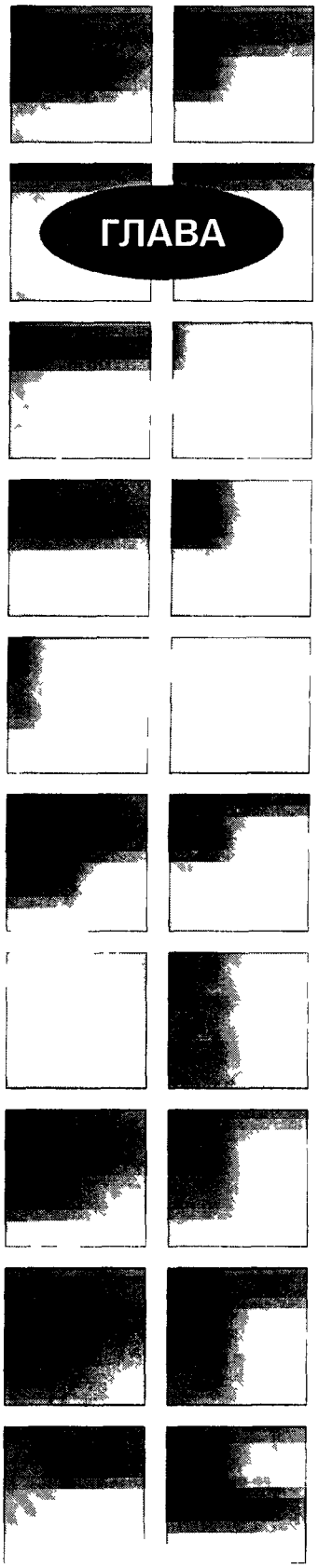
2.2. Скопируйте программу 2.4, переименовав ее в PR80.ASM. Измените эту программу для переключения вашего принтера в 80-колонный режим печати.

2.3. Создайте и распечатайте файлы листингов для программ 2.1–2.4.

2.4. Создайте на дискете или жестком диске каталог для сохранения ваших ассемблерных программ. Создайте в этом каталоге отдельные подкаталоги для каждой программы, назвав их так же, как и программы. Например, для сохранения программы 2.1 вы должны создать подкаталог FF и скопировать в него файлы: FF.ASM, FF.OBJ, FF.EXE, FF.MAP и, при желании, FF.LST.

2.5. Запустите программу 2.4 под управлением Turbo Debugger. Нажимайте <F8> для пошагового выполнения программы. Внимательно наблюдайте за действиями команд в строках от 26 до 32, в результате выполнения которых программа считывает список кодов принтера, останавливаясь при достижении 0, отмечающего конец списка. Воспользуйтесь окном регистров и наблюдайте за регистром ax, особенно при выполнении команды в строке 26. Что вы думаете о происходящем?

2.6. Переделайте программу 2.1 для получения файла в COM-формате.



ГЛАВА 3

Двоичные биты

- Все о памяти
- Сколько битов содержится в байте?
- Двоичная арифметика и логика
- Счет в двоичной системе
- Двоичные и шестнадцатеричные числа
- Запись значения в дополнительном коде
- Логические операторы
- Сдвиги и циклические сдвиги
- Резюме
- Упражнения
- Задания

Все о памяти

"Горючим" для программы на языке ассемблера являются биты и байты. Чем больше вы знаете о битах и байтах, о логических и арифметических операциях, выполняемых над двоичными величинами, тем больше энергии вы сможете извлечь из этого мощного источника всех цифровых вычислений — простых двоичных чисел, или *битов*, 0 и 1.

Конечно, на физическом уровне в памяти процессора не существует никаких двоичных чисел, а только электрические элементы, которые могут находиться либо в возбужденном, либо в невозбужденном состоянии. Однако при программировании удобно не учитывать этот факт, а представлять себе, что в схемах компьютера действительно фигурируют нули и единицы. С помощью наборов двоичных чисел можно представить значения ASCII-символов, управляющие коды принтера, финансовые балансы, даты, время и т.д. Двоичные величины могут использоваться для чтения и записи значений в порты ввода-вывода, они способны обеспечивать переключения между различными схемами, управляющими присоединенным к компьютеру оборудованием. Запись в них битов аналогична включению или выключению света. В ассемблере простая запись определенных значений в определенные места может приводить к включению моторов, выводу символов, пересылке данных на удаленные системы, генерации звукового сигнала.

Учитывая, что двоичные значения играют такую важную роль, особенно при программировании на языке ассемблера, нужно хорошо разбираться в двоичной арифметике и логике. Это не означает, что вы должны вручную складывать колонки шестнадцатеричных чисел, для этого можно использовать калькулятор.

(Более того, именно так поступают многие профессиональные программисты.) Однако знание основ двоичной математики будет помогать вам в написании хороших программ на языке ассемблера. Конечно, пользуйтесь калькулятором, но не забывайте изучать основные принципы. Каждая минута, проведенная за изучением этого предмета, избавит вас от часов замешательства в будущем.

Замечание

Поскольку для программирования на языке ассемблера необходимо хорошее понимание двоичных арифметических и логических операций, в этой главе с самого начала рассматриваются основы двоичной математики. Если вы разбираетесь в двоичной системе, можно пропустить этот материал и обратиться к более серьезным вопросам в конце этой главы.

Сколько битов содержится в байте?

Начнем с краткого обзора. Байт состоит из восьми битов, слово — из двух байтов, двойное слово — из четырех байтов, учетверенное слово — из восьми байтов. Биты нумеруются справа налево — нулевой бит всегда находится справа и называется *младшим значащим разрядом* (LSD). Самый левый бит называется *старшим значащим разрядом* (MSD). Типичное битовое представление байта и слова показано на рис. 3.1.

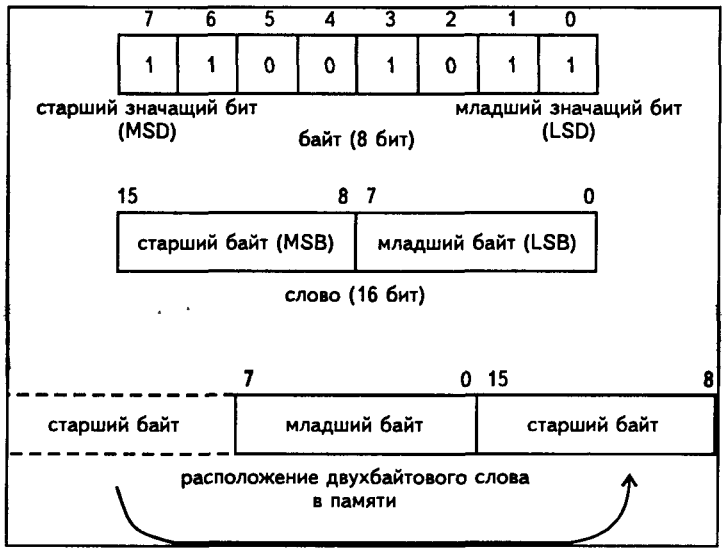


Рис. 3.1. Диаграммы обычного байта и слова

В памяти байты хранятся последовательно, один за другим. Каждый байт имеет соответствующий адрес — уникальное число, которое отличает его местоположение от всех других байтов. Для считывания или изменения значений байтов в памяти в программе на языке ассемблера определяется начальный адрес этих значений, причем не обязательно в виде именованных меток temperature или numCumquats. Возможность использования читаемых меток вместо действительных адресных значений вида 0F00:0014 является одним из основных достоинств языка ассемблера.

При использовании процессора 8086 значения слов записываются в переставленном порядке: *старший значащий байт* (MSB) размещается по более высоким адресам, чем *младший значащий байт* (LSB). В листинге ассемблера значения слов выводятся в порядке, обратном тому, в котором байты действительно хранятся в памяти (см. рис. 3.1.)

Этот переставленный порядок упрощает выполнение арифметических операций с многобайтовыми значениями, так как суммируемые первыми младшие значащие байты хранятся в более младших адресах. Однако этот порядок может вызывать затруднения при нахождении соответствия между напечатанным и хранящимся в памяти значением. Например, для нахождения в памяти слова 0201 в шестнадцатеричном представлении необходимо искать два последовательных байта — 01 и 02, а не 02 и 01.

Двоичная арифметика и логика

Поскольку для представления больших величин используется большое количество битов, вычисление непосредственно в двоичном виде сложных выражений становится весьма затруднительным. К счастью, вам нет необходимости становиться большим специалистом по двоичной арифметике. Некоторые книги требуют от вас изучения сложения, вычитания, умножения и деления двоичных чисел — операций, которые на практике программисты выполняют преимущественно на компьютере. Я сниму перед вами шляпу, если вам эти операции покажутся просты-

ми. В большинстве случаев опытному программисту на языке ассемблера необходимо уметь выполнять только четыре основные операции:

- Самостоятельно считать в двоичном виде от 0 до 16.
- Преобразовывать одно в другое двоичные, шестнадцатеричные и десятичные значения.
- Понимать логические операции AND, OR и XOR.
- Отличать двоичное представление значений со знаком (положительных и отрицательных) от значений без знака (только положительных).

Счет в двоичной системе

В табл. 3.1 представлены двоичные, шестнадцатеричные и десятичные значения от 0 до 16. Постарайтесь запомнить эту таблицу и отметьте страницу. Эти значения часто будут необходимы вам в дальнейшем.

Таблица 3.1 **Представление чисел 0–16 в двоичном, шестнадцатеричном и десятичном виде**

Двоичное	Шестнадцатеричное	Десятичное	Двоичное	Шестнадцатеричное	Десятичное
0000	00	0	1001	09	9
0001	01	1	1010	0A	10
0010	02	2	1011	0B	11
0011	03	3	1100	0C	12
0100	04	4	1101	0D	13
0101	05	5	1110	0E	14
0110	06	6	1111	0F	15
0111	07	7	1 0000	10	16
1000	08	8			

Вы легко научитесь считать и складывать в двоичном виде, если будете помнить одно простое правило сложения двух чисел в любой системе счисления: если значение, получаемое в разряде, превосходит основание системы счисления, то выполняется перенос 1 в левый разряд. Вы знаете, как это делается в десятичной системе. Поскольку в двоичной системе используются только два символа (значения по *основанию два*), перенос влево из одного разряда в другой в ней выполняется чаще, чем в десятичной, в которой имеется десять символов, и большие величины представляются меньшим количеством цифр. Сложение 1+1 не требует переноса в десятичной системе:

```
  1
 +1
------
  2
```

В десятичной системе результат представляется одним символом (2). В двоичной — 0 и 1 — единственные допустимые цифры; поэтому для представления этой суммы необходима дополнительная цифра. Таким образом, сложение 1+1 в двоичной системе вызывает перенос единицы в левый разряд:

```
  1
 +1
------
 10
```

Результат сложения *не равен* десяти. Результатом является *двойка*, представленная значением 10 по основанию два. Как вы знаете, десятичное сложение 1 и 9 (максимальной одиночной цифры с основанием десять) дает 0 в текущем разряде с переносом 1 в следующий левый разряд. Аналогично двоичное сложение 1 с 1 (максимальным одиночным числом с основанием два) также дает 0 в этом разряде с переносом 1 в левый разряд. Двоичное сложение ничем не отличается от десятичного, просто вам чаще приходится выполнять перенос 1 в левый разряд. Помня это правило, вы можете сложить любые два двоичные значения. Постарайтесь выполнить более сложное сложение, записывая переносы:

```
    11 111      (переносы)
  0110 1010     (первое значение)
 +0010 1110     (второе значение)
-------------------------------------------------
  1001 1000     (сумма)
```

Замечание

Во избежание недоразумений не говорите „сто" на двоичное значение 100 или „десять" на значение 10. Произносите в этом случае каждую цифру — „один-нуль-нуль" и „один-нуль".

Степени числа 2

В большинстве числовых систем (по крайней мере, существующих) позиция числа определяет величину, равную произведению этого числа на значимость разряда или порядок. Например, в десятичной системе 3 в числе 300 указывает на количество сотен — степень третьего слева разряда. Самый правый разряд представляет 10 в нулевой степени и записывается как 10^0 и т.д. Для нахождения степени каждого разряда запишите номер позиции разряда (начиная с 0) в качестве показателя степени над основанием системы счисления. Затем для получения значимости разряда возведите в эту степень основание системы. Например, значение 10^3 равняется (10×10×10), или 1000.

Замечание

Традиционно считается, что любое основание, возведенное в нулевую степень, дает 1. На практике это означает, что значение цифры в самом правом разряде равняется значению этой цифры, умноженному на 1.

Двоичные величины, так же как и десятичные, зависят от позиции разряда. Поскольку они выражаются по основанию 2, двоичные разряды представляют степени 2. В двоичном исчислении 1 в числе 100 символизирует значение степени третьего разряда, или 2^2, что равно в десятичном представлении 4 (2×2); следовательно, 100 в двоичной системе эквивалентно 4 в десятичной, 1000 в двоичной равняется 2^3 или 8 (2×2×2) и т.д.

Конечные величины

Компьютерные программы обычно представляют числа с использованием определенного количества битов, одним или более байтами. Благодаря этому запись в память чисел осуществляется в виде побайтовых фрагментов. В то же время ограниченному количеству битов устанавливается предельное число значений, которое может быть выражено с их помощью. Например, одиночный байт из 8 бит может определять значения от 0 до 255. 16-битовое слово может выразить значения от 0 до 65535 и т.д. Для выражения больших значений требуется большее количество битов. Для вычисления максимального значения, которое может быть представлено с помощью фиксированного количества битов n, используется формула 2^n–1. Например, если *n* равняется 8, тогда максимальное значение, которое вы можете выразить с помощью 8 бит, составляет (2×2×2×2×2×2×2×2)–1, или 255. Учитывая 0, получим 256 значений из интервала от 0 до 255; следовательно, формула для вычисления *количества* значений, выраженных с помощью n бит, примет вид 2^n.

Игра с килобайтами

Для представления 1000-байтовых, или *килобайтовых*, величин памяти пользуются удобной сокращенной записью, например 64 Кбайт, 128 Кбайт, 640 Кбайт. Эти степени числа 2, представляемые в двоичной системе в виде 1 с несколькими последующими нулями, принято использовать для точного измерения размера оперативной памяти, несмотря на тот факт, что в действительности 64 Кбайт составляют не 64000, а 65536 байт — полное число значений, которое может быть выражено 16 битами (2^{16}).

Кстати, адресное пространство процессора 8086 составляет 2^{20}, или 1048576 байт — так называемый *мегабайт*. Как вы убедитесь в следующих главах, 8086 использует некоторые хитрости для преобразования двух 16-битовых адресов в один 20-битовый физический адрес, определяющий действительное местоположение байтов в памяти. Процессор 80486 может адресовать до 2^{32} байт. Это составляет четыре *гигабайта* памяти, или точно 4294967296 байт. (Я не знаю, почему биллион байт называют гигабайтом. Может быть, более подходящим названием был бы биллибайт.)

При работе с двоичными адресами памяти представляйте их себе как степени 2. Измерение памяти в килобайтах является быстрым и простым, но в то же время слишком неопределенным для точного мира программирования на языке ассемблера.

Двоичные и шестнадцатеричные числа

Шестнадцатеричные значения представляются по основанию 16, другими словами, с помощью 16 символов — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Шестнадцатеричные числа строятся из десяти десятичных чисел от 0 до 9 и шести букв от A до F.

Замечание

В некоторых, более ранних компьютерных текстах использовался другой набор из шести букв. В одних предлагались U, V, W, X, Y, Z, в других — строчные t, e, d, h, f, l. Верите или нет, но нужно было помнить сокращения t для десяти, e — для одиннадцати, d — для двенадцати, h — для тринадцати, f — для четырнадцати и l — для пятнадцати! К счастью, это не стало общеупотребительным компьютерным стандартом.

Считать в шестнадцатеричной системе достаточно просто (см. табл. 3.1), если вы будете помнить, что 1+F равняется шестнадцатеричной 10 (16 в десятичной системе). Помните, что сумма 1 и последнего символа в любой позиционной числовой системе равняется символу 10, выраженному в этой системе.

Поскольку шестнадцатеричная система использует 16 символов и 16 является степенью 2 (2^4), двоичные значения с помощью подстановки легко преобразуются в шестнадцатеричные значения и обратно. Очевидно, что легче записывать и запоминать шестнадцатеричные значения вида B800, чем их двоичные эквиваленты — 1011 1000 0000 0000. Ниже приводится другой пример:

```
0100 1111 0101 1100
   4    F    5    C  =  4F5C
```

Двоичное значение разбивается справа налево по 4 символа, и каждая такая группа преобразуется путем подстановки из табл. 3.1 в шестнадцатеричное значение. Для преобразования из шестнадцатеричного представления в двоичное воспользуйтесь обратной подстановкой, заменяя шестнадцатеричные цифры их 4-битовыми двоичными эквивалентами.

Преобразования между шестнадцатеричными и десятичными значениями

Преобразования между шестнадцатеричными и десятичными числами требуют больших усилий, чем преобразования между шестнадцатеричными и двоичными числами. Самый простой способ — применять для этого калькулятор, выполняющий такие преобразования. Используйте так же калькулятор, встроенный в некоторые программные продукты, такие как Borland SideKick или Microsoft Windows. При этом вы можете вызвать калькулятор во время набора программы, выполнить вычисления и снова вернуться к работе.

Однако в тех случаях, когда вы не сможете воспользоваться калькулятором, вам необходимо знать, как вручную преобразовывать шестнадцатеричные и десятичные значения. Это не так сложно сделать, как вы, возможно, думаете. Так же как двоичные и десятичные, шестнадцатеричные числа являются позиционными, представляя справа налево возрастающие степени числа 16. Это позволяет быстро выполнять преобразования любого 16-битового значения из шестнадцатеричного в десятичное представление при запоминании только следующих четырех значений:

$16^0 = 1$
$16^1 = 16$
$16^2 = 256$
$16^3 = 4096$

Показатели степени, соответствующие позициям разряда в шестнадцатеричном значении, нумеруются справа (0) налево (3). Для преобразования шестнадцатеричной величины в десятичную умножьте значение каждой шестнадцатеричной цифры на степень ее разряда. Сложите полученные значения, и все готово. Например:

```
8B92=(8*4096)+(11*256)+(9*16)+(2*1)=35730
```

Шестнадцатеричное значение 8B92 равняется 35730 в десятичной системе. Для преобразования перед перемножением шестнадцатеричных чисел A–F в десятичные используйте табл. 3.1.

Для преобразования из десятичного в шестнадцатеричное представление выполняется обратный процесс деления на степени числа 16. Хотя это несколько сложнее, вы также можете произвести вычисления вручную:

```
(35,730/4096)=8.72...   (8*4096)=32,768     (35,730-32,768)=2962
(2,962/256)=11.57...    (11*256)=2816       (2,962-2,816)=146
(146/16)=9.125          (9*16)=144          (146-144)=2
(2/1)=2                 (2*1)=2             (2-2)=0
                        :
                        :
                        8,11,9,2=8B92
```

Не волнуйтесь, это не так сложно, как может показаться на первый взгляд. Читайте каждую строку слева направо и увидите, что сначала десятичное значение с остатком делится на уменьшающиеся степени числа 16, затем целая часть полученного от деления числа умножается на ту же степень и результат вычитается из первоначального

значения. Следующая строка использует результат этого вычисления для следующего деления, повторяя процесс до достижения 0. Если результат первого деления больше или равен 16, то необходимо начинать с более высокой степени. Если результаты последующих делений больше или равны 16, значит, где-то допущена ошибка. Может показаться, что вычисление написанных выше выражений потребует от вас значительных усилий. Однако после некоторой практики и с простым десятичным калькулятором вы сможете выполнять эти преобразования за несколько секунд. Обратите внимание на то, как в процессе деления возникают шестнадцатеричные цифры — 8, 11(B), 9, 2, или 8B92 в шестнадцатеричном представлении.

Запись значения в дополнительном коде

Программист на языке ассемблера должен отличать представление чисел со знаком от значений без знака. Целые беззнаковые числа включают в себя 0 и все положительные значения. Целые числа со знаком состоят из беззнаковых целых и целых со значениями, меньшими 0. С помощью фиксированного количества битов можно выразить фиксированное число знаковых и беззнаковых значений. Например, для 4 бит наименьшее значение равно 0000, наибольшее беззнаковое значение — 1111. В десятичном представлении этому соответствует интервал от 0 до 15 — всего 16 значений, включая 0. Для 8 бит наибольшее беззнаковое значение — 11111111, или 255 в десятичном представлении, что дает 256 возможных значений для одного байта. В математике количество чисел может быть бесконечным, в то время как в программировании количество чисел имеет конечный предел. С помощью фиксированного числа битов вы можете выразить только определенное количество значений, поэтому представление отрицательных значений в двоичной системе требует некоторой хитрости. Знак величины может быть либо положительным (+), либо отрицательным (–), следовательно, для представления знака может использоваться один бит — 1 для отрицательного и 0 для положительного. Оставшиеся биты при этом будут представлять беззнаковое *абсолютное значение*. Такое представление отрицательных чисел в двоичной системе называется *значением в дополнительном коде*, или *двоичным дополнением*.

Замечание

Для простоты 0 рассматривается как положительное значение, хотя, строго говоря, он не является ни положительным, ни отрицательным.

Если в записи в дополнительном коде самый левый бит равен 1, тогда значение отрицательное, если же левый бит равен 0, значение либо положительное, либо нулевое. Для преобразования положительных величин в дополнительный код сначала необходимо выполнить отрицание каждого бита — изменить единицы на нули и нули на единицы, получив промежуточное значение, затем прибавить к этой величине 1. В результате образуется значение в дополнительном коде:

```
 0110 1010 (первоначальное значение)
 1001 0101 (1. отрицание каждого бита)
+        1 (2. прибавление 1)
 --------------
 1001 0110 (значение в дополнительном коде)
```

Эти шаги обратимы. Преобразования значения в дополнительном коде в абсолютное значение выполняются аналогичным образом. Например:

```
 1111 1110 (значение в дополнительном коде)
 0000 0001 (1. отрицание каждого бита)
+        1 (2. прибавление 1)
 --------------
 0000 0010 (абсолютное значение)
```

Другими словами, 1111 1110 — это –2 в десятичной системе, представленное двоичным числом со знаком — значением в дополнительном коде. Такое преобразование выполняется независимо от количества значащих битов — 4, 8, 16 или больше. Левый бит всегда указывает, является значение положительным (0) или отрицательным (1). В случае отрицательного числа для получения его абсолютного значения выполняются операции преобразования дополнительного кода.

Замечание

Другой способ получения значения в дополнительном коде состоит в вычитании двоичного значения из 0, хотя вручную проще выполнить отрицание всех битов и прибавить 1.

Для того чтобы понять назначение дополнительного кода, вспомните числовую прямую, которую вы изучали в школе (рис. 3.2). Справа от нуля расположены положительные значения, слева — отрицательные. Прямая уходит в оба направления дальше, чем это может представить себе человеческий мозг.

В памяти компьютера имеется только фиксированное количество позиций под цифры, поэтому можно представить себе подобную числовую прямую в виде круга (рис. 3.3). Двоичные значения (вне круга) расположены в порядке возрастания по часовой стрелке. Добавление единицы к максимальному значению (1111) приводит к 0. Внутри круга расположены эквивалентные десятичные значения со знаком; беззнаковые десятичные значения расположены над соответствующими двоичными значениями вне круга. На рисунке рассматриваются четырехзначные двоичные значения, хотя такое же построение можно провести для любого фиксированного количества значащих битов.

Из рис. 3.3 видно, что ровно половина значений со знаком являются отрицательными (от –1 до –8), другая половина — положительными (от 0 до 7). Значения без знака (от 0 до 15) представляются теми же двоичными значениями, что и величины со знаком. Это приводит к важному правилу: *отрицательные двоичные значения изображаются так исключительно по договоренности*. При фиксированном количестве битов всем беззнаковым значениям соответствуют значения со знаком, представленные одинаковыми последовательностями битов, такие как (9, –7), (13, –3) и (15, –1). Двоичные значения для отрицательных чисел при этом представляются в форме дополнительного кода.

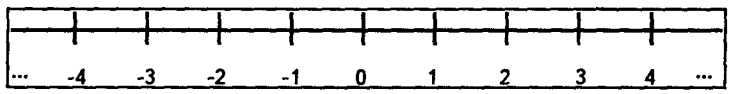


Рис. 3.2. Числовая прямая

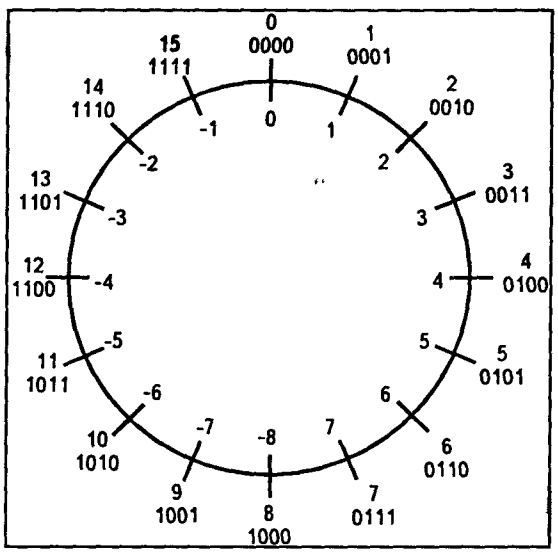


Рис. 3.3. При фиксированном количестве двоичных цифр удобно представлять себе хорошо знакомую числовую прямую в виде круга

Замечание

Может показаться, что здесь отрицательных чисел на одно больше, чем положительных. Но это не так, если учесть, что 0 считается положительным числом, и поэтому отрицательные и положительные числа представлены в равных количествах. Подсчитайте их на рис. 3.3.

Вычитание с помощью сложения

Запись значений в дополнительном коде играет в двоичной арифметике важную роль, так как позволяет на аппаратном уровне заменить вычитание сложением. Выполняя шаги преобразования дополнительного кода, легко найти абсолютное значение отрицательной двоичной величины, представленной в виде записи в дополнительном коде. Если вы поняли принцип круговой числовой прямой (см. рис. 3.3), то легко уловите и эту идею. Очевидно, что результатом десятичного сложения 1+9 является 10 — в двоичном виде 1010, что соответствует значению со зна-

ком –6 на круговой числовой прямой, причем такой же результат получается при вычитании 1–7. Следовательно, вместо вычитания 1–7 можно сложить 1+9 и затем рассматривать отрицательное значение на круговой числовой прямой в качестве результата, записанного в дополнительном коде.

К счастью, процессор 8086 имеет специальные команды для вычитания значений, поэтому при программировании на ассемблере вам не потребуется вместо вычитаний выполнять сложения с преобразованиями дополнительного кода. Но, несмотря на это, необходимо понимать механизм этих преобразований.

Правило следующее: для вычитания одного двоичного значения из другого преобразуйте вычитаемое в двоичный дополнительный код и сложите его с первым значением. Для демонстрации начнем с простого вычитания, результатом которого является положительное число:

```
 1001        1001         9
-0101       +1011        -5
-------    ---------    -----
 0100      1 0100         4
```

Слева 5 (0101) непосредственно вычитается из 9 (1001). В середине 5 в дополнительном коде (1011) складывается с 9. Правая колонка показывает вычитание в десятичном представлении. Два вычисления приводят к одному результату, но, если в центральной колонке не учитывать перенос, результат будет таким же. Теперь посмотрим, что получится при вычитании 5–9, в результате которого получается отрицательное число:

```
 0101          0101
-1001         +0111
--------     ---------
 ?100         0 1100
```

В левой колонке требуется занять из несуществующего разряда. В правой вычитание выполняется путем сложения значения 9, записанного в дополнительном коде, с 5 — получается 1100, значение, которое является отрицательным числом, так как левый бит в нем равен 1. Его абсолютное значение равно 0100 — абсолютному значению числа –4, которое и является результатом вычитания 5–9. Таким образом, преобразование дополнительного кода позволяет вычитать двоичные значения с помощью сложения.

Замечание

Процессор 8086 обеспечивает команды для отрицания каждого бита и создания дополнительного кода двоичных значении Команда not создает отрицание каждого бита Команда neg образует значение в дополнительном коде. Вы снова встретитесь с этими командами в главе 4.

Логические операторы

Три логические операции — AND (логическое И), OR (логическое ИЛИ) и XOR (исключающее ИЛИ) так же характерны для языка ассемблера, как сорная трава для садового участка. Однако в нашем саду они не будут сорняками. AND, OR и XOR дают вам возможность манипулировать отдельными битами в двоичных величинах. Вы можете устанавливать или сбрасывать единичные биты без влияния на другие, выделять из байта или слова один или несколько битов и выполнять другие операции.

Таблица 3 2. **Таблица истинности для AND, OR, XOR**

AND(&)	OR(|)	XOR(x)
a & b=c	a | b=c	a x b=c
0 & 0=0	0 | 0=0	0 x 0=0
0 & 1=0	0 | 1=1	0 x 1=1
1 & 0=0	1 | 0=1	1 x 0=1
1 & 1=1	1 | 1=1	1 x 1=0

Внимательно изучите табл. 3.2. Результат применения оператора AND к двум битам равен 1 только в том случае, если оба бита — *a* и *b* также равны 1.

Результат применения оператора OR к двум битам равен 1, если один бит — *a* или *b* равен 1. Результат применения оператора XOR к двум битам равен 1 только в том случае, если один из битов — *a* или *b* равен 1, а другой 0.

Обнуление битов с помощью оператора AND

AND часто применяется для маскирования (выделения) битов в байтах и словах. Обратившись к истинностной таблице оператора AND (табл. 3.2), можно увидеть, что *a* и *c* одновременно равны 1 только том случае, если равно 1 соответствующее значение в колонке *b*. Используя это наблюдение, вы можете создавать так называемые *фильтры* для извлечения битов из байтов. Ниже приводится типичный пример:

```
  0101 1101 (начальное значение)
& 0000 1111 (AND маска)
-----------------
  0000 1101 (результат)
```

Значение маски равно 0000 1111, или 0F в шестнадцатеричной системе. Пыполнение операции AND над двумя битами дает 1 только в том случае, если оба бита равны 1, поэтому лишь четыре цифры из младшего значащего разряда пройдут через маску без изменений. Четыре цифры старшего значащего разряда будут маскированы 0. Для понимания того, как работает маска, выполните с помощью истинностной таблицы соответствующие операции на каждом разряде этого примера.

Кроме того, AND-маски часто используются для проверки значений отдельных битов. Сначала создайте маску с 1 только в позиции проверяемого бита. Затем выполните операцию AND над маской и проверяемым значением. Предположим, что вы хотите проверить самый левый бит (например, для определения того, является ли данное значение отрицательным или положительным):

```
  0111 1010       1001 1111  (начальное значение)
& 1000 0000     & 1000 0000  (AND маски)
-----------     -----------------
  0000 0000       1000 0000  (результат)
```

Маска (80 в шестнадцатеричной системе) выделяет левую цифру из старшего значащего разряда. Если первоначально в этой позиции содержится 0, результат равен 0. Если первоначальное значение в этой позиции 1, тогда результат не равен 0. Получение ненулевого результата означает, что начальное значение является отрицательным (записанное в дополнительном коде). Как вы узнаете далее, 8086-й процессор обеспечивает другой способ проверки отрицательных значений. Но, даже несмотря на это, очень важно уметь маскировать отдельные биты с помощью описанного выше метода.

Установка битов с помощью оператора OR

В противоположность действию AND, логический оператор OR в основном используется для изменения значений отдельных битов, не влияя при этом на другие биты в байте. Как видно из табл. 3.2, единичный бит в колонке *b* всегда приводит к получению 1 в колонке *c*, в то время как 0 в колонке *b* обеспечивает равенство результата первоначальному значению бита из первой колонки. Заметим, что это действие противоположно операции AND, при выполнении которой единичный бит в маске оставляет значение бита неизменным. Как показывает следующий пример, этот факт позволяет использовать OR для установки значения любого бита в байте:

```
  0010 1011 (начальное значение)
| 1000 0000 (OR маска)
----------------
  1010 1011 (результат)
```

OR-маска (80 в шестнадцатеричной системе) изменяет начальное значение старшей значащей цифры с 0 на 1. (Если этот бит уже был равен 1, его значение не изменяется.) Выполните, используя таблицу истинности, операции OR на каждом разряде этого примера, и вы поймете, как все это работает.

AND и OR часто используются для изменения состояний переключателей устройств, которые представляются отдельными битами памяти, возможно, содержащихся в регистрах интерфейсных плат. (Регистр — небольшая по размеру память специального назначения, обычно расположенная внутри чипа. Процессор 8086, так же как и другие чипы на системной плате вашего компьютера, имеет большое количество таких регистров для содержания в них важных значений.)

Чтобы увидеть, как AND и OR могут использоваться для контроля за устройствами, представим, что ваш компьютер управляет освещением, и предположим, что значение 1 бита 3 в байте некоторого регистра соответствует включенному свету, значение 0 — выключенному. Биты 5, 6 и 7 описывают восемь степеней яркости света от 000 (тусклый) до 111 (яркий). Оставшиеся биты имеют какое-то другое назначение, и вы должны быть внимательными, чтобы случайно не изменить их. Представленные ниже операции приводят к включению света и увеличению яркости до 3, знаком вопроса обозначены запрещенные для изменения биты, *v* — яркость, *s* — состояние переключателя:

```
  7654 3210 (номера позиций битов)

    vvv? s??? (начальное состояние)
  & 0001 0111 (AND маска)
  ----------------
    000? 0??? (результат AND – свет выключен)
  | 0110 1000 (OR маска)
  ----------------
    011? 1??? (результат – свет включен на яркость 3)
```

Сначала маска AND сбрасывает начальные значения единичных битов в позициях 7, 6, 5 и 3 — тех битов, значения которых должны быть изменены. Единицы в маске AND обеспечивают сохранение первоначальных значений в запрещенных позициях — 4, 2, 1 и 0. Затем маска OR устанавливает биты 7, 6, 5 в 011 (3 в десятичной системе) и бит 3 в 1. Обратите внимание на то, как нули в маске OR обеспечивают сохранность необходимых значений (?). Теперь сравним верхнюю и нижнюю строки. Значение яркости *vvv* изменено на 011, переключателя *s* — на 1. Биты, управляющие другими устройствами, оставлены без изменений.

Замечание

При установке отдельных битов в байте вы почти всегда должны последовательно использовать AND и OR. Это одна из наиболее фундаментальных последовательностей в программах на языке ассемблера, ее нужно выучить наизусть.

Исключающее ИЛИ

Третий основной логический оператор, XOR, аналогичен оператору OR, но имеет одно важное отличие. Как можно увидеть из табл. 3.2, результат *c* равен 1 только тогда, когда только одно, а не оба, из начальных значений равно 1. Если оба бита имеют одинаковые начальные значения, результат операции XOR равен 0. Это свойство является удобным инструментом для переключения отдельных битов из состояния on в off и наоборот, причем в этом случае не нужно знать начальные значения этих битов. Так же как и в OR, 0 в маске XOR сохраняет начальное значение бита. Следующий пример поможет вам понять действие маски XOR:

```
  1010 0010 (начальное значение)
⊗ 1110 1011 (XOR маска)
----------------
  0100 1001 (результат)
```

Применение операции XOR к этим двум значениям, когда оба бита равны, дает в результате 0, когда значения битов различны — результат равен 1. Проверьте с помощью табл. 3.2 правильность каждого столбца в этом примере. Теперь посмотрите, что произойдет, если маска XOR будет состоять из одних единиц:

```
  1010 0101 (начальное значение)
⊗ 1111 1111 (XOR-маска)
----------------
  0101 1010 (результат)
```

Сравните верхнюю и нижнюю строки. Первоначальное значение каждого бита в результате меняется на противоположное. Все единицы преобразуются в нули, все нули — в единицы. (Прибавление к этому результату 1 образует значение в дополнительном коде от начального значения.) Еще удивительнее, что повтор той же операции восстанавливает начальное значение:

```
  0101 1010 (результат предыдущего примера)
⊗ 1111 1111 (та же XOR-маска)
----------------
  1010 0101 (начальное значение!)
```

Это становится понятным, если сообразить, что поскольку маска XOR изменяет на противоположный каждый бит начального значения, то повторное применение той же маски к результату восстанавливает первоначальные значения битов. Эта возможность манипулировать значениями с помощью XOR, а затем восстанавливать исходное значение повторным применением XOR часто используется при разработке графического программного обеспечения, позволяя объектам, представленным наборами битов, свободно накладываться одним на другие.

Это применяется также в коммуникациях и шифровании.

Вследствие того, что при выполнении операции XOR каждый единичный бит в маске изменяет на противоположный соответствующий бит в начальном значении, возникает интересный побочный эффект: применение оператора XOR к любым одинаковым значениям всегда дает в результате 0. Например:

```
  0111 1101 (начальное значение)
⊗ 0111 1101 (то же значение в качестве XOR-маски)
-----------------
  0000 0000 (результат)
```

Помните: если биты имеют равные значения, то применение к ним оператора XOR дает в результате 0. Очевидно, что применение оператора XOR к одинаковым величинам может иметь единственный эффект — в результате все нули. Кстати, как вы увидите, этот прием часто используется в программах на языке ассемблера для процессора 8086. Хотя и существуют другие способы обнуления байта, использование операции XOR на одинаковых значениях остается одним из наиболее быстрых способов.

Замечание

При вычитании величины из самой себя в результате также получается 0. Для интереса попробуйте сложить какое-то число со своим значением в дополнительном коде. Что вы получили в результате? Как видите, существует не единственный способ для обнуления байта.

Возвращаясь к примеру с управлением освещения компьютером, для включения и выключения света, вы можете выполнять следующую, не оказывающую воздействие на значения других битов, операцию XOR :

```
  vvv? s??? (начальное значение)
⊗ 0000 1000 (XOR-маска)
-----------------
  vvv? x??? (результат)
```

Единичный бит в маске изменяет соответствующее значение бита *s* в начальной величине на противоположное значение *x*, не влияя при этом на любые другие биты.

Важность этой операции заключается в том, что для выполнения переключения программе нет необходимости знать начальное значение *s*. Заранее известно только то, что в результате получится значение, противоположное начальному. Если свет был включен, то теперь он выключен; если был выключен, то теперь включен.

Сдвиги и циклические сдвиги

Другой важной операцией над двоичными значениями является сдвиг битов влево и вправо. Сдвиг влево обычно обнуляет самую младшую значащую цифру, выталкивая старшую значащую цифру за левую границу байта. Сдвиг вправо выполняется аналогично, но в этом случае обнуляется старшая значащая цифра, приводя к потере младшую значащую цифру. В ассемблере существуют некоторые разновидности этой операции: одни из них записывают теряющийся бит и помещают в позицию обнуляющегося бита значение из другого однобитового флага, другие через однобитовый флаг перемещают теряющуюся значащую цифру в другой конец, вызывая циклический сдвиг байта (ротацию).

Так как операция сдвига является чрезвычайно важной операцией в программировании на языке ассемблера, мы снова вернемся к ее рассмотрению при изучении команд сдвига процессора 8086. Однако сейчас вам необходимо понять: умножение — с помощью сдвига влево, деление — с помощью сдвига вправо. Чтобы разобраться, как с помощью сдвига можно выполнять операции умножения и деления, рассмотрим следующую сумму:

```
  0110 1011 (начальное значение)
+ 0110 1011 (сложение с тем же значением)
---------------
  1101 0110 (сдвиг значения влево)
```

Как показывают верхняя и нижняя строки, сложение значения с самим собой вызывает сдвиг всех битов на одну позицию влево. Другими словами, умножение на 2 эквивалентно сдвигу на один бит влево. Следующее умножение полученного результата на 2, т.е. начального значения на 4, или на 2^2, снова сдвигает биты на одну позицию влево. Это приводит к общему правилу: чтобы умножить любое значение на степень 2, необходимо выполнить сдвиг влево на количество битов, равное показателю степени. Чтобы найти величину $x * 2^4$, т.е. $x*16$, в значении x нужно выполнить сдвиг на четыре бита влево.

Очевидно, что если сдвиг влево равносилен умножению двоичного значения на последовательные степени числа 2, то сдвиг вправо соответствует делению этого значения на 2, 4, 8 и т.д. Для нахождения результата деления

1010 1111 (AF в шестнадцатеричной, или 175 в десятичной системе) на 4 необходимо только дважды сдвинуть биты вправо:

```
1010 1111 (начальное значение)
0101 0111 (деление на два)
0010 1011 (следующее деление на два)
```

В результате получится значение 00101011 (2B в шестнадцатеричной, или 43 в десятичной системе), что при отбрасывании остатка равно частному от деления 175 на 4. По аналогии с умножением деление на степень числа 2 соответствует сдвигу начального значения на количество битов, равное показателю степени.

Однако при использовании этого метода существуют определенные ограничения. Во-первых, вы можете умножать и делить на степени 2 только беззнаковые значения. Во-вторых, произведение должно по размеру соответствовать начальному значению. (Результат умножения 1111 1111 на 2, к примеру, *не равен* 1111 1110, так как для представления правильного значения произведения необходим еще один — девятый бит.) Кроме того, поскольку при сдвиге теряются правые биты, а левые заполняются нулями, деление осуществляется без учета остатка. Несмотря на эти ограничения, все равно имеет смысл выполнять умножение и деление с помощью сдвига, так как это одна из самых быстрых операций, выполняемых процессором. В следующих главах вы увидите соответствующие примеры программ.

Резюме

Биты и байты являются "горючим" для процессора. Байт состоит из восьми битов, слово — из двух байтов, двойное слово — из четырех байтов, учетверенное слово — из восьми байтов. Байты записываются в памяти последовательно, каждый из них имеет свой уникальный адрес. В словах байты хранятся в обратном порядке, старший значащий байт содержится в более высоком адресе памяти.

Счет от 0 до 16 в двоичной системе, преобразования между двоичным, шестнадцатеричным и десятичным представлениями, понимание логических операций AND, OR и XOR, представление значений с помощью преобразований дополнительного кода — это те основы двоичной математики, которые необходимы программисту на языке ассемблера.

Так же как и в других системах счисления, разряды в двоичном значении справа налево соответствуют возрастающим степеням основания системы.

Поскольку 16 является степенью числа 2, вместо двоичных значений удобно использовать их шестнадцатеричные эквиваленты. Преобразования между шестнадцатеричными и двоичными значениями не требуют больших усилий Сложнее преобразовывать одну в другую десятичные и шестнадцатеричные величины; в этом случае, вероятно, лучше воспользоваться калькулятором. Но при этом вы все равно должны уметь выполнять эти преобразования вручную, причем, если вы знаете метод, они не будут для вас очень сложными.

Отрицательные значения в двоичной системе представляются в виде записей в дополнительном коде. Старшая значащая цифра в отрицательном числе всегда равна 1. Нуль для простоты считается положительным числом. Запись в дополнительном коде позволяет процессору выполнять вычитание с помощью сложения, а также упрощает поиск абсолютного значения любого отрицательного числа, представленного в дополнительном коде.

Для работы с отдельными битами в двоичных величинах обычно используются три логические операции: AND, OR и XOR. AND-маски позволяют выделять из двоичных значений один и более битов. OR-маски могут устанавливать отдельные биты в 1. XOR-маски позволяют изменять значения битов с 1 на 0 и наоборот, независимо от их начальных значений. Для задания значения определенного бита без изменения значений других битов обычно используется одна из наиболее характерных для ассемблера цепочек — последовательность операций AND и OR.

Сдвиг битов влево соответствует умножению беззнакового двоичного значения на последовательные степени 2, сдвиг вправо — делению на степени 2 с потерей остатка. Так как компьютер очень быстро выполняет битовые сдвиги, использование этих операций ускоряет двоичные вычисления в программах на языке ассемблера.

Упражнения

3.1. Что означает слово "бит"?

3.2. Сколько битов содержится в байте? Сколько байтов в слове? Сколько слов в учетверенном слове?

3.3. Что такое младшая значащая цифра, старшая значащая цифра, старший двоичный разряд, младший двоичный разряд?

3.4. Чему равна сумма двух двоичных значений 0110 1011 1111 1001 и 1010 1011 1100 1000?

3.5. Найдите шестнадцатеричные эквиваленты для двоичных значений из вопроса 3.4 (включая сумму).

3.6. Чему равно в десятичной системе 2^7? Какой разряд (номер бита) в двоичном значении имеет степень 2^7?

3.7. Чему равно 3ECA в десятичной системе? Чему равно десятичное число 12152 в двоичной системе? Попробуйте выполнить эти преобразования вручную, даже если у вас есть калькулятор. (Подсказка: преобразуйте сначала десятичное значение в шестнадцатеричное, затем с помощью подстановки — в двоичное.)

3.8. Как с помощью AND-маски выделить из 8-битового байта 5-, 3- и 2-й биты? Как использовать OR-маску для установки 7- и 6-го битов в 1? Как с помощью XOR-маски изменять на обратное значение старшей значащей цифры байта?

3.9. [*Повышенной сложности*] Какую комбинацию масок и логических операций необходимо использовать для установки 3- и 7-го битов в 1, изменения на противоположное значения 2-го бита, сохраняя при этом неизменными остальные биты в байте?

3.10. Сколько битов необходимо для представления десятичного числа в 2048?

3.11. Найдите значения в дополнительном коде чисел 1011 1111, 0000 0001, 1000 0000, 1110 0001 и 1111 1111.

3.12. Чему равен десятичный эквивалент двоичного значения со знаком 1111 1001? Чему он равен, если рассматривать это же двоичное число как беззнаковое?

3.13. Какое максимальное значение может быть выражено с помощью шести битов? Сколько значений может быть выражено девятью битами?

3.14. Умножьте 0011 1001 на 4, используя битовый сдвиг. Разделите 1001 1100 на 8. Проверьте полученные значения в десятичной системе. Почему с помощью сдвига нельзя умножать 0101 0101 на 8?

Задания

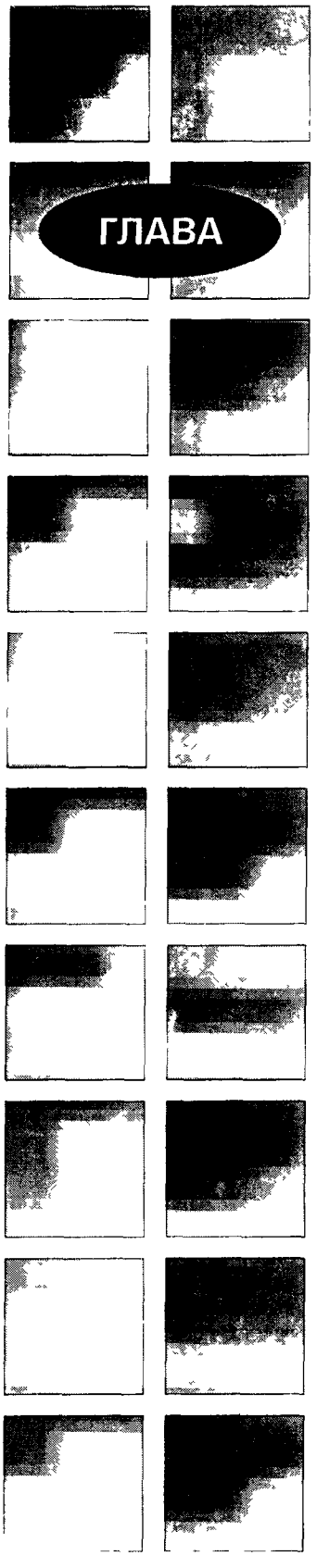
3.1. Посчитайте от 0 до 16 в двоичной и шестнадцатеричной системах без помощи табл. 3.1. Создайте свой собственный карманный справочник по преобразованиям из двоичной в шестнадцатеричную систему.

3.2. Постройте числовые окружности, аналогичные рис. 3 3, для трех- и пятибитовых двоичных значений.

3.3. Зачем, по вашему мнению, процессоры типа 8086 требуют обратного порядка записи байтов в слове?

3.4. Запишите номера битов для 16-битового слова, как это показано на рис. 3.1.

3.5. Напишите таблицы истинности операторов AND, OR и XOR без помощи табл. 3.2.



ГЛАВА 4

Программирование на языке ассемблера

- Сегментация памяти
- Регистры процессора 8086
- Основные группы команд процессора
- Команды передачи данных
- Арифметические команды
- Логические команды
- Команды управления потоком
- Команды управления процессором
- Команды для работы со строками
- Резюме
- Упражнения
- Задания

Сегментация памяти

Перед тем как изучать регистры и команды процессора 8086, необходимо понять, как он адресует память, используя системные *сегменты* и *смещения* — термины, часто вызывающие недоразумения.

Используя для представления значения адреса 20 бит, 8086-й процессор может иметь прямой доступ к 1 Мбайт памяти. DOS, ROM BIOS и другие находящиеся в памяти программы уже занимают определенное место памяти персонального компьютера, поэтому остальные программы запускаются, используя меньшее пространство памяти — до 640 Кбайт. Если вы хотите, чтобы ваша программа работала на возможно большем числе компьютеров — ограничьте пределы необходимой памяти.

Замечание

Последние модели процессоров, такие как 80386, 80486 и Pentium (известный как 80586), эмулируют режим процессора 8086. Методы, описанные в этой главе, применимы ко всем процессорам 80x86

Независимо от того, сколько памяти может адресовать процессор или сколько микросхем памяти установлено в компьютере, наименьшим блоком памяти является 8-битовый байт. Как отмечалось ранее, каждый байт имеет свое уникальное местоположение, называемое *физическим адресом*, по которому программы могут читать и записывать информацию. Очевидно, что для адресации больших объемов памяти необходимо большее число бит под физические адреса. Если в вашем компьютере 64 Кбайт памяти, тогда для задания адреса любого байта достаточно 16 бит, с помощью которых представляются значения от 0 до 65535 (2^{16}–1), или округленно — 64 Кбайт. Для адресации 1 Мбайт памяти персонального компьютера требуется как минимум 20 бит (2^{20}–1 равно 1048575, или, в шестнадцатеричном представлении, FFFFF). Проблема состоит в том, что 8086 процессор использует только 16-битовые регистры. Тогда каким образом процессор 8086 может иметь доступ ко всему мегабайту памяти в обычном персональном компьютере?

Ответом является *сегментация памяти* — метод, используемый процессором 8086 для разделения большого адресного пространства памяти на логические 64 Кбайт фрагменты. С помощью этого метода адрес конкретного байта может быть выражен двумя значениями: адресом фрагмента (сегмента) и 16-битовым смещением от начала сегмента.

Эта комбинация значений сегмента и смещения называется *логическим адресом*. Первый байт сегмента имеет смещение 0000, второй — 0001, третий — 0002 и т.д., независимо от того, где сегмент физически начинается в памяти. Рис. 4.1 демонстрирует эту идею, показывая, что каждая точка памяти имеет как физический адрес (справа), так и логический адрес (слева), выраженный через смещение от начала границы сегмента. С помощью сегментации процессор 8086 может эффективно адресовать до 1 Мбайт памяти, даже используя относительно небольшие 16-битовые регистры. Кроме того, сегментация позволяет легко перемещать программы в новые физические области памяти, изменяя только базовый адрес сегмента. При этом не требуется изменять значения смещений внутри сегмента, что позволяет *перемещаемым программам* исполняться одинаково в разных местах памяти.

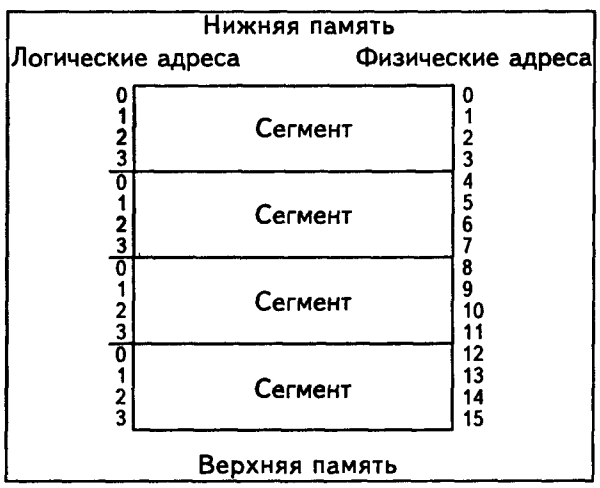


Рис. 4.1. Логические адреса с соответствующими физическими адресами в памяти

Параграфы, сегменты и смещения

Для определения начал сегментов памяти процессор 8086 использует четыре 16-битовых сегментных регистра. Для создания 20-битового физического адреса процессор объединяет значение одного сегментного регистра с 16-битовым смещением (логическим адресом). Это делается путем умножения сегментного значения на 16 (сдвиг влево на 4 бита) и прибавлением к результату смещения (умножение на 16 эквивалентно четырехкратному сдвигу битов влево, как вы помните из главы 3). Исходя из этого получается, что границы сегментов располагаются через каждые 16 байт физических адресов. Каждый из этих 16-байтовых фрагментов называется *параграфом*. Простые вычисления показывают, что максимально может существовать 65536 параграфов и, следовательно, такое же число границ сегментов в одномегабайтовом адресном пространстве процессора 8086 (1048576/16). (Заметим, что это также равняется числу значений, которые могут быть выражены с помощью одного 16-битового сегментного регистра.) Кроме того, запомните несколько важных сведений о сегментах.

- Сегменты физически не выделены в памяти, что является общим заблуждением. Сегменты — это логические окна, через которые программы просматривают области памяти удобными, в 64 Кбайт порциями.
- Начальное положение сегмента (т.е. логический адрес) определяется вами и может иметь любое значение от 0000 до FFFF в шестнадцатеричном представлении. Каждое логическое значение сегмента (0, 1, 2,..., 65535) соответствует физической границе параграфа (0, 16, 32,..., 1048560).

- Размеры сегментов могут изменяться от 16 байт до 64 Кбайт (65536 байт). Действительный размер сегмента определяется вами и вашей программой.
- Сегменты не обязательно располагаются физически в памяти один за другим, хотя так бывает достаточно часто.
- Сегменты могут перекрываться один другим; поэтому один и тот же байт в памяти может иметь различные логические адреса, определяемые разными, но при этом эквивалентными парами сегмент-смещение. Но, несмотря на это, каждый байт имеет только единственный 20-битовый физический адрес.

Почти всех, кто впервые знакомится с сегментацией памяти, этот последний пункт приводит в замешательство. Две разные пары сегмент-смещение могут (и так часто бывает) указывать в памяти на один и тот же байт. Если вы вспомните, как процессор создает 20-битовые физические адреса — умножением значения сегмента на 16 и прибавлением смещения, вы увидите, что шестнадцатеричные значения пар сегмент-смещение 0000:0010 и 0001:0000 указывают на одну и ту же физическую область памяти. Если повторить процесс преобразования этих логических адресов в физические в десятичной системе счисления, то при вычислении (0000×16)+16 и (0001×16)+0 получается один и тот же результат — 16.

Замечание

Обычно логические адреса сегмента и смещения записываются с помощью двух четырехзначных шестнадцатеричных чисел, разделенных двоеточием, например: 0140:001A и F000:0010. При виде подобных значений вы должны понимать, что они являются шестнадцатеричными. Это легко забыть при работе с адресами вида 0100.1024 и 0000 0010, для которых не всегда очевидно, что они представлены в шестнадцатеричной системе. При обозначении шестнадцатеричного числа в программах на языке ассемблера первый символ должен быть десятичной цифрой (от 0 до 9), а после последнего символа приписывается латинская буква „h", например: 137h, 502h, 2h, 0BF8Ch, 0Ah.

Регистры процессора 8086

На рис. 4.2 представлены регистры процессора 8086. Такие же самые регистры имеются во всех моделях процессоров 80x86. (Процессоры 80386, 80486 и Pentium имеют дополнительные регистры и расширения, которые здесь рассматриваться не будут.) Если вы используете только регистры, содержащиеся на рис. 4.2, ваши программы обязательно будут исполняться на всех персональных компьютерах. Регистры делятся на пять категорий:

- Регистры общего назначения (ax, bx, cx, dx)
- Регистровые указатели и индексные регистры (sp, bp, si, di)
- Сегментные регистры (cs, ds, ss, es)
- Регистр командного указателя (ip)
- Флаги (of, df, if, tf, sf, zf, af, pf, cf)

Все регистры 8086 являются 16-битовыми. Кроме того, четыре регистра общего назначения — ax, bx, cx, dx — разделены на старшую и младшую 8-битовые половины. Например, 16-битовый регистр ax состоит из двух 8-битовых частей — ah и al. Регистр bx разделен на bh и bl, cx — на ch и cl, dx — на dh и dl. Такая гибкая организация позволяет вам оперировать либо всем 16-битовым регистром, либо работать отдельно с двумя 8-битовыми половинами регистра. Помните, что изменение значения в 16-битовом регистре ax приводит к изменению значений в его 8-битовых частях — al и ah. И наоборот, при изменении значения в cl изменяется величина cx.

Замечание

В данном тексте регистры записываются строчными буквами — cs, ax, si и т.д. В программах и других руководствах часто можно встретить регистры, написанные прописными буквами — AX, BX, DH. Обе формы записи являются правильными.

Регистры общего назначения

Программы на языке ассемблера обращаются к регистрам, используя их символические имена — ax, cl, ds и подобные им. Но, кроме того, как показано справа на рис. 4.2, регистры имеют также и менее фамильярные названия. (Хотя эти названия никогда не используются непосредственно в программах.) В *регистре-сумматоре* ax обычно содержат результаты сложения, вычитания и т.п. *Базовый* регистр bx часто указывает на начальный адрес (называемый базой) структуры в памяти. *Счетчик* cx определяет количество повторов некоторой операции. Регистр *данных* dx чаще всего содержит данные, передаваемые для обработки в подпрограммы. Эти определения никоим образом не являются полными, и большую часть времени вы сами решаете, как использовать эти регистры общего назначения. Например, несмотря на то что cx назван регистром-счетчиком, ничто не мешает вам использовать для подсчета регистр bx. Однако в ряде случаев некоторые команды процессора 8086 требуют строго определенных регистров.

Рис. 4.2. Регистры 8086

Регистровые указатели и индексные регистры

В отличие от четырех регистров общего назначения, другие регистры 8086 на рис. 4.2 тесно связаны с определенными операциями. *Указатель стека* sp всегда указывает на вершину стека процессора. (Детально стеки мы изучим немного позже.)

Указатель базы bp обычно адресует переменные, хранимые в стеке. *Индекс источника* si и *индекс назначения* di известны под названием *строковых регистров*. Обычно si и di служат для выполнения операций над байтовыми строками.

Замечание

Байтовые строки отличаются от символьных строк в языках высокого уровня. В языке ассемблера байтовая строка представляет собой просто ряд последовательных байтов. Чтобы избежать недоразумений, для обозначения ASCII-строк я буду использовать термин „символьная строка", как это делается в большинстве языков программирования высокого уровня. Под просто строкой будет подразумеваться любая последовательность байтов, которые также могут представлять символы.

Сегментные регистры

Как показано на рис. 4.3, четыре сегментных регистра — cs, ds, ss, es — определяют в памяти начала четырех 64 Кбайтовых сегментов. Программа может распределять более четырех сегментов, но при этом для адресации дополнительных сегментов она должна перемещать соответствующие им правильные значения между одним или несколькими сегментными регистрами.

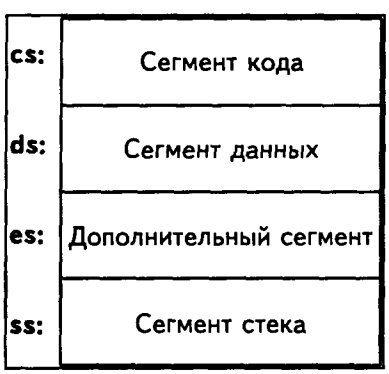


Рис. 4.3. Адреса сегментных регистров, соответствующие четырем сегментам памяти

Сегментные регистры являются строго специализированными. Вы не можете выполнять над ними математические вычисления или хранить в них результаты других операций. *Регистр сегмента кода* cs указывает в памяти на начало машинного кода программы. *Регистр сегмента данных* ds содержит начальный адрес переменных программы. *Регистр сегмента стека* ss определяет начало стекового пространства. *Дополнительный сегментный регистр* es является вспомогательным, но несмотря на то что в большинстве программ в es и ds содержатся одинаковые адреса, он может упрощать некоторые операции, связанные с этими регистрами. Действительный порядок сегментов не обязательно совпадает с порядком, показанным на рис. 4.3. Позже будет объяснено, что сегменты могут в любом порядке храниться в произвольном месте памяти.

Регистр командного указателя

Специальный *регистр командного указателя* ip указывает на следующую исполняемую машинную команду относительно определенного в cs сегмента. Вы редко будете прямо обращаться к ip. Вместо этого для изменения адреса следующей исполняемой инструкции вы будете использовать специальные команды, которые меняют значения в ip (и, возможно, в cs), приводя тем самым к изменению порядка исполнения программы. Например, вызов подпрограммы приводит к загрузке ее адреса в ip (или в пару cs:ip).

Флаги

Хотя *регистр флагов* состоит из 16 бит, из них используются только 9 (см. рис. 4.2), остальные 7 бит программами не используются. Отдельные биты флагов представляются одиночными буквами o, d, i, t, s, z, a, p, c. В некоторых руководствах (включая и это) для них часто применяются обозначения of, df, if и т.п. В табл. 4.1 содержатся полные имена для каждого битового флага.

В основном в битах регистра флагов процессора 8086 отражаются результаты выполнения различных команд и операций. Например, после сложения флаг переноса cf показывает, получился ли в результате перенос. Флаг переполнения указывает, может ли результат сложения чисел со знаком быть правильно представлен с помощью заданного количества битов. Флаги могут также служить для других целей. Например, можно выполнить в регистре сдвиг влево, передав первую значащую цифру для проверки во флаг переноса cf, после чего другие команды могут выполняться в зависимости от установки этого или других битов регистра флагов. Можно так же использовать cf в качестве однобитового устройства, предупреждающего о возникновении ошибок, информируя другие части программы о неполадках. По мере изучения команд языка ассемблера вы также узнаете о той важной роли, которую играют флаги в работе программ.

Таблица 4.1. **Флаги 8086**

Символ	Полное имя
o или of	Флаг переполнения
d или df	Флаг направления
i или if	Флаг прерывания
t или tf	Флаг трассировки
s или sf	Флаг знака
z или zf	Флаг нуля
a или af	Вспомогательный флаг
p или pf	Флаг четности
c или cf	Флаг переноса

Основные группы команд процессора

По своим функциям все команды процессора 8086 делятся на шесть категорий. Далее в этой главе вы найдете описание каждой из этих групп и небольшие программы, которые помогут вам увидеть функционирование большинства команд 8086. (Оставшиеся команды будут введены в следующих главах.) Эти шесть групп имеют следующий вид:

- Команды передачи данных
- Арифметические команды
- Логические команды
- Команды управления потоком
- Команды управления процессором
- Команды для работы со строками

Замечание

В главе 16 содержится список всех команд с примерами программ и полными описаниями элементов данных, обрабатываемых командами. Если вы встречаете новую для себя команду, то за разъяснениями обращайтесь к главе 16.

Команды передачи данных

Команды передачи данных процессора 8086 представлены в табл. 4.2. Эта группа делится на четыре части: общие, ввода-вывода, адресные и флагов. Операнды справа от каждого символического имени команды определяют необходимые ей элементы данных. Для большей части команд определены операнды назначения и источника. Другие команды требуют один операнд или не требуют их вообще.

Рассмотрим первую из команд передачи данных — mov. Вероятно, mov чаще других команд встречается в программах на языке ассемблера. Из табл. 4.2 вы можете увидеть, что для mov необходимы два операнда: операнд-источник и операнд-назначение. Отметим, что источник записывается после назначения, при предположении, что mov работает следующим образом:

```
mov назначение <-- источник
```

Таблица 4.2. **Команды передачи данных**

Мнемокод / Операнды	Описание
Основные команды	
mov назначение, источник	Переслать (скопировать) байт или слово
pop назначение	Извлечение слова из стека
push источник	Занесение слова в стек
xchg назначение, источник	Перестановка байтов и слов
xlat/xlatb таблица	Перекодировка таблицы
Команды ввода-вывода	
in сумматор, порт	Ввод (выборка) байта или слова из порта
out порт, сумматор	Вывод (размещение) байта или слова в порт
Адресные команды	
lds назначение, источник	Загрузка регистра сегмента данных (используется ds)
lea назначение, источник	Загрузка эффективного адреса
les назначение, источник	Загрузка регистра дополнительного сегмента (используется es)
Команды флагов	
lahf	Загрузка флагов в регистр AH
popf	Извлечение регистра флагов из стека
pushf	Занесение регистра флагов в стек
sahf	Установка флагов из регистра AH

Данные из источника передаются в направлении стрелки, справа налево. Будьте внимательны, чтобы не перепутать местами операнды, что является типичной и опасной ошибкой.

Следующая команда в языке ассемблера пересылает значение регистра bx в регистр ax:

```
mov ax, bx                ; ax <-- bx
```

Если значение ax равно 0000, а bx — 0123h, то после выполнения этой команды значение ax станет равным 0123h. Величина bx при этом не изменится. Некоторые программисты для прояснения направления передачи данных любят использовать комментарии. Например:

```
mov cx, [numPages]                ; cx <-- [numPages]
```

Данная команда пересылает значение, содержащееся в [numPages], в регистр cx. Скобки вокруг numPages имеют важное значение. Метка numPages определяет адрес в памяти. Но со скобками [numPages] соответствует данным, записанным по этому адресу. Это понятие, т.е. метки, определяют адреса, по которым данные хранятся в памяти, — жизненно необходимо вам для понимания принципов программирования на языке ассемблера. Каждый раз вы должны быть очень внимательны при определении того, будет ли команда работать со значениями адресов или с данными, хранящимися по этим адресам. Скобки являются простым инструментом для этих целей, но вы должны помнить, как правильно их использовать.

Вы можете также пересылать данные из регистров в память. Например, следующая команда записывает значение 8-битового регистра dl по адресу, определенному в level:

```
mov [level], dl                   ; [level]<----- dl
```

Скобки подсказывают вам, что значение из dl пересылается по адресу, на который указывает метка level. Такая пересылка данных — копирование значения одного регистра в другой и перемещение данных из регистра в память — является одной из наиболее общих операций в программировании на языке ассемблера. Однако mov не может пересылать данные непосредственно между двумя адресами памяти. Операция, представленная ниже, является некорректной:

```
mov [count], [maxCount]           ; ???
```

Пересылка значения, записанного в maxCount по адресу count, вместо этого требует выполнения двух шагов и использования промежуточного регистра:

```
mov ax, [maxCount]              ; ax<---- [maxCount]
mov [count], ax                 ; [count]<----- ax
```

Пример использования команды mov

Программа 4.1 показывает, как работает команда mov. Выполните ассемблирование и компоновку, после чего загрузите программу в Turbo Debugger, используя для этого следующие команды:

```
tasm /zi mov
tlink /v mov
td mov
```

Программа 4.1. MOV.ASM

```
1:  %TITLE "Использование команды MOV -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL   small
6:          STACK   256
7:
8:          DATASEG
9:
10: exCode  DB      0
11: speed   DB      99      ; Однобайтовая переменная
12:
13:         CODESEG
14:
15. Start:
16.         mov     ax, @data       ; Установка в DS
17:         mov     ds, ax          ; адреса сегмента данных
18:
19.         mov     ax, 1           ; Переслать конкретные данные
20:         mov     bx, 2           ; в регистры
21:         mov     cx, 3
22.         mov     dx, 4
23:
24:         mov     ah, [speed]     ; Загрузить значение speed в al
25:         mov     si, offset speed ; Загрузить адрес speed в si
26:
27: Exit:
28:         mov     ah, 04Ch        ; Функция DOS : Выход из программы
29:         mov     al, [exCode]    ; Возврат значения кода ошибки
30:         int     21h             ; Вызов DOS. Останов программы
31:
32:         END     Start           ; Конец программы / точка входа
```

Исполнение программы MOV под управлением Turbo Debugger

Итак, вы загрузили программу MOV в Turbo Debugger. Выполнение следующих шагов поможет вам разобраться в том, как работает команда mov.

1. Для открытия окна CPU нажмите последовательно <Alt+V+C>, затем распахните его на весь экран, нажав <F5>. В окне CPU отображается большое количество важной информации — стек, регистры, флаги, память и команды, поэтому именно это окно вы должны использовать для исполнения большинства программ на языке ассемблера из этой книги.
2. После того как вы прочитаете следующие описания, нажмите <F8> для пошагового (покомандного) выполнения программы. (Нумерация строк соответствует тексту программы 4.1.)
3. Строки 16–17 инициализируют сегментный регистр ds, занося в регистр предопределенное значение @data и пересылая его затем в регистр ds. (В сегментный регистр вы можете записывать значения исключительно

через регистры общего назначения, переменные, содержащиеся в памяти, или стек. Нельзя прямо присваивать сегментному регистру литеральные значения.)

4. После выполнения строк 19–22 в регистры общего назначения ax, bx, cx и dx записываются литеральные значения 1, 2, 3, 4. Нажимайте <F8>, пока стрелка, указывающая на исполняемые команды в Turbo Debugger (справа от адресов вида cs:0011), не остановится напротив mov ah, [speed]. Если вы случайно проскочили эту команду, нажмите для перезапуска программы <Ctrl+F2>, после чего снова нажимайте <F8>, пока не вернетесь в нужное место.)
5. Команда mov ah, [speed] в строке 24 загружает значение, содержащееся по адресу speed, в 8-битовый регистр ah. Обратите внимание на текст в двойной рамке в верхней части экрана — ds:0001= 63. Он указывает шестнадцатеричное значение (63), которое будет загружено в регистр ah. Запись ds:0001 отображает адрес, по которому было записано значение. Подобно другим адресам, он состоит из двух частей: значения сегмента (содержащегося в ds) и смещения 0001.
6. Для выполнения команды в строке 24 нажмите <F8> и наблюдайте в правом верхнем углу экрана за изменением значения в регистре ax. Обратите внимание, что запись ds:0001=63 при этом исчезла. Чтобы снова увидеть ее, воспользуйтесь клавишами "стрелка вверх" и "стрелка вниз" для перемещения выделенного курсора. Перемещая прямоугольный курсор на любую отдельную команду, вы всегда можете наблюдать за результатами загрузки значений в регистры или память.
7. Найдите в правой верхней трети окна CPU регистр si. Снова нажмите <F8>, выполнив команду mov si, offset speed в строке 25. Как вы видите, эта команда устанавливает 0001 в регистре si, т.е. значение смещения адреса из предыдущего шага. Ключевое слово OFFSET в команде mov предупреждает ассемблер, что вы собираетесь использовать смещение адреса метки. (OFFSET на вашем экране может записываться прописными буквами.)
8. Продолжайте нажимать <F8> до окончания программы. В строках 28–30 выполняются три шага, завершающие любую EXE-программу. Сначала в регистр ah загружается код функции выхода DOS (04Ch). Затем в регистр al записывается содержимое переменной exCode, которая может возвращаться программой в DOS в качестве индикатора ошибки. Нулевое значение означает отсутствие ошибок. Команда int 21h в строке 30 вызывает DOS с параметрами, содержащимися в ah и al, завершая программу.
9. Для выхода из Turbo Debugger нажмите <Esc> и <Alt+X>.

Замечание

Строчная буква h в конце значений типа 21h или 04Ch показывает Turbo Assembler, что эти величины выражены в шестнадцатеричной системе, значения которой всегда начинаются с десятичной цифры. Другими словами, вы не можете написать FFFh, вместо этого нужно записать 0FFFh.

Стековая память

Стек — это специальный сегмент памяти, который используется некоторыми командами процессора 8086. Так же как и в случае с другими сегментами, положение стека и его размер (до 64 Кбайт) определяются вами и вашей программой. Самый простой способ создания стека на языке ассемблера состоит в использовании директивы STACK, как это и делается в большинстве примеров программ из этой книги. Если вы не создадите стек, то получите предупреждение от Turbo Linker. Стек выполняет три основные функции:

- Временное сохранение значений регистров
- Хранение адресов возврата из подпрограмм
- Хранение динамических переменных

Последний из этих пунктов более характерен для программирования на языках высокого уровня, когда через стек осуществляется передача переменных в функции и процедуры. В стеке также могут сохраняться временные переменные. Это редко используется при программировании на чистом языке ассемблера, однако при желании вы, конечно, можете пользоваться стековой памятью для записи переменных.

Как работает стек

Образно стек напоминает гору тарелок на кухне ресторана. Верхнюю тарелку взять легко, но чтобы добраться до тарелки, находящейся внизу, необходимо снять все тарелки над ней. Помещение новой порции на вершину стека называется *проталкиванием* в стек (push). Удаление порции из вершины стека, приводящее к поднятию на один уровень остальных порций, называется *выталкиванием* из стека (pop). При проталкивании в стек последней порции она первой и выталкивается, поэтому такой тип стека называется LIFO-стек (Last-In-First-Out — последний-пришел-первый-ушел).

В отличие от тарелок, значения в памяти компьютера физически не могут передвигаться вниз или вверх. Поэтому для моделирования передвижения значений в стеке необходимо использовать регистры, которые определяют базовый адрес стека и смещение вершины стека, т.е. местоположение верхнего значения в стеке. При программировании под 8086 процессор сегментный регистр ss содержит адрес начала стекового сегмента. В регистре sp содержится смещение вершины стека относительно этого сегмента.

На рис. 4.4 показано, как располагается в памяти небольшой стек из 12 байт. В регистре ss содержится адрес начала стекового сегмента — 0F00. Регистр sp указывает на смещение относительно этого начального адреса в интервале от 0000 до 000A. Последний байт стека имеет смещение 000B (на рисунке он расположен справа от байта со смещением 000A). Под элементы стека выделяются 2-байтовые слова. Чтобы использовать этот стек, в программе достаточно декларировать STACK 12, после чего ассемблер, компоновщик и DOS сами точно определят, где расположить стек в памяти. Вам не нужно инициализировать регистры ss и sp. Это делает за вас DOS при загрузке ассемблерной программы. На рисунке sp1 показывает, куда при запуске программы указывает sp. Обратите внимание, что логический адрес ss:sp указывает на байт, расположенный *ниже* последнего байта в стеке.

Замечание

Поскольку основание стека расположено в более высоких адресах памяти, чем вершина стека, термины „основание", „выше", „ниже" могут вызывать затруднения. Но для стека эти термины устоялись, поэтому остается лишь примириться с этой двусмысленностью. Только помните, что в памяти стек растет в направлении уменьшения адресов памяти и уменьшается в противоположном направлении.

Вернитесь снова к рис. 4.4. К требуемому результату приведет выполнение следующих команд:

```
mov         ax, 100
push ax                 ; sp2
mov         bx, 200
push bx                 ; sp3
```

Команда push выполняет два действия.

1. Уменьшает на 2 значение sp.
2. Заданное значение регистра записывается по адресу [ss:sp].

Порядок этих шагов существен. Сначала push вычитает 2 из sp. На рис. 4.4 первый оператор push записывает sp2 в sp, помещая затем туда значение из регистра ax. Заметим, что после выполнения этих действий указатель стека адресует последнее помещенное в стек значение.

Замечание

Привыкайте к записи [ss:sp], под которой понимается значение со смещением sp внутри стекового сегмента. Помните, что скобки указывают на значение, расположенное по данному адресу.

Пример использования стека

Лучшим способом для понимания работы стека является наблюдение за ним в действии с помощью Turbo Debugger. Для этого используйте программу 4.2, в которой представлено наиболее характерное использование стека — для сохранения значений регистров. Выполните ассемблирование, компоновку и загрузку программы в Turbo Debugger с помощью следующих команд:

```
tasm /zi pushpop
tLink /v pushpop
td pushpop
```

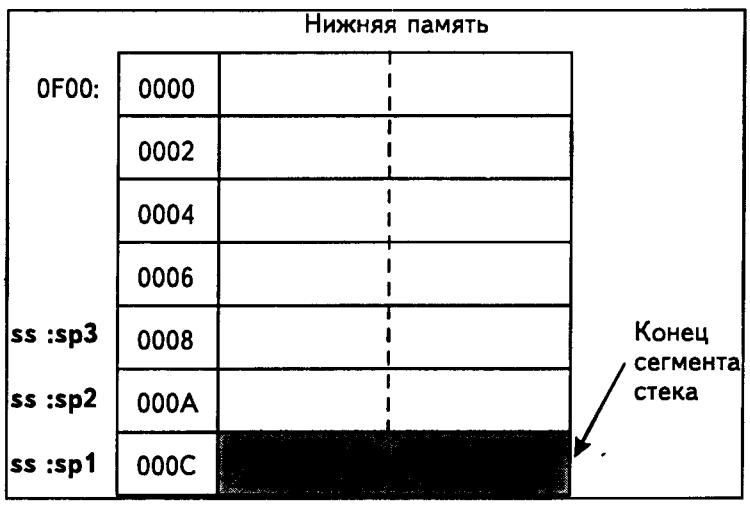


Рис. 4.4. Стековый сегмент

Программа 4.2. PUSHPOP.ASM

```
1:  %TITLE "Использование операторов PUSH/POP -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:          DATASEG
9:
10: exCode  DB      0
11:
12:         CODESEG
13:
14: Start:
15:         mov     ax, @data       ; Установка в DS адреса
16:         mov     ds, ax          ; сегмента данных
17:
18:         push    ax              ; Сохранить ax и bx
19:         push    bx              ; в стеке
20:
21:         mov     ax, -1          ; Установить тестовые значения
22:         mov     bx, -2
23:         mov     cx, 0
24:         mov     dx, 0
25:
26:         push    ax              ; Протолкнуть ax в стек
27:         push    bx              ; Протолкнуть bx в стек
28:         pop     cx              ; Вытолкнуть cx из стека
29:         pop     dx              ; Вытолкнуть dx из стека
30:
31:         pop     bx              ; Восстановить из стека
32:         pop     ax              ; сохраненные значения ax и bx
33:
34: Exit:
35:         mov     ah, 04Ch        ; Функция DOS : Выход из программы
36:         mov     al, [exCode]    ; Возврат значения кода ошибки
37:         int     21h             ; Вызов DOS. Останов программы
38:
39:         END     Start           ; Конец программы / точка входа
```

Ниже приведены подробные инструкции по выполнению программы под управлением Turbo Debugger.

Исполнение демонстрационной программы PUSHPOP

Итак, вам необходимо запустить на исполнение программу PUSHPOP в Turbo Debugger. Для наблюдения за работой стека выполните следующие шаги.

1. Откройте и распахните на весь экран окно CPU последовательным нажатием <Alt+V+C> и <F5>. Дважды нажмите <F8>, чтобы оказаться на 18-й строке. Запомните значения регистров ax и bx.
2. Наблюдайте за значениями стека в правом нижнем углу — в окне с адресами, первый из которых — ss:. Чтобы протолкнуть значение ax в стек, нажмите один раз <F8>. Снова нажмите <F8> для проталкивания значения bx. Вершина стека, расположенная внизу окна, отмечается стрелкой. (Наверно, только разработчикам Turbo Debugger известно, почему "вершина" стека появляется в "основании" окна. Я уже говорил о странности этих терминов.)
3. Нажмите <F8> четыре раза, выполнив строки 21–24 и загрузив в регистры ax, bx, cx и dx тестовые значения
4. Снова нажмите <F8> для выполнения строки 26 и проталкивания в стек значения ax. Наблюдайте за содержанием стека и значением sp до и после проталкивания. Еще раз нажмите <F8> для проталкивания bx.
5. Строки 28 и 29 выталкивают стек, выбирая значение [ss:sp] и увеличивая sp на 2, адресуя тем самым следующее слово. Для выполнения этих двух строк дважды нажмите <F8>. Обратите внимание на то, что вы можете вытолкнуть значения из стека в регистры, отличные от тех, которые проталкивались ранее.
6. Дважды нажмите <F8> для выполнения строк 31–32. Эти команды возвращают в регистры bx и ax значения, которые хранились там до выполнения строк 18–19.
7. Выйдите из Turbo Debugger, нажав <Alt+X>. В данном случае нет необходимости выполнять программу до конца.

Управление стеком

Принцип правильного управления стеком прост: каждая команда push в программе должна иметь соответствующую команду pop. Такое равенство обеспечивает правильность указателя стека — другими словами, при этом программа может корректно записывать и восстанавливать значения.

Замечание

Существует ряд исключений из правила о том, что каждая команда push должна иметь компенсирующую ее команду pop. Например, для того чтобы зарезервировать в стеке пространство под временные переменные, вы можете добавлять и вычитать значения из sp. Более того, можно завершить программу с помощью функции DOS 4C даже в том случае, если стек не синхронизирован. Однако всегда старайтесь поддерживать стек в известном состоянии Невнимательное управление стеком является одной из основных причин серьезных ошибок.

Рассмотрим, что произойдет, если вы не будете придерживаться соответствиям между pop и push. В этом случае следующие проталкивания приведут все к большему и большему росту стека, возможно, вызывая переполнение сегмента, выделенного вашей программой. Эта серьезная ошибка обычно приводит к зависанию системы, так как, вследствие некорректного значения указателя стека, начинают перекрываться области памяти. Аналогичная ошибка возникает, если количество команд pop превышает число команд push, вызывая исчезновение стека и обычно приводя к зависанию системы.

Хороший способ избежать таких проблем состоит в том, чтобы писать программы в виде небольших модулей или подпрограмм. В каждом модуле протолкните в стек все регистры, которые вы планируете в нем использовать. Перед тем как закончить код, вытолкните в обратном порядке из стека эти же регистры.

Например, можно создать такой типичный фрагмент кода:

```
push ax                         , Занести ax, bx, dx, в стек
push bx
push dx


, -------- Здесь содержится программа


pop dx                          , Извлечение dx, bx, ax, из стека
pop bx
pop ax
```

Вероятно, команды между push и pop будут использовать регистры ax, bx и dx; поэтому для сохранения первоначальных значений они проталкиваются в стек. Позже те же регистры в обратном порядке выталкиваются из стека, восстанавливая исходные значения и поддерживая стек в правильном состоянии. Вспоминая аналогию с тарелками, можно увидеть, что для восстановления ранее сохраненных значений в нужных регистрах необходимо обязательно выталкивать их в обратном порядке. Последнее значение, которое протолкнулось в стек (dx), выталкивается первым, в то время как первая порция (ax), которую протолкнули, выталкивается последней.

Замечание

После выталкивания значения из стека не пытайтесь вычитать 2 из sp и снова читать это же значение. Это всегда неверно, даже несмотря на то, что при просмотре стека в Turbo Debugger можно заметить, что „вытолкнутое" значение могло остаться в стековой памяти, имея смещение, меньшее, чем sp. Гарантируется сохранность только тех значений, которые расположены между sp и основанием стека. Все другие значения в стековом сегменте могут изменяться либо DOS, либо, что вероятнее, прерываниями, которые выполняются параллельно с вашей программой. (Подробнее обработка стека и прерываний объясняется в главе 10.) Несоблюдение этого правила, скорее всего, приведет к неправильной работе вашей программы. Не допустите этого!

Перестановка данных

Рассмотрим другую команду из табл. 4.2 — xchg, которая меняет местами значения двух регистров либо значение регистра с байтом или словом, хранимым в памяти. Предположим, вы хотите переставить значения в регистрах dx и ax. Используя xchg, можно просто записать:

```
xchg ax, dx             ,ax <- dx, dx <- ax
```

Хотя в табл. 4.2 для xchg приводятся операнд-источник и операнд-назначение, их порядок не играет никакой роли, так как команда просто меняет местами значения операндов. Перестановка двух регистров без использования xchg требует или проталкивания в стек, или третьего регистра. В качестве примера ниже приводится менее эффективный метод для перестановки двух 16-битовых регистров, использующий стек как промежуточное место хранения одного из значений:

```
push ax          ; стек <- ax
mov  ax, dx      ; ax <- dx
pop  dx          ; dx <- стек (первоначальное значение ax)
```

Для перестановки двух 8-битовых значений необходим третий регистр, так как в стек можно проталкивать и выталкивать только 16-битовые слова. Чтобы переставить два байта в al и ah, не используя xchg, вы должны записать:

```
mov bh, ah       ; bh <- ah
mov ah, al       ; ah <- al
mov al, bh       ; al <- bh
```

Конечно, при xchg во всем этом нет никакой необходимости. (Однако полезно знать, как для этих целей может использоваться стек и другие регистры.) Кроме перестановки значений регистров, xchg также может менять местами значение в регистре со значением, хранящимся в памяти. Ниже приводятся два примера:

```
xchg ax, [things]        ; ax <--> [things]
xchg [oldCount], cx      ; cx <--> [oldCount]
```

В первой строке значение ax меняется местами со значением, содержащимся по адресу things. Во второй строке переставляются cx и oldCount. При этом порядок операндов снова несуществен.

Замечание

Перестановка полных 16-битовых значений регистров осуществляется несколько быстрее, если одним из регистров является сумматор ax Turbo Assembler преобразует команды вида xchg ax,bx и xchg cx,ax в быстрые однобайтовые машинные коды. Другие перестановки, не использующие ax, приводят к двухбайтовым машинным кодам. Имейте в виду, что не все ассемблеры столь изящны, как Turbo. Например, ассемблер в DOS DEBUG для генерации однобайтового машинного кода требует, чтобы ax был последним операндом в команде. Кроме того, перестановка между регистрами во много раз быстрее перестановки между регистром и памятью. Уделяя внимание подобным незначительным, на первый взгляд, деталям, вы сможете достичь максимальной скорости для вашей программы.

Арифметические команды

Компьютеры в основном предназначены для выполнения математических операций, поэтому наличие в языке ассемблера только нескольких относительно примитивных математических операторов может вызывать удивление. В набор команд процессора 8086 не входят операции с плавающей точкой, возведения в степень, взятия квадратного корня, функции SIN и COS. Математические команды в языке ассемблера ограничиваются сложением, умножением, делением и вычитанием знаковых и беззнаковых целых двоичных чисел. Математические команды процессора 8086 представлены в табл. 4.3.

Существует два способа увеличения вычислительных возможностей языка ассемблера. Во-первых, вы можете приобрести (или написать) необходимые вам пакеты подпрограмм, которые реализуют функции высокого уровня. Другое решение связано с покупкой для вашего персонального компьютера математического сопроцессора, хотя это и требует определенных затрат для процессоров 80286 и 80386, которым необходимы соответствующие 80287 и 80387 математические сопроцессоры. Процессор 80486 содержит в себе встроенный 80387 сопроцессор. Третий, и возможно лучший способ, состоит в использовании для вычисления выражений с плавающей запятой языков высокого уровня типа Turbo Pascal или Turbo C. Эти языки автоматически определяют наличие математического сопроцессора и могут переключаться на программный эмулятор для систем, в которых он отсутствует. После написания программы вы можете объединить откомпилированный высокоуровневый код с программой на языке ассемблера (см. главы 12 и 13). Так как математические сопроцессоры выдвигают строгие требования к форматам данных и команд, то большинство компиляторов генерируют оптимизированный машинный код, и поэтому не имеет большого смысла писать выражения с плавающей запятой непосредственно на языке ассемблера.

Однако это не уменьшает вычислительных достоинств языка ассемблера. Даже без математической библиотеки или сопроцессора вы можете использовать множество встроенных в 8086 команд для работы с целыми значениями. На самом деле многие хорошие программы не нуждаются в таких математических возможностях. Чтобы вычислить размер директории на диске или подсчитать число слов в текстовом файле, вам не требуются никакие числа с плавающей запятой. Для этих и других операций более чем достаточно целочисленной математики. При этом в языке ассемблера такие операции обычно выполняются быстрее, чем эквивалентный код на языках высокого уровня.

Таблица 4 3. **Арифметические действия**

Мнемокод / Операнды	Описание
Команды сложения	
aaa	Коррекция ASCII-формата для сложения
adc назначение, источник	Сложение с переносом
add назначение, источник	Сложение байт или слов
daa	Десятичная коррекция для сложения
inc назначение	Инкремент
Команды вычитания	
aas	Коррекция ASCII-формата для вычитания
cmp назначение, источник	Сравнение
das	Десятичная коррекция для вычитания
dec назначение	Декремент
neg назначение	Изменение знака числа (преобразование в дополнительный код)
sbb назначение, источник	Вычитание с заемом
sub назначение, источник	Вычитание
Команды умножения	
aam	Коррекция ASCII-формата для умножения
imul источник	Умножение величин со знаком
mul источник	Беззнаковое умножение
Команды деления	
aad	Коррекция ASCII-формата для деления
cbw	Преобразование байта в слово
cwd	Преобразование слова в двойное слово
div источник	Беззнаковое деление
idiv источник	Деление величин со знаком

Команды сложения

В табл. 4.3 представлено пять команд сложения. Две из них, add и adc, суммируют два байта или слова. Inc (инкремент) — команда быстрого добавления 1 к регистру или значению в памяти. (Другие две команды, aaa и daa, выполняют коррекцию значений, представленных в *двоично-десятичном* формате, с которым мы познакомимся позже.) Для сложения 8-битового значения в ah с 8-битовым значение в bh вы можете записать:

```
add ah, bh              , ah <- ah + bh
```

Так же как и для команды mov, для add требуются операнд-источник и операнд-назначение. Команда суммирует два значения и записывает результат в заданное место, заменяя первоначальное значение. В этом примере результат записывается в ah. Команда adc работает аналогично, но добавляет к результату флаг переноса cf:

```
adc ah  bh              ; ah <- + ah + bh + cf
```

Если cf равно 1, то результат получается таким же, как и при добавлении 1 к сумме ah и bh. Флаг cf устанавливается в 1, если в предыдущем сложении возникало переполнение. Поэтому adc чаще используется после add, складывающего многобайтовые значения, информируя о переносах, которые возникают при сложении отдельных байтов. Хотя слова можно складывать непосредственно, для суммирования в регистре ax отдельных байтов 16-битового значения, записанного по адресу sum, вы можете использовать следующие команды, которые удваивают слово в sum:

```
mov ax, [word sum]              ; Переслать в ax значение [sum]
add al, [byte sum]              ; Прибавить младший байт
adc ah, [byte sum + 1]          ; Прибавить старший байт с возможным переносом
mov [word sum], ax              ; Переслать полученное значение обратно в память
```

Помните, что в слове байты хранятся в обратном порядке. В этом примере первая строка загружает значение слова в ax. Вторая строка складывает один с другим младшие значащие байты, записывая результат в al, и в случае возникновения переноса устанавливает cf в 1.

Третья строка добавляет этот возможный перенос к сумме старших значащих байтов. И наконец, четвертая строка записывает конечный результат обратно в память. Поскольку 8086 может непосредственно обрабатывать значения слов, вы можете выполнить это же сложение, используя более простые команды:

```
mov ax, [word sum]          ; Переслать в ax значение [sum]
add [word sum], ax          ; Прибавить [sum] к самому себе
```

Перед сложением вы должны загрузить [sum] в регистр, так как add не может прямо складывать два значения, записанные в памяти, — по меньшей мере, необходимо определить один регистр. Заметьте, что в этих примерах операторы word и byte информируют ассемблер о типах данных, адресуемых sum. В некоторых случаях ассемблер делает это без их помощи, в других эти операторы необходимы. Тем не менее их можно всегда без вреда использовать. (*Форматы данных и операторы подробнее рассматриваются в главе 5.*)

Как add, так и adc могут прибавлять к регистрам и значениям в памяти непосредственные (литеральные) значения. Например, следующий оператор прибавляет 5 к текущему значению bx и записывает результат в bx:

```
add bx, 5                   , bx <- bx + 5
```

Для того чтобы быстрее прибавить 1 к некоторому значению, вместо add *используйте* inc. *Обратите внимание,* что inc требует только одного операнда. Следующие команды увеличивают на 1 значения четырех регистров общего назначения:

```
inc ax                      , ax <- ax + 1
inc bx                      , bx <- bx + 1
inc cx                      , cx <- cx + 1
inc dh                      ; dh <- dh + 1
```

В последнем примере увеличивается значение dh, а dl остается неизменным. В трех других примерах увеличиваются на 1 выбранные 16-битовые регистры. Помните, что вы можете оперировать 8-битовыми половинами регистров общего назначения, не оказывая влияния на другие половины.

Команды вычитания

По своей форме вычитание в языке ассемблера аналогично сложению. Команда sub вычитает значения байтов или слов. Команда sbb делает то же самое, но учитывает возможный заем при предыдущем вычитании многобайтовых или многословных значений. *Следующий пример показывает, как вычесть* bx *из* ax *и записать результат в* ax:

```
sum ax, bx                  , ax <- ax - bx
```

Как и в случае с add и adc, вы можете вычитать два регистра или регистр и значение, записанное в памяти. Вы можете вычитать также непосредственные значения. Прочитав комментарии справа в каждой строке, вы должны разобраться в следующих примерах:

```
sub cx, 5                   , cx <- cx - 5
sub dx, [score]             , dx <- dx - [score]
sub [answer], 3             ; [answer] <- [answer] - 3
sub ax, 1                   , ax <- ax - 1
```

Последний из этих примеров можно заменить более быстрой командой dec, которая вычитает 1 из регистра или значения в памяти. Как показывают следующие примеры, вы можете уменьшать на 1 значения, представленные байтом или словом:

```
dec ax                      ; ax <- ax - 1
dec dl                      ; dl <- dl - 1
dec si                      ; si <- si - 1
dec [balance]               ; [balance] <- [balance] - 1
```

Пример программы на сложение и вычитание

Программа 4.3 представляет работу четырех команд — add, sub, inc и dec. Выполните ассемблирование, компоновку и запустите программу на исполнение под управлением Turbo Debugger, используя следующие команды:

```
tasm /zi addsub
tlink /v addsub
td addsub
```

Программа 4.3. ADDSUB.ASM

```
1:  %TITLE "Использование команд ADD, SUB, INC, DEC -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:          DATASEG
9:
10. exCode  DB      0
11. count   DW      1
12:
13          CODESEG
14:
15: Start:
16:         mov     ax, @data       ; Установка в DS адреса
17:         mov     ds, ax          ; сегмента данных
18.
19          mov     ax, 4
20.         mov     bx, 2
21:         add     ax, bx          ; ax <- ax + bx
22.
23:         mov     cx, 8
24.         add     cx, [count]     ; cx <- cx + [count]
25.
26.         add     [count], cx     ; [count] <- cx + [count]
27.
28.         inc     [count]         ; [count] <- [count] + 1
29.         dec     [count]         ; [count] <- [count] - 1
30:         inc     ax              ; ax <- ax + 1
31:         dec     cx              ; cx <- cx + 1
32:
33: Exit:
34:         mov     ah, 04Ch        ; Функция DOS : Выход из программы
35          mov     al, [exCode]    ; Возврат значения кода ошибки
36.         int     21h             ; Вызов DOS. Останов программы
37.
38.         END     Start           ; Конец программы / точка входа
```

Исполнение демонстрационной программы ADDSUB

Нажмите <Alt+V+C> и <F5> для просмотра окна CPU. Выполняя программу пошагово с помощью нажатия <F8>, наблюдайте за изменением значений регистров. Старайтесь предсказывать значения в регистрах и памяти перед исполнением соответствующих команд.

Строки 19–21 показывают, как сложить значения регистров ax и bx и записать результат в регистр ax. Попробуйте изменить начальные значения (4 и 2) и перезапустить программу. В строках 23–26 складываются значения регистра cx и переменной [count]. Для экспериментирования с командой sub сделайте резервную копию ADDSUB.ASM, замените все add на sub, снова выполните ассемблирование, компоновку и запустите программу под Turbo Debugger.

В строках 28–31 показано, как с помощью команд inc и dec увеличиваются и уменьшаются на единицу значения переменных и регистров. Наблюдайте в верхней части окна CPU за изменением значений в памяти. Вы должны увидеть там значение, записанное в [count]. К сожалению, после выполнения строки 29 это значение исчезает (так как следующая команда не ссылается на адрес count). Ниже описывается метод для более простого просмотра переменных.

Замечание

Для выхода из Turbo Debugger нажмите <Alt+X>.

Наилучший способ просмотра переменных

В Turbo Debugger имеется специальное "окно просмотра" для наблюдения за переменными. При выполнении команд, изменяющих значения в памяти, в окне просмотра также изменяются находящиеся там значения. Это упрощает наблюдение за результатами выполнения команд языка ассемблера, работающими с переменными. Загрузите программу 4.3 с помощью команды td addsub, но не открывайте окно CPU.

Выполните следующие шаги для инспектирования переменной count (строка 11).

1. Нажмите <Ctrl+F7>, наберите count и нажмите <Enter>. Turbo Debugger найдет в памяти переменную count и покажет ее начальное значение в нижней части экрана в окне просмотра.
2. Нажимайте <F8>, пока не окажитесь на строке 26 (add [count], cx). Снова нажмите <F8> и наблюдайте за изменением в окне просмотра значения переменной count.

Замечание

При видимом окне CPU нельзя таким способом просматривать переменные, но для показа окна просмотра вы можете несколько раз нажать <F6> или <Alt+F2>.

При выполнении других программ из этой книги можно также добавлять имена переменных в окно просмотра. Кроме этого, в Turbo Debugger существуют и другие способы просмотра памяти — например, в нижнем левом углу окна CPU показываются последовательные байты, начиная с любого местоположения. Однако проще использовать окно просмотра, в котором переменные показываются с именами. Более того, без повторного ассемблирования программы вы можете изменять значения переменных. Для этого нажмите <Ctrl+F2> для перезагрузки ADDSUB (или запустите Turbo Debugger командой td addsub) и выполните следующие действия.

1. Нажимайте <F6>, пока окно просмотра не окажется в двойной рамке, отражая свое активное состояние. Наберите count и нажмите <Enter>. Это другой способ помещения имени переменной в окно просмотра. (Вы можете пропустить этот шаг, если count уже находится в окне просмотра.)
2. Нажмите <Ctrl+C> (команда изменения значения в окне просмотра) и введите для count новое значение. Вместо начального значения count (1, как показано в строке 11) программа начинает работать с определенным вами новым значением.
3. С помощью <F8> выполняйте программу пошагово. Команды при этом используют новое значение count. Нажмите <Ctrl+F2> для перезагрузки программы, активизируйте с помощью <F6> окно просмотра и введите новое значение для count. Повторяйте эти действия, пока до конца не разберетесь с данной возможностью.

Эти команды Turbo Debugger сохраняют ваше время, позволяя без повторного ассемблирования кода изменять значения переменных. При изменении значений переменных вы можете вводить новые числа в шестнадцатеричном, десятичном или двоичном представлении. Во всех случаях первый символ числа должен быть десятичной цифрой. Последний символ для десятичных чисел — d, шестнадцатеричных — h, двоичных — b. По умолчанию число является шестнадцатеричным. Например, вы можете ввести в окно просмотра следующие значения:

```
100     шестнадцатеричное (256 в десятичном представлении)
0ffh    шестнадцатеричное (255 в десятичном представлении)
256d    десятичное
1001b   двоичное (9 в десятичном представлении)
Ffh     ошибочное - первый символ должен быть 0-9
```

Странные вычитания

При просмотре табл. 4.3 вам может показаться, что команды neg и cmp находятся не на своем месте. Neg изменяет знак двоичного значения. Cmp сравнивает две величины. Почему эти команды расположены вместе с командами вычитания?

В случае с neg процессор 8086 вычитает соответствующее значение из 0. Это значение может быть записано в регистре или памяти. Как вы помните, при вычитании значения из 0 получается значение в дополнительном коде или, другими словами, двоичное дополнение этого значения, что эквивалентно изменению всех нулей на единицы, единиц — на нули и прибавлению 1. В языке 8086 ассемблера для этого проще использовать команду neg. Ниже приводятся два примера:

```
neg  ax             ; Получение двоичного дополнения ax
neg  [value]        ; Получение двоичного дополнения [value]
```

Связь между cmp и вычитанием не настолько очевидна — для этого вам необходимо понять, что большинство цифровых процессоров выполняют сравнение путем вычитания одного значения из другого, проверяя затем результат. После выполнения таким способом операции сравнения устанавливаются биты различных флагов, описывая

тем самым результат — например, является ли он нулевым, отрицательным или положительным. Cmp выполняет вычитание аналогично команде sub, однако сохраняет при этом только значения флагов, которые могут проверяться другими командами. (Это станет понятно при рассмотрении команд управления потоком далее в этой главе.) Сейчас запомните, что cmp аналогична команде sub, однако вместо сохранения результата вычитания только изменяет значения флагов.

Умножение и деление беззнаковых значений

Для правильного выполнения умножения и деления требуется повышенное внимание. Вы должны быть уверены, что помещаете значения в подходящие регистры. После операции вы должны очень аккуратно извлекать ответ. Лучший способ хорошо разобраться в этом —запустить программу в Turbo Debugger и наблюдать за работой команд mul, imul, div и idiv. Выполните ассемблирование и компоновку программы 4.4 и загрузите программу в Turbo Debugger с помощью команд:

```
tasm /zi muldiv
tlink /v muldiv
td muldiv
```

Программа 4.4. MULTDIV.ASM

```
1:  %TITLE "Использование команд MUL, DIV, IMUL, IDIV -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL   small
6:          STACK   256
7:
8:          DATASEG
9:
10: exCode  DB      0
11: opByte  DB      8
12: opWord  DW      100
13: sourceByte      DB      64
14: sourceWord      DW      4000
15:
16:         CODESEG
17:
18: Start:
19:         mov ax, @data           ; Установить в DS адрес
20:         mov ds, ax              ; сегмента данных
21:
22:         mov al, [opByte]
23:         mul [sourceByte]        ; ax <- al * [sourceByte]
24:
25:         mov ax, [opWord]
26:         mul [sourceWord]        ; ax,dx <- ax * [sourceWord]
27:
28:         mov ax, [opWord]
29:         mul ax                  ; ax,dx <- ax * ax
30:
31:         mov ax, [opWord]
32:         div [sourceByte]        ; al <- ax div [sourceByte]
33:
34:         mov ax, [opWord]
35:         mov dx, [opWord]
36:         div [sourceWord]        ; ax <- ax,dx div [sourceWord]
37:
38: Exit:
39:         mov ah, 04Ch            ; Функция DOS: Выход из программы
40:         mov al, [exCode]        ; Возврат значения кода ошибки
41:         int 21h                 ; Вызов DOS. Останов программы
42:
43:         END Start               ; Конец программы / точка входа
```

Выполнение демонстрационной программы MULDIV

Кроме exitCode, в строках 11–14 программы MULDIV описаны четыре тестовые переменные. Поместите эти переменные в окно просмотра Turbo Debugger. (Быстрее всего — нажать <F6> и набрать имена переменных.) Затем, в зависимости от того, что вы предпочитаете, откройте либо окно регистров, либо окно CPU. С помощью <F8> выполните пошагово каждую команду. Для повторного перезапуска программы нажмите <Ctrl+F2>. Поэкспериментируйте с различными значениями, принимая во внимание следующие нюансы.

1. В строках 22–23 перемножаются два беззнаковых байта. Один из байтов должен находиться в регистре al. Другой может располагаться либо в памяти, либо, как в этом примере, в другом 8-битовом регистре. Результат умножения записывается в 16-битовый регистр ax. Переполнение при этом невозможно, так как 255*255 равно 65025, что заведомо помещается в 16-битовое слово. Убедитесь в этом, изменив opByte и sourseByte на 0ffh и снова запустив программу на исполнение.
2. Аналогичные действия выполняются в строках 25–26, но здесь перемножаются два 16-битовых слова. Результат, который может достигать длины в 32 бит, содержится в двух регистрах, dx и ax, при этом в dx находится старшая значащая часть результата, в ax — младшая. Так же как и в случае с перемножением байтов, переполнение здесь невозможно.
3. В строках 28–29 значение регистра возводится в квадрат путем перемножения ax на самого себя. Таким же образом вы можете возвести в квадрат 8-битовое значение, умножая al на самого себя.
4. Строки 31–32 демонстрируют деление беззнаковых значений. Содержимое операнда-источника команды div делится на 16-битовый делитель, содержащийся в регистре ax. Частное помещается в регистр al, остаток — в ah.
5. Аналогичное деление выполняется в строках 34–36, но в этом случае 32-битовое значение, содержащееся в регистрах dx и ax делится на 16-битовое значение sourceWord. В регистре dx содержится старшее значащее слово первоначального значения, в ax — младшее. После деления частное записывается в ax, остаток — в dx.

Замечание

Экспериментируя с новыми значениями, не пытайтесь делить на 0. При этом процессор будет генерировать сигнал, вызываемый прерыванием „деление на 0" (об этом позже в главе 10), останавливая программу. На самом деле это достаточно условное название, так как это прерывание может возникать в любом случае, когда результат деления не помещается в заданный операнд назначения. Например, прерывание „деление на 0" возникает в строках 31–32 при opword = 0F000h и sourcebyte = 1, поскольку 0F000h больше, чем максимальное значение, которое может выражаться одним байтом. Если такое сообщение возникает в Turbo Debugger, попробуйте перезапустить программу нажатием <Ctrl+F2> либо выйдите из отладчика и снова повторите процесс загрузки.

Как вы можете убедиться, беззнаковое умножение и деление в языке 8086 ассемблера — не очень дружественные операции. Вы должны использовать только определенные регистры и иметь в виду, что 32-битовый результат и операнды записываются в два регистра — dx и ax.

Операнд-источник для команд mul и div (см. строки 23, 26, 29, 32 и 36) может располагаться в памяти, как это делается в большинстве примеров, либо в любом регистре общего назначения. Так как размер операнда-источника определяет размер результата, то случайное перемножение вместо байтовых переменных, переменных длиной в слово (как в строке 26) вызовет изменение регистра dx.

Умножение и деление знаковых значений

Команды знакового умножения (imul) и деления (idiv) работают аналогично и используют те же регистры, что и mul и div. (Буква i в символическом имени используется для целых чисел, показывая, что допускаются как положительные, так и отрицательные значения.) Единственное отличие состоит в допустимых диапазонах значений:

- Знаковый байт изменяется от –128 до +127
- Знаковое слово изменяется от –32768 до 32767

Попробуйте в программе 4.4 заменить mul на imul и div на idiv. Задайте новые отрицательные и положительные тестовые значения, либо отредактировав строки 11–14, либо набрав их в окне просмотра Turbo Debugger. Как вы увидите, использование знакового умножения и деления требует определенного внимания. Вам должны помочь следующие замечания:

- Помните, что отрицательный результат представляется в двоичном дополнительном коде.
- Любой остаток (ah — для 8-битового деления и dx — для деления слов) имеет тот же знак, что и частное.
- Если вы попытаетесь выполнить деление на 0 или если результат от деления не будет помещаться в заданный операнд-назначение, то будет генерироваться прерывание деления на 0, останавливающее программу.

Преобразования байтов, слов и двойных слов

При использовании знаковых двоичных значений часто необходимо преобразовывать 8-битовые значения байтов в 16-битовые слова, в частности при подготовке к умножению или делению. Значение может быть отрицательным числом, представленным в дополнительном коде, поэтому выполнять такое преобразование достаточно сложно, поскольку нужно аккуратно сохранять как первоначальное значение, так и его знак. Для упрощения используйте команды cbw (преобразовать байт в слово) и cwd (преобразовать слово в двойное слово). Чтобы проследить за работой этих команд, замените в программе 4.4 строки 22–36 на приведенные ниже. Выполните ассемблирование и запустите программу под управлением Turbo Debugger, экспериментируя с разными значениями для sourceByte и sourceWord:

```
mov  al, [sourceByte]        ; Загружает исходный байт в al
cbw                          ; Размножает знаковый бит по всему ax
mov  ax, [sourceWord]        ; Загружает исходное слово в ax
cwd                          ; Дублирует знаковый бит ax через dx
```

Попробуйте установить sourceByte в −3 (в десятичной системе) и выполнить первые две из этих команд. До выполнения команды cbw в регистре al содержалось шестнадцатеричное значение FD. После значение в ax становится равным FFFD — тому же значению −3, но выраженному с помощью не восьми, а шестнадцати битов. Команда cbw *расширяет* 8-битовое значение (включая знак) до 16-битового. Аналогично cwd расширяет 16-битовое значение до 32-битового двойного слова. Эти команды выполняют одинаковые действия, за исключением количества обрабатываемых битов

Используя эти команды, нужно придерживаться нескольких ограничений. Исходное значение для cbw должно содержаться в регистре al. 16-битовый результат всегда помещается в ax. Начальное значение для cwd должно храниться в регистре ax. 32-битовый результат всегда помещается в dx и ax. Обычно вы будете использовать cbw и cwd вместе с командами imul и idiv при необходимости деления или умножения байтовой величины на слово. Конечно, можно использовать эти команды и для других целей.

Логические команды

В табл. 4.4 представлены логические команды процессора 8086, организованные в виде двух разделов: логические команды и команды сдвига/ротации (циклического сдвига). Логические команды позволяют создавать выражения, объединяющие байты и слова с AND, OR и с другими логическими операторами. Команды сдвига/ротации выполняют сдвиг и ротацию в байтах и словах. Эти понятия вводились в главе 3.

Простейшая логическая команда not изменяет в байтах и словах значения битов с нулей на единицы и с единиц на нули. (Добавление к этому результату 1 образует значение в дополнительном коде, хотя для этих целей проще использовать команду neg.) Одно из применений оператора not состоит в переключении между истинными и ложными значениями. Если ложь представляется нулевым значением, а ненулевое значение означает истину, тогда следующие команды переводят регистр dx из состояния "истина" в состояние "ложь" и затем снова возвращают в состояние "истина":

```
mov  dh, -1          ; Присвоить dh истинное значение (не нуль)
not  dh              ; Присвоить dh значение "не истина", т.е  ложь
not  dh              ; Присвоить dh значение "не ложь", т.е. истина
```

Таблица 4 4 **Логические команды 8086**

Мнемокод/Операнды	Описание
Логические команды	
and назначение, источник	Логическое И
not назначение	Логическое НЕ (отрицание)
or назначение, источник	Логическое ИЛИ
test назначение, источник	Проверка бита
xor назначение, источник	Логическое исключающее ИЛИ

Окончание таблицы 4.4

Мнемокод/Операнды	Описание
Команды сдвига	
rcl назначение, количество	Циклический сдвиг влево через перенос
rcr назначение, количество	Циклический сдвиг вправо через перенос
rol назначение, количество	Циклический сдвиг влево
ror назначение, количество	Циклический сдвиг вправо
sar назначение, количество	Арифметический сдвиг вправо
sal назначение, количество	Арифметический сдвиг влево
shl назначение, количество	Сдвиг влево
shr назначение, количество	Сдвиг вправо

Замечание

Помните, что neg вычитает значение из 0, а не переключает значения битов, как not — это принципиально разные операции. Будьте внимательны, чтобы не перепутать эти команды, так как это всегда приводит к труднообнаружимым ошибкам.

Логические выражения

В главе 3 объясняются основы логических операций AND, OR и XOR на двоичных значениях. Эти же логические действия выполняются одноименными командами процессора 8086, оперирующими значениями байтов и слов в соответствии с правилами таблицы истинности из табл. 3.2. В программе 4.5 показана работа этих команд на языке ассемблера. Выполните ассемблирование, компоновку и запустите программу в Turbo Debugger, используя следующие команды:

```
tasm /zi andorxor
tlink /v andorxor
td andorxor
```

Программа 4.5. ANDORXOR.ASM

```
1:  %TITLE "Использование команд AND, OR, XOR -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL   small
6:          STACK   256
7:
8:          DATASEG
9:
10: exCode  DB      0
11: sourceWord      DW      0ABh    ; 16-битовый источник
12: wordMask        DW      0CFh    ; 16-битовая маска
13:
14:         CODESEG
15:
16: Start:
17:         mov     ax, @data       ; Установить в DS адрес
18:         mov     ds, ax          ; сегмента данных
19:
20:         mov     ax, [sourceWord]; Установить ax, bx, cx и dx
21:         mov     bx, ax          ; в [sourceWord]
22:         mov     cx, ax
23:         mov     dx, ax
24:
25:         and     ax, [wordMask]  ; ax <- ax AND маска
26:
27:         or      bx, [wordMask]  ; bx <- bx OR маска
28:
29:         xor     cx, [wordMask]  ; cx <- cx XOR маска
30:
31:         xor     dx, dx          ; dx <- 0000
```

```
32:
33:  Exit:
34:          mov     ah, 04Ch        ; Функция DOS : Выход из программы
35:          mov     al, [exCode]    ; Возврат значения кода ошибки
36:          int     21h             ; Вызов DOS. Останов программы
37:
38:          END     Start           ; Конец программы / точка входа
```

Выполнение демонстрационной программы ANDORXOR

Чтобы увидеть команды and, or и xor в действии, загрузите ассемблированную программу ANDORXOR в Turbo Debugger и выполните следующие действия.

1. Откройте окно CPU Turbo Debugger (<Alt+V+C>) и распахните его на весь экран (<F5>).
2. Для упрощения ввода новых тестовых значений поместите переменные sorceWord и wordMask в окно просмотра (<Ctrl+F7>). При необходимости нажмите <F6>, чтобы активизировать окно просмотра.
3. Нажимайте <F8> для пошагового исполнения программы, остановившись после выполнения команды xor в строке 31. Постарайтесь предсказать результаты выполнения команд and, or и xor в строках 25–29, после чего сравните ваш прогноз со значениями в регистре ax — для операции and, bx — для or и cx — для xor.
4. Для использования новых тестовых значений нажмите <Ctrl+F2>, перезапустив программу. Затем установите в активном окне просмотра прямоугольный курсор на переменную, которую вы хотите изменить, и нажмите <Ctrl+C>. Введите новое значение и нажмите <Enter>. Затем повторите все действия, начиная с шага 3.

Команда xor в строке 31 программы 4.5 устанавливает регистр dx в 0, что является часто используемым приемом в программировании 8086. Чтобы убедиться, что в этой строке всегда в результате получается 0, задавайте различные тестовые значения в регистре dx.

Проверка 0001 0010 0011

В результате выполнения операции AND над двумя битами получается 1 только тогда, когда оба бита равны 1, поэтому команда and часто используется для проверки, равен ли в байте или слове один или более битов 1. Например, для того чтобы определить, установлен ли второй бит, вы можете использовать маску числа 4:

```
0011 0111 (Тестируемое значение)
0000 0100 (AND-маска)
-------------
0000 0100 (Результат)
```

Если в результате получается 0, значит, второй бит в начальном значении равен 0. Если, как в этом примере, результат не равен 0, значит, второй бит в начальном значении равен 1. К сожалению, команда and изменяет начальное значение. Для того чтобы сохранить тестируемое значение (это удобно при выполнении последовательной проверки битов без перезагрузки регистра), используйте вместо and команду test:

```
mov ah, [testValue]             ; Загрузить [testValue] в ah
test ah, 04h                    ; Проверить бит 2

;------------Выполнить действия, соответствующие установке бита 2

mov dh, 80h                     ; Загрузить маску в dh
test ah, dh                     ; Проверить, установлен ли маскируемый бит

;------------Выполнить действия, соответствующие установке бита 7

test ah, [testBit]              ; Проверить бит с различными масками

;------------Выполнить действия, соответствующие проверяемым битам
```

Как показывают эти примеры, вы можете проверять литеральными значениями, 04h или 80h, значения в регистрах либо в памяти. Команда test выполняет на операндах операцию логического И, но не влияет при этом на результат, оставляя неизменным значение операнда-назначения, но устанавливая те же флаги, что и команда and. После выполнения команды test вы можете использовать команды переходов (описываемые далее), выполняя определенные действия, зависящие от результатов проверки. Обратите внимание на сходство между операторами test и cmp, который выполняет вычитание, не изменяя при этом результирующий операнд. Test выполняет команду and, но не оказывает влияния на результирующий операнд.

Сдвиги битов

В наборе команд процессора 8086 имеется несколько команд сдвига-ротации Как показано в табл 4 4, существуют команды для сдвига битов влево и вправо и ротации через флаг переноса cf Команды делятся на четыре подгруппы·

- Простые сдвиги (shl, shr)
- Простые ротации (rol, ror)
- Ротации через cf (rcl, rcr)
- Арифметические сдвиги (sal, sar)

Эти группы применяют для сдвига вправо и влево битов в байтах и словах различные правила Несмотря на тонкие отличия, команды используют одинаковое количество однотипных операндов Научившись применять одну команду, вы сможете пользоваться и остальными Для демонстрации, воспользуемся наиболее общей командой сдвига shl В ней определяется регистр или значение в памяти и счетчик сдвигов n

```
shl  ax  n                    , Сдвиг ax влево на n=1 бит
```

Это достаточно странно, но n должно равняться 1, в противном случае вы получите сообщение об ошибке (В более новых моделях процессоров типа 80386 в качестве n может использоваться беззнаковая 8-битовая константа) Единственная допустимая форма оператора сдвига в языке ассемблера 8086 имеет вид

```
shl  ax  1                    , Сдвиг ax влево на один бит
```

Сдвиг за раз более чем на один бит на процессоре 8086 выполняется в два этапа сначала в регистр cl загружается счетчик сдвигов, затем cl используется в качестве второго операнда команды сдвига

```
mov  cl 5                     , Загрузить счетчик сдвигов в cl
shl  ax  cl                     Сдвиг ax влево на cl бит
```

Для этих целей вы должны использовать только cl — никакой другой регистр не может быть вторым операндом Вы можете также выполнять сдвиги в значениях, расположенных в памяти и в 8-битовых половинах регистров Например

```
mov  cl  2                    , Загрузить счетчик сдвигов в cl
shl  bh  cl                   , Сдвиг bh влево на cl бит
shl  [seconds]  1             , Сдвиг [seconds] влево на один бит
shl  [minutes]  cl            , Сдвиг [minutes] влево на cl бит
```

Несколько экспериментов и диаграмм прояснят различия между многочисленными командами сдвигов Используйте следующие команды для ассемблирования и исполнения программы 4 6 в Turbo Debugger

```
tasm /zi shift
tlink/v shift
td shift
```

Программа 4 6 SHIFT ASM

```
1    %TITLE  Использование команд сдвига -- by Tom Swan
2
3            IDEAL
4
5            MODEL    small
6            STACK    256
7
8            DATASEG
9
10   exCode  DB       0
11   operand DB       0AAh
12
13           CODESEG
14
15   Start
16           mov      ax  @data        , Установка в DS адреса
17           mov      ds  ax           , сегмента данных
18
19           shl      [operand]  1     , Сдвиг влево
```

```
20.             shr     [operand], 1    ; Сдвиг вправо
21:             rol     [operand], 1    ; Ротация влево
22:             ror     [operand], 1    ; Ротация вправо
23              rcl     [operand], 1    ; Ротация влево через перенос
24:             rcr     [operand], 1    ; Ротация вправо через перенос
25:             sal     [operand], 1    ; Арифметический сдвиг влево
26:             sar     [operand], 1    ; Арифметический сдвиг вправо
27:
28:  Exit:
29.             mov     ah, 04Ch        ; Функция DOS : выход из программы
30.             mov     al, [exCode]    ; Возврат значения кода ошибки
31:             int     21h             ; Вызов DOS. Останов программы
32:
33:             END     Start           ; Конец программы / точка входа
```

Выполнение демонстрационной программы SHIFT

Выполнение предыдущих шагов подразумевает, что вы выполнили ассемблирование программы SHIFT.ASM и загрузили ее в Turbo Debugger. Следующие действия помогут прояснить вам тонкости в командах сдвига на процессоре 8086.

1. В программе 4.6 исполняется каждая из семи команд сдвига 8086, представленных в табл. 4.4. По причинам, которые я объясню далее, названия shl и sal обозначают одну и ту же команду, поэтому, несмотря на восемь мнемонических названий, в действительности существует только семь команд сдвига.
2. На рис. 4.5 показано, как работают команды простого сдвига shl и shr. Выполните пошагово (<F8>) строки 19–20. Каждый бит в операнде назначения сдвигается на один или cl бит влево или вправо. При выполнении команды shl седьмой бит (старший значащий разряд) перемещается во флаг переноса, а правый освобождающийся бит заполняется нулевым значением. При выполнении команды shr нулевой бит (младший значащий разряд) перемещается во флаг переноса, а левый освобождающийся бит заполняется нулевым значением.

Замечание

Несмотря на то что на рис. 4.5–4.8 показаны 8-битовые байты, все команды сдвигов также могут работать и на 16-битовых значениях. По этой причине на диаграммах не указаны номера битов.

3. На рис. 4.6 показаны отличия между командами ротаций rol и ror и простыми сдвигами. Они не обнуляют биты справа или слева, а перемещают вместо этого значение младшего или старшего значащего разряда на противоположный конец. Остальные биты сдвигаются в указанном направлении. При выполнении команды rol начальный старший значащий разряд перемещается на место младшего значащего разряда. При выполнении команды ror младший значащий разряд становится на место старшего значащего разряда. Так же как и при выполнении команд shl и shr, эти биты заносятся во флаг переноса. Для экспериментирования с этими командами выполните пошагово строки 21–22.
4. Рис. 4.7 представляет команды ротации через перенос — rcl и rcr. Для обеих этих команд однобитовый флаг переноса служит как бы расширением регистра или памяти при выполнении ротации. При выполнении команды rcl старший значащий разряд сдвигается во флаг переноса, а предыдущее значение флага переноса становится на место младшего значащего разряда. При выполнении команды rcr младший значащий разряд сдвигается во флаг переноса, а старое значение флага переноса помещается на место старшего значащего разряда. Остальные биты сдвигаются в указанном направлении. Для экспериментирования с командами rcl и rcr выполните пошагово строки 23–24.
5. На рис. 4.8 представлена последняя, достаточно странная команда сдвига — sar. Команда sar работает аналогично shr, за исключением того, что старший значащий разряд сохраняет свое первоначальное значение. Кроме того, старший значащий разряд копируется в правый от себя бит. Проще это увидеть на нескольких примерах двоичных значений:

```
10001000
11000100
11100010
11110001
11111000
```

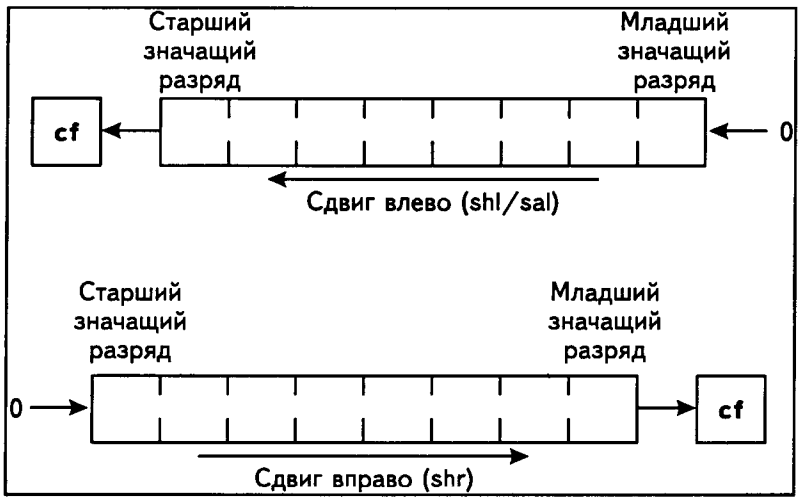


Рис 4.5. Команды простого сдвига *shl/sal* и *shr*

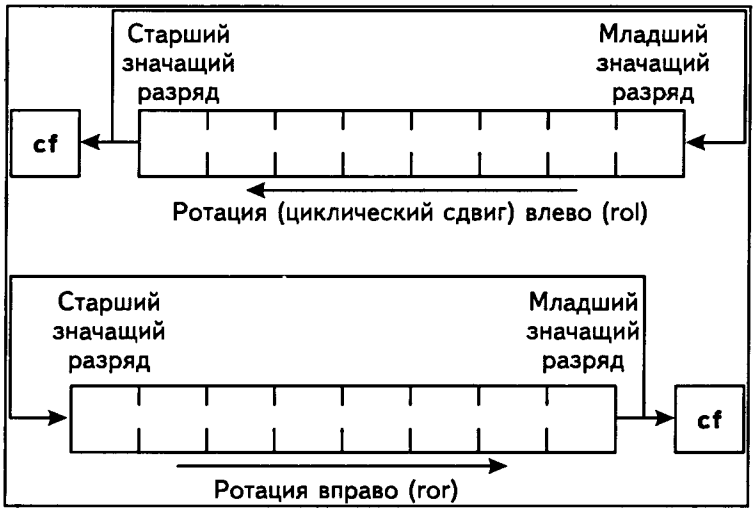


Рис. 4.6. Команды ротации *rol* и *ror*

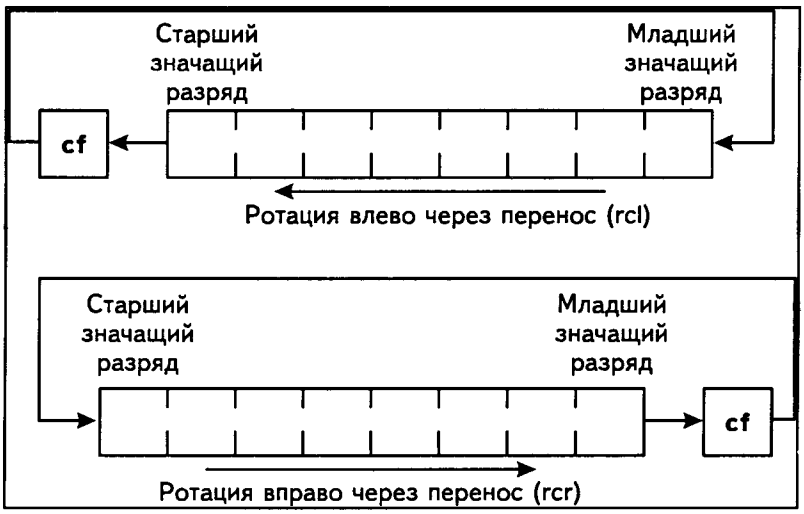


Рис. 4.7. Команды ротации через перенос *rcl* и *rcr*

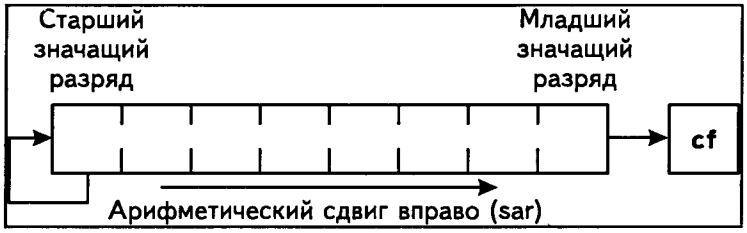


Рис. 4.8. Команда *sar*

Начиная со второго значения, каждая следующая строка является результатом применения оператора sar к предыдущей строке. Как и в команде shr, биты сдвигаются вправо, но при этом старший значащий разряд сохраняет свое предыдущее значение и распространяется вправо. Поэтому команда sar полезна при выполнении деления отрицательных чисел, представленных в форме дополнительного кода, на степени числа 2. Например, при последовательном применении команды sar к шестнадцатеричным значениям получится следующий ряд:

```
8000        -32768
C000        -16384
E000         -8192
F000         -4096
F800         -2048


FFFE            -2
FFFF            -1
```

Применение команды sar к шестнадцатеричному значению FFFF не дает никакого результата, в отличие от команды idiv, которая при делении –1 на 2 дает, как это и нужно ожидать, в итоге 0.

В отличие от других парных команд сдвига, которые для каждого левого сдвига имеют аналогичный правый сдвиг, у команды sar отсутствует соответствующий левый аналог. Для устранения этого недостатка команде shl дается второе мнемоническое название — sal. Причина, по которой арифметический сдвиг влево не отличается от логического сдвига влево, становится очевидной при рассмотрении в обратном порядке предыдущей шестнадцатеричной последовательности. Если рассматривать ее снизу вверх, то эти же значения получаются и при использовании оператора shl. (Если вам сложно это представить, попробуйте преобразовать шестнадцатеричные значения в двоичные) Функционально команда sar заменяется командами shl/sal, которые могут умножать отрицательные значения, представленные в виде дополнительного кода на степени числа 2, и поэтому нет необходимости в отдельной команде.

Замечание

При просмотре команд sal в Turbo Debugger в окне CPU они часто трактуются как shl Так получается потому, что отладчик не может знать контекста, в котором вы использовали ту или иную мнемоническую команду, и поэтому показывает наиболее общее имя.

Почему сдвиг?

Существует много причин для использования в программах команд сдвига, хотя две из них явно выделяются:

- Передвижение битов в заданные положения
- Деление и умножение на степени 2

Передвижение битов на заданные позиции и использование логических операторов для размещения результатов сдвига в других значениях — типичная операция в языке ассемблера. Например, предположим, что dh первоначально равен 3, dl — 5 и программе требуется разместить эти два числа в регистре dh, причем 3 должна размещаться, начиная со старшего значащего разряда, а 5 — в младшей значащей части байта. Это можно выполнить следующим образом:

```
mov  dh, 3          ; dh <- 3
mov  dl, 5          ; dl <- 5
mov  cl, 4          ; Загрузить счетчик в cl
shl  dh, cl         ; Сдвиг dh влево на четыре бита
or   dh, dl         ; dh <- dh OR dl
```

Замечание

Если у вас возникли сложности при рассмотрении этого примера, замените строки 19–26 в программе 4.6 этими пятью командами и запустите программу в Turbo Debugger. Наблюдайте за регистром dh по ходу выполнения каждой строки. Команда shl сдвигает dh влево на 4 бита, перемещая 4-битовое значение в верхнюю позицию и заполняя биты справа нулевыми значениями. Затем команда or объединяет сдвинутое значение со значением в dl, размещая 4-битовое значение в 8-битовом байте.

Деление и умножение с помощью сдвига

Очень полезно знать, как, используя только операторы сдвига, можно выполнять умножение и деление на степени числа 2. (Основы этого вы изучали в главе 3.) В большинстве случаев сдвиги выполняются быстрее, чем команды mul, imul, div и idiv, поэтому при соответствующих обстоятельствах вы всегда должны их использовать. Например, для умножения какого-либо значения на 8 (или 2^3) вам необходимо просто сдвинуть это значение три раза влево:

```
mov  ax  6          , ax <- 6
mov  cl  3          , Загрузить счетчик в cl
shl  ax, cl         , ax <- ax * 8
```

Для деления на 16 (2^4) сдвиньте соответствующее значение вправо на 4 бита:

```
mov  cl, 4          , Загрузить счетчик в cl
shr  ax, cl         , ax <- ax / 16
```

Единственной проблемой при умножении таким способом является возможность возникновения переполнения, которое не учитывается в этих примерах. Если значение флага после сдвига влево на один бит равняется 1, значит, результат не помещается в назначенный регистр или область памяти. При сдвиге более чем на 1 определять переполнения становится сложно. Кроме того, при делении теряется остаток — деление 3 на 2 с помощью сдвига на один бит вправо дает в результате 1 и найти остаток не представляется возможным.

Команды управления потоком

В табл. 4.5 представлены команды управления потоком 8086, или команды *переходов*, которые позволяют изменять в программе адрес следующей выполняемой машинной команды. В случае команд управления потоком программа просто с головокружительной скоростью исполнялась бы от начала до конца без остановок, циклов и взаимодействия с другими программами. Имея команды управления потоком, в программе можно принимать решения, проверять флаги и основывать свои действия на результатах предыдущих операций — проверки битов, логических сравнений и арифметических вычислений. Кроме этого, команды управления потоком позволяют при выполнении определенных условий исполнять какие-то фрагменты программы повторно, экономя память с помощью циклического повторения одного и того же фрагмента кода.

Таблица 4 5 **Команды управления потоком 8086**

Мнемокод / Операнды	Описание
Команды безусловного перехода	
call адрес	Вызов процедуры
jmp адрес	Безусловный переход
ret значение	Возврат из процедуры
retn значение	Возврат из близлежащей процедуры
retf значение	Возврат из дальней процедуры
Команды условного перехода	
ja/jnbe короткий адрес	Переход, если выше/не ниже или равно
jae/jnb короткий адрес	Переход, если выше или равно/не ниже
jb/jnae короткий адрес	Переход, если ниже/не выше или равно
jbe/jna короткий адрес	Переход, если ниже или равно/не выше

Окончание таблицы 4.5

Мнемокод / Операнды	Описание
jc короткий адрес	Переход, если был перенос
je/jz короткий адрес	Переход, если равно/нуль
jg/jnle короткий адрес	Переход, если больше/ не меньше или равно
jge/jnl короткий адрес	Переход, если больше или равно/не меньше
jl/jnge короткий адрес	Переход, если меньше/не больше или равно
jle/jng короткий адрес	Переход, если меньше или равно/не больше
jnc короткий адрес	Переход, если нет переноса
jne/jnz короткий адрес	Переход, если не равно/не нуль
jno короткий адрес	Переход, если нет переполнения
jnp/jpo короткий адрес	Переход, если нет паритета/паритет нечетный
jns короткий адрес	Переход, если нет знака
jo короткий адрес	Переход по переполнению
jp/jpe короткий адрес	Переход, если есть паритет или паритет четный
js короткий адрес	Переход, если есть знак
Команды цикла	
jcxz короткий адрес	Переход, если cx равно нулю
loop короткий адрес	Цикл пока cx <> 0
loope/loopz короткий адрес	Цикл, если равно или нуль
loopne/loopnz короткий адрес	Цикл, если не равно/не нуль
Команды прерывания	
int тип прерывания	Прерывание/вызов прерывания
into	Прерывание по переполнению
iret	Возврат из обработки прерывания

Хотя может показаться, что в табл. 4.5 содержится невообразимое количество команд перехода, необходимо запомнить только несколько легко выделяющихся групп. В этой главе рассматриваются первые две из них: условные и безусловные переходы. В следующих главах вводятся циклы и команды управления прерываниями.

Безусловные переходы

Безусловный переход изменяет адрес следующей исполняемой команды. Он работает подобно одностороннему ответвлению со скоростной дороги — попав туда, хотите вы того или нет, вы обязаны дальше продолжать движение. И если процессор выполняет безусловный переход, то следующей обязательно будет выполняться команда, на которую указывает этот переход. При безусловных переходах в регистр ip, а в некоторых случаях также в регистр кодового сегмента cs загружаются значения новых адресов. Вместе cs:ip определяют адрес следующей исполняемой команды. Изменение одного или обоих регистров приводит к изменению адреса следующей исполняемой команды, нарушая обычный ход исполнения программы сверху вниз.

Вызов подпрограмм

Одним из наиболее полезных инструментов в языке ассемблера является *процедура,* или подпрограмма — набор взаимосвязанных команд, обычно выполняющих одну часто встречающуюся операцию. Процедура может выводить на экран строку символов, суммировать наборы значений или инициализировать выходной порт. Некоторые подпрограммы живут замечательной жизнью: играют в шахматы или обеспечивают доступ к удаленному компьютеру. Другие имеют более скромную роль: выводят отдельный символ или считывают код клавиши с клавиатуры.

Некоторые программисты создают длинные подпрограммы, выполняющие сразу большое количество работы, объясняя это тем, что использование многочисленных маленьких подпрограмм может замедлять скорость выполнения программы. Не делайте этого. Объединяя команды в большие подпрограммы, вы получите только незначительное увеличение скорости, но, что более вероятно, это приведет к ошибочной и трудноотлаживаемой программе, размышляя над которой вы можете пожалеть о том, что стали программистом.

Хорошая подпрограмма выполняет одну и только одну работу. Хорошая подпрограмма коротка, насколько это возможно, и велика, насколько это необходимо. Хорошая подпрограмма умещается на одной или двух печатных страницах. Хорошая подпрограмма начинается не с кода, а с комментариев, описывающих ее назначение, результаты, ожидаемые входные данные и используемые регистры. В хорошей подпрограмме можно разобраться, не зная, что делает вся программа. Другими словами, хорошая подпрограмма коротка, изящна и точна.

В тексте программы 4.7 показано, как пишутся подпрограммы в языке ассемблера. Выполните ассемблирование, компоновку и загрузите программу в Turbo Debugger аналогично другим примерам из этой главы, используя команды:

```
tasm /zi subdemo
tlink /v subdemo
td subdemo
```

Программа 4.7. SUBDEMO.ASM

```
1:  %TITLE "Использование подпрограмм -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL   small
6:          STACK   256
7:
8:          DATASEG
9:
10: exCode  DB      0
11:
12:         CODESEG
13:
14: Start:
15:         mov     ax, @data       ; Установить в DS адрес
16:         mov     ds, ax          ; сегмента данных
17:
18:         mov     al, 1           ; Загрузка суммируемых значений в AL-DL
19:         mov     bl, 2
20:         mov     cl, 3
21:         mov     dl, 4
22:         call    AddRegisters    , AX <- AL+BL+CL+DL
23:         call    AddRegisters    ; снова
24:         call    AddRegisters    ; и снова!
25:
26: Exit:
27:         mov     ah, 04Ch        ; Функция DOS : Выход из программы
28:         mov     al, [exCode]    ; Возврат значения кода выхода
29:         int     21h             ; Вызов DOS. Останов программы
30:
31: ;-----------------------------------------------------------------
32: ; AddRegisters Суммирование al, bl, cl и dl
33: ;-----------------------------------------------------------------
34: ; Вход:
35: ; al, bl, cl, dl = Четыре 8-битовых складываемых значения
36: ; Выход:
37: ; ax = al + bl + cl + dl
38: ; Регистры:
39: ; Изменяются ax, bh, ch, dh
40: ;-----------------------------------------------------------------
41: PROC    AddRegisters
42:         xor     ah, ah          ;Установить ah в 0
43:         xor     bh, bh          ; Установить bh в 0
44:         xor     ch, ch          ; Установить ch в 0
45:         xor     dh, dh          ; Установить dh в 0
46:         add     ax, bx          ; AX <- AX + BX
47:         adc     ax, cx          ; AX <- AX + CX + CF
48:         adc     ax, dx          ; AX <- AX + DX + CF
49:         ret                     ; Возврат в вызывающую программу
50: ENDP    AddRegisters
51:
52:         END     Start           ; Конец программы / точка входа
```

Выполнение демонстрационной программы SUBDEMO

Главная часть программы SUBDEMO расположена в строках 14–29. Подпрограмма находится в строках 31–50. В тексте программы встречается несколько новых понятий.

- Комментарии в строках 31–40 описывают имя подпрограммы, ее назначение, входные и выходные данные и используемые регистры. Пунктирная черта не является обязательной, а служит только для того, чтобы выделить начало подпрограммы в объемном листинге. Для многих программистов индивидуальный стиль заголовка подпрограммы — это их фирменный знак. Вы свободны в использовании своего формата, но включайте в него хотя бы информацию, указанную здесь.
- Директивы PROC и ENDP (строки 41, 50) отмечают начало и окончание подпрограммы.
- Команда ret (строка 49) должна быть в любой подпрограмме, но не обязательно, как в этом примере, в последней строке.

Директивы PROC и ENDP не являются обязательными, но я настойчиво рекомендую использовать их для выделения начала и конца каждой вашей подпрограммы. PROC и ENDP являются директивами Turbo Assembler, а не командами 8086. После директивы PROC следует имя подпрограммы, помечающее адрес первой команды, в нашем случае — строку 42. Директива ENDP завершает подпрограмму, за ней может следовать имя той же метки, что и в предыдущей директиве PROC. Включение имени подсказывает, какая процедура заканчивается, но, при желании, вы можете его не указывать. В строке 22 главная программа *вызывает* подпрограмму с помощью команды call с меткой AddRegisters. При исполнении команды call выполняются два важных действия:

- *Адрес возврата* на следующую после call команду проталкивается в стек.
- Адрес подпрограммы помещается в регистр ip или, в ряде случаев, в регистровую пару cs:ip.

Перед началом выполнения вызванной подпрограммы процессор 8086 проталкивает в стек адрес следующей за call команды. Этот адрес называется адресом возврата, так как он отмечает место, в которое подпрограмма должна передать управление после своего окончания. В нашем примере первый такой адрес возврата указывает на команду в строке 23, т.е. на другую команду call. После проталкивания этого адреса в стек процессор выполняет безусловный переход на требуемую метку, выполняя команду в строке 42. Затем с этой точки программа продолжает исполняться и выполняет команды подпрограммы.

Причина, по которой адрес возврата проталкивается в стек, проясняется при выполнении в подпрограмме последней команды — ret в строке 49. Подобно call команда ret выполняет два важных действия:

- Адрес возврата выталкивается из стека в регистр ip (или в cs:ip).
- Программа продолжает выполняться, начиная с команды, следующей за вызывающей подпрограмму командой call.

Действие трех команд call из строк 22–24 программы 4.7 показано на рис. 4.9.

При каждом вызове call начинают исполняться команды подпрограммы вплоть до команды ret, которая возвращает управление следующей после call команде. Одна и та же подпрограмма может вызываться из различных мест программы. Чтобы увидеть это на вашем компьютере, загрузите программу 4.7 в Turbo Debugger и выполните следующие действия:

1. Находясь в окне CPU, шесть раз нажмите <F8>, остановившись перед выполнением команды call в строке 22. Обратите внимание, что значения в регистрах al, bl, cl и dl будут переданы на обработку в подпрограмму.
2. Вместо <F8> нажмите для выполнения команды call <F7> — клавишу "трассировки". При этом вы должны увидеть, что командный маркер перейдет на команду xor в строке 42, показывая готовность подпрограммы к выполнению. Вы можете также увидеть, как протолкнулся в стек адрес возврата (в нижнем правом углу экрана).
3. Повторно нажимайте <F7>, пока не дойдете до команды ret в строке 49. После этого снова нажмите <F7>, выполнив ret и, тем самым, вернув управление на следующую после call команду в строке 22.
4. Снова нажмите <F7> для вызова той же подпрограммы. Затем, как и на предыдущем шаге, повторно нажимайте <F7>, остановившись в этот раз после выполнения команды ret.
5. Сейчас командный маркер должен находиться на строке 24, готовый к выполнению последнего call. В этот раз вместо <F7> нажмите <F8> — клавишу, которую вы обычно используете для пошагового исполнения программы. <F8> — клавиша "пошагового исполнения" — выполняет подпрограмму как одно целое, останавливаясь только после возврата из подпрограммы, а не показывая ее внутренние команды. Помните, что для *пошагового исполнения* подпрограммы необходимо на команде call нажимать клавишу <F7>. Для выполнения всей подпрограммы как одного оператора нажимайте <F8>. Клавиша <F8> полезна, когда вы уверены, что подпрограмма работает правильно и вы не хотите терять время на пошаговое исполнение внутренних команд подпрограммы.

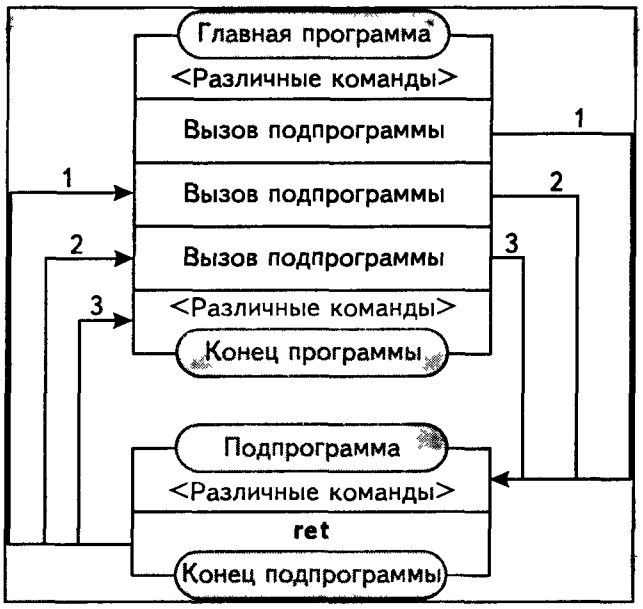


Рис. 4.9. Вызовы и возвраты из подпрограммы

У вас уже достаточно знаний, чтобы разобраться, как работает подпрограмма AddRegisters в программе 4.7. Там нет незнакомых команд, но если вы нуждаетесь в подсказках, то прочитайте соответствующие комментарии. Команды xor в строках 42–45 обнуляют значения в верхних половинах складываемых регистров. Затем выполняется сложение и сложение с переносом четырех значений — al, bl, cl, dl, и сумма помещается в ax.

Дальние и ближние вызовы

Хотя в табл. 4.5 приводятся три команды возврата — ret, retf и retn, на самом деле их только две — retf и retn. Встречая общую мнемоническую команду ret, Turbo Assembler сам решает, какая из двух других команд возврата соответствует используемой модели памяти. Для понимания различий между retf и retn сначала необходимо разобраться в различиях между *внутрисегментными* и *внесегментными* вызовами подпрограмм:

- При внутрисегментном вызове подпрограмма находится в том же кодовом сегменте, что и команда call. Другими словами, при передаче управления на новую точку значение сегментного регистра cs остается неизменным, поэтому для запуска подпрограммы необходимо изменять только значение в ip. Соответственно адрес возврата также представляется одним 16-битовым словом, которое и проталкивается в стек.
- При внесегментном вызове подпрограмма и вызывающий ее оператор call располагаются в различных кодовых сегментах. В этом случае должны изменяться значения в обоих регистрах — cs и ip и в стек проталкивается полный 32-битовый адрес команды, следующей за call.

Поскольку ассемблер знает, является вызываемая подпрограмма ближней (расположенной в том же сегменте) или дальней (расположенной в другом сегменте), в обоих случаях используется одна и та же мнемоническая команда call. Однако существует две мнемонические команды возврата: retn — для *ближних*, внутрисегментных вызовов, и retf — для *дальних*, внесегментных вызовов, позволяя вам выбирать между ближними и дальними подпрограммами, изменяя команду, которая по умолчанию генерируется Turbo Assembler для ret.

Лучший способ не запутаться в этих деталях — позволить Turbo Assembler самому сгенерировать для вас правильный код. Для определения ближней подпрограммы используйте в определении PROC оператор NEAR:

```
PROC SubName NEAR
;-------------Вставить команды подпрограммы
     ret
ENDP SubName
```

Для написания внесегментной подпрограммы измените NEAR на FAR. Тогда Turbo Assembler будет рассматривать ее как дальнюю и заменит команду ret на retf.

Замечание

Использование малой модели памяти, как это делается в большинстве примеров программ в этой книге, подразумевает, что подпрограммы являются ближними (расположенными в том же сегменте, что и их вызовы). Следовательно, при определении подпрограммы необязательно использовать оператор NEAR.

Передача значений в подпрограммы и из них

В тексте программы 4.7 вы можете увидеть, что подпрограмме AddRegisters для хранения складываемых значений необходимы четыре 8-битовых регистра. Подпрограмма возвращает полученную сумму в регистре ax. Передача значений в подпрограмму через регистры является наиболее общим методом получения подпрограммами данных для обработки. Кроме того, существует еще два метода:

- Запись данных в глобальные переменные
- Передача данных через стек

Подпрограммы могут работать непосредственно с переменными, описанными в сегменте данных, например с байтом exCode из строки 10, хотя обычно это не лучший выбор. Изменение глобальных переменных внутри подпрограмм может приводить к осложнениям, связанным с тем, что становится трудно определить, какое значение в какой процедуре было изменено. В сложных программах с сотнями и тысячами подпрограмм, вызывающих одна другую в различных последовательностях, подпрограммы, оказывающие влияние на одни и те же глобальные значения, могут вызывать ошибки очень неприятного типа, называемые *эффектом наведения*. Такие проблемы возникают, если программа (или другая подпрограмма) вызывает подпрограмму, изменяющую глобальное значение, которое в данный момент используется для других целей.

Передача данных через стек позволяет избежать эффекта наведения, особенно в подпрограммах с большим числом параметров. Вы можете изменить программу 4.7, используя такую схему. Перед вызовами подпрограмм (строки 22–24) вместо загрузки регистров al, bl, cl и dl воспользуйтесь следующими командами:

```
mov ax, 1                ; Первый элемент
push ax                  ; Поместить в стек
mov ax,2                 ; Второй элемент
push ax                  ; Поместить в стек
mov ax,3                 ; Третий элемент
push ax                  ; Поместить в стек
mov ax, 4                ; Четвертый элемент
push ax                  ; Поместить в стек
call AddValues
```

Обратите внимание, что сначала вы должны загрузить значение в регистр (в данном случае ax), а затем протолкнуть его в стек, так как непосредственное проталкивание в стек литеральных значений не допускается. Однако следующее действие является неверным:

```
PROC AddValues
     pop dx              ; ???
     pop cx
     pop bx
     pop ax
     :                   ; Команды подпрограммы
     ret
ENDP AddValues
```

Первая команда pop удаляет из стека адрес возврата, помещенный туда командой call, в результате чего подпрограмма, во-первых, суммирует неверные значения, а во-вторых, не может вернуть управление в точку вызова.

Правильное решение состоит в том, чтобы сначала сохранить адрес возврата, вытолкнуть параметры и затем снова вернуть его в стек. Для этого необходим дополнительный регистр:

```
PROC AddValues
     pop si              ; Сохранить адрес возврата в si
     pop dx              ; Извлечь 4 параметра
     pop cx
     pop bx
     pop ax
     push si             ; Вернуть адрес возврата в стек
     :                   ; Команды подпрограммы
     ret
ENDP AddValues
```

Как видите, передача параметров через стек является более сложной процедурой, чем передача данных непосредственно через регистры. Параметры в стеке можно прочитать с использованием метода, применяемого в языках высокого уровня и объясняемого в главах 12 и 13. Специальная форма команды ret может извлекать параметры из стека до выталкивания адреса возврата, несколько упрощая описанный выше метод.

Проталкивать или не проталкивать

В комментарии в строке 39 программы 4.7 говорится, что в подпрограмме изменяются различные флаги и регистры — ax, bh, ch, dh. Если любой из этих регистров или флагов еще будет использоваться в вызывающей программе в своих собственных целях, то тогда такой конфликт надо разрешать. Возможны два решения:

- Сохранить начальные значения регистров до вызова подпрограммы
- Сохранить начальные значения регистров в самой подпрограмме

Спросите шестерых программистов и вы получите шесть мнений о том, какой из этих двух методов сохранения регистров лучший. В первом случае используемые регистры сохраняются до вызова изменяющих их подпрограммы. Например, если в программе 4.7 вызывающая программа использует регистры bh и ch, тогда вызов подпрограммы осуществляется следующим образом:

```
push bx                    Сохранить bx в стеке
push cx                  , Сохранить cx в стеке
call AddRegisters        , Вызов подпрограммы
pop  cx                    Извлечь cx из стека
pop  bx                  , Извлечь bx из стека
```

Вы должны сохранять регистры целиком (ax, bx и т.д.), даже если вам необходимы только их 8-битовые половины (ah, bl и т.п.) Проталкивание регистров в стек до вызова подпрограммы временно сохраняет там их значения, восстанавливая после окончания подпрограммы прежние значения регистров. Не забывайте, что команды pop должны следовать в обратном порядке, чем команды push.

Во втором случае за сохранение и восстановление изменяемых регистров ответственны сами подпрограммы, конечно, кроме регистров, которые используются для возврата значений в вызывающую программу. Для этого необходимо изменить строки 41–49 в AddRegisters:

```
PROC AddRegisters
     pushf                 , Сохранить флаги
     push bx               , Сохранить изменяемые регистры
     push cx
     push dx
                             Команды подпрограммы
     pop dx                , Извлечь регистры
     pop cx
     pop bx
     popf                  , Извлечь флаги
     ret                   , Выход из подпрограммы
ENDP AddRegisters
```

Безусловный jmp работает аналогично оператору call, за исключением того, что в стек не проталкивается адрес возврата. Команда jmp использует один параметр — метку, указывающую на место в программе, в которое будет передано управление. Для примера измените программу 4.7, вставив между строками 21 и 22 команду

```
jmp Exit
```

При пошаговом исполнении программы в Turbo Debugger вы увидите, что команда jmp пропускает выполнение трех операторов call в строках 22–24, непосредственно переходя по метке Exit на команду mov. Это все, что делает jmp. Вы можете всегда использовать эту команду, если вам необходимо перейти с одного места программы в какое-либо другое. Аналогично случаю с оператором call, это место может находиться как в том же кодовом сегменте, так и в другом. Turbo Assembler всегда правильно применяет оператор jmp, выполняя либо внутрисегментный переход (на другое смещение в одном и том же кодовом сегменте, изменяя только регистр ip), либо внесегментный переход (в другой сегмент и смещение, изменяя как cs, так и ip). В основном вы будете использовать jmp для переходов внутри одного кодового сегмента, по меньшей мере при работе с малой моделью памяти.

Замечание

Используйте jmp очень осмотрительно, чтобы избежать создания хорошо известного среди программистов кода типа „спагетти", когда многочисленные команды jmp создают цепочку переходов, которые тянутся подобно пасте из тюбика. Кроме того, вам будет очень сложно разобраться в том, что делает эта программа.

Условные переходы

В табл. 4.5 представлено 18 команд условных переходов, некоторые из которых используют для одной команды по два мнемонических представления, например je/jz и jg/jnle, доводя общее количество команд до 30. Может показаться, что такое чрезмерное количество команд условных переходов нереально для изучения, но, подобно спряжению глаголов, различные формы запоминаются легко, если отделить корень (j для jmp) от окончаний (a, nbe, e, z и т.п.). Как показано в табл. 4.6, каждое из этих окончаний представляет определенное условие. Запомнив их значения, вы легко разберетесь в различных командах условных переходов. В таблице окончания справа имеют противоположный смысл, чем окончания слева. (Два условных перехода, jpe и jpo, не имеют соответствующих противоположных пар.)

Условные переходы требуют целевого адреса — метки, обозначающей место в программе, с которого она продолжит исполняться в случае выполнения заданного условия. Например, сравнивая значения регистров с помощью команды cmp, для передачи управления в заданное место при их равенстве вы должны использовать je (переход, если равно). Чтобы показать это, предположим, что подпрограмма должна возвращать 1 в регистре cx, если ax=bx, или 0, если ax<>bx. Это происходит следующим образом:

```
PROC RegEqual
            mov cx, 1               ; Записать 0001 в cx
            cmp ax, bx              ; Сравнение ax с bx
            je Continue             ; Переход, если ax = bx
            xor cx, cx              ; Иначе, установить cx в 0000
Continue
            ret                     ; Возврат в вызывающую программу
ENDP RegEqual
```

Таблица 4.6 **Окончания команд условных переходов**

Окончание	Значение	Окончание	Значение
a	Выше	na	Не выше
ae	Выше или равно	nae	Не выше или равно
b	Ниже	nb	Не ниже
be	Ниже или равно	nbe	Не ниже или равно
c	Перенос	nc	Нет переноса
e	Равно	ne	Не равно
g	Больше	ng	Не больше
ge	Больше или равно	nge	Не больше или равно
l	Меньше	nl	Не меньше
le	Меньше или равно	nle	Не меньше или равно
o	Переполнение	no	Нет переполнения
p	Паритет	np	Нет паритета
pe	Паритет четный	--	
po	Паритет нечетный	--	
s	Знак	ns	Нет знака
z	Нуль	nz	Не нуль

Сначала cx устанавливается в 1, т.е. значение, означающее равенство ax и bx, хотя в этот момент подпрограмме это еще не известно. Затем команда cmp сравнивает ax и bx. Помните, что cmp выполняет вычитание (ax–bx), не оказывая влияния на результирующий операнд и устанавливая флаг нуля zf в 1 в случае нулевого результата или в 0,

если результат вычитания не равен 0. Условный переход je проверяет флаг нуля, передавая управление на мет Continue в случае выполнения условия, а именно, zf=1, который показывает, что ax равен bx и поэтому сохраняет в cx ранее установленное значение. Если условие не выполняется (zf=0), тогда команда xor устанавливает cx в 0 обоих случаях последней выполняется команда ret, возвращающая управление следующей после call команде.

Переход вниз по программе, как в этом примере, пропускающий инициализацию регистра или, возможно, выч другой подпрограммы, вероятно, является наиболее распространенным использованием условного перехода Н тем не менее вы можете также использовать переходы для создания в программе циклов. Например, в следующе фрагменте значение ax каждый раз увеличивается на 1, вызывается подпрограмма Print (не показанная здесь), тех пор пока ax не станет равно 10:

```
        xor     ax,ax           , Установить ax в 0
Count
        inc     ax              , ax <- ax + 1
        call    Print           , Вызов подпрограммы
        cmp     ax, 10          , ax = 10 ?
        jne     Count           , Перейти, если ax <> 10
        :                       , Продолжение программы
```

Операторы, расположенные между Count: и jne, исполняются циклически до тех пор, пока ax не станет равн (ne) 10. Как и в предыдущем примере, команда cmp устанавливает флаги, которые проверяются при следующем у ловном переходе. Если условие не удовлетворяется, другими словами, если ax все еще не равно 10 — управлени передается назад к Count и цикл снова начинает исполняться, начиная с команды inc. Когда ax достигает 10, усл вие нарушается и jne уже не передает управление на целевую метку, вместо этого выполняется следующая за j команда

Двойные названия переходов

Как вы можете увидеть из табл. 4.5, многие условные переходы, выполняющие одну и ту же команду, имеют дв названия. Вы всегда можете взаимозаменяемо использовать любое из этих мнемонических названий: например, je jz ассемблируются в одинаковые машинные коды.

Зачем тогда нужны два различных имени? Ответ: сделать программирование более простым Jz расшифровыва ется как "переход, если флаг нуля равен 1", в то время как je означает "переход, если равно". Причина двух различ ных расшифровок становится очевиднее при рассмотрении использования команды перехода. После операции cmp в случае нулевого результата, флаг нуля устанавливается в 1. Зная это, вы можете использовать jz для проверки флага 0 и перехода в другую точку. Чтобы не заставлять вас на каждом шаге программы заниматься подобной ум ственной гимнастикой, в наборе команд процессора 8086 предусмотрены альтернативные мнемонические названия имеющие больший смысл в данной ситуации. После cmp можно использовать je для проверки равенства операндов либо jne для проверки операндов на неравенство. В большинстве случаев вам вообще не нужно знать, какие флаги установлены и проверяются.

Замечание

Иногда, конечно, вам будет необходимо знать проверяемые при условном переходе флаги. В этих случаях обращайтесь к мнемоническим названиям команд в главе 16. В этой главе также описаны точные комбинации битов флагов, проверяемые каждой командой условного перехода

Использование условных переходов

Какой условный переход лучше использовать в конкретной ситуации можно выяснить только на практике. В этом вам поможет чтение ассемблерных программ из этой книги, в которых вы увидите, как работает большинство команд перехода. Обязательно запомните окончания из табл. 4.6. Поймите также различие между двумя фразами, выше-ниже и меньше-больше, используемыми в таких командах, как jb и jge. Запомните два следующих положения:

- Используйте переходы выше-ниже, такие как ja и jbe, для *беззнаковых* значений
- Используйте переходы меньше-больше, такие как jle и jg, для значений *со знаком*

Вследствие особенностей арифметических операций с двоичными значениями, выражаемыми с помощью фик сированного количества битов, между сравнением знаковых и беззнаковых величин существуют важные различия (Например, сумма 1 и шестнадцатеричного значения FFFF равна 0 при использовании 16 бит. В десятичной системе это соответствует странному, однако верному уравнению: 66535+1=0.) Следующие несколько примеров помогут прояснить важные детали. Предположим, вы вычитаете два регистра и хотите перейти в определенное место, если в результате получается значение меньше 0. Вот правильный способ для достижения этой цели:

```
    sub  ax, bx          ; ax <- ax - bx
    jl   Negative        ; Перейти если ax < bx
```

Если при вычитании получается отрицательное значение, тогда условие jl выполняется и управление передается на адрес с меткой Negative. Очевидно, что если ax меньше, чем bx, тогда результат вычитания bx из ax будет отрицательным. Однако если заменить jl на jb, то следующие команды работать не будут:

```
    sub  ax, bx          ; ax <- ax - bx
    jb   Negative        ; ???
```

Условные переходы типа выше-ниже проверяют результаты сравнений и других операций на беззнаковых (положительных) значениях. Даже если bx больше, чем ax, результат вычитания беззнакового bx из ax все равно останется беззнаковым значением. Чтобы проверить, является ли беззнаковое значение ax большим, чем беззнаковое bx, вы можете записать:

```
    cmp  ax, bx          ; Беззнаковое ax > bx?
    ja   Greater         ; Перейти, если ax > bx
```

Команда ja (переход, если выше) корректно проверяет результат сравнения двух беззнаковых значений. Переход происходит только тогда, когда ax больше, чем bx. Если ax ниже или равно bx, тогда переход не выполняется. С другой стороны, если ax и bx являются значениями со знаком, тогда в этом случае оператор ja не подходит — вместо него, вероятно, нужно использовать знаковый условный переход jg.

Замечание

Возьмите в привычку использовать „выше-ниже" — для беззнаковых сравнений и „больше-меньше" — для знаковых. Делайте это в программах, разговорах, в ваших заметках и комментариях. Не существует легкого способа для освоения этих различий — вы просто должны запомнить соответствующие правила

Ограничения при использовании условных переходов

Все условные переходы имеют одно существенное ограничение: они могут передавать управление только на очень короткие расстояния — строго до 128 байт назад (в нижние адреса) или до 127 байт вперед (в верхние адреса) от первого байта команды, непосредственно следующей за командой перехода. Учитывая 2 байта, занимаемые условным переходом, вы можете перейти вперед несколько дальше, чем назад — небольшая деталь, которая редко имеет большое значение. Не волнуйтесь. Turbo Assembler сообщит вам, если вы попытаетесь перейти слишком далеко.

Расстояние до целевой точки условного перехода в интервале от –128 до 127 байт называется смещением — значение, вычисляемое за вас ассемблером от метки, которую вы поставили в тексте программы. Смещение не является настоящим адресом целевой метки, оно вставляется в ассемблированный машинный код для данной команды перехода. Вам никогда не придется вычислять смещение вручную, но вы должны знать, что так как целевой адрес выражается через смещение, условные переходы обладают удивительным свойством одинаково выполняться в любом месте памяти, что отражает интересный факт в программировании под 8086 процессор.

Замечание

Код, использующий только условные переходы, может исполняться в любом месте памяти. Он перемещаем, т.е. такой код можно разместить в другом месте памяти и это не повлияет на его исполнение.

Хотя перемещаемые условные переходы обычно обладают преимуществами, однако, если вам необходимо выполнить условный переход на значительное расстояние, ограничение на диапазон смещения становится неприятным. Для перехода более чем на 127 байт необходимо комбинировать условные и безусловные переходы. Например, предположим, что при dx, равном 1, программа должна переходить на выполнение подпрограммы с метки Error, которая, возможно, выдает сообщение о прерывании программы. Итак, вы должны записать:

```
        cmp     dx, 1           ; dx = 1 ?
        jne     Continue        ; Переход, если dx <> 1
        jmp     Error           ; Ошибка, останов (dx = 1)
Continue:
        :                       ; Продолжение программы, если нет ошибок
```

Если dx равно 1, тогда условие jne нарушается, выполняется безусловный jmp, который передает управление на метку Error, предположительно расположенную вне допустимого диапазона смещения команды jne. При таком соединении переходов внимательно следите за логикой — в этом случае легко неверно подобрать условный переход, что является характерным источником ошибок. Чтобы избежать неприятностей, запомните следующее пожелание.

Совет

Используйте условный переход, противоположный тому, который вы могли применять, если бы целевая точка попадала в допустимый диапазон. После этого переходите в нужное место с помощью команды jmp.

Этот совет можно проиллюстрировать на предыдущем примере, если метка Error находится в допустимом диапазоне условного перехода. Более простой вариант программы тогда будет иметь такой вид:

```
cmp  dx, 1          ; dx = 1?
je   Error          , Ошибка, останов (dx = 1)
```

Очевидно, что в этом фрагменте переход на Error осуществляется при условии, что dx равно 1. Для условного перехода на метку вне допустимого диапазона необходимо использовать противоположный условный переход (jne вместо je) с последующим безусловным jmp в нужное место.

Подробнее об условных переходах

Чтобы подробнее узнать о работе каждого условного перехода, запустите один из предыдущих примеров в Turbo Debugger. С этого момента вы уже должны делать это сами. Возьмите одну из тестовых программ, которые вы вводили ранее, и замените ее внутреннюю часть фрагментами из этого текста или, что еще лучше, создайте свой собственный пример. (Вы должны предусмотреть метки для каждого перехода и вызова подпрограммы.)

В главе 16 детально описаны все условные переходы. Обращаясь к этой главе, вы сможете узнать о том, какие флаги используются каждой командой. Кроме того, размышляйте логично. После сравнения у вас должен возникнуть вопрос. Хотите вы перейти по условию меньше-больше (для знакового результата) или по условию выше-ниже (для беззнакового результата)? Выполняйте переходы на минимально возможные расстояния и избегайте их чрезмерного использования. Ниже приведена типичная ошибка при написании такого кода:

```
      cmp     bx, 5               ; bx = 5 ?
      jne     Not5                ; Нет, перейти на Not5
      mov     ax, [count5]        ; Да, загрузить в ax [count5]
      jmp     Continue            ; Пропустить следующую команду
Not5
      mov     ax, [count]         , Загрузить в ax значение [count]
Continue.
      :                           ; Продолжение программы
```

В этом фрагменте для загрузки в ax различных значений, в зависимости от того, равно ли bx пяти, используются две метки и две команды перехода. Это слишком много. Для загрузки в ax одного из двух возможных значений из этого текста можно устранить метку и безусловный переход:

```
      mov ax, [count 5]       ; Установить в ax < --[count5]
      cmp bx, 5               ; bx = 5?
      je Continue             , Да, в ax содержится правильное значение, значит, переход
      mov ax, [count]         , Нет, загрузить в ax другое значение
Continue.
      :                       , продолжение программы
```

Это не просто короче и проще для чтения, когда bx не равно 5, а данный код работает быстрее. (Используемая здесь команда jmp занимает больше процессорного времени, чем пересылка значений между регистром и памятью, поэтому использование двух команд mov не расточительно, как можно подумать при первом чтении.)

Команды управления процессором

Команды, представленные в табл. 4.7, непосредственно работают с процессором. Во всех случаях, кроме одного, эти команды управления процессором ассемблируются в однобайтовые коды и не требуют операндов. Большинство команд либо устанавливают, либо очищают отдельные биты флагов. Другие синхронизируют процессор с внешними событиями, и, в одном случае, команда nop действительно не выполняет никакой работы.

Таблица 4.7. **Команды управления процессором 8086**

Мнемокод / Операнды	Описание
Команды флагов	
clc	Сброс флага переноса
cld	Сброс флага направления
cli	Сброс флага прерывания
cmc	Переключение флага переноса
stc	Установка флага переноса
std	Установка флага направления
sti	Установка флага прерывания
Команды внешней синхронизации	
esc	Переключение на сопроцессор
hlt	Останов процессора
lock	Блокировка шины доступа к данным
wait	Установка процессора в состояние ожидания
Прочие команды	
nop	Нет операции

Операции с флагами

Первая группа команд в табл. 4.7 устанавливает и очищает отдельные биты флагов. Флаг считается установленным, если его значение равно 1, и очищенным в случае нулевого значения. Вы можете устанавливать и очищать флаг переноса (stc и clc), флаг направления (std и cld) и флаг прерывания (sti и cli). Вы можете также дополнять с помощью cmc флаг переноса, переключая cf с 1 на 0 и с 0 на 1.

Команды для работы с флагом направления используются исключительно с командами обработки строк из табл. 4.8. Эти команды объясняются в главе 5. Флаг прерывания обычно устанавливается или очищается внутри программ обработки прерываний (далее, в главе 10). Вообще, sti разрешает большую часть прерываний, в то время как cli запрещает их выполнение.

Одно из типичных использований stc и clc состоит в установке флага переноса для возврата результата из подпрограммы. Например, можно написать следующую подпрограмму для проверки установки некоторого бита в значении, находящемся в регистре dl:

```
PROC TestBit
     test dl, 08h                ; Проверить бит 3
     jz Exit                     ; Выйти, если бит 3 = 0
     stc                         ; Установить флаг переноса
Exit:
     ret                         ; Возврат из подпрограммы
ENDP TestBit
```

Эта подпрограмма проверяет, равен ли третий бит единице, устанавливая в этом случае флаг переноса в 1. Команда test сбрасывает флаг переноса независимо от значений операндов. Кроме того, если результат равен 0, т.е. в данном примере значение третьего бита в регистре dl равно 0, то также устанавливается в 1 флаг нуля. В этом случае команда jz выполняет переход непосредственно на Exit, оставляя cf=0. В противном случае команда stc устанавливает флаг переноса, возвращая cf=1. Подпрограмма может вызываться из главной программы следующим образом:

```
mov dl, [testvalue]          ; Переслать проверяемое значение в dl
call TestBit                 ; Вызвать подпрограмму проверки
jc BitIsSet                  ; Перейти, если бит 3 = 1
:                            ; Продолжить программу, если бит 3 = 0
```

После вызова TestBit команда jc передаст управление на BitIsSet только в том случае, если cf=1. Возвращение таким способом из подпрограммы флага переноса является типичным приемом программирования на языке ас-

семблера. Кроме этого, вы часто будете встречать программы, которые используют cf в качестве индикатора ошибок. Для вызова гипотетической подпрограммы DiskRead и перехода к ее обработке в случае возникновения ошибки чтения вы должны написать примерно следующее:

```
        call DiskRead           ; Чтение с диска ( подпрограмма не показана )
        jnc Continue            ; Продолжить программу, если нет ошибок (CF= 0)
        jmp Error               ; Иначе, перейти к обработке ошибки (CF = 1)
Continue.
        :                       ; Продолжение программы
```

Синхронизация

Команды внешней синхронизации процессора 8086 применяются достаточно редко и, возможно, вам вообще не придется их использовать. Команда hlt переводит процессор в состояние останова. Продолжение работы происходит только после возникновения одного из двух типов прерываний. (Дополнительную информацию по прерываниям см. в главе 10.) Обычно hlt используется для того, чтобы заставить процессор подождать сигнала от внешнего устройства и продолжать работу только после разрешения от этого устройства.

Wait и esc используются для создания интерфейса с математическим сопроцессором. Esc — единственная команда управления процессором, которая требует для себя операндов.

Lock приводит к генерации (включению) сигнала, информирующего интерфейсные схемы о занятости шины данных. (Шина — это набор соединений между процессором, памятью и любым другим оборудованием, по которым своими разнообразными путями перемещаются биты данных.) В действительности, Lock является не отдельной командой, а префиксом для других команд, чаще всего — для xchg. В компьютерах с несколькими процессорами, имеющими доступ к одной и той же памяти, lock используется для предотвращения потенциальных конфликтов, возникающих в случае одновременной записи обоими процессорами информации в одинаковые адреса памяти. Ели вам необходима эта возможность, обращайтесь к документации фирмы Intel. Обычно при программировании на персональном компьютере необходимости в операторе lock не возникает.

Что-то из ничего

Nop, вероятно, самая странная из всех команд 8086. Из ее названия вытекает, что nop вообще ничего не делает. Так оно и есть! Исполнение nop подобно ускорению машины на нейтральной передаче — вы вдавливаете до пола педаль газа, но быстрее от этого не едете. Но иногда в странном мире программирования на языке ассемблера даже "ничего" имеет свое назначение. Обычно nop используется в двух случаях:

- Для временного устранения некоторой команды.
- Для резервирования пространства для последующих операторов jmp.

Команда nop наиболее полезна в случае, если вы хотите убрать из программы какую-либо команду, не выполняя повторного ассемблирования и линковки. Помещение на место этих команд нескольких машинных кодов, соответствующих команде nop (90 в шестнадцатеричном представлении), — полезный прием отладки. Пытаясь обнаружить источник ошибок, попробуйте заменить одну или две подозрительные команды на nop, в надежде, что при этом выявятся скрытые ошибки. Часто, устраняя команду, можно разобраться в вызываемых ею эффектах. Например, предположим, вы хотите проверить, что случится, если в строке 42 программы 4.7 не будет обнуляться регистр ah. Вы можете для этого удалить команду из текста программы, выполнить повторное ассемблирование и компоновку. Можно просто загрузить уже ассемблированный код в Turbo Debugger и выполнить следующие шаги.

1. Откройте окно CPU и поставьте курсор на команду xor в начале AddRegisters. Запомните адрес слева, вероятно, напоминающий cs:001D.
2. Нажмите <Tab> для перемещения курсора на область дампа памяти в левом нижнем углу окна CPU.
3. Нажмите <Ctrl+G> для выбора команды Goto. Затем введите адрес из шага 1, например cs:111Dh. (Не забывайте добавлять h для шестнадцатеричных значений!)
4. Теперь курсор должен находиться на первых двух байтах, 32 и E4, двоичного машинного кода команды xor ah, ah. Проверьте это, сравнив байты в области дампа памяти с дезассемблированным кодом.
5. Измените значения байтов, набрав 090h 090h, нажмите <Enter> и посмотрите на дезассемблированный код. 2-байтовая команда xor заменилась двумя однобайтовыми командами nop.
6. Используйте <F7> для пошагового исполнения модифицированной программы, наблюдая при этом, что произойдет (или, наоборот, не произойдет) с ah при выполнении команды nop. После окончания подпрограммы (по команде ret) в ax не будет правильной суммы четырех регистров. Как показывает этот опыт, для получения верного результата необходимо обнуление регистра ah.
7. Для перезапуска программы нажмите <Ctrl+F2> либо замените команды nop первоначальными машинными кодами, 032h и 0E4h.

Резервирование пространства для переходов

Turbo Assembler для резервирования пространства под jmp иногда может вставлять в программы команды nop. Вы знаете, что jmp безусловно передает управление на целевой адрес. Но, в зависимости от дальности перехода, Turbo Assembler представляет его в одной из нескольких форм машинных кодов, добавляя к ассемблированной программе от 2 до 5 байт. Обычно вы не учитываете этот факт, позволяя ассемблеру выбирать наиболее эффективную форму, что он всегда и делает. Но, даже несмотря на это, поскольку Turbo Assembler является однопроходным, т.е. генерирует объектный код, просматривая исходный текст программы только один раз, с приведенными ниже последовательностями команд возникают определенные проблемы:

```
    or      ax  bx          , ax = bx ?
    jz      Skip            , Переход на Skip, если да
    jmp     Elsewhere       , Иначе перейти к Elsewhere
Skip
    mov     ax  1           , Переслать 1 в ax, если ax = bx
    jmp     Continue        , Пропустить следующую команду
Elsewhere
    mov     ax ,2           , Переслать 2 в ax, если ax <> bx
Continue
                            , Продолжение программы
```

Хотя эта последовательность не имеет практического смысла, она иллюстрирует типичную проблему. Когда Turbo Assembler встречает первую команду jmp, которая в нашем случае выполняет переход на более высокие адреса памяти, он не знает, насколько далеко от jmp расположен целевой адрес Elsewhere. Как пессимист, Turbo Assembler рассчитывает на наихудший вариант, предполагая, что Elsewhere находится на расстоянии, большем, чем 127 байт. Поэтому ассемблер резервирует пространство под 3-байтовый jmp, который допускает расстояния до 32 Кбайт. Дойдя до Elsewhere, Turbo Assembler понимает свою ошибку — Elsewhere расположена достаточно близко и можно использовать более короткий 2-байтовый jmp, работающий в интервале от 128 байт вверх до 127 байт вниз Поскольку 2-байтовый jmp работает быстрее 3-байтового, Turbo Assembler возвращается назад и заменяет jmp его 2-байтовым аналогом. Чтобы не выполнять повторного ассемблирования команд, расположенных между jmp и Elsewhere, ассемблер заменяет третий байт командой nop и продолжает ассемблирование оставшейся части программы. Если вы выполните ассемблирование этого короткого примера, то получите код, подобный следующему:

```
cs  0000    EB 04           jmp     Elsewhere
cs  0002    90              nop
```

Вставленный оператор nop не делает ничего, кроме занятия пространства. Вследствие предыдущего безусловного перехода, nop никогда и не исполняется.

Чтобы избавиться от бесполезной команды nop и сохранить 1 байт, поместите перед целевым адресом в команде jmp директиву SHORT:

```
jmp  SHORT   Elsewhere
```

Это приведет к тому, что Turbo Assembler будет использовать 2-байтовую версию jmp. Конечно, если позже Elsewhere выйдет за границу 127 байт, вы получите сообщение об ошибке и должны будете убрать директиву SHORT.

Использование директивы JUMPS

Если где-нибудь в начале вашей программы вы вставите директиву JUMPS, то Turbo Assembler позволит вам использовать команды условных переходов в точки, расположенные дальше обычного ограничения в 127 байт. Однако у этой команды есть свои подводные камни. Предположим, вы написали:

```
    JUMPS
    or      ax, ax          ; ax = 0 ?
    je      There           ; Перейти, если ax = 0
    mov     ax  5           ; Иначе переслать 5 в ax
There
```

При ассемблировании команды je, учитывая директиву JUMPS, Turbo Assembler вставляет следующие строки:

```
    je There
    nop
```

```
        nop
        nop
There:
```

Три оператора nop резервируют место под дополнительный код, который вставляется ассемблером в случае, е ли целевая метка There расположена вне пределов обычной досягаемости оператора je:

```
        jne     Temp
        jmp     There
Temp.
```

Вместо ассемблирования написанной вами команды je Turbo Assembler вставляет обратную ей команду jne с по следующим безусловным переходом — таким образом, как это объяснялось раньше. Метка Temp использова здесь исключительно для иллюстрации, в текст программы она не вставляется. Проблема с директивой JUMPS з ключается в тех лишних операторах nop, которые могут без необходимости вставляться в программу. По этой пр чине я предпочитаю исключительно двойные переходы, хотя директива JUMPS может оказаться полезным вр менным инструментом. После окончания разработки программы вы можете преобразовать дальние переходы двойные и устранить из конечного варианта директивы JUMPS.

При этом будут удалены бесполезные операторы nop.

Команды для работы со строками

Команды для работы со строками, представленные в табл. 4.8, — компактный, но мощный инструмент для об работки всех типов данных, а не только символьных строк. Помните, что строки в языке ассемблера являются по следовательностями байтов, которые могут либо представлять, либо не представлять ASCII-символы. Несмотря н наводящие названия, командам работы со строками 8086 все равно, что означают эти байты. Строковые команд делятся на три группы:

- Команды пересылки строк
- Команды проверки строк
- Команды префикса повторения

Таблица 4.8. **Команды обработки строк 8086**

Мнемокод / Операнды	Описание
Команды пересылки строк	
lods источник	Загрузка строки байтов или строки слов
lodsb	Загрузка строки байтов
lodsw	Загрузка строки слов
movs назначение, источник	Пересылка строки байтов или строки слов
movsb	Пересылка строки байтов
movsw	Пересылка строки слов
stos назначение	Запись строки байтов или строки слов
stosb	Запись байта в строку
stosw	Запись слова в строку
Команды проверки строк	
cmps назначение, источник	Сравнение строк по байтам или словам
cmpsb	Сравнение строк по байтам
cmpsw	Сравнение строк по словам
scas назначение	Поиск байта или слова в строке
scasb	Поиск байта в строке
scasw	Поиск слова в строке

Окончание таблицы 4.8

Мнемокод / Операнды	Описание
Команды префикса повторения	
rep	Повтор
repe/repz	Повтор, пока равно 0
repne/repnz	Повтор, пока не равно 0

Команды пересылки строк пересылают байты или слова из памяти в регистр, из регистра в память или непосредственно из памяти в память. Команды *проверки строк* позволяют вам сравнивать либо находить байты или слова. Команды *префикса повторения* могут присоединяться к другим строковым командам, создавая команды, которые повторяют строковую операцию определенное число раз до выполнения заданного условия. Префиксные команды позволяют быстро заполнять значениями тысячи байтов, копировать строки из одного места в другое и искать необходимые значения в больших блоках памяти.

Несмотря на большое число мнемонических названий в табл. 4.8, на самом деле существует только пять строковых команд: lods, stos, movs, scas, cmps. Остальные являются сокращенными мнемоническими названиями для этих же самых команд. Как вы видите из таблицы, сокращенные названия типа lodsb или cmpsw не требуют операндов и поэтому более просты для использования. Аналогично, существует только два префикса повторения: команда rep идентична командам repe и repz. Команды repne и repnz также представляют одинаковый префикс. Взаимозаменяемые названия просто помогают вам разбираться в действиях программы.

Строковые индексные регистры

Все строковые команды используют для выполнения своих функций определенные регистры. В отличие от других команд, для которых вы сами выбираете используемые регистры, строковые команды требовательнее и всегда работают с одними и теми же комбинациями регистров — ds:si и es:di — с индексными регистрами строки-источника и строки-назначения, которые определяют смещения в данных и сегментах.

Замечание

Если ds и es адресуют один и тот же сегмент данных, как это часто и бывает, тогда при выполнении строковых операций вам не нужно беспокоиться о корректной адресации сегментов памяти. Если ds и es адресуют различные сегменты, вы должны быть внимательными, указывая нужные сегменты для выполняемых операций. Кроме того, индекс назначения di определяется всегда относительно сегмента, адресуемого es. Индекс источника si вычисляется обычно относительно сегмента, адресуемого ds, если вы не переопределили это, используя адресацию es:si.

Пять строковых команд загружают, записывают, пересылают, сравнивают и выполняют поиск байтов и слов. Выполняя это, каждая команда также увеличивает или уменьшает значения в используемых ею регистрах. При операциях с байтами происходит вычитание или добавление 1 к регистру si или di (или обоим); при операциях со словами добавляется или вычитается 2. Например, если si равно 0010 в шестнадцатеричной системе, тогда после выполнения операции lodsw значение si должно возрасти до 0012 (или уменьшиться до 000E, в зависимости от направления строковой операции). Вследствие этого добавление к строковой команде префикса повторения позволяет программе обрабатывать целые наборы данных с помощью единственной команды.

Флаг направления df определяет, должна ли строковая команда приводить к увеличению или уменьшению si и di. Если df = 1, тогда индексы уменьшаются в сторону меньших адресов. Если df = 0, тогда индексы увеличиваются в сторону более высоких адресов. Для очистки флага df и автоматического увеличения si и di в сторону более высоких адресов применяйте команду cld. Использование std приводит к установке флага df и автоматическому уменьшению si и di в сторону меньших адресов.

Замечание

Вы можете установить или очистить флаг df в начале программы, но, поскольку его можно изменять другими подпрограммами, лучше всегда устанавливать или очищать флаг направления непосредственно перед каждой строковой операцией. Это не требует много времени, однако помогает предотвратить возможные ошибки.

Загрузка строк

Команда lods загружает данные, адресуемые ds:si или es:si в регистр al для байтовых операций или в ax для операций со словами. После этого, в зависимости от установки флага направления df, увеличивается или уменьшается

значение si. При операциях с байтами si изменяется на 1, при операциях со словами — на 2. С помощью этих команд вы можете создать простой цикл для поиска определенного значения байта:

```
        cld                             ; Автоувеличение si
Repeat:
        lods [byte ptr ds:si]           ; al <-- [ds si], si <-- si + 1
        or al, al                       ; al = 0 ?
        jne Repeat                      ; Повторять, если al <> 0
```

Сначала cld очищает df, готовясь к автоувеличению si после выполнения каждой команды lods, копирующих в al байты, адресуемые ds:si. Затем значение в si увеличивается, адресуя следующий байт памяти. После загрузки каждого байта команда or проверяет al на равенство нулю. В случае неравенства нулю по команде jne выполняется возврат на метку Repeat:, повторяя те же действия до нахождения нулевого байта. (Кстати, если в сегменте ds не существует нулевого байта, этот цикл будет повторяться бесконечно. Будьте внимательны, чтобы не внести в свои программы подобные ошибки.)

Замечание

Автоувеличение либо уменьшение si или di на краях сегментов приводит к переходу на другой конец сегмента. Другими словами, если si или di равно 0FFFFh, добавление 1 „увеличивает" значение регистра до 0. И наоборот, если регистр равен 0000, то вычитание 1 „уменьшает" его значение до 0FFFFh.

Использование сокращенных мнемонических названий строковых команд

Поскольку команда lods обычно работает со значениями, адресуемыми ds:si, Turbo Assembler предоставляет для нее два сокращенных мнемонических названия, не требующих операндов, — lodsb и lodsw. В этих и других сокращенных мнемонических названиях sb используется для обозначения байта из *строки байтов*, sw — слова из *строки слов*. В табл. 4.9 представлены полные эквивалентные формы для каждого сокращенного мнемонического названия.

Таблица 4.9. **Сокращенные строковые команды**

Сокращение	Эквивалентная строковая команда	Сокращение	Эквивалентная строковая команда
lodsb	lods [byte ptr ds:si]	movsw	movs [word ptr es:di], [word ptr ds:si]
lodsw	lods [word ptr ds:si]	scasb	scas [byte ptr es:di]
stosb	stos [byte ptr es:di]	scasw	scas [word ptr es:di]
stosw	stos [word ptr es:di]	cmpsb	cmps [byte ptr ds:si], [byte ptr es:di]
movsb	movs [byte ptr es:di], [byte ptr ds:si]	cmpsw	cmps [word ptr ds:si], [word ptr es:di]

Адресация строк через метки

Для полных форм строковых команд из табл. 4.9 Turbo Assembler позволяет использовать метки. Например, для загрузки в al первого байта строки s1 вы можете записать:

```
DATASEG
string db 'This is a string', 0

CODESEG
mov si, offset string           ; Присвоить si адрес строки
lods [string]                   ; Получить первый байт строки
```

Но команда lods [string] ассемблируется не так, как вы могли подумать. Turbo Assembler преобразует эту команду в lodsb, предполагая, что перед этим вы загрузили в si смещение string. Помните, что все строковые команды требуют для адресации используемых данных строго определенных регистров. Даже если вы определили переменную по имени, как в этом примере, все равно нужно загрузить в si и di соответствующие адреса. Определение переменной по имени просто позволяет Turbo Assembler проверить, адресуется ли эта переменная соответствующими регистрами. Ассемблер не инициализирует за вас индексный регистр.

Запись данных в строки

Команда stos (сокращенные мнемонические названия stosb и stosw) записывает байт из регистра al или слово из регистра ax в память по адресу es:di. Так же как и lods, в зависимости от установки df и того, состоят данные из байтов или слов, stos увеличивает либо уменьшает на 1 или 2 значение в регистре di. Использование в цикле команд lods и stos позволяет переместить строку из одного места памяти в другое:

```
     cld                    , Автоувеличение si и di
Repeat
     lodsw                  , ax <-- [ds si], si <-- si + 2
     cmp ax  0FFFFh         , ax = 0FFFFh?
     je Exit                  Переход, если ax = 0FFFFh
     stosw                    [es di] <-- ax, di <-- di +2
     jmp Repeat             , Повторять до окончания
Exit
```

В этом примере сначала команда cld обеспечивает подготовку к автоувеличению si и di. Затем lodsw загружает в ax слово, расположенное по адресу ds.si, и увеличивает значение si на два. Если ax равно значению 0FFFFh, предположительно помещенному в память ранее в качестве маркера конца данных, то команда je обеспечивает выход из цикла В противном случае stosw записывает слово из ax в память по адресу es di, также увеличивая di на 2. Последний jmp обеспечивает повторение этих действий до момента нахождения маркера 0FFFFh. Снова возникает опасность, если в сегменте данных не существует значения 0FFFFh. Позже вы увидите, что существуют другие способы выполнения этой операции, устраняющие данную проблему.

Пересылка строк

Для пересылки байтов и слов между областями памяти используйте команду movs или ее сокращенные формы movsb и movsw. Эти команды не требуют промежуточных регистров для перемещения данных в памяти, они явля-

5 ГЛАВА

Простые структуры данных

- Адресация данных в памяти
- Способы адресации памяти
- Выражения и операторы
- Простые переменные
- Инициализированные и неинициализированные данные
- Строковые переменные
- Локальные метки
- Пакет для работы с ASCIIZ-строками
- Программирование по частям
- Пакет для ввода-вывода строк
- Компоновка модулей в программу
- Изучение модуля STRINGS
- Резюме
- Упражнения
- Задания

Адресация данных в памяти

В программировании на языке ассемблера 8086 самой сложной среди всех тем является изучение многочисленных способов адресации данных в памяти. Можно, однако, избежать этих затруднений, если помнить, что все обращения к данным делятся на три типа:

- Обращения к непосредственным данным
- Обращения к данным в регистрах
- Обращения к данным в памяти

Непосредственные данные — это значения, записанные прямо в машинном коде команд. Например, когда вы записываете:

```
mov  ax, 5              , ax <- 5
```

ассемблер генерирует машинный код, соответствующий команде mov, которая загружает в регистр ax непосредственное значение 5. Значение 5 записывается в ассемблированном машинном коде команды mov. В большинстве случаев непосредственные данные являются либо единственным операндом, либо вторым из двух операндов. (Исключение составляет команда out, в которой непосредственные данные могут находиться на месте первого из двух операндов.) Непосредственные данные не должны изменяться при выполнении программы.

Замечание

Конечно, можно написать программу, изменяющую машинные коды, записанные в памяти. Используя этот подход, можно определить место, в котором записано непосредственное значение, и изменить его до выполнения команды. Однако этот прием считается плохим тоном. Такие самоизменяющиеся коды часто сложны для отладки и, что еще хуже, не могут быть записаны в ROM — нестираемой памяти, где значения запечатлены раз и навсегда. Кроме того, поскольку процессоры семейства 8086 одновременно загружают некоторые команды в небольшую область внутренней памяти, называемую командным кэшем, то модификация кода в процессе работы очень ненадежна. Преодолейте соблазн написания самоизменяющихся программ, только в единичных случаях результат может оправдать риск.

Регистровые данные ссылаются на данные, содержащиеся в регистрах процессора. Вы уже встречали много примеров такого способа обращения к данным. Машинный код, создаваемый ассемблером для регистровых данных, содержит соответствующие значения, позволяющие команде оперировать заданными регистрами, как в следующей команде:

```
add  ax,bx              , ax <- ax + bx
```

Данные памяти являются третьим типом обращения к данным, и для них существует несколько вариантов. Во избежание осложнений при изучении этих вариантов помните, что цель состоит в том, чтобы помочь процессору вычислить 16-битовое беззнаковое значение, называемое *эффективным адресом*, или ЭА. ЭА показывает смещение, начиная от базового сегмента, адресуемого одним из четырех сегментных регистров: cs, ds, es, ss. Как было отмечено в главе 4, сегментный регистр и смещение образуют 32-битовый логический адрес, который в дальнейшем преобразуется в 20-битовый физический адрес, однозначно определяющий положение байта в памяти.

Вам никогда не нужно будет вычислять ЭА или образовывать 20-битовый физический адрес — эту работу выполняет исключительно процессор. Вы ответственны только за то, чтобы дать процессору данные, необходимые для вычисления ЭА, расположив переменные в памяти. Для этого можно использовать один из семи показанных ниже способов адресации.

Замечание

В главе 16 для каждой команды приводятся допустимые способы адресации памяти. Всякий раз, когда вы не уверены в том, допускает ли команда заданный способ, обращайтесь к этому справочнику.

Способы адресации памяти

В табл. 5.1 представлены семь способов адресации памяти, доступных при программировании на 8086-м процессоре. Кроме строковой адресации и адресации портов ввода-вывода, имеющих определенные ограничения, эти способы адресации могут использоваться со всеми командами, которые допускают обращения к данным в памяти. Хотя в табл. 5.1 в качестве примера применяется команда mov, вы можете использовать аналогичные обращения и с другими командами, такими как add, inc или xor. В следующих разделах описаны первые пять способов адресации. Строковая адресация и адресация портов ввода-вывода будет рассмотрена позже.

Таблица 5.1 **Способы адресации 8086**

Способ адресации	Пример
Прямой	mov ax, [count]
Косвенный	mov ax, [bx]
Базовый	mov ax, [record + bp]
Индексный	mov ax, [array + si]
Базово-индексный	mov ax, [recordArray + bx + si]
Строковый	lodsw
Порт ввода-вывода	in ax, dx

Прямые адреса

Прямой адрес — это литеральный адрес смещения переменной в памяти относительно любого базового сегмента. Например, для обращения к переменным в сегменте данных можно воспользоваться командами следующего типа.

```
inc  [MyMoney]          ; Прибавить 1 к значению [MyMoney]
```

Запись [MyMoney] ассемблируется в адрес смещения, по которому записана переменная MyMoney. Все подобные обращения к прямым адресам зафиксированы в ассемблированном коде и не должны изменяться при исполнении программы. (Самоизменяющиеся программы могут менять ссылки на прямой адрес, но по описанным выше причинам это ненадежный метод.)

Замечание

При прямом обращении к памяти фиксируется только адрес смещения. Сегмент, в котором записана переменная MyMoney, может начинаться с границы любого параграфа, поэтому нет никаких гарантий, что MyMoney будет записана по определенному физическому адресу.

Переназначения

Обычно обращения к непосредственному адресу выполняются относительно сегмента, адресуемого регистром ds. Чтобы это изменить, можно определить *сегментное переназначение* следующим образом:

```
mov  ch, [es:OverByte]
```

Эта команда загружает байт с меткой OverByte, записанный в сегменте, адресуемом es. Команда переназначения сегмента на es: требуется для того, чтобы изменить используемую по умолчанию процессором адресацию базового сегмента ds. Можно также использовать аналогичные переназначения и для доступа к данным в других сегментах. Ниже приводятся три соответствующих примера:

```
mov  dh, [cs:CodeByte]          ; dh <- байт из кодового сегмента
mov  dh, [ss:StackByte]         ; dh <- байт из стекового сегмента
mov  dh, [ds DataByte]          ; dh <- байт из сегмента данных ???
```

В первой строке в dh загружается байт из кодового сегмента. Поскольку большинство переменных находится в сегменте данных, обращение к данным, записанным в кодовом сегменте, крайне редко. Вторая строка загружает байт из сегмента стека. Хотя это и допустимо, но на практике происходит не часто. В третьей строке излишне определяется сегмент ds — непосредственные данные всегда по умолчанию располагаются в сегменте, адресуемом ds. Ниже приведены несколько дополнительных советов, которые помогут вам правильно использовать переназначения.

- Несмотря на то что вы определяете переназначение как часть обращения к данным, в действительности оно занимает байт машинного кода и вставляется только перед командой, на которую воздействует. Переназначения являются командными префиксами и изменяют поведение следующей исполняемой команды.
- Переназначение действует только в пределах одной команды. Нужно использовать переназначение при каждом обращении к данным, расположенным в сегментах, отличных от установленных по умолчанию для данной команды.
- В режиме Ideal Turbo Assembler адрес, включая сегментное переназначение, целиком берется в скобки. Хотя режим MASM обеспечивает более свободный стиль, простые синтаксические требования режима Ideal полностью совместимы с режимом MASM.
- Вы должны быть уверены, что переменные действительно располагаются в определяемых вами сегментах, и следить, чтобы адреса этих сегментов находились в сегментных регистрах es и ds. Регистр стека ss и регистр кодового сегмента cs инициализации не требуют.

Косвенная адресация

Вместо обращения к переменной в памяти по имени можно использовать один из трех регистров — bx, si или di в качестве указателя на данные в памяти. Поскольку для адресации различных областей памяти программа может изменять значения регистров, *косвенная адресация* позволяет одной команде оперировать многими переменными. Загрузив адрес смещения в соответствующий регистр, можно обращаться к данным, записанным в памяти с помощью следующих команд:

```
mov  cx, [WORD bx]          ; Скопировать слово из [bx] в cx
dec  [BYTE si]              ; Уменьшить значение байта в [si]
```

Операторы WORD и BYTE необходимы, если Turbo Assembler не может определить, что адресует регистр в памяти — байт или слово. В первой строке данные, адресуемые bx, пересылаются в 16-битовый регистр cx, следовательно, в операторе WORD нет необходимости, так как ассемблер знает размер данных из контекста команды. Излишнее определение оператора, как в этом примере, не приносит никакого вреда. Во второй строке необходимо использовать оператор BYTE, так как у ассемблера нет другого способа узнать — уменьшать значение байта или слова.

Замечание

Встречая команду inc [si], Turbo Assembler выдает предупреждение, но продолжает ассемблировать программу, подразумевая, что si адресует в памяти слово. Чтобы устранить неоднозначную адресацию и уменьшить вероятность введения труднонаходимых ошибок, используйте всегда операторы WORD и BYTE.

При косвенной адресации по умолчанию используется сегмент, адресуемый ds. Так же как и при прямой адресацией, вы можете использовать переназначения для изменения этого сегмента на любой из трех других. Это показано на следующих примерах:

```
add  [WORD es bx], 3        ; Прибавить 3 к слову в es:bx
dec  [BYTE ss si]           ; Уменьшить байт по адресу ss si
mov  cx, [cs di]            ; Загрузить слово из сегмента кода
```

Как объяснялось ранее, при использовании переназначений вы должны быть уверены, что адресуемые вами данные действительно находятся в заданных сегментах. И хотя допускается переназначение на стековый сегмент, это используется крайне редко.

Замечание

Строковые команды используют es по умолчанию в качестве сегментного регистра для индекса di. При косвенной адресации для di используется по умолчанию сегмент ds. Не путайте эти два совершенно различных способа адресации, хотя они и применяют один и тот же индексный регистр.

Базовые адреса

Для *базовой адресации* используются два регистра — bx и bp. Обращения к bx проводятся относительно сегмента данных, адресуемого ds. Ссылки на bp осуществляются относительно стекового сегмента ss и обычно применяются для чтения и записи значений в стек. Для обращения к данным из любых других сегментов можно использовать сегментные переназначения, как это было описано выше.

При базовой адресации к области, адресуемой bx или bp, добавляется значение перемещения. Это перемещение является знаковым 16- или 8-битовым значением, представляющим дополнительное смещение вверх или вниз от смещения, определенного в заданном регистре. Обычно базовая адресация используется для нахождения полей в структуре данных. Например:

```
mov  bx, OFFSET Person      ; Точка начала символьной переменной Person
mov  ax, [bx + 5]           ; Получить слово, расположенное на
                            ; расстоянии в 5 байт от начала Person
```

После размещения в bx адреса смещения переменной с именем Person второй оператор mov загружает в ax значение, записанное на расстоянии 5 байт от начала Person. Аналогично, для обращения к переменным в стеке можно использовать следующие команды:

```
inc [WORD bp + 2]           ; Увеличение значения слова в стеке
dec [BYTE bp - 8]           ; Уменьшение значения байта в стеке
```

Помните, что ссылки на bp делаются относительно стекового сегмента ss. (В главах 12 и 13 детально описывается использование bp и базовой адресации для доступа к стековым переменным.) Как показано во второй строке, значение перемещения тоже может быть отрицательным. Поскольку перемещения являются 16-битовыми значениями, то эффективный диапазон составляет от –32768 до 32767 байт от смещения, указанного в bx или bp.

Замечание

При нулевом перемещении базовая адресация совпадает с косвенной адресацией для регистра bx. „Зная" это, Turbo Assembler преобразует обращения вида [bx+0] в более эффективные [bx] (нет перемещения). Это неверно для обращений, использующих bp в выражении вида [bp+0], для которого [bp] просто является синонимом, а не отличным способом адресации.

Индексные адреса

Индексная адресация идентична базовой адресации за исключением того, что адреса смещений содержатся в регистрах si и di. До тех пор пока вы не определили сегментное переназначение, все обращения к индексным адресам проводятся относительно сегмента данных, адресуемого ds. Обычно индексная адресация используется для доступа к простым массивам. Например, для увеличения пятого байта в массиве 8-битовых значений можно написать:

```
inc  [BYTE si + 4]                ; Добавить 1 к пятому элементу массива
```

Поскольку si+0 указывает на первый элемент массива, для нахождения пятого байта в массиве необходимо использовать перемещение 4, а не 5. Так же как и в случае адресации, перемещения являются знаковыми значениями и поэтому могут быть отрицательными:

```
mov  dx, [WORD di - 8]            ; Загрузить слово, находящееся на
                                  ; расстоянии 8 байт до [di]
```

Замечание

При нулевом перемещении индексная адресация совпадает с косвенной адресацией для обоих регистров — si и di. „Зная" это, Turbo Assembler сводит ссылки вида [si+0] и [di+0] к более эффективным косвенным эквивалентам — [si] и [di].

Базово-индексные адреса

Базово-индексная адресация объединяет два регистра и добавляет необязательное значение перемещения для формирования обращения к памяти по составному смещению — соединяя таким образом черты базового и индексного способов адресации. Первым регистром должен быть либо bx, либо bp. Вторым регистром должен быть si или di. Смещения, начинающиеся с bx, определяются относительно сегмента данных ds, смещения, начинающиеся с bp, — относительно сегмента стека ss. Так же как и при других способах адресации, вы можете это изменять, используя переназначения. Несколько примеров помогут объяснить этот важный метод адресации:

```
mov  ax, [bx + si]                ; Загрузить в ax слово из сегмента данных
mov  ax, [bx + di]                ; Загрузить в ax слово из сегмента данных
mov  ax, [bp + si]                ; Загрузить в ax слово из сегмента стека
mov  ax, [bp + di]                ; Загрузить в ax слово из сегмента стека
```

Turbo Assembler позволяет вам менять порядок регистров, например, записывая [si+bx] и [di+bp]. Это не различные способы адресации, а только разные формы одного и того же обращения. В любом из четырех предыдущих вариантов можно добавлять необязательное значение перемещения:

```
mov  ax, [bx + si + 5]            ; Загрузить в ax слово из сегмента данных
mov  ax, [bx + di + 5]            ; Загрузить в ax слово из сегмента данных
mov  ax, [bp + si + 5]            ; Загрузить в ax слово из сегмента стека
mov  ax, [bp + di + 5]            ; Загрузить в ax слово из сегмента стека
```

Кроме того, в каждом из этих восьми основных вариантов базово-индексной адресации вы можете использовать переназначения и ссылаться на данные в сегментах, отличных от установленных по умолчанию:

```
mov  ax, [es:bx + si + 8]         ; Использовать es вместо ds
mov  ax, [cs.bp + di]             ; Использовать cs вместо ss
```

Базово-индексная адресация является самым мощным методом обращения к памяти. С его помощью вы можете определить начальное смещение в bx или bp (возможно, адрес массива), добавить к нему значение индекса из si или di (возможно, определяющий элемент массива), после чего прибавить значение перемещения (может быть, для нахождения поля записи в этом заданном элементе массива). Изменяя значения базового и индексного регистров, программа может адресовать в памяти сложные структуры данных.

Замечание

В режиме MASM базово-индексная адресация (и другие методы адресации) могут иметь более свободное представление типа 5[bx+si] и 5[bp][di], наводя многих на мысль о какой-то отдельной и загадочной форме адресации. Но это не так. Как показано выше, существует только восемь основных форм базово-индексной адресации. Вы избежите многих неприятностей (и ничего при этом не потеряете), если будете придерживаться стандартных форм, описанных здесь, и требуемых режимом Ideal.

Использование директивы ASSUME

Директива ASSUME указывает Turbo Assembler сегмент памяти, на который ссылается сегментный регистр. Назначение ASSUME состоит в том, чтобы позволить ассемблеру при необходимости автоматически вставлять команды переназначения. Помните, что ASSUME является командой ассемблера и не генерирует никакого кода.

При использовании упрощенной сегментной адресации, как это делается в большинстве примеров из книги, вам редко потребуется применять ASSUME. Кроме того, используя сегментные переназначения, можно совсем устранить необходимость в директиве ASSUME. Однако, несмотря на это, нужно понимать, как она работает. Предположим, вы напишите:

```
CODESEG
     jmp     There                    ; Пропустить описание v1
v1   db      5                        ; Записать 5 в кодовом сегменте
There:
     mov     ah,     [cs v1]          ; Загрузить 5 в ah
```

Этот фрагмент кода иллюстрирует один из способов записи данных внутри кодового сегмента — необычный, но допустимый на практике. Команда jmp пропускает описание байтовой переменной v1. (При смешивании данных и кода, конечно, вы не хотите, чтобы переменные случайно выполнялись как команды.) Для загрузки значения v1 в ah команда mov использует сегментное переназначение (cs:). Переназначение необходимо, так как по умолчанию при прямом обращении к данным используется сегмент данных ds.

Поскольку Turbo Assembler "знает", что cs ссылается на кодовый сегмент, он позволяет вам заменить команду mov более простой командой:

```
mov  ah, [v1]                 ; Загрузить в ah 5 из кодового сегмента
```

Хотя явное переназначение не используется, Turbo Assembler проверяет свой список переменных, определяет, что v1 записана в кодовом сегменте, и автоматически вставляет необходимое переназначение. В других случаях, когда Turbo Assembler не "знает", какие сегментные регистры на какие сегменты памяти ссылаются, вы должны либо использовать явные переназначения, либо сообщить это ассемблеру с помощью директивы ASSUME. Вот другой пример:

```
CODESEG
     jmp     There                    ; Пропустить описание v1
v1   db      5                        ; Записать 5 в этом месте
There:
     mov     ax, @code                ; Поместить в регистр es
     mov     es, ax                   ; адрес кодового сегмента
ASSUME es:_TEXT
     mov     ah, [v1]                 ; Загрузить 5 в ah из сегмента es
```

Снова байт 5 записывается прямо в кодовый сегмент. В этом примере сегментный регистр es инициализируется адресом кодового сегмента, помещая предопределенный символ @code в ax и затем пересылая это значение в es. Директива ASSUME показывает Turbo Assembler, куда в данный момент указывает es, используя для кодового сегмента название из малой модели памяти _TEXT. Наконец, mov загружает в ah значение v1. Хотя такая же строка имеется и в предыдущем примере вследствие директивы ASSUME, на самом деле ассемблируется команда

```
mov  ah, [es:v1]
```

Несмотря на то что v1 записана в кодовом сегменте и [es:v1], [cs:v1] корректно определяют одну и ту же переменную, ASSUME только позволяет ассемблеру автоматически вставлять команды переназначения.

Замечание

Сегментные имена типа _TEXT встречаются при описании директивы MODEL в вашем руководстве по Turbo Assembler. При использовании упрощенных моделей памяти, как объясняется в главе 2, обычно отпадает необходимость в ссылках на эти имена и применении директив ASSUME.

Выражения и операторы

Выражения в языке ассемблера преследуют одну цель — сделать программу легкой для понимания и, следовательно, простой для модификации. Например, может быть несколько макроопределений, связывающих значения с идентификаторами:

```
RecSize     EQU     10
NumRecs     EQU     25
```

Вы можете использовать эти идентификаторы в любом месте в выражениях, например для записи в память значения, равного произведению RecSize на NumRecs:

```
BufSize     dw      RecSize * NumRecs
```

При обработке этой директивы Turbo Assembler перемножает RecSize на NumRecs и записывает полученную константу (250) в переменную типа слова BufSize. Важно понимать, что эти вычисления происходят при ассемблировании, а не при исполнении программы. В ассемблированном коде все выражения заменяются константами. В языках высокого уровня выражения типа (Columns * 16) вычисляются при исполнении, возможно, с новым, введенным оператором, значением для переменной Columns. В языке ассемблера выражения приводятся к константным значениям при ассемблировании текста программы, а не при выполнении программы. Это отличие может вначале смущать, особенно если вы в большей степени владеете программированием на высоком, а не на низком уровне.

В табл. 5.2 представлены операторы для построения выражений в режиме Ideal Turbo Assembler, с помощью которых вы можете вычислять константные значения различных типов. Операторы для режима MASM (приведенные в руководстве по Turbo Assembler) имеют аналогичный вид. Не путайте операторы AND, OR, XOR и NOT с соответствующими мнемониками языка ассемблера. Мнемоники являются командами языка ассемблера, работающими во время исполнения программы. Операторы, используемые в выражениях, вычисляются во время ассемблирования. В этой и других главах вы увидите в действии многие из них.

Простые переменные

В предыдущих примерах программ из этой книги простые переменные создавались с помощью директив db и dw. Эти директивы принадлежат к семейству команд, имеющих одинаковое общее назначение — определить (в смысле зарезервировать) в памяти пространство под значения. Директивы отличаются только количеством пространства, которое они могут определить, и заданными вами типами начальных значений. В табл. 5.3 приведены все семь этих полезных директив в порядке возрастания резервируемого ими пространства. Даются также типичные примеры, хотя ими не ограничивается использование директив. Вы можете набирать эти директивы прописными либо строчными буквами. DB и db имеют одно и то же значение.

Резервирование большого пространства

Для резервирования большого количества пространства вы можете либо последовательно использовать несколько db, dw или других определяющих память директив, либо обычно более удобный оператор DUP. Этот оператор имеет следующий формат:

```
[Label] directive count DUP (expression [, expression]    )
[метка] директива счетчик DUP (выражение [, выражение]    )
```

Таблица 5.2 **Операторы для построения выражений**

Оператор	Описание	Оператор	Описание
()	Скобки	LT	Меньше, чем
*	Умножение	MASK	Битовая маска
/	Деление	MOD	Остаток от деления
+	Сложение/Унарный плюс	NE	Не равно
—	Вычитание/Унарный минус	NEAR	Ближний указатель
.	Элемент структуры	NOT	Логическое отрицание
:	Переназначение сегмента	OFFSET	Адрес смещения
?	Неинициализированные данные	OR	Логическое ИЛИ

Окончание таблицы 5.2

Оператор	Описание	Оператор	Описание
[]	Ссылка на память	PROC	Ближний/дальний указатель
AND	Логическое И	PTR	Размер выражения
BYTE	Указание размера байта	PWORD	32-битовый дальний указатель
CODEPTR	Размер адреса подпрограммы	QWORD	Учетверенный размер
DATAPTR	Размер, зависящий от модели	SEG	Адрес сегмента
DUP	Дублировать переменную	SHL	Сдвиг влево
DWORD	Указание размера двойного слова	SHORT	Короткий указатель
EQ	Равно	SHR	Сдвиг вправо
FAR	Дальний указатель	SIZE	Размер раздела
FWORD	Размер длинного слова	SMALL	16-битовое смещение
GE	Больше или равно	SYMTYPE	Символьный тип
GT	Больше, чем	TBYTE	Десятибайтовый размер
HIGH	Вернуть верхнюю часть	THIS	Ссылка на следующий раздел
LARGE	32-битовое смещение	TYPE	Тип раздела
LE	Меньше или равно	UNKNOWN	Удалить информацию о типе
LENGTH	Количество элементов	WIDTH	Ширина битового поля
LOW	Нижняя часть	WORD	Размер слова
		XOR	Исключающее ИЛИ

Таблица 5 3 **Директивы выделения памяти**

Директива	Название	Минимальное число выделяемых байтов	Характерное использование
db	Определить байт	1	Байты, строки
dw	Определить слово	2	Целые числа
dd	Определить двойное слово	4	Длинные целые
dp	Определить указатель	6	32-битовый указатель
df	Определить дальний указатель	6	48-битовый указатель
dq	Определить учетверенное слово	8	Вещественные числа
dt	Определить десять байтов	10	BCD-числа

Для создания многобайтового пространства начните строку с метки и директивы определения памяти и табл. 5.3. Следующим разместите счетчик, т.е. число, показывающее, сколько раз вы хотите повторить взятое скобки выражение. Ключевое слово DUP вставляется между счетчиком и выражением. Например, каждая из сл дующих директив резервирует 10-байтовые области памяти, обнуляя все 10 байт:

```
Ten1 dt     0                   ; Десять нулевых байтов
Ten2 db     10 DUP (0)          ; То же самое
```

Разделение нескольких выражений или константных значений запятыми приводит к их поочередному повтор нию, увеличивая суммарный размер выделяемого пространства во столько раз, сколько выражений или конста находится в скобках. Поэтому, несмотря на то что счетчик равен 10, следующая директива создает 20-байтовую п ременную — по десять повторяющихся байтов со значениями 1 и 2:

```
Twenty1     db      10 DUP (1,2)            ; 20 байт -1, 2, 1, 2, ..., 2
```

Для создания больших буферов, инициализированных константными значениями, вы можете использовать DUP в выражениях. Например, каждая из следующих директив резервирует 20-байтовые области и присваивает каждому байту значение 255:

```
Twenty2     db      10 DUP (2 DUP (255))      ; 20 байт со значениями 255
Twenty3     db      20 DUP (255)              ; То же самое
```

Эти же примеры работают со всеми директивами определения памяти, резервируя различные количества пространства. Чаще всего вы будете использовать db и dw для целых, строковых и байтовых переменных, оставляя другие директивы для применения в особых случаях, приведенных в табл. 5.3, хотя вы можете пользоваться этими директивами по своему усмотрению. Например, для создания 20-байтовой переменной, состоящей из одних нулей, можно либо, как и раньше, воспользоваться директивой db, либо применить директиву dt:

```
Twenty4     dt      2 DUP (0)
```

Из всех директив определения памяти только db имеет специальную возможность выделять пространство под символьные строки, записывая по одному ASCII-символу в каждом байте памяти. Следующий пример, заканчивающийся нулевым байтом, иллюстрирует типичную конструкцию, называемую ASCIIZ-строкой:

```
Astring             db      'String things', 0
```

Объединение строковых возможностей директивы db с оператором DUP является полезным приемом для заполнения буфера текстом, который затем легко находится в окне дампа памяти Turbo Debugger. Вы можете создать 1024-байтовый буфер следующим образом:

```
Buffer      db      128 DUP ('=Buffer=')            ; 1024 байт
```

DUP 128 раз повторяет 8-байтовую строку, взятую в скобки, резервируя таким образом в сумме 1024 байт. Находясь в Turbo Debugger, воспользуйтесь командой View-Dump, с помощью <F5> распахните окно на весь экран, нажмите <Alt+F10> и выберите Goto для просмотра сегмента данных с адреса DS:0000. Затем используйте клавишу <PgDn> для поиска в памяти этого или подобного буфера. Существуют и другие способы поиска переменных в Turbo Debugger, однако этот старый метод отладки все еще является полезным приемом.

Инициализированные и неинициализированные данные

Если вы знаете, что ваша программа по ходу своего исполнения присваивает переменным новые значения и поэтому их первоначальные значения вас не волнуют, вы можете определить их как неинициализированные переменные, т.е. такие, которые не имеют определенных значений перед запуском программы. Для этого используйте на месте определяющей память константы знак вопроса (?):

```
stuff       db      ?               ; Байт с неизвестным значением
moreStuff           dw      ?       ; Слово с неизвестным значением
anyStuff            dt      ?       ; Десять байтов с неизвестными значениями
```

Для получения большего неинициализированного пространства используйте знак вопроса в скобках на месте DUP-выражений, что является полезным приемом для создания больших буферов, например:

```
BigBuf      dp      8000 DUP (?)    ; 8000-байтовый буфер
```

При запуске программы созданный этой командой 8000-байтовый буфер состоит из байтов, не имеющих определенных значений. Когда DOS загружает вашу программу, в буфере будут находиться те значения, которые находились в памяти.

Замечание

При ассемблировании и компоновке программ с помощью команд tasm /zi <filename> и tlink /v <filename> Turbo Debugger обнуляет неинициализированные данные. Не полагайтесь на это в конечной программе. При ассемблировании и компоновке без этих ключей неинициализированные переменные имеют неопределенные значения.

Главная причина для использования неинициализированных переменных состоит в уменьшении размера исполняемого файла. Вместо записи бесполезных байтов на диск во время исполнения выделяется неинициализированное пространство. Для этого вы должны придерживаться двух правил:

- Размещайте все неинициализированные переменные последними в сегменте данных
- Предваряйте неинициализированные переменные директивой UDATASEG

Обычно проще всего разместить неинициализированные переменные последними в сегменте данных, после переменных с заданными начальными значениями. Когда это неудобно, используйте директиву UDATASEG, в соответствии с которой Turbo Assembler переставляет неинициализированные переменные за последнюю инициализированную переменную в сегменте данных, независимо от того, где они встречались в тексте программы. Например можно написать:

```
DATASEG
var1        db          1
var2        db          2
UDATASEG
array       db          1000 DUP (?)
DATASEG
var3        db          3
```

Директива UDATASEG размещает array в памяти после var3, как будто вы определили большую неинициализированную переменную не между двумя инициализированными переменными var2 и var3, а сразу после них. Без UDATASEG большой массив "попадался бы в ловушку" между var2 и var3, бесполезно увеличивая на 1000 байт размер вашего исполняемого файла

Замечание

Во многих свободно распространяемых исходных текстах программ на языке ассемблера неинициализированные переменные содержатся между другими инициализированными переменными. Получив такую программу, постараитесь переместить неинициализированные переменные в конец сегмента данных. Иногда это значительно уменьшает размер исполняемого файла

Имейте в виду, что при использовании UDATASEG не подразумевается, что переменные физически располагаются в памяти одна за другой, как это обычно и бывает. Некоторые программы предполагают, что в памяти переменные располагаются в порядке, определенном в тексте программы, и в этих случаях перемещение переменных является серьезной ошибкой. Чтобы избежать этой проблемы в своих собственных программах и прояснить исходный код, организовывайте свой сегмент данных следующим образом:

```
DATASEG
; инициализированные переменные
UDATASEG
; неинициализированные переменные
```

Строковые переменные

Несмотря на то что директива db может создавать строковые переменные, в языке ассемблера отсутствуют встроенные строковые команды для чтения и записи строк, удаления символов или сравнения одной строки с другой. В программе 5.1 приведены эти и другие процедуры на языке ассемблера. Однако сначала изучим несколько характерных строковых форматов.

Вероятно, самым общим строковым форматом является *ASCII$-строка* — набор ASCII-символов, заканчивающийся знаком доллара. Для создания ASCII$-строки используйте директиву db:

```
myString            db              Welcome to my program   '$'
```

Знак доллара необязательно отделять от главной строки — вы можете просто добавить $ между "m" и закрывающейся одинарной кавычкой. Представленное выше раздельное написание ставит своей целью только обратить ваше внимание на то, что знак доллара является не просто символом, а ограничителем строки. Для вывода этой строки на экран используйте 09-ю функцию DOS:

```
mov   dx  OFFSET myString       ; Строка размещена по адресу ds:dx
mov   ah, 09                    ; Определить 09 функцию DOS
int   21h                       ; Вызвать DOS для вывода строки
```

Первая строка определяет адрес смещения myString в сегменте данных программы, адресуемом ds. В ah записывается 09 — номер функции DOS "Вывод символьной строки", которая активизируется командой int 21h. Команда int (программное прерывание) работает аналогично вызову подпрограммы и после того, как DOS завершит исполнение определенной в ah функции, она возвращает управление в вашу программу на команду, следующую за int 21h. *Детально этот и другие типы прерываний обсуждаются в главе 10*

Основная проблема при использовании ASCII$-строк очевидна — не существует простого способа для вывода знака доллара! Кроме этого, в такие строки сложно считывать символы с клавиатуры или дисковых файлов. По этим причинам я редко использую ASCII$-строки. Вместо них я предпочитаю ASCIIZ-строки, заканчивающиеся нулевым байтом, — формат, используемый большинством компиляторов с языков высокого уровня C и C++. С помощью ASCIIZ-строки можно следующим образом создать сообщение об ошибке:

```
diskErr             db      "Disk read error!", 0
```

ASCIIZ-строки могут иметь любую длину — от одного до тысяч символов. Первый байт строки является либо ASCII-символом, либо нулевым байтом, также называемым *нулевым символом* ASCII. Если первый символ равен 0, тогда строка пустая. На этой основе с помощью оператора DUP легко создавать строковые переменные нулевой длины:

```
stringVar   db      81 DUP (0)              ; 80-символьная строка + нуль
```

При создании строки таким способом всегда устанавливайте счетчик DUP на единицу больше максимального числа символов, которое вы планируете записать в эту строку, оставляя место для нуля, которым она всегда должна заканчиваться. Единственным недостатком ASCIIZ-строк является то, что в DOS нет стандартных программ для чтения и записи строковых переменных в этом формате. Пакет для работы со строками, рассматриваемый далее в данной главе, устраняет этот недостаток, предоставляя подпрограммы, которые можно использовать для чтения и записи ASCIIZ-строк.

Кавычки в кавычках

Строки, описанные с помощью директивы db, могут браться либо в апострофы, либо в двойные кавычки. В наборе символов ASCII апостроф совпадает с одинарной закрывающейся кавычкой. На вашей клавиатуре эти символы печатаются в виде прямых вертикальных линий. Однако на экране, в зависимости от операционной системы и набора символов текстового редактора, символ апострофа может быть завернут влево.

Замечание

Не берите строку в открывающиеся одинарные кавычки (`), которые обычно получаются при нажатии клавиши, расположенной в верхнем левом углу клавиатуры. Использование открывающихся кавычек в качестве разделителей строк не допускается.

Существует несколько возможностей для включения кавычек внутрь строки. Простейший метод состоит в использовании разных типов кавычек для выделения всей строки и ее отдельных частей:

```
Quotedb     'When "quoting" speech, you can surround' , 0
Unquote     db      "the text with ' quote marks' like this " , 0
```

Двойные кавычки в первой строке вставляются в качестве простых символов так же, как и одинарные кавычки во второй строке. Другой способ состоит в повторении тех же кавычек, но уже в качестве разделителя строки. Это удобно при создании строк, содержащих как двойные, так и одинарные кавычки:

```
CrazyQuotes db      'This ' ' string ' ' contains four "quote" marks' , 0
```

Повторяющиеся одинарные кавычки вокруг слова *string* рассматриваются в качестве символов одинарных кавычек даже несмотря на то, что ими выделена и вся строка. Это же самое вы можете проделывать с двойными кавычками.

Локальные метки

До сих пор примеры программ использовали в кодовых сегментах метки Start: и Repeat:. Такие метки являются глобальными для программы, в которой они описаны. Другими словами, если пометить команду в начале программы меткой Here:, она будет доступной для call, jmp и других команд из любого места программы. При этом вы постоянно должны придумывать новые имена, чтобы избежать конфликта с ранее используемыми метками. Как показано в следующем примере, для коротких переходов это является серьезным неудобством:

```
    cmp     ax, 9               ; ax = 9 ?
    je      SkipIt              ; Пропустить строку и перейти вниз, если ax = 9
    add     cx, 10              ; Иначе прибавить к cx 10
SkipIt :
```

Короткие переходы, такие как je, на метку SkipIt: являются наиболее характерными для программирования на ассемблере. Скорее всего, никакие другие команды не будут выполнять переходы на эту же метку, следовательно SkipIt: не нужна нигде, кроме этого места. В больших программах могут осуществляться сотни и тысячи подобных переходов, заставляя вас изобретать для каждого из них все новые и новые имена. Для упрощения этого Turbo Assembler позволяет создавать *локальные метки*, существующие только в той части кода, в которой они действительно нужны.

Локальные метки идентичны любым другим меткам, за исключением того, что они начинаются с двух знаков @@ Примерами локальных меток могут служить такие имена, как @@10:, @@Here:, @@Tempo:, @@x . Локальная метка существует только до следующей нелокальной метки. Поскольку сюда же попадают метки, определенные в директивах PROC, выделение ваших подпрограмм с помощью PROC и ENDP приведет к тому, что локальные метки подпрограмм будут доступны только внутри их кода. Эти же метки затем вы снова можете использовать в любом другом месте программы. Следующий пример поможет прояснить ситуацию:

```
     jmp     There           ; переход на глобальную метку
@@10
     inc     ax
     cmp     ax, 10
     jne     @@10            ; переход на локальную метку вверх
There
     cmp     ax, 20
     je      @@10            ; переход на локальную метку вниз
     xor     cx, cx
@@10 .
```

Не пробуйте запустить этот пример на исполнение — это только иллюстрация. Первый jmp выполняет переход на глобальную метку There: — на глобальные метки вы можете переходить из любого места программы. Следующий — jne, переходит на локальную метку @@10:. Но на какую? Их существует две. Ответ — на первую метку @@10:, действие которой распространяется только до глобальной метки There:. Следовательно, jne может "видеть" только первую метку @@10:. По той же причине следующая команда je выполняет переход на вторую метку @@10., так как расположенная выше глобальная метка There: делает первую локальную метку недоступной. Локальные метки имеют следующие преимущества.

- Они экономят память при ассемблировании программ, позволяя Turbo Assembler повторно использовать память для других локальных меток. Глобальные метки постоянно хранятся в памяти, даже если они использовались только один раз. Локальные метки исчезают каждый раз при появлении новой нелокальной метки.
- Локальные метки делают программу более "читабельной", ясной. Например, при быстром просмотре программы их легко отличить от глобальных.
- Они способствуют уменьшению количества ошибок, усложняя написание программ с многочисленными переходами на большие расстояния из одного места программы в другое. Если вы выделяете ваши подпрограммы с помощью директив PROC и ENDP, то у вас не будет соблазна выполнить переход на метку в теле подпрограммы, что является известным источником ошибок.

Замечание

Подобно глобальным меткам локальные метки должны заканчиваться двоеточием, например @@ABC:. Когда команда ссылается на локальную метку, двоеточия не должно быть, например jmp @@ABC.

Пакет для работы с ASCIIZ-строками

В главе 4 были введены команды для работы со строками на процессоре 8086. Программа 5.1 (STRINGS.ASM) является пакетом, состоящим из 12 подпрограмм обработки ASCIIZ-строк, многие из которых используют эти строковые команды. В строках 18–29 содержатся названия и краткие описания подпрограмм пакета, который организован несколько иначе, чем до сих пор встречавшиеся вам тексты программ. STRINGS.ASM является *библиотечным модулем*, который должен ассемблироваться отдельно и затем компоноваться с другими программами. В отличие от предыдущих примеров программ модуль STRINGS сам по себе не исполняется. Вместо этого, как показано в следующих примерах, для STRINGS необходима головная программа, использующая его подпрограммы. Для ассемблирования STRINGS применяйте команду `tasm string`

Если вы предполагаете использовать для изучения работы программ строковый пакет Turbo Debugger, тогда применяйте команду `tasm /zi strings`

Имейте в виду, что при использовании опции /zi к ассемблированному коду добавляется отладочная информация, что в результате может приводить к значительному разрастанию конечного файла. Для уменьшения размера используйте предыдущую команду (без опции /zi).

Какую бы команду вы ни применяли, в результате получается файл с именем STRINGS.OBJ, содержащий строки ассемблированного кода, готовые к компоновке с вашей головной программой. Как это сделать, описано после текста программы STRINGS.ASM. Однако для исполнения других программ из этой книги, многие из которых требуют пакет STRINGS, вам необходимо записать код STRINGS.OBJ в *библиотечный файл*. Введите следующую команду, не обращая внимания на возможное предупреждение "STRINGS [was] not found in [the] library" (STRINGS в библиотеке не найдена): `tlib /E mta- +strings`.

Замечание

Если у вас нет жесткого диска, вы, естественно, захотите записать MTA.LIB на свою дискету с Turbo Assembler. Если дискета находится в дисководе A:, то вместо mta далее везде набирайте a:mta. После этого можно ассемблировать другие программы и модули, требующие MTA.LIB, не заботясь о том, имеются ли необходимые OBJ-файлы.

Результатом выполнения команды tlib является файл с именем MTA.LIB, содержащий пакет STRINGS. Опция */E* записывает в библиотечный файл *расширенный словарь*, ускоряя компоновку и обеспечивая TLINK дополнительной информацией о библиотечных символах. Команда `-+strings` указывает TLIB на то, что любые предыдущие версии STRINGS заменяются новыми OBJ-файлами. Позже вы добавите в MTA.LIB новые объектные файлы, которые значительно упростят ассемблирование и компоновку программ, использующих подпрограммы из STRINGS и других отдельно ассемблируемых модулей. При внесении любых изменений в программу STRINGS.ASM для обновления и замены старого объектного кода в файле MTA.LIB снова воспользуйтесь командами tasm и tlib.

Программа 5.1. STRINGS.ASM

```
1:    %TITLE "Процедуры для работы со строками -- Copyright 1989,1995 by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:
7:          CODESEG
8:
9:          PUBLIC MoveLeft, MoveRight, StrNull, StrLength
10:         PUBLIC StrUpper, StrCompare, StrDelete, StrInsert
11:         PUBLIC StrConcat, StrCopy, StrPos, StrRemove
12:
13:   ;---------------------------------------------------------------------
14:   ; Для создания файла STRINGS.OBJ воспользуйтесь командой TASM STRINGS.
15:   ; Для использования процедур добавьте в вашу программу операторы EXTRN <procedure>:PROC,
16:   ; где <procedure> является одним из следующих идентификаторов:
17:   ;
18:   ;     MoveLeft              - переместиться в сторону возрастания индексов
19:   ;     MoveRight             - переместиться в сторону убывания индексов
20:   ;     StrNull               - удалить все символы из строки
21:   ;     StrLength             - возвратить количество символов в строке
22:   ;     StrUpper              - преобразовать символы строки в прописные
23:   ;     StrCompare            - сравнить две строки в алфавитном порядке
24:   ;     StrDelete             - удалить символы из строки
25:   ;     StrInsert             - вставить символы в строку
26:   ;     StrConcat             - присоединить одну строку к другой
27:   ;     StrCopy               - скопировать одну строку в другую
28:   ;     StrPos                - найти в строке позицию подстроки
29:   ;     StrRemove             - удалить из строки подстроку
30:   ;
31:   ; После ассемблирования вашей программы скомпонуйте ее совместно с STRINGS.OBJ.
32:   ; Например, если ваша программа называется MYPROG, сначала выполните ассемблирование MYPROG
33:   ; в MYPROG.OBJ и затем скомпонуйте ее с помощью команды TLINK MYPROG+STRINGS.
34:   ;
35:   ; СТРОКОВЫЕ ПЕРЕМЕННЫЕ:
36:   ; Строка является простым массивом символов по одному символу на 8-битовый байт.
37:   ; Последним символом строки должен быть нулевой символ (ASCII 0).
38:   ; Пустая строка содержит только нулевой символ.
39:   ; Строковые переменные описываются следующим образом:
40:   ;
```

```
41:    ; STRING DB 81 DUP (0) ; 80-символьная строка + нуль
42:    ;
43:    ; СТРОКОВЫЕ КОНСТАНТЫ:
44:    ; Оставляйте всегда один дополнительный байт для нулевого окончания.
45:    ; Символьные константы (которые могут использоваться в качестве переменных)
46:    ; должны также заканчиваться соответствующим образом. Например:
47:    ;
48:    ; C1 db This is a test string., 0
49:    ;
50:    ; СЕГМЕНТНЫЕ РЕГИСТРЫ.
51:    ; В подпрограммах этого пакета подразумевается, что ES и DS адресуют
52:    ; один и тот же сегмент. Установите ES=DS до вызова любой из этих подпрограмм.
53:    ;-------------------------------------------------------------------
54:
55     ASCNull EQU 0 ; Символ ASCII нуля
56:
57     %NEWPAGE
58:    ;-------------------------------------------------------------------
59:    ; MoveLeft Переместить в памяти блок байтов влево (вниз)
60:    ;-------------------------------------------------------------------
61:    ; Вход.
62     ; si = адрес строки источника (s1)
63:    ; di = адрес строки назначения (s2)
64:    ; bx = индекс s1 (i1)
65:    ; dx = индекс s2 (i2)
66:    ; cx = количество перемещаемых байтов (count)
67.    ; Выход.
68:    ; count байтов из s1[i1] перемещены в позицию,
69.    ; начинающуюся с s2[i2]
70:    ; Регистры.
71:    ; не используются
72:    ;-------------------------------------------------------------------
73.    PROC MoveLeft
74:          jcxz   @@99                    ; Выход, если count = 0
75:          push   cx                      ; Сохранить изменяемые регистры
76:          push   si
77:          push   di
78:
79:          add    si, bx                  ; Индекс строки источника
80:          add    di, dx                  ; Индекс строки назначения
81:          cld                            , Автоувеличение si и di
82:          rep    movsb                   , Перемещать, пока cx <> 0
83:
84:          pop    di                      ; Восстановить регистры
85:          pop    si
86:          pop    cx
87.    @@99
88           ret                            ; Возврат в вызывающую программу
89     ENDP MoveLeft
90:    %NEWPAGE
91:    ;-------------------------------------------------------------------
92:    ; MoveRight Перемещение в памяти блока байтов вправо (вверх)
93:    ,-------------------------------------------------------------------
94:    ; Вход:
95:    ; (такой же, как в MoveLeft)
96:    ; Выход
97:    , (такой же, как в MoveLeft)
98:    ; Регистры:
99:    ; не используются
100:   ;-------------------------------------------------------------------
101    PROC MoveRight
102:         jcxz   @@99                    ; Выход, если count = 0
103:         push   cx                      ; Сохранить изменяемые регистры
104:         push   di
105:         push   si
106:
107:         add    si, bx                  ; Индекс строки источника
108:         add    di, dx                  ; Индекс строки назначения
109:         add    si, cx                  ; Настроиться на последний байт источника
110:         dec    si
111:         add    di, cx                  ; Настроиться на последний байт назначения
112:         dec    di
113:         std                            ; Автоуменьшение si и di
114:         rep    movsb                   ; Перемещать, пока cx <> 0
```

```
115:
116:        pop    si                          ; Восстановить регистры
117:        pop    di
118:        pop    cx
119:    @@99:
120:        ret                                ; Возврат в вызывающую программу
121:    ENDP MoveRight
122:    %NEWPAGE
123:    ;---------------------------------------------------------------
124:    ; StrNull Удалить все символы в строке
125:    ;---------------------------------------------------------------
126:    ; Вход:
127:    ; di = адрес строки (s)
128:    ; Выход:
129:    ; s[0] <- нулевой символ (ASCII 0)
130:    ; Регистры:
131:    ; не используются
132:    ;---------------------------------------------------------------
133:    PROC StrNull
134:        mov    [byte ptr di], ASCNull      ; Вставить нуль в s[0]
135:        ret                                ; Возврат в вызывающую программу
136:    ENDP StrNull
137:    %NEWPAGE
138:    ;---------------------------------------------------------------
139:    ; StrLength Подсчитать в строке количество ненулевых символов
140:    ;---------------------------------------------------------------
141:    ; Вход:
142:    ; di = адрес строки (s)
143:    ; Выход:
144:    ; cx = количество ненулевых символов в s
145:    ; Регистры:
146:    ; cx
147:    ;---------------------------------------------------------------
148     PROC StrLength
149:        push   ax                          ; Сохранить изменяемые регистры
150:        push   di
151:
152:        xor    al, al                      ; al <- искомый символ (null)
153:        mov    cx, 0ffffh                  ; cx <- максимальная глубина поиска
154:        cld                                ; Автоувеличение di
155:        repnz  scasb                       ; Искать al, пока [di]<>null & cx<>0
156:        not    cx                          ; Логическое отрицание cx
157:        dec    cx                          ; вычитание 1 дает длину строки
158:
159:        pop    di                          ; Восстановить регистры
160:        pop    ax
161:        ret                                ; Возврат в вызывающую программу
162.    ENDP StrLength
163:    %NEWPAGE
164:    ;---------------------------------------------------------------
165:    ; StrUpper Преобразовать символы строки в прописные
166:    ;---------------------------------------------------------------
167:    ; Вход:
168:    ; di = адрес преобразуемой строки (s)
169:    ; Выход:
170:    ; строчные символы строки преобразуются в прописные
171:    ; Регистры:
172:    ; не используются
173:    ;---------------------------------------------------------------
174:    PROC StrUpper
175:        push   ax                          ; Сохранить изменяемые регистры
176:        push   cx
177:        push   di
178:        push   si
179:        call   StrLength                   ; Установить cx = длина строки
180:        jcxz   @@99                        ; Выход, если длина = 0
181:        cld                                ; Автоувеличение si, di
182:        mov    si, di                      ; Установить si = di
183:    @@10:
184:        lodsb                              ; al <- s[si]; si <- si + 1
185:        cmp    al, a                       ; al >= a?
186:        jb     @@20                        ; Если нет, то продолжить поиск
187:        cmp    al, z                       ; al <= z?
188:        ja     @@20                        ; Если нет, то продолжить поиск
```

```
189:          sub    al, a-A                    ; Преобразовать строчные символы в прописные
190:    @@20.
191:          stosb                             ; s[di] <- al; di <- di + 1
192:          loop   @@10                       ; cx <- cx-1; цикл, если cx <> 0
193:    @@99:
194:          pop    si                         ; Восстановить регистры
195:          pop    di
196:          pop    cx
197:          pop    ax
198:          ret                               ; Возврат в вызывающую программу
199:    ENDP StrUpper
200:    %NEWPAGE
201:    ;---------------------------------------------------------------
202:    ; StrCompare Сравнить две строки
203:    ;---------------------------------------------------------------
204:    ; Вход:
205:    ; si = адрес строки 1 (s1)
206:    ; di = адрес строки 2 (s2)
207:    ; Выход:
208:    ; набор флагов для условных переходов jb, jbe,
209:    ; je, ja, or jae.
210:    ; Регистры:
211:    ; не используются
212:    ;---------------------------------------------------------------
213:    PROC StrCompare
214:          push   ax                 ; Сохранить изменяемые регистры
215:          push   di
216:          push   si
217:          cld                       ; Автоувеличение si
218.    @@10:
219:          lodsb                     ; al <- [si], si <- si + 1
220:          scasb                     ; Сравнить al и [di]; di <- di + 1
221:          jne    @@20               ; Выход, если найдены неравные символы
222:          or     al, al             ; al=0? (т.е. конец s1)
223:          jne    @@10               ; Если нет, то переход, иначе выход
224:    @@20:
225.          pop    si                 ; Восстановить регистры
226:          pop    di
227:          pop    ax
228:          ret                       ; Возвратить флаги в вызывающую программу
229:    ENDP StrCompare
230:    %NEWPAGE
231:    ;---------------------------------------------------------------
232:    ; StrDelete Удалить символы из строки
233:    ;---------------------------------------------------------------
234:    ; Вход:
235:    ; di = адрес строки (s)
236:    ; dx = индекс (i) первого удаляемого символа
237:    ; cx = количество удаляемых символов (n)
238:    ; Выход:
239:    ; В строке, начиная с позиции s[i], удалены n символов
240:    ; Замечание: предотвращает удаление конца строки
241:    ; Регистры:
242:    ; не используются
243:    ;---------------------------------------------------------------
244:    PROC StrDelete
245.         push   bx                  ; Сохранить изменяемые регистры
246:         push   cx
247:         push   di
248:         push   si
249:
250:    ; bx = SourceIndex
251:    ; cx = Count / Len / CharsToMove
252:    ; dx = Index
253:
254:         mov    bx, dx              ;Присвоить индекс строки bx
255:         add    bx, cx              ; SourceIndex <- index + count
256:         call   StrLength           ; cx <- length(s)
257:         cmp    cx, bx              ; length > index?
258:         ja     @@10                ; Если да, то переход на удаление символов
259:         add    di, dx              ; иначе вычислить индекс конца строки
260:         mov    [byte ptr di], ASCNull      ; и вставить null
261:         jmp    short @@99          ; Переход на выход
262:    @@10:
```

```
263.          mov     si, di                  ; Источник = назначение
264:          sub     cx, bx                  ; CharsToMove <- Len - SourceIndex
265:          inc     cx                      ; Прибавить единицу для нулевого символа
266:          call    MoveLeft                ; Переместить символы на место удаленных
267:    @@99:
268:          pop     si                      ; Восстановить регистры
269:          pop     di
270:          pop     cx
271:          pop     bx
272:          ret                             ; Возврат в вызывающую программу
273:    ENDP StrDelete
274:    %NEWPAGE
275:    ;---------------------------------------------------------------
276:    ; StrInsert Вставить одну строку в другую строку
277:    ;---------------------------------------------------------------
278:    ; Вход:
279:    ; si = адрес строки 1 (s1)
280:    ; di = адрес строки 2 (s2)
281:    ; dx = индекс вставки для s2 (i)
282:    ; Замечание: s2 должна быть достаточно большой, чтобы вместить в себя (s1)!
283:    ; Выход:
284:    ; символы из строки s1 вставлены в строку s2[i]
285:    ; s1 не изменяется
286:    ; Регистры:
287:    ; не используются
288:    ;---------------------------------------------------------------
289.    PROC StrInsert
290:          push    ax                      ; Сохранить изменяемые регистры
291:          push    bx
292:          push    cx
293:
294:    ; ax = LenInsertion
295:    ; cx = CharsToMove
296:
297:          xchg    si, di                  ;Переставить si и di
298:          call    StrLength               ; и найти длину s1
299:          xchg    si, di                  ; Восстановить si и di
300:          mov     ax, cx                  ; Сохранить length(s1) в ax
301:
302:          call    StrLength               ; Найти длину s2
303:          sub     cx, dx                  ; cx <- length(s2) - i + 1
304:          inc     cx                      ; cx = (CharsToMove)
305:
306:    ; bx = индекс s1
307:
308:          push    dx                      ; Сохранить индекс (dx) и si
309:          push    si
310:          mov     si, di                  ; Поместить в si адрес s2
311:          mov     bx, dx                  ; Установить индекс s1 в dx (i)
312:          add     dx, ax                  ; Установить индекс s2 в i+LenInsertion
313:          call    MoveRight               ; Подготовить место для вставки
314:          pop     si                      ; Восстановить индекс (dx) и si
315:          pop     dx
316:
317.          xor     bx, bx                  ; Установить индекс s1 (источника) в нуль
318.          mov     cx, ax                  ; Установить cx в LenInsertion
319:          call    MoveLeft                ; Вставить s1 целиком в s2
320:
321:          pop     cx                      ; Восстановить регистры
322:          pop     bx
323:          pop     ax
324:          ret                             ; Возврат в вызывающую программу
325:    ENDP StrInsert
326:    %NEWPAGE
327:    ;---------------------------------------------------------------
328:    ; StrConcat Объединение двух строк
329:    ;---------------------------------------------------------------
330:    ; Вход:
331:    ; si = адрес строки источника (s1)
332:    ; di = адрес строки-назначения (s2)
333:    ; Замечание : s2 должна быть достаточно большой, для того чтобы поместить в себе (s1)!
334:    ; Выход:
335:    ; символы строки s1 добавлены в конец s2
336:    ; Регистры:
```

```
337:  ; не используются
338:  ;---------------------------------------------------------------
339:  PROC StrConcat
340:        push   bx                  ; Сохранить изменяемые регистры
341:        push   cx
342:        push   dx
343:
344:  ; dx = назначение s2
345:
346:        call   StrLength           ; Найти длину строки-назначения (s2)
347:        mov    dx, cx              ; Установить dx в индекс конца строки
348:        xchg   si, di              ; Переставить si и di
349:        call   StrLength           ; Найти длину строки-источника (s1)
350:        inc    cx                  ; Учесть нулевое окончание
351:        xchg   si, di              ; Восстановить si и di
352:        xor    bx, bx              ; Индекс источника = 0
353:        call   MoveLeft            ; Скопировать строку-источник в назначение
354:
355:        pop    dx                  ; Восстановить регистры
356:        pop    cx
357:        pop    bx
358:        ret                        ; Возврат в вызывающую программу
359   ENDP StrConcat
360   %NEWPAGE
361:  ;---------------------------------------------------------------
362:  ; StrCopy Скопировать одну строку в другую
363:  ;---------------------------------------------------------------
364:  ; Вход:
365:  ; si = адрес строки-источника (s1)
366:  ; di = адрес строки-назначения (s2)
367:  ; Выход:
368:  ; Символы из s1 скопированы в s2
369:  ; Замечание: s2 должна по меньшей мере иметь длину Length(s1)+1
370:  ; Регистры:
371:  ; не используются
372:  ;---------------------------------------------------------------
373   PROC StrCopy
374         push   bx                 ; Сохранить изменяемые регистры
375.       push   cx
376:        push   dx
377:
378:        xchg   si, di              ; Переставить si и di
379:        call   StrLength           ; Найти длину строки-источника (s1)
380:        inc    cx                  ; Добавлением единицы учесть нулевое окончание
381:        xchg   si, di              ; Восстановить si и di
382:        xor    bx, bx              ; Индекс строки-источника = 0
383:        xor    dx, dx              ; Индекс строки-назначения = 0
384:        call   MoveLeft            ; Копировать источник в назначение
385:
386:        pop    dx                  ; Восстановить регистры
387:        pop    cx
388:        pop    bx
389:        ret                        ; Возврат в вызывающую программу
390:  ENDP StrCopy
391:  %NEWPAGE
392:  ;---------------------------------------------------------------
393:  ; StrPos Найти вхождение в строку подстроки
394:  ;---------------------------------------------------------------
395:  ; Вход:
396:  ; si = адрес искомой подстроки
397:  ; di = адрес проверяемой целевой строки
398:  ; Выход:
399:  ; если zf = 1, тогда dx = индекс подстроки
400:  ; если zf = 0, тогда подстрока не найдена
401:  ; Замечание: если zf = 0, то значение в dx не имеет никакого смысла
402:  ; Регистры:
403:  ; dx
404:  ;---------------------------------------------------------------
405:  PROC StrPos
406:        push   ax                  ; Сохранить изменяемые регистры
407:        push   bx
408:        push   cx
409:        push   di
410:
```

```
411:          call   StrLength           ; Найти длину целевой строки
412:          mov    ax, cx              ; Сохранить length(s2) в ax
413:          xchg   si, di              ; Переставить si и di
414:          call   StrLength           ; Найти длину подстроки
415:          mov    bx, cx              ; Сохранить length(s1) в bx
416:          xchg   si, di              ; Восстановить si и di
417:          sub    ax, bx              ; ax = последний возможный индекс
418:          jb     @@20                ; Выход, если длина строки < длины искомой подстроки
419:          mov    dx, 0ffffh          ; Инициализировать dx в -1
420:    @@10:
421:          inc    dx                  ; От i = 0 до последнего возможного индекса
422:          mov    cl, [byte bx + di]  ; Сохранить символ s[bx] в cl
423:          mov    [byte bx + di], ASCNull     ; Заменить символ нулем
424:          call   StrCompare          ; Сравнить si с измененным di
425:          mov    [byte bx + di], cl  ; Восстановить замененный символ
426:          je     @@20                ; Перейти, если найдено совпадение, dx=index, zf=1
427:          inc    di                  ; Иначе увеличить индекс целевой строки
428:          cmp    dx, ax              ; В случае равенства все позиции проверены
429:          jne    @@10                ; Продолжать поиск
430:
431:          xor    cx, cx              ; Подстрока не найдена. Установить zf = 0,
432:          inc    cx                  ; чтобы показать отсутствие совпадений
433:    @@20:
434:          pop    di                  ; Восстановить регистры
435:          pop    cx
436:          pop    bx
437:          pop    ax
438:          ret                        ; Возврат в вызывающую программу
439:    ENDP StrPos
440:    %NEWPAGE
441:    ;---------------------------------------------------------------
442:    ; StrRemove Удалить из строки подстроку
443:    ;---------------------------------------------------------------
444:    ; Вход:
445:    ; si = адрес удаляемой подстроки
446:    ; di = адрес строки, из которой удаляется подстрока
447:    ; Выход:
448:    ; если zf = 1, тогда подстрока удалена
449:    ; если zf = 0, тогда подстрока не найдена
450:    ; Замечание: строка в si не изменяется
451:    ; Замечание: если zf = 0, тогда строка в di не изменяется
452:    ; Регистры:
453:    ; не используются
454:    ;---------------------------------------------------------------
455:    PROC   StrRemove
456:          push   cx                  ; Сохранить изменяемые регистры
457:          push   dx
458:
459:          call   StrPos              ; Найти подстроку, установив dx=индекс
460:          jne    @@99                ; Выход, если подстрока не найдена
461:          pushf                      ; Сохранить флаг zf
462:          xchg   si, di              ; Переставить si и di
463:          call   StrLength           ; Найти длину подстроки
464:          xchg   si, di              ; Восстановить si и di
465:          call   StrDelete           ; Удалить cx символов из di[dx]
466:          popf                       ; Восстановить флаг zf
467:    @@99:
468:          pop    dx                  ; Восстановить регистры
469:          pop    cx
470:          ret                        ; Возврат в вызывающую программу
471:    ENDP   StrRemove
472:
473:          END                        ; Конец модуля STRINGS.ASM
```

Программирование по частям

До перехода к описаниям подпрограмм модуля STRINGS вы должны знать о том, как можно объединять STRINGS с программами и другими объектными модулями. Модули, подобные STRINGS, могут описывать подпрограммы, переменные и константы для совместного использования с программами и другими модулями. Объектные модули являются независимыми пакетами, которые ассемблируются отдельно от других текстов и компонуются с главной программой, создавая на диске конечный исполняемый файл.

Разделение больших программ на модули приводит к значительной экономии времени. Вместо переассемблирования одного и того же кода вы можете записать его в отдельный модуль, ассемблировать на диск и только затем компоновать со своей программой. При модификации существующих программ переассемблируйте только измененные модули. Модули способствуют также упрощению сложных команд, позволяя вам сконцентрироваться на меньших и более простых для понимания фрагментах кода. Кроме того, можно записать объектные модули в библиотечные файлы, сделав свои любимые подпрограммы мгновенно доступными для новых программ.

В исходном виде отдельные модули слабо отличаются от текста главной программы. Обратившись к тексту программы 5.1, можно увидеть, что первые строки такие же, как и в предыдущих программах (например, см. программу 4.7), за исключением директивы STACK. Стековый сегмент может определяться только в главной программе, в отдельных модулях это никогда не происходит.

Другое отличие состоит в том, что в отдельных модулях отсутствует инициализация регистров сегмента данных и возврат управления в DOS по окончании программы. Вместо этого, как вы можете видеть, программа 5.1 содержит набор процедур, помеченных директивами PROC и ENDP. Текст заканчивается директивой END, но при этом к ней не добавляется метка точки входа, как это должно быть сделано в файле с главной программой (например, см строку 52 в программе 4.7). Точка входа может определяться только в главной программе.

Директива PUBLIC

Строки 9–11 STRINGS описывают названия процедур как символы с директивами PUBLIC. Эти символы совпадают с названиями, используемыми в качестве меток в заголовках процедур PROC. Любой символ (идентификатор), который вы хотите экспортировать из модуля во внешний мир, должен описываться, как показано выше, директивой PUBLIC. Используя отдельные директивы PUBLIC, можно описывать символы по одному или в строках, разделяя их запятыми, как в этом примере. Символы могут быть названиями числовых констант, описанных с помощью знака равенства (=), переменными или кодовыми метками. Константы, описанные с помощью EQU, экспортироваться не могут.

Замечание

В режиме IDEAL EQU-константы при ассемблировании трактуются в качестве текста, тогда как константы, определенные с помощью знака равенства (=), рассматриваются как значения. В режиме MASM некоторые EQU-константы – числовые и поэтому могут экспортироваться. Другие типы EQU-констант должны оставаться собственностью модуля. Это не означает, что режим Ideal налагает дополнительные ограничения на экспортируемые символы. Просто в режиме Ideal вы всегда знаете, какие константы являются экспортируемыми.

Все другие символы, не описанные в PUBLIC (например, ASCNull в строке 55), — локальные и не могут использоваться другими программами. Локальные символы могут повторяться в модулях и программах, не конфликтуя с символами, описанными в других модулях. Вне модуля видимы только символы, описанные в директивах PUBLIC. Обратите внимание, что символы в директиве PUBLIC не имеют идентификаторов типа данных — ничто не указывает на то, чем являются эти символы. В следующих примерах будет показано, что ответственность за это несет импортирующая символы программа.

Замечание

Некоторые программисты описывают отдельные директивы PUBLIC перед каждым заголовком PROC. Я предпочитаю собирать все PUBLIC-символы в одном месте в начале файла, где их можно легко найти и изменить. Оба метода — правильные и приводят к одному и тому же результату.

Ассемблирование и компоновка отдельных модулей

Ассемблировать отдельные модули очень просто. Наберите tasm module, где module — имя ассемблируемого текстового файла. При этом нет необходимости в определении после имени файла расширения ASM. Для ассемблирования модуля с дальнейшим применением Turbo Debugger воспользуйтесь командой tasm -zi module, которая включает в OBJ-файл дополнительную информацию, позволяющую Turbo Debugger находить переменные и подпрограммы по именам.

Для ассемблирования программ, использующих тексты из отдельных модулей, применяйте эти же команды. Можно ассемблировать главную программу и все ее модули в любом порядке, при этом не требуется, чтобы на диске находились объектные файлы каких-либо других модулей. После ассемблирования всех модулей вы получите на диске набор OBJ-файлов. Следующий шаг состоит в сборке этих отдельных кусочков для создания конечного исполняемого файла. Например, если ваша главная программа называется THEMEAT.ASM, модули – LETTUCE.ASM и MUSTARD.ASM, тогда сначала вы должны ассемблировать каждый модуль:

```
tasm lettuce
tasm themeat
tasm mustard
```

Эти шаги можно выполнить в любом порядке. Если в вашем текущем каталоге содержатся только ASM-файлы, вы можете использовать более простую команду tasm *.ASM для ассемблирования всех этих файлов сразу. После ассемблирования вы получите на диске файлы THEMEAT.OBJ, LETTUCE.OBJ и MUSTARD.OBJ. Затем можно скомпоновать эти объектные файлы командой

```
tlink themeat lettuce mustard
```

Первое имя после tlink должно относиться к главной программе. Последующие имена ссылаются на отдельные модули, используемые программой. Имена модулей могут записываться в любом порядке и разделяться пробелами. (Вы можете также использовать знак плюс — tlink themeat+lettuce+mustard.) В результате компоновки все модули и главная программа будут объединены в один окончательный файл с исполняемым кодом, в этом примере — в THEMEAT EXE Имя результата совпадает с именем первого объектного файла после TLINK, но его расширение изменяется на EXE. Для определения другого имени, например SANDWICH.EXE, добавьте запятую и новое имя после списка объектных файлов:

```
tlink themeat lettuce mustard, sandwich
```

Имя нового EXE-файла должно быть отделено от списка объектных файлов запятой. При компоновке TLINK создает файл отображения (карту использования памяти), содержащий информацию о символах и их адресах в конечном коде. Пока вы не определите другое имя, файл отображения будет иметь то же имя, что и EXE-файл, но заканчиваться расширением MAP. Следующая команда ассемблирует объектные файлы (представленные здесь в виде <obj-files>) и создает как SANDWICH.EXE, так и SANDWICH.MAP:

```
tlink <obj-files>, sandwich, sandwich
```

Если вы не хотите создавать файл отображения, тогда перед списком объектных файлов используйте опцию /x. Это приводит к экономии пространства на диске и немного ускоряет процесс компоновки. Turbo Debugger не требует файла отображения, однако он необходим некоторым отладчикам и утилитам для работы с исходными текстами других компаний. Кроме того, вы можете захотеть сохранить файл отображения как часть программной документации. Следующая команда запрещает генерацию файла отображения:

```
tlink /x <obj-files>
```

Последней опцией, которую можно определить в строке TLINK, является имя одного или нескольких библиотечных файлов, содержащих отдельно ассемблированные объектные модули. Между именами библиотек должны вставляться пробелы. Например, если у вас имеется две библиотеки, BUTTER.LIB и BREAD.LIB, то полная команда компоновки примет вид

```
tlink <obj-files>,,, butter bread
```

Расширения LIB определять не нужно. Обратите внимание на три запятые после списка объектных файлов. Эти запятые показывают Turbo Assembler, что для пропущенных имен нужно использовать имена, установленные по умолчанию. Без запятых Turbo Linker не мог бы знать, что BUTTER и BREAD являются файлами библиотек, и ошибочно считал бы их OBJ-файлами. Вместо необязательных имен, которые вы не определяете, должны добавляться запятые. Вместе с квадратными скобками, представляющими список опций, полный синтаксис TLINK 6.0 имеет следующий вид:

```
tlink [options] objfiles, exefile, mapfile, libfiles, deffile, resfiles
```

В этой команде objfiles обозначает ассемблированные объектные файлы, exefile — имя конечного исполняемого файла, mapfile содержит список общих символов и другую информацию, libfiles ссылаются на библиотеки типа MTA.LIB (находящиеся на диске), которые содержат многочисленные объектные файлы, deffile является файлом определения компоновщика, resfiles определяет ресурсы, объединяемые в конечном коде. Два последних раздела, deffile и resfiles, требуются только для Windows-программ.

Пакет для ввода-вывода строк

Несмотря на то что модуль STRINGS можно использовать автономно, для вывода и чтения строк с клавиатуры необходим другой модуль. Этот второй модуль упрощает эксперименты со STRINGS и является полезен сам по себе. Выполните ассемблирование программы 5.2, STRIO.ASM и добавьте его объектный код в ваш файл библиотеки MTA.LIB с помощью команд:

```
tasm /zi strio
tlib /E mta -+strio
```

Чтобы затем запускать главную программу в Turbo Debugger, здесь, а также при ассемблировании STRINGS вы должны использовать опцию /zi. Для уменьшения размера получаемого кода воспользуйтесь командой tasm strio и переинсталируйте STRIO в библиотеке. При выполнении команды tlib не обращайте внимания на возможное предупреждение о том, что STRIO не найден в библиотеке. Это предупреждение вы увидите только при первом добавлении STRIO к MTA.LIB. С этого момента в MTA.LIB вы имеете два модуля: STRINGS и STRIO. Для просмотра содержимого файла библиотеки введите команду

```
tlib mta, con
```

Для того чтобы распечатать результат на принтере, замените con на prn. Кроме того, вы можете записать результат в файл на диске с помощью команды вида tlib mta, temp.txt. Будьте внимательны — TLIB без предупреждения сотрет с диска существующий там файл с таким же названием.

Программа 5.2. STRIO.ASM

```
1:      %TITLE "Процедуры ввода-вывода строк -- by Tom Swan"
2:
3:              IDEAL
4:
5:              MODEL  small
6:
7:
8:      ;----- Макроопределения
9:
10      BufSize         EQU             255         ; Максимальный размер строки (<=255)
11      ASCnull         EQU             0           ; ASCII нуль
12      ASCcr           EQU             13          ; ASCII-символ возврата каретки
13      ASClf           EQU             10          ; ASCII-символ прогона строки
14
15:
16:     ;----- Структура буфера для DOS функции 0Ah
17:
18:     STRUC StrBuffer
19      maxlen db BufSize               ; Максимальная длина буфера
20:     strlen db 0  ; Длина строки
21.     chars db BufSize DUP (?)        ; Буфер для StrRead
22.     ENDS strBuffer
23:
24.
25.             DATASEG
26:
27:     buffer StrBuffer <>             ; Переменная буфера для ReadStr
28:
29:
30:             CODESEG
31:
32:     ;----- Из: STRINGS.OBJ
33:
34              EXTRN StrLength:proc, StrCopy:proc
35:
36              PUBLIC StrRead, StrWrite, StrWrite2, NewLine
37:
38:     %NEWPAGE
39:     ;---------------------------------------------------------------
40:     ; StrRead Чтение строки с редактированием
41:     ;---------------------------------------------------------------
42:     , Вход:
43:     ; di = адрес строки назначения
```

```
44:     ; cl = максимальная длина строки, включая нулевой ограничитель
45:     ; Замечание: если cl = 0, StrRead ничего не делает
46:     ; Замечание: переменная буфера должна иметь длину cl+1 байт
47:     ; Замечание: длина строки ограничена 255 символами
48:     ; Выход:
49:     ; Строка, скопированная со стандартного устройства ввода в ваш буфер
50:     ; Регистры:
51:     ; не используются
52:     ;-----------------------------------------------------------------
53:     PROC StrRead
54:           or cl, cl              ; cl = 0?
55:           jz @@99                ; Если да, переход на exit
56:
57:           push ax                ; Сохранить изменяемые регистры
58:           push bx
59:           push dx
60:           push si
61:
62:           mov [buffer.maxlen], cl ; Установить байт maxlen
63:           mov ah, 0ah            ; Функция DOS чтения из буфера
64:           mov dx, offset buffer.maxlen     ; Поместить адрес struc в ds:dx
65:           int 21h                ; Вызов DOS для чтения строки
66:           xor bh, bh             ; Обнулить старший байт bx
67:           mov bl, [buffer.strlen] ; bx = # chars в буфере
68:           mov [bx+buffer.chars], ASCnull   ; Изменить cr на null
69:           mov si, offset buffer.chars      ; Поместить адрес буфера в si
70:           call StrCopy                     ; Скопировать chars в строку пользователя
71:
72:           pop si                           ; Восстановить регистры
73:           pop dx
74:           pop bx
75            pop ax
76.     @@99:
77:           ret                              ; Возврат в вызывающую программу
78.     ENDP StrRead
79:     %NEWPAGE
80:     ;-----------------------------------------------------------------
81:     ; StrWrite/StrWrite2 Записать строку на стандартное устройство вывода
82:     ;-----------------------------------------------------------------
83:     ; Вход:
84:     ; di = адрес строки (s)
85:     ; cx = количество записываемых символов (только для StrWrite2 )
86:     ; Выход:
87:     ; строка s копируется на стандартное устройство вывода
88:     ;
89:     ; Регистры:
90:     ; cx (только для StrWrite)
91:     ;-----------------------------------------------------------------
92:     PROC StrWrite
93            call StrLength           ; Установить cx=длина строки
94:
95:     PROC StrWrite2                 ; Переменная точка входа
96:           push ax                  ; Сохранить изменяемые регистры
97:           push bx
98:           push dx
99:
100:          mov bx, 1                ; Стандартное устройство вывода
101:          mov dx, di               ; Строка адресуется в ds:dx
102:          mov ah, 40h              ; DOS запись в файл или на устройство
103:          int 21h                  ; Вызов DOS (on ret ax=# chars written)
104:
105:          pop dx                   ; Восстановить регистры
106:          pop bx
107:          pop ax
108:          ret                      ; Возврат в вызывающую программу
109.    ENDP StrWrite2                 ; Конец промежуточной процедуры
110:    ENDP StrWrite                  ; Конец обычной процедуры
111:
112:    %NEWPAGE
113:    ;-----------------------------------------------------------------
114:    ; NewLine Перейти на новую строку на стандартном устройстве вывода
115:    ;-----------------------------------------------------------------
116:    ; Вход:
117:    ; нет
```

```
118:    ; Выход:
119:    ; на устройство вывода посылается символ возврата каретки и прогона строки
120:    ; Регистры:
121:    ; ah, dl
122:    ;---------------------------------------------------------------
123.    PROC NewLine
124             mov ah, 2               ; Функция записи символа DOS
125             mov dl, ASCcr           ; Загрузить в dl символ возврата каретки
126.            int 21h                 ; Вывести символ возврата каретки
127             mov dl, ASClf           ; Загрузить в dl символ прогона строки
128:            int 21h                 ; Вывести символ прогона строки
129             ret                     ; Возврат в вызывающую программу
130:    ENDP NewLine
131:
132:            END                     ; Конец модуля STRIO
```

Процедуры модуля STRIO

В модуле STRIO содержатся три процедуры, используемые многими программами из этой книги, а именно:

- StrRead — Чтение ASCIIZ-строки
- StrWrite — Вывод ASCIIZ-строки
- NewLine — Переход на новую строку

Для первых двух процедур требуется представлять строки в форме ASCIIZ — той же форме, которая используется в модуле STRINGS. Все три подпрограммы применяют стандартные файлы ввода-вывода DOS — обычно клавиатуру и дисплей. Как будет показано в следующих программах, существуют более быстрые способы отображения текста на экране, чем SrtWrite. Но этот небольшой модуль полезен для чтения и записи строковых данных.

Использование модуля STRIO

Вы легко разберетесь в трех процедурах STRIO.ASM (см. программу 5.2). За исключением структуры данных в строках 18–22, вы уже встречались с большинством элементов из этого текста. В данном разделе объясняется, как использовать в ваших собственных программах подпрограммы из STRIO для чтения и записи ASCIIZ-строк в стандартные файлы входа-выхода, обычно с клавиатуры (вход) на дисплей (выход). (Мы снова вернемся к этой программе в главе 6.)

StrRead (39–78)

Поместите адрес любой ASCIIZ-переменной, состоящей от 1 до 255 символов плюс один байт для нулевого ограничителя в es:di. В принципе, ASCIIZ-строка может иметь любую длину. Однако вследствие ограничений DOS максимальный размер читаемой строки составляет только 255 символов. Установите также в cl максимально допустимое число вводимых символов. Если cl равно 0, StrRead не выполняет никаких действий. Ниже показано, как вы можете использовать StrRead для обеспечения ввода с клавиатуры некоторых данных:

```
DATASEG
response      db       81 dup (0)        , 80-символьная строка + 0
CODESEG
mov  di, OFFSET response                 ; Поместить адрес response в es:di
mov  cl, 80                              , Разрешить ввод от 0 до 80 символов
call StrRead                             , Прочитать строку
```

Обратите внимание, что несмотря на то что строковая переменная имеет длину 81 байт, cl устанавливается в 80. Таким образом, оставляется 1 байт для нулевого ограничителя в конце строки. Не забывайте это важное правило – вы должны оставить место для байта окончания строки. StrRead вызывает в строке 65 функцию DOS 0Ah, которой необходима структура данных, определенная в строках 18–22 (это будет объяснено позже в главе 6).

Подпрограмма StrWrite (строки 80–110)

Для того чтобы послать ASCIIZ строку на стандартное устройство вывода (обычно дисплей), вызовите StrWrite, установив в es:di адрес строки. Если длина строки вам известна, вы можете записать ее в cx и вместо StrWrite вызвать StrWrite2 — пример *вложенной процедуры*. Обратите внимание, как процедура в строках 95–109 вкладывается внутрь внешней процедуры, расположенной в строках 92–110. Отличие между двумя процедурами состоит в том, что после вызова StrWrite2 значение в cx не изменяется, а после вызова StrWrite cx равняется длине строки. Вложенная процедура определяет *переменную точку входа* в подпрограмму.

Замечание

Не обязательно определять переменную точку входа с помощью вложенной процедуры — можно просто добавить новую метку и вызвать переход на эту метку. Использование вложенных процедур проясняет поведение программы и является предпочтительным, даже если допустимы другие подходы.

Типичное использование StrWrite состоит в том, чтобы вывести на экран приглашение программы:

```
cr   EQU     13                              ; ASCII символ возврата каретки
lf   EQU     10                              ; ASCII символ прогона строки
DATASEG
welcom              db      cr, lf, ' Welcome to Noware Land '
                    db      cr, lf, ' (C) 1998 by Nobody, Inc.' , cr, lf, lf, 0
CODESEG
mov  ax, @data
mov  ds, ax                                 ; Инициализация ds
mov  es, ax                                 ; Установить es = ds
mov  di, OFFSEET welcom                     ; Поместить адрес строки в di
call StrWrite                               ; Вывести строку
```

Здесь есть ряд интересных моментов, которые заслуживают более пристального внимания. Сначала два макроопределения приписывают символам cr и lf ASCII-значения возврата каретки и прогона строки. Затем в сегменте данных создается строковая переменная с добавлением при необходимости cr и lf. Таким способом в языке ассемблера гибкий оператор db позволяет вам легко добавлять управляющие символы непосредственно к строкам. Поскольку обе переменные записываются в памяти последовательно, в действительности здесь есть только одна строковая переменная, несмотря на то, что она описана в двух отдельных директивах db. Единственный нулевой ограничитель располагается в конце второй строки; поэтому это одна строка, а не две. Заметьте также, что строка заканчивается возвратом каретки и двумя прогонами строки. Первый возврат каретки перемещает курсор на левый край дисплея. После этого последовательные прогоны строки дважды перемещают курсор вниз (или прокручивают дисплей). При этом нет необходимости добавлять другой возврат каретки. Возможность обрабатывать такие гибкие структуры данных является одной из привлекательных особенностей языка ассемблера.

Три первые команды в кодовом сегменте этого примера инициализируют регистры ds и es адресом сегмента данных программы. Выполняйте эти шаги во всех программах, использующих модуль STRIO (так же как и другие модули из этой книги). После этого команда mov помещает в di адрес строки welcome. Затем для вывода двухуровневой строки вызывается StrWrite.

Код StrWrite в STRIO чрезвычайно простой. В строках 102–103 вызывается функция DOS 40h с cx, равным длине строки, bx, равным 1 (представляющей стандартный файл вывода DOS), и адресом строки в ds:dx. Остальные команды сохраняют и восстанавливают изменяемые регистры (кроме cx).

Подпрограмма NewLine (строки 113–130)

Последней процедурой в STRIO является NewLine. Эта процедура вызывается для перехода на новую строку экрана. Процедура работает, передавая коды управляющих символов возврата каретки и прогона строки через регистр dl в функцию DOS 2, которая записывает по одному символу в стандартное устройство вывода. Заметьте, что эта процедура изменяет ah и dl.

Компоновка модулей в программу

Теперь вы располагаете двумя полезными пакетами для чтения и записи ASCII-строк — процедурами, которые широко используются другими программами из этой книги и найдут применение в ваших программах. Но вы должны ввести еще одну программу, иллюстрирующую применение процедур из отдельных модулей. Для этих целей выполните ассемблирование и компоновку ECHOSTR.ASM (программа 5.3), используя команды:

```
tasm        /zi echostr
tlinkechostr,,,mta
```

Как было описано ранее, три запятые, вставленные на месте пропущенных имен в команде tlink, информируют Turbo Linker, что mta является именем файла библиотеки. Кроме этого, если вы хотите исполнять ECHOSTR под управлением Turbo Debugger, то должны использовать опцию /zi. Для запуска программы из DOS просто наберите echostr. Затем наберите любую строку символов и нажмите Enter. В доказательство того, что модуль STRIO работает, вы должны увидеть ту же строку, повторенную после введенной. Ясно, что это очень простой пример. Позже вы увидите, что с помощью STRINGS и STRIO можно сделать значительно больше.

Программа 5.3. ECHOSTR.ASM

```
1:      %TITLE "Проверка чтения строки -- by Tom Swan"
2:
3:              IDEAL
4:
5:              MODEL  small
6:              STACK  256
7:
8:      MaxLen          EQU             128         ; 128-символьная строка
9:      cr      EQU     13                          ; ASCII-символ возврата каретки
10:     lf      EQU     10                          ; ASCII-символ прогона строки
11:
12:
13:             DATASEG
14:
15:     exCode          db              0
16:     welcome         db              Welcome to Echo-String, cr, lf
17:                     db              Type any string and press Enter, cr,lf,lf, 0
18:     testString      db              MaxLen DUP (0), 0 ; MaxLen символов + 0
19:
20:
21:             CODESEG
22:
23:     ;----- Из STRIO.OBJ:
24:
25:             EXTRN StrRead:proc, StrWrite:proc, NewLine:proc
26:
27:     Start:
28:             mov     ax, @data               ; Поместить в DS
29:             mov     ds, ax                  ; адрес сегмента данных
30:             mov     es, ax                  ; Сделать ds=es
31:
32:             mov     di, offset welcome      ; Вывести приглашение
33:             call    StrWrite
34:
35:             mov     di, offset testString   ; di = адрес testString
36:             mov     cx, MaxLen      ; cx = максимальная длина
37:             call    StrRead         ; Читать строку с клавиатуры
38:             call    NewLine         ; Перейти на новую строку
39:             call    StrWrite        ; Вывести строку на дисплей
40:
41:     Exit:
42:             mov     ah, 04Ch        ; функция DOS: выход из программы
43:             mov     al, [exCode]    ; Возвратить значение кода выхода
44:             int     21h             ; Вызов DOS. Останов программы
45:
46:             END     Start           ; Конец программы с указанием точки входа
```

Новые особенности в ECHOSTR.ASM

Пакеты STRINGS и STRIO требуют, чтобы ds и es адресовали один и тот же сегмент данных. Это требование удовлетворяется в строке 30 ECHOSTR путем записи в es значения, присвоенного в предыдущей строке ds. EXESHELL.ASM (см. программу 2.3) включает в себя эту команду, так что при необходимости вы не забудете об этом важном шаге.

В строке 25 ECHOSTR показано, как импортируются символы, описанные директивами PUBLIC из других модулей. Директива EXTRN сообщает Turbo Assembler, что данные символы — внешние для этой программы и что их действительные адреса и значения будут определяться позже, при сборке программы и всех модулей. Используя EXTRN, нужно иметь в виду следующее.

- Все символы директивы EXTRN первоначально должны описываться в директивах PUBLIC в модулях, компонуемых к программе. В противном случае Turbo Linker выдаст сообщение об ошибке.
- В директиве EXTRN определяется *тип* символа. В строке 25 все три символа имеют тип proc, в результате ассемблер "понимает", что они являются метками подпрограмм и, следовательно, могут использоваться в качестве целей в командах call и jmp. Вы можете также описать кодовые метки как near или far, вынуждая ассемблер генерировать либо внутрисегментные, либо внесегментные вызовы подпрограмм.

- При описании внешних переменных допускаются типы: byte, word, dword, fword, pword, dataptr, qword и tbyte, соответствующие директивам определения данных из табл. 5.3. Нужно вставлять директивы EXTERN в подходящие сегменты данных, обычно сразу после DATASEG. Если вы случайно опишите внешние переменные внутри CODESEG, компоновщик не сможет вычислить для них правильные адреса.
- Внешние числовые макроопределения всегда имеют тип abs (абсолютное значение). Эти EXTRN символы разумно размещать до директивы DATASEG.
- Объектные модули также могут использовать директивы EXTRN. Например, см. строку 34 в STRIO.ASM (в программе 5.2), которая импортирует две процедуры из модуля STRINGS. Любой модуль может экспортировать свои собственные символы через директиву PUBLIC и импортировать внешние символы из любого другого модуля с помощью директив EXTRN.
- Когда несколько модулей (включая главную программу) ссылаются на одни и те же символы, то к конечному файлу компонуется единственная копия из объектного модуля, содержащего эти символы (идентификаторы).
- Нужно описывать только символы, использующиеся в программе. Не требуется описывать все символы, встречающиеся в директиве PUBLIC модуля. Однако, несмотря на это, Turbo Linker всегда компонует к конечному коду все модули целиком, даже если вы использовали из них только одну или две процедуры (или другие описания).
- Для создания конечного исполняемого файла вы должны компоновать все модули, содержащие символы, описанные в директивах EXTRN других модулей программы. Запись объектных кодов в файлы библиотек делает компоновку более простой, позволяя Turbo Linker просматривать только необходимые объектные модули. Библиотека целиком не компонуется в ваш код, компонуются только необходимые записанные в ней модули.

Упрощенный пример использования внешних переменных

Несколько простых примеров помогут прояснить детали при экспортировании и импортировании макроопределений, переменных и меток процедур. (Вы можете не вводить и не исполнять эти примеры, хотя при желании можете это сделать.) Ниже приводится текст объектного модуля:

```
      IDEAL
      MODEL   small
      PUBLIC  Maximum
Maximum =     100h
              DATASEG
              PUBLIC  counter
counter       db      0fh
              CODESEG
              PUBLIC  subroutine
PROC          subroutine
              ret
ENDP          subroutine
      END
```

После переключения в режим Ideal и задания малой модели памяти в модуле описывается общее числовое макроопределение Maximum. В сегменте данных в качестве общей также описана байтовая переменная count. В кодовом сегменте экспортируется третий символ, subroutine — метка процедуры. Обратите внимание на то, что директивы PUBLIC размещены в разумных местах. Главная программа может импортировать эти символы следующим образом:

```
        IDEAL
        MODEL small
        STACK 256
EXTRN   Maximum abs
        DATASEG
EXTRN   counter byte
        CODESEG
EXTRN   Subroutine proc
Start   mov     ax, @data               ; Поместить в ds адрес
        mov     ds, ax                  ; сегмента данных
        mov     ax  Maximim             , Установить ax = Maximum
        mov     cl, [counter]           ; Получить значение счетчика
        mov     bx, OFFSET counter      ; Получить адрес счетчика
        call    Subroutine              , Вызвать внешнюю подпрограмму
Exit
        mov     ax, 04C00h              ; Функция DOS  Выход из программы
        int     21h                     ; Вызов DOS  Останов программы
        END     Start                   ; Конец программы/точка входа
```

Внимательно посмотрите на расположение директив EXTRN, в особенности для counter и Subroutine. Эти символы помещены в сегменты данных и кода, так что компоновщик способен правильно определять их адреса. Числовое макроопределение имеет тип abs. Db определяет переменную типа байт. Если в другом модуле переменная будет описана с помощью dw, тогда она будет иметь тип слово. Метке Subroutine задан тип proc. В тексте основной программы эти символы используются так, как будто они непосредственно в ней и описывались. Если вы хотите ассемблировать и запускать конечную программу в Turbo Debugger, в предположении, что имя модуля – MODULE.ASM, а главной программы — MAIN.ASM, используйте следующие команды:

```
tasm /zi module
tasm /zi main

tlink /v main module

td   main
```

Изучение модуля STRINGS

Теперь, когда вы знаете, как писать, ассемблировать и компоновать отдельные модули, вы готовы к изучению 12 процедур из программы 5.1, STRINGS. Все процедуры в STRINGS оперируют ASCIIZ-строками — последовательностями символов, которые заканчиваются нулевым байтом. Кроме того, две процедуры, MoveLeft и MoveRight, можно использовать и для неограниченных строк из байтов. В интересах скорости — и поэтому в духе программирования на языке ассемблера — в STRINGS выполняется только минимальная проверка ошибок. Например, при копировании одной строки в другую вы отвечаете за то, чтобы строка назначения вместила в себя копируемые символы.

Следующие главы описывают каждую их подпрограмм STRINGS. Нумерация строк совпадает с нумерацией в тексте программы 5.1.

Замечание

Модули STRINGS и STRIO подразумевают, что сегментные регистры ds и es адресуют в памяти один и тот же сегмент данных. Если до вызова любой из подпрограмм этих модулей вы не установите ds=es, это, скорее всего, приведет к серьезным ошибкам.

MoveLeft (58–89)

MoveRight (91–121)

Обе эти подпрограммы перемещают байты из одной области памяти в другую. MoveLeft и MoveRight вызываются другими программами для копирования строк, присоединения одной строки к другой и вставки в строку символов. Вы можете также использовать эти подпрограммы для заполнения буферов и копирования блоков памяти из одного места в другое.

И MoveLeft, и MoveRight используют повторяющуюся строковую команду movsb в строках 82 и 114. Другие команды сохраняют и восстанавливают значения регистров и подготавливают si, di и флаг df к перемещению блоков памяти. Обратите внимание на то, как команда jcxz в строке 74 предотвращает случайное перемещение 65536 байт в случае, если cx равно 0, переходя при этом на локальную метку @@99: в строке 87. По этим же причинам аналогичная команда в строке 102 осуществляет переход на строку 119. (Помните, что область действия локальных меток распространяется вверх и вниз до следующей нелокальной метки; поэтому метка @@99: может использоваться повторно в строках 193, 267 и 467.)

Замечание

При просмотре в Turbo Debugger повторяющихся строковых команд типа rep movsb для полного выполнения команды нажмите <F8>. Для пошагового исполнения нажимайте <F7>.

В комментариях к MoveLeft и MoveRight в строках 58–72 и 91–100 описываются необходимые регистры и объясняются последствия вызова каждой подпрограммы. MoveLeft применяет те же входные параметры, что и MoveRight. При использовании этой или любой другой процедуры из STRINGS всегда внимательно проверяйте раздел "Регистры" в заголовке процедуры, в котором описаны все потенциально изменяемые регистры. В данном случае MoveLeft и MoveRight являются дружественными — они возвращают все первоначальные значения регистров без изменений. Однако это верно не для всех процедур. Кстати, директивы %NEWPAGE, расположенные в начале каждой процедуры в STRINGS, записывают в файл листинга символы прогона страницы, если вы создаете его с помощью команды /l Turbo Assembler. Это делает листинг аккуратным, каждый раз новая процедура начинается с отдельной страницы.

Вызывайте MoveLeft, записав в si адрес *строки-источника*, а в di — адрес *строки-назначения*, т.е. того места, в которое вы хотите скопировать байты. Если вы хотите скопировать байты не в начало строки, занесите значения соответствующих индексов в bx и dx. Например, чтобы скопировать 20-байтовую переменную v1 в середину 40-байтовой переменной v2, вы должны написать:

```
DATASEG
v1   db      ' 12345678901234567890 ', 0      ; 20-байтовая строка
v2   db      40 dup (0)                       ; 40-байтовая строка
CODESEG
mov  si, OFFSET v1                            ; Присвоить исходный адрес v1
mov  di, OFFSET v2                            ; Задать адрес назначения
mov  bx, 0                                    ; Установить индекс источника (v1[0])
mov  dx, 10                                   ; Установить индекс назначения (v2[10])
mov  cx, 20                                   ; Определить количество перемещаемых байтов
call MovLeft                                  ; Переместить байты из v1[0] в v2[10]
```

MovLeft копирует байты слева (из младших адресов) направо (в старшие адреса). Когда адреса источника и назначения перекрываются (это может быть, например, при перемещении байтов внутри одной строковой переменной), *направление перемещения может иметь важное значение, как показано в следующем примере:*

```
mov  [buffer], 0              ; Установить первый байт буфера в 0
mov  si, OFFSET buffer        ; si указывает на начало буфера
mov  di, si                   ; di указывает на тот же буфер
xor  bx, bx                   ; Установить индекс источника в 0
mov  dx, 1                    ; Установить индекс назначения на второй байт
mov  cx, (LENGTH buffer) - 1  ; Установить счетчик = длина буфера -1
call MoveLeft                 ; Заполнить буфер нулями
```

Первая команда mov устанавливает первый байт буфера в 0. В регистры si и di заносится один и тот же адрес смещения этой переменной. После этого индекс источника bx устанавливается в 0 (индекс позиции первого байта в buffer), а dx — в 1 (индекс второго байта в buffer). Затем, используя оператор LENGTH, который возвращает количество байтов в переменной, cx устанавливается на 1 меньшим, чем длина buffer. Вызов MovLeft с этими параметрами копирует байт из buffer[0] в buffer[1], затем из buffer[1] в buffer[2] и т.д., заполняя весь буфер значением, первоначально располагавшимся по индексу 0.

Замечание

Для заполнения буфера значением байта лучше использовать повторяющиеся команды stosb или stosw. MovLeft выполняется не так быстро, как отдельная строковая команда!

Если адреса источника и назначения перекрываются и вы не хотите размножать байты источника в назначении, тогда вам *необходимо начинать перемещение в противоположные концы переменных. Это действие выполняется* процедурой MoveRight, добавляющей cx–1 к si и di (см. строки 109–112). Затем std подготавливает автоматическое уменьшение si и di при выполнении повторяющейся строковой команды в строке 114. Это предотвращает сдвиг байтов источника на назначение, что особенно полезно в случае перемещения байтов в более высокие адреса переменной — например, при выполнении вставки в большой текстовый буфер. Ниже приводится еще несколько советов, которые помогут вам максимально использовать возможности MoveRight и MoveLeft:

- Если адреса источника и назначения перекрываются и источник расположен в более низких адресах, чем назначение, *для предотвращения случайного дублирования данных источника в назначении вызывайте* MoveRight.
- Если адреса источника и назначения перекрываются и источник расположен в более высоких адресах, чем назначение, для предотвращения случайного дублирования данных источника в назначении вызывайте MoveLeft.
- Если адреса источника и назначения не перекрываются, всегда вызывайте MoveLeft. Эта процедура выполняется несколько быстрее, так как ей не нужно увеличивать si и di на cx–1.

StrNull (123–136)

Для удаления символов строки, расположенной по адресу в di, вызывайте процедуру StrNull. StrNull записывает в начало строки нулевой байт (строка 134). Исследуйте выражение в скобках, повторенное для удобства ниже:

```
mov  [byte ptr di], ASCNull
```

Оператор byte ptr информирует Turbo Assembler о том, что di адресует 8-битовый байт. Если di должно адресовать 16-битовое слово, замените byte на word. Слово ptr — необязательное, и вы можете привести строку к виду

```
mov  [byte di] ASCNull
```

Для использования StrNull запишите адрес строковой переменной в di и вызовите процедуру. Например, вы можете применять StrNull для установки в 0 длины неинициализированной строковой переменной:

```
UDATASEG
string      db 81 dup (?)     , Неинициализированная 80-символьная строка
CODESEG
mov  di, OFFSET string        , Записать адрес строки в di
call StrNull                  ; Установить длину строки в 0
```

Поскольку в ASCIIZ-строке нулевой длины на месте первого символа находится нулевой ограничитель и StrNull не нужно знать максимальный размер строки, следовательно, эта процедура может работать с переменными *произвольной длины*.

StrLength (138–162)

StrLength вычисляет количество символов, записанных в ASCIIZ-строке, адресуемой di. StrLength возвращает это значение в регистре cx, после чего оно может передаваться в другие процедуры, для которых необходимо знать длину строки (Обратите внимание, что строка 146 информирует вас об изменении cx. Если вы используете cx для иных целей и вам необходимо вызвать StrLength, нужно любым способом сохранить cx, например в стеке, а позже восстановить его первоначальное значение.)

Предположим, вы хотите перейти в конец программы, если после некоторых действий длина строки стала равна 0. Вы можете написать:

```
DATASEG
string      db      ' Sample user response string '  0
CODESEG
mov  di, OFFSET string, Поместить адрес строки в di
call StrLength        , Установить в cx значение длины строки
or   cx, cx             cx = 0 ?
jz   Exit             , Переход на Exit, если cx = 0
```

StrLength иллюстрирует использование строковой команды scasb, введенной в главе 4. Scasb применяется для поиска определенного значения в строке байтов, для строки, состоящей из слов, используется команда scasw Искомое значение должно находиться в al при поиске байта или в ax при поиске слова Начальный адрес для поиска поместите в es:di, а в cx установите максимальное количество проверяемых байтов. И scasb и scasw сравнивают байт в al или слово в ax с данными в es.di, эффективно выполняя команду cmp. С этими командами можно придумать циклы для поиска значений байтов или слов:

```
cld                   , Подготовка к автоувеличению di
mov  di  buffer       , Поместить адрес буфера в es di
mov  cx  lenbuffer    , Установить cx = длина буфера
mov  al, searchval    , Установить al = искомое значение
repne scas            , Повторять до нахождения одинаковых байтов
je   Match            , Совпадение найдено
jmp  NoMatch          , Совпадение не найдено
```

В этом коде префикс repne повторно выполняет scasb, пока байты в al и в es:di "не равны (ne)", уменьшая cx и останавливаясь, если cx=0. По окончании проверяется, закончился ли поиск нахождением совпадающих байтов, выполняются переходы на соответствующие метки (не показанные здесь). Так как scas устанавливает те же флаги, что и cmp, после поиска можно использовать условные переходы, как показано выше.

В результате повторения операции в строке 155 процедуры StrLength выполняется просмотр ASCIIZ-строки Он заканчивается, если байт в es:di становится равным 0 либо cx уменьшается до 0, предотвращая таким образом зацикливание, которое может происходить, если вы случайно передали в процедуру неинициализированную строку или в сегменте данных отсутствует нулевой байт, что печально, однако возможно.

Циклические вычисления

В строках 156–157 StrLength используется странный метод для вычисления количества повторений строковой команды. Метод требует, чтобы cx инициализировался в 0FFFFh (–1 в двоичном дополнительном коде), что показано в строке 153. После повторяющегося поиска (строка 155) количество повторений предыдущей операции вычис

ляется с помощью простой логической команды. Чтобы понять, как это работает, сначала рассмотрим классический метод вычисления количества повторений выполнения строковой команды:

```
mov  cx  -1                    , Инициализация cx в -1
repnz scasb                    ; Повторять пока [di] <> 0 , cx <>0
not  cx                        , Произвести логическое отрицание cx
```

В результате этих действий cx будет равно количеству повторений команды repnz scasb. Чтобы понять, почему это так, исследуем результат одной итерации Так как cx первоначально равен –1 и если scasb останавливается после одного повторения, то значение в cx становится равным –2, или FFFE в шестнадцатеричном представлении. (Префикс repnz при каждом повторении scasb вычитает из cx 1.) Абсолютное значение (значение в дополнительном коде) –2, естественно, равняется 2, что много для одной итерации. Поэтому для получения правильного ответа вы должны вычесть из абсолютного значения 1 (2–1=1 итерация), но, вспомнив из главы 2, что значение в дополнительном коде равно логическому отрицанию плюс 1, в качестве конечного результата можно просто рассматривать логическое отрицание Кстати, это также верно и для положительных чисел Если после просмотра cx равно 32766, то это означает, что было выполнено 32769 циклов. Чтобы проверить это, запишите в двоичной системе логическое отрицание от 32766 (7FFEh).

Так как классический метод учитывает еще нулевой ограничитель в конце строки, то, исходя из назначения StrLength, для получения конечного ответа в строке 157 из cx вычитается 1.

StrUpper (164–199)

StrUpper преобразует строчные буквы строки в прописные, не изменяя другие, не входящие в алфавит, символы. Поместите адрес строки в di и вызовите процедуру следующим образом:

```
DATASEG
lc   db              ' abcdefghijklmnopqrstuvwxyz ', 0
CODESEG
mov  di  OFFSET lc     , Поместить адрес строки в es di
call StrUpper          , Преобразовать символы в прописные
```

В процедуре применяются две распространенные строковые команды — loadsb и stosb, введенные в главе 4 вместе с новой командой loop (см. строку 192) Команда loop вычитает из cx 1 и, если cx еще не равно 0, переходит на заданный целевой адрес. В StrUpper целевым адресом является локальная метка @@10: в строке 183. Loop за один шаг эффективно выполняет ту же работу, что и следующие команды:

```
dec  cx                        , cx <- cx-1
jnz  Target                    ; Переход на Target если cx <> 0
```

Две другие разновидности команды loop имеют вид loopne/loopnz и loope/loopz. В мнемонических парах для одних и тех же команд используются различные названия, что сделано по тем же причинам, по которым имеют двойные имена такие команды, как repne/repnz и jnz/jne. Loopne и loopnz также выполняют переход на целевую метку, если после уменьшения cx значение в этом регистре еще не равно 0. Одновременно проверяется флаг zf, предварительно установленный или очищенный при предыдущем сравнении. Например, для поиска в буфере байта, равного 0FFh, можно использовать следующий код:

```
    mov     cx, LENGTH buffer
    mov     bx, OFFSEET buffer + LENGTH buffer
@@20
    dec     bx
    cmp     [BYTE bx], 0ffh
    loopne  @@20
    je      Match
```

В регистре cx устанавливается максимальное число проверяемых байтов, в bx — адрес предпоследнего элемента буфера. Затем три команды после метки @@20: уменьшают указатель на индексы bx, сравнивая каждый байт, расположенный по этому адресу, с 0ffh Команда loopne вычитает из cx 1 и переходит на @@20: только в том случае, если cx не равно 0 и cmp не нашел байт 0ffh. После окончания поиска по команде je осуществляется переход на метку Match (здесь не показана), если в буфере было найдено значение 0ffh. Аналогично, для поиска байтов или слов, не совпадающих с определенным значением, можно использовать команду loope.

Как видите, loop, loope и loopne удобны для осуществления циклического поиска. Возвращаясь к модулю STRINGS, после размещения в cx длины строки, в случае, если она равна нулю, происходит немедленный выход из процедуры StrUpper (см. строки 175–182), строки 183–192 используют для поиска операторы lodsb, stosb и loop, проверяя каждый символ с помощью двух команд cmp. Если найдена строчная буква, в строке 189 ее ASCII-значение дополняется до прописной.

Обратите внимание, как в sub al, 'a'–'A' из al вычитается выражение, означающее числовую разность между строчными и прописными ASCII-буквами. В языке ассемблера символы эквивалентны числам, и, как показывает этот пример, вы можете прямо использовать их в числовых выражениях. (Программисты на Basic и Pascal могут найти это немного странным.) Помните, что Turbo Assembler вычисляет эти и другие константные выражения еще во время ассемблирования, а не при исполнении программы. Те же действия можно выполнить, использовав команду sub al, 32, но тогда назначение этой команды будет менее понятным.

StrCompare (201–229)

Как вы можете увидеть из текста процедуры StrCompare, сравнение двух строк в алфавитном порядке удивительно простое дело. Для использования StrCompare поместите адреса двух строк в si и di и вызовите процедуру. Затем для проверки результатов сравнения используйте одну из команд беззнакового условного перехода — jb, jbe, je, ja, jae. Например, для сравнения строк s1, s2 и перехода на метку StringsLess, если s1 в алфавитном смысле меньше s2, можно написать:

```
DATASEG
s1          db      80 dup (0)      ; строковые ASCIIZ-переменные,
s2          db      40 dup (0)      ; содержащие произвольные символы
CODESEG
mov  si, OFFSET s1                  ; Поместить адрес первой строки в si
mov  di, OFFSET s2                  ; Поместить адрес второй строки в di
call StrCompare                     ; Сравнить s1 и s2
jb   StringsLess                    ; Перейти, если s1 < s2
jg   StringsGreater                 ; Перейти, если s1 > s2
                                    ; Здесь s1 = s2
```

Как показано здесь, вместо повторного вызова StrCompare вы можете использовать несколько команд переходов. Строковые переменные могут иметь разные размеры, и их длина может равняться 0. Обе строки должны заканчиваться нулевым байтом, иначе поведение StrCompare будет непредсказуемым.

Процедура работает, используя в строках 219–220 операторы loadsb и scasb, поодиночно загружая символы в al и сравнивая их ASCII-значения с символами в [es:di]. Обе команды также на 1 увеличивают si и di (вследствие предыдущей команды cld в строке 217). В строке 221 по команде jne в случае неудачного сравнения осуществляется выход из цикла. Очевидно, что результат сравнения в алфавитном порядке становится известным, как только в строках найдены два первых отличающихся символа. Команда or в строке 222 проверяет, равняется ли al нулю, показывая, что в ds:si обнаружен конец первой строки до достижения в es:di конца второй строки. Если конец не достигнут, команда jne в строке 223 продолжает сравнение; в противном случае цикл заканчивается.

Возможно вы захотите узнать, что случится, если строка в es:di короче первой строки в ds:si. В этом случае, предположив, что до конца короткой строки все символы в обеих строках были равны, команда scasb в строке 220 сравнивает символ из первой строки в ds:si с нулевым ограничителем второй строки в es:di. Очевидно, что символы различны, поэтому в результате получается, что более длинная строка в алфавитном смысле больше короткой. Другими словами, в сравнении участвует нулевой ограничитель, не являющийся символом строки, однако результат получается верным. Чтобы понять все это при простом чтении текста и программы, нужны некоторые усилия. Для получения лучшей картины работы StrCompare исполните в Turbo Debugger небольшую тестовую программу, сравнивая одну с другой различные строки. В частности, наблюдайте при выполнении цикла в строках 218–223 за значениями флагов cf и zf.

StrDelete (231–273)

StrDelete удаляет, начиная с любой позиции строки, один или несколько символов и упрощает такую работу, как выделение расширения из имени файла или ограничение выводимой информации определенным числом символов. Поместите в di адрес любой ASCIIZ строковой переменной, установите в dx индекс первого удаляемого символа, в cx — количество удаляемых символов. Например, в следующем фрагменте из строки удаляется фраза "and tigers" и один пробел:

```
DATASEG
string      db              ' Lions and tigers and bears, oh my ! ', 0
CODESEG
mov  di, OFFSET string      , Поместить адрес строки в es:di
mov  dx, 6                  ; Индекс символа "a" из слова and
mov  cx, 11                 ; Количество символов в "and tigers"
call StrDelete              , Удалить 11 символов, начиная со string[6]
```

Замечание

Несмотря на то что StrDelete не позволяет удалить больше символов, чем их существует в строке, dx должен адресовать на символ в строке или указывать на нулевой ограничитель. Другими словами, dx должен быть меньшим или равным длине строки. Игнорирование этого правила может привести к разрушению других переменных или кода в памяти.

StrDelete работает в два этапа. Строки 259–261 обрабатывают ситуацию, если вы попытаетесь удалить больше символов, чем располагает строка. В этом случае команда mov вставляет нуль в конец новой строки и осуществляется выход. В строках 263–266 происходит удаление символов с помощью вызова MoveLeft, при этом si и di адресуют одну и ту же строку. Таким образом, конец строки (включая нулевой ограничитель) перемещается на место удаляемых символов. Обратите внимание на оператор short (строка 261), добавленный к целевому адресу jmp, который сообщает Turbo Assembler, что метка @@99: находится на расстоянии, не большем 127 байт. Это помогает ассемблеру генерировать более эффективный код для jmp по сравнению с более дальними переходами.

StrInsert (275–325)

StrInsert предназначена для вставки символов одной строки в другую, начиная с любой позиции. Поместите в si адрес строки источника (строки, вставляемой в другую), в di — адрес строки назначения (строки, в которую будет осуществлена вставка). В dx запишите индекс символа в строке назначения, после которого вы хотите сделать вставку. Помните, что первый символ имеет индекс 0. Строка источник не изменяется. В следующем примере строка "tab-A" вставляется в другую строку:

```
DATASEG
destination  db      'Insert into slot-B              ', 0
source       db      ' tab-A ', 0
CODESEG
mov  si, OFFSET source                  ; Поместить адрес строки источника в ds:si
mov  di, OFFSET destination             ; Поместить адрес строки назначения в es:di
mov  dx, 7                              ; dx = индекс "I" в строке назначения
call StrInsert                          ; Вставить источник в назначение
```

Замечание

Чтобы предотвратить изменение в памяти кода и других переменных, строка назначения должна быть достаточно большой для размещения в ней вставляемой строки источника. Выполняйте это условие при использовании StrInsert.

К этому моменту, применяя комментарии текста программы, вы должны понимать все команды в StrInsert. Подсказка: вызовом MoveRight в строке 313 в строке назначения создается место, необходимое для вставки. Затем, при вызове MoveLeft в строке 319, на это место копируются символы строки источника. Другие команды выполняют инициализацию регистров, необходимую для перемещения этих двух блоков.

StrConcat (327–359)

StrConcat выполняет присоединение одной строки к другой. Строка назначения в es:di должна быть достаточно большой, чтобы в ней уместились ее символы плюс символы строки источника из ds:si. Строка-источник при этом не изменяется. В следующем примере, строка "This is" изменяется на "This is the end!":

```
DATASEG
source       db              ' the end! ', 0
destination  db              ' This is ',0,'          ' , 0
CODESEG
mov  si, OFFSET source       ; Поместить адрес источника в ds:si
mov  di, OFFSET destination  ; Поместить адрес назначения в es:di
call StrConcat               ; Присоединить источник к назначению
```

StrConcat вызывает StrLength в строках 346 и 349, первый раз — для поиска конца строки назначения, второй — для определения длины строки-источника. Обратите внимание, как команда xchg в строке 348 для этих вызовов подпрограмм временно переставляет местами значения si и di. После этих действий присоединение выполняется с помощью вызова MoveLeft в строке 353.

StrCopy (361–390)

StrCopy копирует одну строковую переменную в другую, которая, по меньшей мере, должна иметь длину, равную длине первоначальной строки плюс 1 байт для нулевого ограничителя. Использовать эту процедуру просто: поместите в si адрес строки источника, а в di – назначения. Затем вызовите StrCopy. При этом в строке назначения будут изменены соответствующие символы. Например, чтобы скопировать символы из одной строки в неинициализированную строковую переменную, можно написать:

```
DATASEG
s1   db          ' Original string', 0
s2   db          80 dup (?)          ; Неинициализированная строковая переменная
CODESEG
mov  si, OFFSET s1                   ; Поместить адрес строки источника в si
mov  di, OFFSET s2                   ; Поместить адрес строки назначения в di
call StrCopy                         ; Скопировать строку s1 в s2
```

Разобраться в коде StrCopy несложно. Команда xchg в строке 378 переставляет si и di, для того чтобы StrLength, которая использует di, могла возвращать длину строки-источника. Первоначальные значения регистров затем восстанавливаются вторым xchg в строке 381. Другие команды процедуры подготавливают регистры к вызову MoveLeft, которая и выполняет копирование, перемещая байты s1 в s2.

StrPos (392–439)

StrPos — наиболее сложная процедура модуля STRING, хотя составляющие ее команды уже для вас знакомы. StrPos определяет, существует ли в целевой строке, находящейся по адресу es:di, подстрока из ds:si, и если существует, то в каком месте. Если StrPos возвращает zf, равный 1, тогда в dx содержится индекс целевой строки, с которого начинается подстрока. Если zf равно 0, тогда подстрока в целевой строке не найдена и значение в dx не имеет определенного смысла. Следующий пример показывает, как использовать StrPos для определения, содержится ли в строке названия файла расширение ASM:

```
DATASEG
extension    db              ' ASM ', 0
filename     db              ' MYTEST ASM ', 0
CODESEG
mov  si, OFFSET extension    ; Поместить адрес подстроки в ds si
mov  di, OFFSET filename     ; Поместить адрес целевой строки в es:di
call StrPos                  ; Искать подстроку
jz   foundExtension          ; Переход, если подстрока найдена в dx
jmp  notfound                ; Переход, если подстрока не найдена
```

После того как процедура определит, что длина подстроки меньше или равна длине целевой строки — в противном случае не имеет смысла продолжать поиск, в строках 421–429 циклически вызывается StrCompare, пока не будет найдена подстрока или не будет достигнут конец целевой строки. Команды mov в строках 422, 423 и 425 временно заменяют нулями символы целевой строки, используя мощный базово-индексный способ адресации, индексируя строку в bx через регистр di. Путем повторения этой операции и последовательного продвижения по символам целевой строки проверяются все возможные позиции, в которых может находиться подстрока.

StrRemove (441–471)

Из трех других подпрограмм модуля STRING StrRemove наиболее удобна для удаления из строки подстроки; ее также просто использовать. Поместите в ds:si адрес удаляемой подстроки. Адрес целевой строки установите в es:di. Затем вызовите StrRemove. Если подстрока будет найдена в целевой строке, то ее символы будут удалены; в противном случае целевая строка останется неизменной. Подстрока не изменяется ни при каких условиях. Так же как и в StrPos, флаг zf отражает результат удаления: 1, если подстрока найдена и удалена, 0 — в противном случае. В следующем примере из строки с телефонным номером удаляется код страны:

```
DATASEG

phoneNumber  db      ' (800) - 555 - 1212 ', 0  ; Целевая строка
areaCode     db      ' (800) - ', 0             ; Удаляемая строка
CODESEG
mov  si, OFFSET areaCode                        ; Адрес удаляемой строки
mov  di, OFFSET phoneNumber                     ; Адрес целевой строки
call StrRemove                                  ; Удалить подстроку из целевой строки
```

В StrRemove представляют интерес команды pushf и popf в строках 461 и 466, которые сохраняют и восстанавливают в стеке регистр флагов. Это позволяет процедуре возвращать флаг zf, получившийся в результате вызова в строке 459 StrPos, так как он изменяется затем при вызовах StrLength и StrDelete.

Резюме

Существует только три формы обращения к данным: непосредственная, к регистрам и к памяти. Непосредственные данные хранятся прямо в командах машинного кода. Регистровые данные относятся к значениям, содержащимся в регистрах, таких как ax и ch. Допускается пять вариантов обращения к памяти: прямой, косвенный, базо-

вый, индексный и базово-индексный. Несмотря на многообразие возможных методов адресации, их цель состоит в том, чтобы помочь процессору сформировать эффективный адрес, 16-битовое беззнаковое смещение от начала сегмента памяти, адресуемого одним из четырех сегментных регистров.

При ассемблировании выражения приводятся к константным значениям, которые могут использоваться программой. В отличие от выражений в языках высокого уровня, выражения в языке ассемблера не вычисляются во время исполнения программы. В выражениях могут использоваться разнообразные операторы, объединяющие разными способами метки, адреса и другие значения.

Простые переменные создаются путем резервирования пространства в сегменте данных с помощью таких директив, как db и dw. Для резервирования блоков памяти к этим директивам может добавляться оператор DUP. Инициализированные данные записываются в исполняемом файле программы на диске. Место размещения неинициализированных данных определяется во время исполнения программы и они не имеют заранее установленных значений. Для выделения пространства под строковую переменную, которая берется в одиночные или двойные кавычки, может использоваться директива db.

Область действия локальных меток распространяется только до следующей сверху или снизу нелокальной метки. Локальные метки аналогичны глобальным, но только начинаются с символов @@. Локальные метки способствуют экономии памяти, позволяя ассемблеру повторно использовать RAM для других локальных меток. Благодаря им не нужно придумывать новые названия для временно используемых меток.

Модульное программирование позволяет разделить большой проект на легко управляемые части. Модули ассемблируются отдельно и затем компонуются к главной программе, создавая конечный код. Модули могут экспортировать код, числовые константы и метки переменных в директивах PUBLIC для их совместного использования другими модулями и программами. Программы и модули импортируют символы из других модулей с помощью директив EXTRN. Утилита TLIB позволяет записывать объектные модули в файлы библиотек, благодаря которым упрощается компоновка многочисленных модулей.

Упражнения

5.1. Приведите примеры команд, которые используют непосредственные обращения к данным, регистровые данные и данные в памяти.

5.2. Приведите примеры команд, использующих каждый из пяти способов адресации памяти.

5.3. Постройте сегмент данных, включающий байтовую, строковую, переменную типа слова и одну 1024-байтовую буферную переменную. Разместите буфер в неинициализированной области сегмента данных.

5.4. Напишите процедуру инициализации буфера из упражнения 5.3 последовательными значениями байтов от 0 до 255.

5.5. Вставьте вашу подпрограмму из упражнения 5.4 в объектный модуль. Затем создайте главную программу, вызывающую вашу подпрограмму. Какие шаги необходимо предпринять для ассемблирования и компоновки вашего модуля и программы?

5.6. Какие преимущества дает запись объектных модулей в файлы библиотек?

5.7. Что делает директива PUBLIC? Для чего нужна директива EXTRN?

5.8. На какую локальную метку ссылается jmp?

```
@@40
        inc     ax
Repea
        jmp     @@40
        cmp     ax, 0
        jl      Repeat
        lodsb
        je      @@Exit
@@40
        xor     cx, cx
@@Exit.
        mov     ax, 04Ch
        int     21h
```

5.9. Какие из следующих макроопределений могут экспортироваться директивой PUBLIC? Что необходимо записать в директиве EXTRN для импортирования этих символов в программу?

```
IDEAL
MaxCount        = 1000
cr              EQU 13
lf              EQU 10
```

```
YesAnswer       = 'Y'
MaxSize         EQU 4
BufferSize      = MaxCount * MaxSize
```

5.10. Приведите три способа описания 20-символьной строковой переменной.

5.11. Предположим, что у вас имеются модули GETDATA, PRINTER, READTEXT и файл библиотеки MTA.LIB. Какие команды необходимо применять для ассемблирования и компоновки главной программы, использующей эти три модуля плюс модули STRINGS и STRIO из библиотеки?

5.12. Какую команду TLIB можно применять для помещения в библиотеку трех модулей из упражнения 5.11?

5.13. Предположим, что в кодовом сегменте записана байтовая переменная с именем Flag. Какую команду или команды необходимо использовать для загрузки этого байта в регистр dh?

5.14. Опишите следующую строку, применяя директиву db: "This 'string' can't have 'too' many quotes," she said.

Задания

5.1. Напишите улучшенную версию процедур MoveLeft и MoveRigth в модуле STRING, перемещая за один раз 16-битовое слово с помощью movsw, если счетчик байтов в cx является четным.

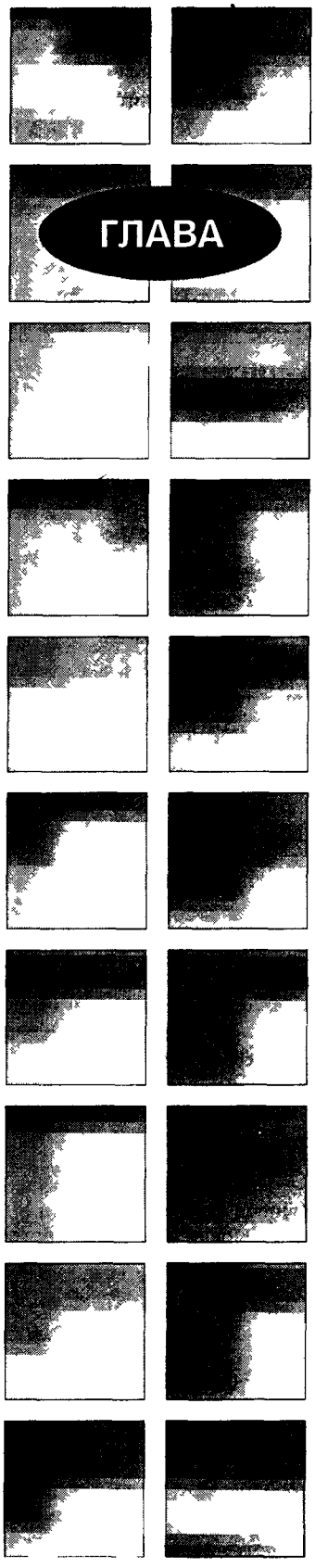
5.2. Напишите ряд тестовых процедур для исследования работы модулей STRING и STRIO.

5.3. Переделайте StrConcat таким образом, чтобы вместо MoveLeft в ней вызывалась StrInsert. Проверьте, чтобы процедура работала так же, как и оригинальная.

5.4. Напишите модуль для посылки ASCIIZ-строки на принтер.

5.5. Напишите программу, использующую модуль для работы с принтером из задания 5.4 и устанавливающую различные опции печати на вашем принтере.

5.6. [Повышенная сложность] Создайте новый модуль STRINGS для работы со строками, содержащими в первом байте байт-длину строки, в котором и записана ее длина. Второй и следующий байты содержат символы строки. Строка не заканчивается нулевым ограничителем и ее длина ограничена 255 символами. Ваш модуль STRINGS должен использовать те же имена процедур, что и ASCIIZ STRINGS модуль из этой главы.



ГЛАВА 6

Сложные структуры данных

- Структуры
- Еще о числовых переменных
- Массивы в языке ассемблера
- Объединения и записи
- Эффективные логические операции
- Использование заранее указываемых макроопределений
- Выполнение VERSION
- Преобразования чисел и строк
- Программирование преобразователя между различными системами счисления
- Резюме
- Упражнения
- Задания

Структуры

Структуры являются именованными переменными, содержащими в себе другие переменные, называемые *полями* Структура начинается с ключевого слова STRUC, после которого в той же строке следует любое имя, выбранное по вашему желанию, например MyStruct После последнего поля структура закрывается соответствующим ключевым словом END После END вы можете либо оставить пустое место, либо написать имя структуры — аналогично тому, как после ENDP вы могли повторить имя процедуры. Например, следующая структура состоит из трех полей, представляющих дату.

```
STRUCDate
day         db  1    , Поле День - начально установленное значение = 1
month       db  ?    , Поле Месяц - значение не установлено
year        dw  1991 , Поле Год - начально установлено в значение = 1991
ENDS        Date     , " Date " здесь является необязательной
```

Внутрь структуры вы можете вставлять поля любого типа, используя те же методы, что и при определении обычных переменных. В примере имеется три поля: *day*, *month*, *year*. Первые два поля представляются одиночными байтами, значение первого из них равно 1. Второе байтовое поле неинициализированное. Третье поле является словом и имеет значение 1991. Все поля структуры легко различимы. При определении структур помните три важных положения:

- Структура не является переменной, она — только схема переменной.
- Структуры могут быть описаны в любом месте. Директива STRUC не обязана находиться в сегменте данных программы, хотя это возможно.
- Структура информирует Turbo Assembler о конструкции переменных, которые вы планируете описывать позднее или которые уже существуют где-нибудь в памяти.
- Несмотря на то что для определения типов полей вы используете директивы типа db и dw, структуры не резервируют пространство в сегменте данных и не записывают ни одного байта в конечную программу.

Описание переменных в структурах

Для использования структур вы обязаны зарезервировать в памяти пространство под их поля. Результатом будет переменная, имеющая тип структуры. Каждое такое описание переменной начинается с метки, за которой следует имя структуры, и заканчивается списком начальных значений в угловых скобках <>. Чтобы использовать начальные значения, определенные ранее при описании структуры, скобки нужно оставить пустыми. Снова возвращаясь к примеру структуры Date, отметим, что в сегменте данных программы переменная типа Date может быть описана следующим образом:

```
DATASEG
birthDay      Date     <>              , 1 - 0 - 1991
```

Описание переменной начинается с метки birthDay. Затем на месте, в котором обычно используется директива db, располагается имя структуры — Date. Пустые угловые скобки приводят к использованию начальных значений полей, определенных в структуре. Неинициализированные начальные значения полей — как здесь в случае с полем month, устанавливаются в 0, если не все поля в структуре неинициализированные и переменная описана в неинициализированной области сегмента данных. В противном случае действительные значения полей являются неопределенными. Ниже приводится еще несколько примеров:

```
DATASEG
today               Date    <5  10>          , 5 - 10 - 1991
dayInDayOut         Date    < 11  12  1912>  , 11 - 12 - 1912
monthOfSurdays      Date    <   8 , >        , 1 - 8 - 1991
```

Первых два начальных значения — day и month в переменной today изменяются соответственно на 5 и 10. Пропущенное третье поле подразумевает использование начального значения, определенного в самой структуре, в данном случае — 1991. Во второй переменной dayInDayOut замещаются все три начальных значения. Третья переменная monthOfSundays определяет новое значение для month (8), используя для других полей первоначально установленные значения. Первая запятая необходима для "извлечения" второго поля структуры. Использовать вторую запятую необязательно и можно записать:

```
monthOfSundays Date < , 8 >
```

Пример использования структуры

Наилучший способ подробнее изучить структуры состоит в исследовании нескольких примеров структурированных переменных в Turbo Debugger с использованием программы 6.1, STRUC.ASM. После ассемблирования, компоновки и загрузки программы в Turbo Debugger с помощью команд:

```
tasm /zi struc
tlink /v struc
td struc
```

выполните действия, перечисленные после текста программы.

Программа 6.1 STRUC ASM

```
1    %TITLE  TD Structure Demo -- by Tom Swan
2
3.           IDEAL
```

```
4:
5:              MODEL small
6:              STACK 256
7:
8:      STRUC Date
9:      day     db 1      ; Day-начальное значение = 1
10:     month   db ?      ; Month-начальное значение не определено
11:     year    dw 1991   ; Year-начальное значение = 1991
12:     ENDS Date
13:
14:     STRUC CityState
15:     city    db '####################', 0       ; до 20 символов
16:     state   db '##',                   0       ; 2 символа
17:     ENDS CityState
18:
19:
20:             DATASEG
21:
22:     exCode  db 0
23:
24:              today  Date     <>
25:              birthDay        Date    <8,8,1988>
26:              earthDay        Date    <1,1,2001>
27:              newYear         Date    <,,1990>
28:
29:              address         CityState       <>
30:              glitterTown     CityState       <'Hollywood','CA'>
31:              pennState       CityState       <'Pennstate','PA'>
32:              hotSpot         CityState       <'Brownsville', 'TX'>
33:              defaultState    CityState       <,'NH'>
34:              defaultCity     CityState       <'New York City'>
35:
36:
37:             CODESEG
38:
39:     Start:
40:             mov     ax, @data       ; Поместить в DS адрес
41:             mov     ds, ax          ; сегмента данных
42:
43:     ; Замечание: исполняйте программу в Turbo Debugger - она не выполняет никаких действий
44:
45:     Exit:
46:             mov     ah, 04Ch                ; Функция DOS: Выход из программы
47:             mov     al, [exCode]            ; Возврат значения кода выхода
48:             int     21h                     ; Вызов DOS. Останов программы
49:
50:             END     Start                   ; Конец программы / точка входа
```

Выполнение демонстрационной программы STRUC

К этому моменту вы должны ассемблировать STRUC и загрузить ее в Turbo Debugger. Выполните предлагаемые эксперименты, чтобы увидеть, как хранятся в памяти структурированные переменные.

1. Нажмите <Alt+V> и клавишу <V> для выбора команды View:Variables. В результате появится окно со списком всех имен переменных программы.
2. Нажмите <Tab> для перемещения курсора на список переменных, после чего используйте клавишу "стрелка вниз" для перевода курсора на "today". Обратите внимание на значения, приведенные справа от названий полей, которые позволяют вам быстро просмотреть данные, хранящиеся в структурированных переменных.
3. Нажмите <Ctrl+I> для инспектирования переменной today. (В этом месте вы можете нажать <F5> для того, чтобы распахнуть небольшое окно на весь экран и обеспечить более удобный просмотр). Turbo Debugger показывает каждое поле на отдельной строке, используя имена из определения STRUC и показывая их действительные значения, хранящиеся в памяти. Поскольку db может резервировать пространство как под ASCII символы, так и под байты, отладчик показывает эти значения обоими способами. Просто не рассматривайте символы, соответствующие несимвольным байтовым полям. Целые значения показаны в десятичном представлении и в скобках — в шестнадцатеричном.
4. Нажмите <Alt+F3> или <Esc> для закрытия окна инспектирования. Переместите курсор вниз на следующую переменную (birthDay) и нажмите <Enter> — самый простой способ для вызова окна просмотра. Сравните

приведенные значения полей со значениями из строки 25 программы. Нажмите <Alt+F3> или <Esc> и повторите эти же шаги для двух оставшихся переменных из строк 26–27.

5. В строках 14–17 описана другая структура CityState с двумя строковыми полями — city и state. Начальные значения вы можете увидеть в Turbo Debugger, эти строки инициализированы знаками штриховки. Обычно вы будете инициализировать значения строк менее экзотическими символами, такими как пробелы или нули. Снова начните с окна Variables, поместите курсор на adress и нажмите <Enter>.
6. В окне инспектирования теперь будут отображаться два начальных поля. Внизу окна приводится информация о типе и размере структуры и отдельных полей. Используйте клавиши управления курсором для перемещения на первое поле (наблюдайте, как изменяется нижняя строка окна) и снова нажмите <Enter>. В результате откроется новое окно инспектирования, позволяющее вам просматривать отдельные байты в переменной поля. Переместите курсор вниз на любой байт и один или несколько раз нажмите <Enter> для открытия других окон инспектирования, на этот раз показывающих адреса отдельных байтов. Возможность пошагово добраться до значений байтов структурированной переменной является одной из лучших отличительных черт Turbo Debugger, так как нахождение в памяти структур данных иногда может быть чрезвычайно затруднено.
7. Нажимайте <Esc> до тех пор, пока на экране не останется одно активное окно Variables (в двойной рамке). Переместите курсор вниз на значение следующей переменной pennState и нажмите <Enter>. С помощью <F5> распахните окно на весь экран. Сравните показанные строки с их начальными значениями в строке 31. Заметьте, что текст из угловых скобок заменил только головную часть в строковом поле. Остаток строки *набит* (заполнен) первоначальными символами из определения STRUC.
8. Строки 15 и 16 описывают поля структуры как ASCIIZ-строки, заканчивающиеся нулевым символом. Однако на дисплее нуль не показывается. Причина этого несоответствия состоит в том, что Turbo Debugger показывает только первоначальные значения полей. Чтобы доказать, что нули находятся там, где они и должны быть, переместите курсор на city и нажмите <Enter>. Затем нажимайте <PgDn> до тех пор, пока курсор не остановится на последнем байте этого поля (в строке 19). Снова нажмите <Enter> и запомните адрес (у меня 6C89:0060). Дважды нажмите <Esc>, после чего выберите поле field. Нажмите <Enter>. Адрес на моем экране — 6C89:0062 показывает, что по адресу 6C89:0061 находится невидимый байт. Вы нашли нуль!
9. Чтобы увидеть нули в строковых переменных, несколько раз нажмите <Esc> для возврата в окно Variables. Нажмите <Alt+V> и <D> для выбора команды View:Dump. Нажмите <Ctrl+G> и введите адрес строки из шага 8 — в моем случае 6C89h:0060h. После значений адресов сегмента и смещения вы *должны* использовать строчные буквы *h*. Нажмите <F5> для распахивания окна. Теперь вы видите значения структурированных переменных в том виде, в котором они записаны в сегменте данных программы. Постарайтесь найти нули, разделяющие отдельные строковые поля.
10. Не нужно запускать эту программу на исполнение — она не выполняет никаких действий, а только показывает, как ассемблируются структуры. Когда вы закончите, нажмите <Alt+X> для возврата в DOS.

Как вы поняли из этих замечаний, строковые поля в структурах имеют фиксированную длину. Знаки штриховки (#) в начальных значениях в строках 15–16 заменяются новыми значениями, определенными в угловых скобках в строках 29–34. В режиме Ideal Turbo Assembler заполняет остаток строки начальными символами из определения структуры. (В режиме MASM любые оставшиеся символы изменяются на пробелы — даже в том случае, если вы этого не хотите.) В Turbo Debugger вы можете видеть только первое из списка значений, описанных в директивах db и dw. Чтобы увидеть каждое значение, вы можете изменить определение структуры в строках 14–17, поместив каждое значение поля на отдельную строку:

```
STRUC       CityState
city        db      ' ################## '
cnull       db      0
state       db      ' ## '
snull       db      0
ENDS        CityState
```

Из-за дополнительных полей, резервирующих байты для нулевых ограничителей строк, вы также должны изменить описания переменных в строках 29–34, добавив для каждого поля новые значения. Если вы не сделаете этого, то получите при ассемблировании ошибку переполнения, которая возникает при попытке заменить начальное значение, представленное единственным байтом, многобайтовой строкой. Измените, как показано ниже, строки 29–34, выполните повторное ассемблирование и проинспектируйте новые переменные в Turbo Debugger:

```
address      CityState      <>
glitterTown  CityState      < ' Hollywood ' , 0 , ' CA ' , 0 >
pennState    CityState      < ' Pennstate ' , 0 , ' PA ' , 0 >
hotSpot      CityState      < ' Brownsville ' , 0 , ' TX ' , 0 >
defaultState CityState      < , , ' NH ' >
defaultCity  CityState      < ' New York City ' >
```

Использование структурированных переменных

Как объяснялось в главе 5, использовать поля структурированных переменных не намного сложнее, чем простые переменные. Допускаются все те же режимы адресации. Так как имена полей содержатся в определении структуры, то для обращения к ним, помимо их имени, вы должны использовать имя структуры, разделяя их точкой. Вернемся к программе 6.1. Для задания нового значения полю дня в today вы можете присвоить ему непосредственное значение с помощью команды:

```
mov  [today day], 5            ; Изменить день на 5
```

Вы можете также загружать значения полей в регистры, как это делается в следующей команде, записывающей в ax значение года:

```
mov  ax, [today year]          ; Присвоить в ax значение года
```

Возможны и другие варианты. Вы можете складывать, вычитать, считывать, записывать и логически объединять поля и регистры. Помните, что во всех случаях вы должны задавать имена переменной и структуры, чтобы ассемблер мог сгенерировать для ваших полей правильные адреса. Ниже приводится еще несколько примеров:

```
inc  [earthDay.day]            ; Прибавить 1 к полю день
add  [newYear.year], cx        ; Прибавить cx к полю год
cmp  [today month], 8          ; Месяц = 8 ?
```

Замечание

В режиме Ideal Turbo Assembler имена полей являются локальными и уникальными для структуры, в которой они определены. Это означает, что можно создавать многочисленные определения STRUC с одинаковыми именами полей. Например, можно иметь две различные структуры, каждая из которых содержит поля day, month и year. Вы не можете аналогично поступить в более ограничительном режиме MASM, в котором все имена полей глобальные, т.е. каждое имя может встречаться только в одном определении структуры на протяжении всей программы. По этой причине в режиме MASM нельзя иметь две структуры, такие как Customer и Personal, с одинаковыми названиями полей Name — вместо этого нужно изобретать новые имена полей типа Cname и Pname. В режиме Ideal использовать структуры проще, хотя, поскольку имена могут быть неуникальными, при всех обращениях нужно использовать и имя структуры и имя поля, разделенные точкой.

Структуры из STRIO

В главе 5 я пообещал объяснить использование структуры StrBuffer в строках 18–22 в программе 5.2, STRIO.ASM. Для удобства эта структура повторяется ниже:

```
BufSize     EQU     255          ; Максимальный размер строки
STRUC       StrBuffer
maxLen      db BufSize           ; Максимальная длина буфера
strlen      db 0                 , Длина строки
chars       db BufSize DUP (?)   , Буфер для StrRead
ENDS        StrBuffer
```

BufSize является макроопределением, равным 255 — максимальной длине строки, которая может быть прочитана DOS. Структура StrBuffer использует это значение для определения трех полей в виде, требуемом функцией DOS 0Ah, которая считывает строки из стандартного входного файла (обычно клавиатуры). StrRead вызывает ее, чтобы ввести строковое значение переменной (см. строки 39–78 в программе 5.2). Эта входная последовательность затем преобразуется в ASCIIZ-формат для использования в процедурах из STRINGS, STRIO и других модулей этой книги.

В строке 27 программы 5.2 описывается переменная buffer структуры StrBuffer, использующая начальные значения из определения структуры. StrRead передает адрес этой переменной в DOS, где обеспечивается вся обработка клавиатуры, ограничивая результат максимальной длиной, определенной в поле maxLen, записывая действительный размер строки в поле strLen и вставляя символы (если таковые имеются) в поле chars.

Замечание

Поскольку StrRead для ввода строки вызывает функцию DOS, можно редактировать вводимую строку с помощью тех же самых функциональных клавиш, которые вы привыкли использовать в командной строке DOS.

После окончания набора нажатие <Enter> приводит к тому, что DOS устанавливает в поле strLen число, равное количеству введенных вами символов. DOS также добавляет к концу строки ASCII-символ возврата каретки. Поскольку это неправильный ограничитель для ASCIIZ-формата, в строках 66–68 StrRead перед записью строки в переменную программы (строки 69–70) символ возврата каретки заменяется на ASCII нуль.

Обратите внимание, как программа обращается к строковым полям в строках 62, 64, 67, 68 и 69, используя как прямой, так и базовый способы адресации. В каждом случае после имени структуры через точку следует имя поля. В строке 62 значение из cl записывается в поле maxLen переменной buffer. Строка 64 показывает, как находить адрес смещения поля maxLen. В строке 68 значение регистра bx прибавляется к первому символу поля chars, определяя тем самым адрес находящегося в chars возврата каретки.

Еще о числовых переменных

В программах на языке ассемблера вы можете представлять значения в шестнадцатеричном, двоичном или десятичном виде. Однако, так как эти три системы счисления используют одни и те же цифровые символы, вы должны указать ассемблеру, какую систему вы подразумеваете. К двоичным числам добавляйте в конце букву *b*, к шестнадцатеричным — *h*. Обычно по умолчанию все числа считаются десятичными, поэтому к ним можно либо ничего не добавлять, либо дописывать букву *d*. Например, следующие переменные представляют одно и то же значение в трех числовых системах:

```
v1   dw      0100111101011100b      ; Двоичное
v2   dw      04F5Ch                 ; Шестнадцатеричное
v3   dw      20316                  ; Десятичное (подразумевается)
v4   dw      20316d                 ; Десятичное
```

Обратите внимание, что шестнадцатеричное значение 04F5Ch начинается с 0. Этот 0 требуется только в том случае, если первая цифра числа попадает в интервал от A до F, например, как в значении 0FACEh. Однако это неплохая идея — начинать шестнадцатеричное число с 0 в любом случае, когда это допустимо. Шестнадцатеричные значения должны начинаться с десятичных цифр, так как ассемблер не может сам определить, является ли FACEh меткой или значением. Поэтому при написании числовых значений вы обязаны соблюдать строгое правило: любое число в двоичной системе должно начинаться с 0 или 1, в десятичной и шестнадцатеричной — с 0 до 9.

Добавление 0 в начало шестнадцатеричного значения удовлетворяет этому правилу.

Использование RADIX

До тех пор пока вы не закончите число буквой *b* или *h*, Turbo Assembler считает его десятичным. Директива RADIX используется для изменения этого первоначально установленного поведения (*RADIX* означает "основание системы счисления"). Например, чтобы установить по умолчанию шестнадцатеричное основание системы, можно использовать команду

```
RADIX 16               ; По умолчанию устанавливается шестнадцатеричная
                       ; система счисления
```

Для большинства целей, вероятно, лучше придерживаться устанавливаемой ассемблером по умолчанию десятичной системы счисления, используя для задания шестнадцатеричных и двоичных значений буквы *h* и *b*. Если в новой программе вы забудете изменить RADIX на шестнадцатеричную, то легко ошибочно принять 100 за десятичное 256. Вы не ошибетесь, если шестнадцатеричное значение будет представлено в виде 0100h.

Замечание

Значение, следующее за директивой RADIX, всегда выражается в десятичной системе и должно быть 2 (для двоичной), 8 (для восьмеричной), 10 (для десятичной) или 16 (для шестнадцатеричной), независимо от текущей установленной системы счисления. Если вы измените установленную по умолчанию систему счисления, не забывайте заканчивать каждое десятичное значение буквой d.

Знаковые и беззнаковые целые

При описании значений с помощью db и dw нужно понимать разницу между знаковыми и беззнаковыми величинами, как это объяснялось в главе 3. В отличие от языков высокого уровня, язык ассемблера не накладывает никаких ограничений на диапазоны чисел со знаком; поэтому, пока определенное вами значение помещается в выделенном под него пространстве, ассемблер исполняет все ваши желания и команды. Например, вы можете написать:

```
v1   dw      32768          ; 08000h
v2   dw      -32768         ; 08000h
v3   dw      -1             ; 0FFFFh
v4   dw      65535          ; 0FFFFh
```

Когда Turbo Assembler записывает эти значения в памяти, результат может быть не таким, каким вы ожидаете его увидеть. Переменная v1 записана как беззнаковое значение 32,768, или 08000h. (Замечание: запятые в числах используются исключительно для простоты чтения. В программе добавлять запятые к числам запрещается.) Заметьте, что это значение эквивалентно значению со знаком –32,768 — по крайней мере, в мире двоичных значений фиксированной длины в памяти компьютера. Аналогично, –1 и 65,535 ассемблируются в одно и то же значение 0FFFFh. Это показывает, что, несмотря на допустимый диапазон значений от –32,768 до +65,535, значения от –32,768 до –1 и от 32,768 до 65,535 имеют одинаковое представление в двоичной системе. Доскональное понимание двоичного представления и записи в дополнительном коде является лучшим способом избежать недоразумений с этими тонкостями в языке ассемблера.

Числа с плавающей запятой

Вы можете описать числа с плавающей запятой с помощью директивы dt, которая резервирует 10 байт памяти, подобно тому, как dw резервирует 2 байта. Результатом определения директивой dt значения с плавающей запятой является двоичное 10-байтовое действительное число в стандартном формате IEEE. Эти значения совместимы с форматом, используемым 8087, 80287 и 80387 математическими сопроцессорами. Кроме этого, для обработки выражений с плавающей запятой вы также можете обмениваться этими значениями из вашей программы на языке ассемблера с большинством языков высокого уровня.

Без пакета подпрограмм для вывода и обработки значений с плавающей запятой применять эти значения в языке ассемблера достаточно тяжело. Для описания числа с плавающей запятой используйте dt следующим образом:

```
fp   dt      3.14159        ; 4000C90FCF80DC33721Dh
```

Двоично-десятичные числа

Другая область применения dt состоит в определении упакованных двоично-десятичных (BCD) чисел. Эти значения особенно полезны при финансовых вычислениях, в которых часто используются большие числа, но не допускается ошибка при округлении значений с плавающей запятой. BCD значения занимают больше места (10 байт каждое) и требуют большего времени на обработку, чем целые размером в байт или слово, так что этот формат нужно применять в исключительных случаях. (Детально BCD описаны в главе 11.) Для описания упакованного BCD значения используйте ту же директиву db, что и для значений с плавающей запятой, но только без десятичной точки. Например:

```
bcd1 dt      1234
bcd2 dt      9876543210
bcd3 dt      250000
```

Каждое из этих описаний резервирует 10 байт памяти, записывая в каждый байт по две цифры из начального значения. Другими словами, в BCD значении может быть до 20 цифр. Значение записывается в обратном порядке, так что каждая цифра занимает 4 бита в байте:

```
nnnn 0000    34 12 00 00 00 00 00 00 00 00
nnnn 0000    10 32 54 76 98 00 00 00 00 00
nnnn 0000    00 00 25 00 00 00 00 00 00 00
```

Массивы в языке ассемблера

Не существует специальных команд, структур или методов для описания и использования массивов в программах на языке ассемблера. В языках высокого уровня, таких как Pascal или C, вы можете описать массив и затем обращаться к его элементам по индексу. Например, в программе на Pascal может быть определен массив из десяти целых чисел, индексированных от 0 до 9:

```
VAR  IntArray:       ARRAY[ 0 .. 9 ] OF Integer;
```

Операторы программы могут затем обращаться к массиву, возможно, используя в цикле FOR переменную индекса для задания значений каждому элементу массива:

```
FOR  I := 0 TO 9 DO
     IntArray[ I ] := I;
```

Для тех, кто не знаком с Pascal, этот оператор присваивает целым элементам массива значения от 0 до 9. Программисты на C и Basic также имеют аналогичные способы для создания и использования массивов. В языке ассемблера управление массивами немного более сложное, но в то же время и более гибкое, так как вы сами пишите код для доступа к их значениям. Например, целый массив можно создать, используя оператор DUP:

```
anArray     db      10 DUP (?)              , Массив из 10 целых элементов
```

Так же вы можете определить последовательность из десяти значений, в один прием описывая и инициализируя массив:

```
anArray     db      0  1, 2, 3  4, 5, 6, 7, 8, 9
```

Создание массивов, состоящих из составных элементов, таких как строки и переменные типа STRUC, требует дополнительных усилий. Предположим, вам необходим массив из четырех 20-байтовых строк. Поскольку массив достаточно короткий, для его описания вы можете использовать четыре отдельные переменные:

```
anArray     db      20 DUP (?), 0           , anArray[0]
            db      20 DUP (?)  0           , anArray[1]
            db      20 DUP (?)  0           , anArray[2]
            db      20 DUP (?), 0           , anArray[3]
```

Четыре переменные записываются в памяти последовательно, поэтому эти же четыре 20-байтовые строки (плюс один байт для ограничителя строки) могут рассматриваться как отдельные переменные или как структура из четырех строк, образующая массив. Если вы не любите набирать длинные программы, этот подход является неприемлемым для создания больших массивов. Рассмотрим, как можно создать пространство для ста 20-байтовых строк. Используя две новые директивы — LABEL и REPT, вы можете написать:

```
LABEL       anArray Byte
REPT        100
    db       20 DUP (?), 0
ENDM
```

Первая строка описывает метку anArray типа Byte. Здесь можно использовать и другие типы, такие как Word, Dword, Fword, Pword, DataPtr, Qword и Tbyte, а также имя структуры. Директива LABEL указывает ассемблеру, как адресовать последующие данные — в памяти она не резервирует никакого пространства. В данном примере следующие данные являются строками, которые всегда адресуются как одиночные байты. Команда REPT (Повторить) определенное количество раз (здесь 100) повторяет любой оператор языка ассемблера. Все, что находится между REPT и ENDM (Конец макро), повторяется так, как если бы вы набрали эти строки необходимое число раз. (Команда ENDM также завершает макроопределения; предмет изучения главы 8).

Один удобный прием — это изменять каждый раз описание в определении. Например, для создания массива из десяти целых элементов и задания каждому элементу массива значений от 0 до 9 можно использовать следующее описание:

```
value = 0
LABEL       anArray Word
REPT        10
 dw  value
 value = value + 1
ENDM
```

В результате получится массив целых чисел типа Word со значениями 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Числовое макроопределение value инициализируется 0. Как вы помните из главы 5, символы, описанные с помощью знака равенства, могут позднее переопределяться, что является ключом к этому методу. Внутри REPT директива db определяет одно слово в памяти, равное value. После этого value при следующем проходе увеличивается на 1. Помните, что выражения, такие как value=value+1, вычисляются во время ассемблирования и все описанные действия имеют место при ассемблировании, а не при выполнении программы. Результатом является массив из десяти слов, инициализированный последовательными значениями. Эти команды не генерируют никакого кода.

Замечание

Элементы массива, описанные, как показано здесь, с помощью директив REPT, *не отображаются в окне* Variables в Turbo Debugger Чтобы увидеть массив, используйте команды View Dump для просмотра памяти, начиная с адреса массива

Изменение типов с помощью LABEL

Директива LABEL чаще используется для назначения одним и тем же данным в памяти двух или более меток разных типов. При этом можно читать и записывать переменные как байты в одних командах и как слова (или другие типы) — в других местах. Директива состоит из трех частей:

```
LABEL идентификатор тип
```

Идентификатор аналогичен любой другой метке. *Тип* может быть near, far, proc, byte, word, dword, fword, pword, dataptr, qword или tbyte. *Тип* также может быть именем структуры данных STRUC. Используя LABEL, вы можете описать значение как 2 байта, но просматривать его как 16-битовое слово:

```
LABEL       ByteValue       byte
WordValue   dw              01234h
```

Шестнадцатеричное значение 01234h имеет метку WordValue и описано с помощью dw как 16-битовое слово. Но предыдущая LABEL создает другую метку типа byte, адресующую то же значение в памяти. Это позволяет писать такие команды, как:

```
mov  ax, [WordValue]          , Получить полное 16 битовое значение
mov  bl, [ByteValue]          , Получить 8-битовый младший значащий байт
mov  bh, [ByteValue + 1]      , Получить 8-битовый старший значащий байт
```

Первая команда mov загружает полное 16-битовое значение, устанавливая ax в 01234h, вторая — только первые 8 бит этого же значения, устанавливая bl в 034h, третья — следующие 8 бит, устанавливая bh в 012h. Таким образом, две последние команды устанавливают в bx то же значение, которое находится в ax. (Помните, что слова записываются в обратном порядке байтов — значение 01234h записывается в памяти как два байта — 034h и 012h.)

Использование LABEL для задания переменным меток различных типов удобно для адресации структур как коллекций типизированных полей, а также как потоков 16-битовых слов. Применяя структуру Date из предыдущего раздела, вы можете написать:

```
LABEL       DayMonth        word
OneDay      Date            <>
```

OneDay является одиночной структурированной переменной типа Date. Метка DayMonth адресует эту же память, но рассматривает данные как тип word. В тексте программы вы можете обращаться к первым двум полям обычным образом — как OneDay.day и OneDay.month. Благодаря дополнительной метке можно непосредственно загружать эти два байтовых поля в один 16-битовый регистр:

```
mov  ax, [DayMonth]           , Загрузить день и месяц в ax
mov  al, [OneDay day]         , Загрузить день в al
mov  ah, [OneDay month]       ; Загрузить месяц в ah
```

Первая команда mov выполняет те же действия, что и две последние команды. Иногда, как показано здесь, использование LABEL может помочь разделить одну команду на две, и если одна из них повторяется чаще, это может повысить производительность программы.

Индексирование массивов

Теперь, когда вы знаете, как описывать массивы, следующим шагом будет изучение способов чтения и записи их значений Например, как обратиться к пятому элементу? Ключом к ответу является понимание того, что в языке ассемблера индексы массивов — обычные адреса, поэтому, независимо от типа данных, записанных в массив, индексация отдельных значений сводится к следующим двум шагам:

- Умножить размер элемента массива на индекс массива I.
- Прибавить результат к базовому адресу массива.

Например, в простом массиве байтов, если *I* равно 0, тогда *Ix2*(0) плюс адрес array определяет первое значение *array[0]*. Второе значение (*array[1]*) расположено по базовому адресу массива плюс 1 и т.д. Как показывает рис. 6.1, цель состоит в преобразовании значений индекса массива в адреса памяти. Индекс 0 эквивалентен адресу 000D, т.е. базовому адресу всего массива. Индекс 1 соответствует 000E, индекс 2 — 000F и так до индекса 9, который определяет значение со смещением 0016. Практический пример поможет уяснить этот процесс. Проще всего работать с массивами байтов, поэтому с них и начнем. Для загрузки в al 64-го элемента 100-байтового массива можно написать:

```
DATASEG
anArray db 100 DUP (0)
CODESEG
mov al, [anArray + 63]
```

В этом примере 63 является правильным значением, так как первый элемент массива имеет смещение 0. Индекс 64 ошибочно определял бы 65-ю, а не 64-ю позицию массива. Вы избежите недоразумений и разочарований, если всегда будете помнить, что для массива из 100 элементов индексы находятся в диапазоне от 0 до 99, а не от 1 до 100.

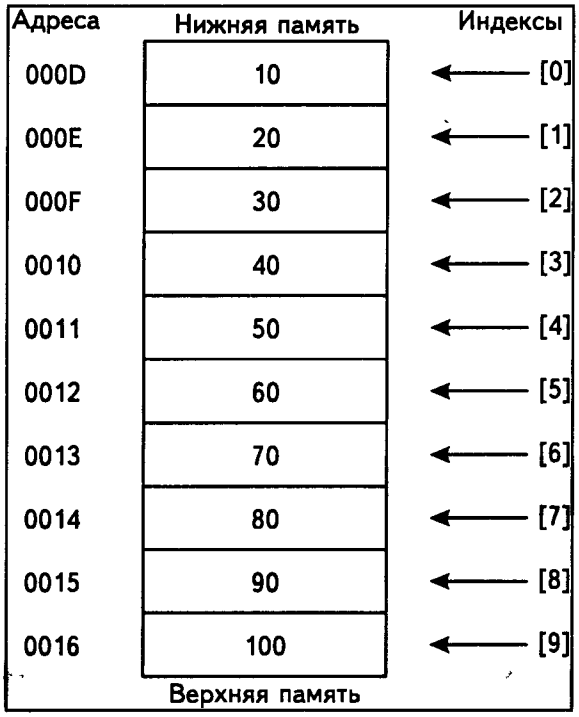


Рис. 6.1. Возможное расположение в памяти простого массива байтов

Прибавление литерального значения, как это было сделано в предыдущем примере, во многом лишает вас гибкости. В большинстве случаев под индексы массивов вы будете использовать регистры или переменные в памяти. Применяя базовый способ адресации, введенный в главе 5, можно записывать значения индексов массива в регистр bx. Например, предположим, что имеется переменная с именем index и вы хотите загрузить в регистр значение anArray[index]. Тогда можно написать:

```
DATASEG
index       dw      ?
anArray     db      100 DUP (?)
CODESEG
mov  bx, [index]            ; Получить значение индекса
mov  al, [bx + anArray]     ; al <- anArray[index]
```

Два описания данных резервируют пространство под 16-битовый индекс и 100-байтовый неинициализированный массив. В кодовом сегменте первая команда mov загружает в bx текущее значение index. Вторая команда mov прибавляет bx к базовому адресу массива, определяя нужный байт и загружая значение элемента массива в al. Для этих же целей можно использовать регистры si и di:

```
mov  si, [index]            ; Получить значение индекса
mov  al, [si + anArray]     ; al <- anArray[index]
mov  di, [index]            ; Получить значение индекса
mov  al, [di + anArray]     ; al <- anArray[index]
```

Верхние две строки выполняют те же действия, что и две нижние. Технически — это *индексный* и *базовый* способ адресации, хотя, как вы можете видеть, между этими двумя методами практически нет различий.

Замечание

Для адресации массива вы также можете применять регистр bp, но помните, что для него устанавливается по умолчанию сегмент ss, а не ds, использующийся для bx, si и di при базовом и индексном режиме адресации

Многобайтовые значения массивов

Когда значения элементов массивов занимают более одного байта, адресация массивов становится изощреннее. Вследствие двоичной природы компьютера вычисление адресов многобайтовых элементов массива упрощается, если размеры элементов являются степенями числа 2. В этом случае для перемножения индекса на значение размера элемента в байтах можно использовать быстрые команды сдвига. Прибавление результата этого перемножения к базовому адресу массива определяет любой элемент массива, как это делается в следующем фрагменте:

```
DATASEG
index       dw      ?
anArray     dw      100 DUP (?)
CODESEG
mov  bx  [index]            , Получить значение индекса
shl  bx  1                  , bx <- index * element-size (2)
mov  ax  [bx + anArray]       ax <- anArray[index]
```

В этом примере каждый элемент имеет размер в 2 байта, поэтому проще (и быстрее) для умножения значения индекса на 2 сдвинуть его на один бит влево Сравните рис. 6 2 с рис. 6.1. Как вы видите, адреса слева увеличились в два раза Для вычисления адреса четвертого 2-байтового значения массива (с индексом 4) сначала нужно перемножить 4x2, а затем сложить результат с базовым адресом массива для получения окончательной величины смещения 0018h.

Если размер элемента не является степенью 2, то вычисление адресов индексов для получения максимально быстрого кода требует некоторой творческой работы. Конечно, для выполнения начального перемножения всегда можно использовать команду mul. Рассмотрим массив элементов, каждый из которых занимает 5 байт. Для установки в bx адреса смещения элемента с номером index требуется выполнить несколько шагов:

```
mov  ax  [index]            , Переслать значение индекса в ax
mov  bx  5                  , Установить bx = размер элемента
mul  bx                     , dx   ax <- index * element size
mov  bx  ax                 , Переслать результат в bx (игнорируя dx)
add  bx  OFFSET anArray     , Установить bx <- адрес смещения элемента
```

Важен только младший значащий байт полученного произведения — верхние 16 бит 32-битового результата не используются. (Предварительно другая часть этой программы проверяет, попадает ли значение index в допустимые пределы) Проблемой в этом подходе является команда mul, для выполнения которой может потребоваться до 118 машинных циклов. Поэтому следует разложить ее в ряд по степеням 2 и вычислять адреса значений элементов массива, используя комбинации сдвигов и других быстрых команд:

```
mov  bx, [index]            , Получить значение индекса в bx
mov  ax  bx                 , Такое же значение в ax
shl  bx, 1                  , bx <- bx * 2
shl  bx, 1                  , bx <- bx * 4 ( в сумме)
add  bx, ax                 , bx <- bx * 5 ( в сумме)
add  bx  OFFSET an Array    , Установить bx <- адрес элемента
```

В комментариях к этому фрагменту показано, как в bx получается требуемый результат. Сначала с помощью двух сдвигов влево bx умножается на 4. Окончательно умножение на 5 завершается прибавлением этого результата к начальному значению index. Тот же самый результат, что и при умножении некоторого значения на 5, можно получить, умножив это значение на 4 и прибавив это же значение к полученному произведению. Так как 4 является степенью 2, то программа может выполнить первую часть перемножения с помощью быстрых команд сдвига, после чего получить окончательный результат, используя простое сложение. Данная последовательность команд выполняется во *много* раз быстрее, чем отдельная команда mul.

Приемы, подобные этому, проходят не всегда. Но если вы можете использовать вместо умножения сдвиги, результат получается быстрее. Лучше всего пользоваться массивами с размерами элементов, являющимися степенями 2 Когда это невозможно, тогда для получения требуемого результата попробуйте найти комбинацию сдвигов и других команд.

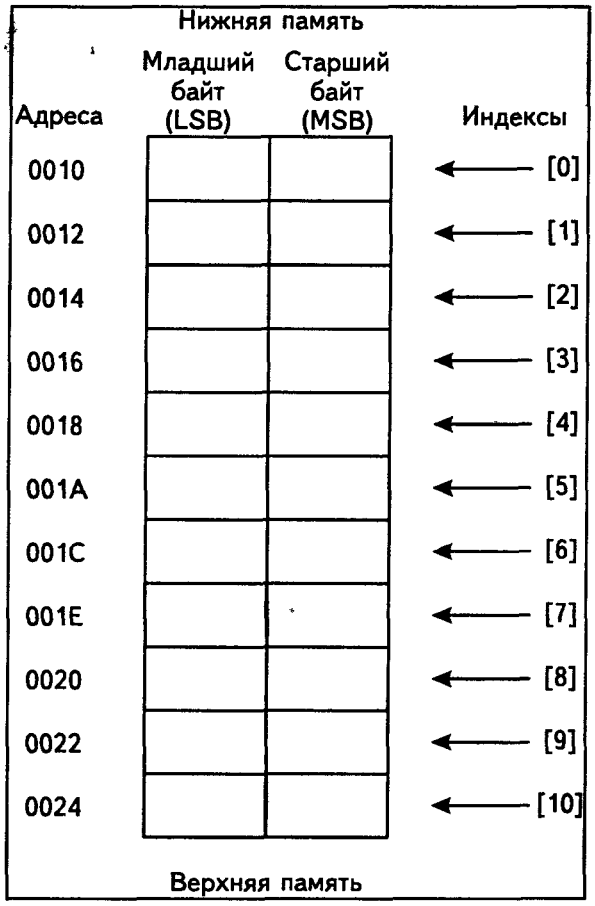


Рис. 6 2 Если размеры элементов массива являются степенями 2, перевод индексов в адреса смещения выполняется относительно просто

Объединения и записи

Определяемые с помощью директивы UNION *объединения* имеют такой же вид, что и структуры STRUC. Подобно структурам объединения содержат именованные поля, часто имеющие разные типы данных. Различие между объединением и структурой заключается в том, что поля объединения накладываются одно на другое внутри переменной. Другими словами, объединение с тремя байтовыми полями в действительности занимает только один байт Как показано в следующем примере, можно использовать эту особенность для создания переменных, в которых могут содержаться данные, принадлежащие к разным типам, аналогично тому, как это делалось ранее с помощью директивы LABEL:

```
UNIONByteWord
 aByte        db      ?
 aWord        dw      ?
ENDS          ByteWord
```

Объединение заканчивается директивой ENDS. В этом примере aByte накладывается на первый байт aWord Если бы это была структура, тогда aByte и aWord располагались бы в памяти последовательно. Но поскольку это объединение, то aByte и aWord хранятся в одном и том же месте памяти. Поэтому при помещении значения в aByte изменяется младший значащий байт aWord:

```
mov [aByte], bh , Записать bh в aByte и в младший значащий байт aWord
```

При объединении со структурами объединения становятся мощным средством для обработки переменных. Например, на рис. 6.3 представлена комбинация структуры и объединения, которую можно использовать для обращения к переменным как к 16-битовым словам, так и 8-битовым байтам.

```
STRUCTwoBytes
     loByte          db      ?
     hiByte          db      ?
ENDS         TwoBytes

UNIONByteWord
     asBytes         TwoBytes        <>
     asWord          dw              ?
ENDS         ByteWord
```

Рис. 6 3. Объединение с вложенной структурой

Структура TwoBytes определяет два байтовых поля — loByte и hiByte. Два поля также определены и в объединении ByteWord. Первое поле — asByte имеет тип определенной перед этим структуры TwoBytes, а второе — asWord описано как 16-битовое слово. Переменные типа ByteWord позволяют легко обращаться как к словам, так и к значениям образующих их байтов, "не забывая", что в слове байты хранятся в обратном порядке — это нужно помнить при использовании директивы LABEL. Для применения вложенного объединения сначала опишите переменную, в нашем случае ей присваивается значение 0FF00h:

```
DATASEG
data         ByteWord        <, 0FF00h >
```

Теперь вы можете обращаться к data как к структуре TwoBytes или как к 16-битовому слову. Следующий короткий пример показывает, как загружать данные из одного и того же места памяти либо в 8-битовые, либо в 16-битовые регистры. Так как структура TwoBytes вложена внутрь объединения, то для доступа к ее байтовым полям необходимо использовать две точки. Обратите внимание на то, как имена полей уменьшают опасность случайной загрузки в 8-битовый регистр неверного байта из слова:

```
CODESEG
mov  al, [data asBytes loByte]        ; Загрузить в al младший значащий байт
mov  ah, [data asBytes hiByte]        , Загрузить в ah старший значащий байт
mov  ax  [data asWord]                , Тот же результат
```

Битовые поля

При программировании на языке ассемблера вам часто бывает необходимо проверять или изменять один или более битов в значении байта или слова. Вы уже знаете, как это делается с помощью логических команд, таких как or, and и xor, устанавливающих и очищающих отдельные биты без воздействия на остальные. Например, для установки в регистре бита с номером 2 можно использовать команду

```
or   al  00000100b
```

При этом удобно записывать значения в двоичной системе, не забывая заканчивать их буквой *b*. Кроме того, как вы знаете, команда and может маскировать значения, устанавливая один или несколько битов в 0.

```
and  al, 11110000b
```

Хотя двоичное представление и помогает прояснить, на какие биты оказывается воздействие командами, вам все равно приходится подсчитывать их и тратить время на визуализацию вашей логики. В сложных программах легко установить или сбросить неверный бит, а такие ошибки находятся затем очень тяжело. Для упрощения работы с битами Turbo Assembler предоставляет два средства — RECORD и MASK.

Описание типов RECORD

RECORD является директивой, позволяющей вам давать имена битовым полям в байтах и словах. Вы просто определяете ширину каждого поля — другими словами, число битов, занимаемое полем. После этого Turbo Assembler сам подсчитывает положение поля. Например, следующая директива RECORD определяет signedByte как 8-битовое значение с двумя полями:

```
RECORD       signedByte      sign  1, value  7
```

После директивы RECORD следует имя записи и список имен полей. Каждое имя поля заканчивается двоет чием и шириной поля в битах. Поле sign в этом примере имеет размер 1 бит, поле value — 7 бит. Поля разделяют запятыми. Если суммарное количество битов меньше или равно 8, Turbo Assembler считает запись байтом, в про тивном случае — словом. Вы не можете создавать записи, большие, чем слово, хотя аналогичный результат можн получить с помощью структуры, содержащей многочисленные битовые поля. Не обязательно определять строго или 16 бит, хотя большинство программистов поступают именно так, вставляя для заполнения битовой записи пус тые поля и учитывая каждый бит, независимо от того, используется он или нет.

Переменные типа RECORD создаются аналогично переменным структур и объединений. Фактически эти тр формы определения идентичны, что скрадывает разницу между структурами и записями. Данная ситуация прояснн ется в следующих примерах:

```
DATASEG
v1    signedByte      <>              ; Значения, устанавливаемые по умолчанию
v2    signedByte      <1>             , sign = 1, value - по умолчанию
v3    signedByte      <,5>            , sign - по умолчанию, value = 5
v4    signedByte      <1,127>         ; sign = 1, value = 127
v5    signedByte      <3,300>         , sign = 1, value = 44
```

Описание переменной записи состоит из трех частей: метки, имени записи и двух угловых скобок с необяз тельными значениями внутри. В первом примере v1 описывается как переменная типа signedByte. Значения в скоб ках отсутствуют, поэтому для всех полей используются значения, устанавливаемые по умолчанию. (В данном слу чае таким значением является 0. Далее вы увидите, как для этих целей устанавливаются другие значения.) Во вто ром примере бит sign переменной v2 устанавливается в 1, поле value инициализируется по умолчанию. В третьей строке value устанавливается в 5, в поле sign используется значение по умолчанию. В четвертой строке задаются значения для обоих полей переменной, sign устанавливается в 1, value — в 127. В пятой строке показано, что про изойдет, если вы попытаетесь использовать значения, выходящие за допустимый диапазон, такие как 3 и 300 В этом случае действительные значения будут равны остатку от деления значений, которые вы пытаетесь вставить, н 2*n, где n равно числу битов в поле.

Установка новых значений по умолчанию для битовых полей

Обычно по умолчанию поля переменных RECORD имеют значения 0. Чтобы это изменить, добавьте к ширин поля знак равенства и новое желаемое значение. Например, для создания записи, в которой по умолчанию старша значащая цифра равна 1, а значение второго поля — 5, можно написать:

```
RECORD      minusByte       msign . 1 = 1, mvalue  7 = 5
```

В результате при описании переменной этого типа с пустыми угловыми скобками поле msign устанавливается 1, поле mvalue — в 5. Эти значения можно изменить, определив новые значения в угловых скобках, как это объя нялось выше. Обратите внимание на то, что здесь используются разные имена полей. Несмотря на то что имена со держатся в определении RECORD, Turbo Assembler считает их глобальными, т.е. активными во всех местах про граммы или модуля. Поэтому в одном модуле во всех определениях RECORD вы должны использовать уникал ные имена полей.

Замечание

В отличие от имен полей RECORD, имена полей STRUC и UNION — не глобальные. Вы можете повторно использо вать для других целей имена полей структур и объединений, но это нельзя делать с именами полей записи, которые долж ны быть уникальными на протяжении всей программы. Вероятно, в следующих версиях Turbo Assembler это несоответст вие будет устранено и имена полей RECORD станут для записи локальными.

Использование переменных RECORD

Описав тип RECORD и несколько переменных данного типа, можно использовать различные способы для чт ния и записи значений в битовые поля этих переменных. Чтобы проиллюстрировать это, сначала определим новы тип RECORD:

```
RECORD      person  sex : 1, married : 1, children : 4, xxx : 1, age : 7, school : 2
```

Записи, подобные этой, могут размещать в небольшом пространстве большое количество информации. В это примере для записи пяти пунктов анкеты требуется только 16 бит — поле sex равно 0 для мужского пола и 1 — д женского, married равно 0, если нет, или 1, если да, значения поля children может изменяться от 0 до 15, однобит

вое поле xxx зарезервировано для будущего использования, значения поля age находятся в диапазоне от 0 до 127, school изменяется от 0 до 3, представляя четыре уровня образования. На рис. 6.4 показано, как эти поля упаковываются в одно 16-битовое слово. Так же как и во всех 16-битовых значениях, два 8-битовых байта этой переменной записываются в памяти в обратном порядке — биты 0–7 (младший значащий байт) в более низких адресах, чем биты 8–15 (старший значащий байт).

Что скрывается за именем поля?

Turbo Assembler преобразует имена битовых полей в значение, равное числу сдвигов вправо, необходимых, чтобы переместить поле в правый конец байта или слова. Это значение равно номеру позиции младшего значащего разряда данного поля. Возвращаясь к записи person, имеем: sex=15, married=14, children=10, xxx=9, age=2 и school=0 (рис. 6.4). Можно использовать эти константы имен полей как простые макроопределения EQU, хотя обычно пользуются этими значениями для сдвига битовых полей в правый конец регистра, упрощая этим обработку значений отдельных полей. Этот процесс можно выполнять также в обратную сторону. Если значение битового поля children уже находится в правом конце ax, сдвиг ax влево на количество битов, соответствующее имени children, перемещает это значение в надлежащую позицию для упаковки в запись.

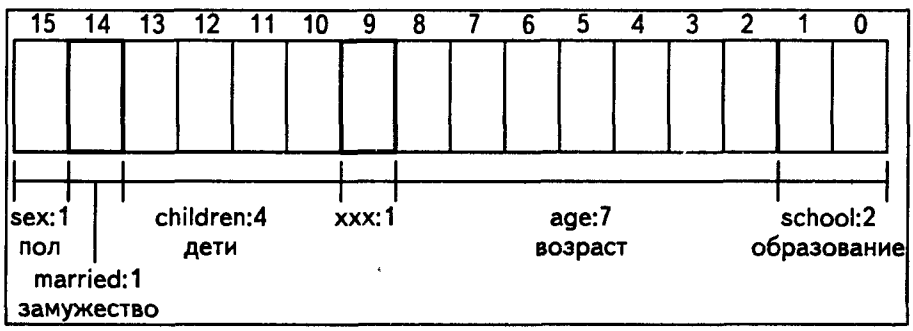


Рис. 6.4. Упакованная запись с шестью битовыми полями, хранящая в небольшом пространстве большое количество информации

Использование имен полей избавляет от ручного подсчета битов, экономит время и помогает предотвращать ошибки. Например, чтобы увеличить на 1 поле age, нужно сдвинуть значение соответствующего битового поля в самый правый край 16-битового регистра, увеличить значение в регистре на 1 и затем вернуть полученный результат на старое место. До этого, конечно, вы должны удалить из переменной остальные биты. Для выполнения этого шага Turbo Assembler предоставляет оператор с именем MASK, который берет имя битового поля и генерирует маску and, с битами, равными 1 во всех позициях, соответствующих этому полю. При создании масок хорошо использовать имена, аналогичные именам связанных с ними полей:

```
maskSex      = MASK sex
maskMarried  = MASK married
maskChildren = MASK children
maskAge      = MASK age
maskSchool   = MASK school
```

Каждый новый идентификатор — например maskSex и maskMarried, определяет маску для каждого битового поля (кроме поля xxx, которое просто игнорируется). Такие имена способствуют более простому запоминанию различных символов, хотя можно использовать имена по своему усмотрению. Вы не обязаны начинать идентификатор с "mask". С помощью имен битовых полей и масок легко выделять и обрабатывать информацию без подсчета позиций полей в записи. Это объясняется в следующем примере. Сначала опишем переменную с именем subject типа person:

```
DATASEG
subject              person          <>
```

Затем, чтобы установить однобитовые поля в 1, используем or для соединения маски с текущим значением записи:

```
CODESEG
or   [subject], maskSex       ; Установить поле sex = 1
or   [subject], maskMarried   ; Установить поле married = 1
```

Для сброса однобитовых полей в 0 применим к битовой маске оператор NOT, изменив значения всех ее битов на противоположные. Работа выполняется в два этапа, показанных в следующих строках:

```
and  [subject], NOT maskSex    , Обнулить поле sex
mov  ax, [subject]             , Загрузить subject в ax
and  ax, NOT maskMarried       ; Обнулить поле married
mov  [subject], ax             ; Записать результат обратно в память
```

Выделение битовых полей

Для битовых полей, больших 1 бита, процесс аналогичен, однако для выделения необходимых значений требуются дополнительные шаги. Вы можете использовать различные методы, но всегда подходят следующие шаги.

1. Скопировать начальное значение переменной в регистр.
2. Выполнить операцию AND над регистром и маской поля.
3. Сдвинуть регистр вправо на константу, соответствующую имени поля.

После копирования переменной в регистр (в 8- или 16-битовый, в зависимости от размера переменной) шаг 2 выделяет биты поля, удаляя из записи остальные поля и устанавливая все биты, кроме содержащихся в требуемом поле в 0. На шаге 3 биты выделенного поля сдвигаются в самый правый край регистра. Чтобы добавить новый член в нашу семью subject, используйте следующие шаги:

```
mov  ax, [subject]             , Шаг 1 - скопировать переменную
and  ax, maskChildren          ; Шаг 2 - выделить битовое поле
mov  cl, children              , Подготовить счетчик сдвига
shr  ax, cl                    , Шаг 3 - сдвиг поля вправо
inc  ax                        , Прибавить 1 к количеству детей
```

Команды mov и and копируют переменную subject в ax и удаляют из значения другие поля, оставляя только биты, используемые для children. После загрузки счетчика сдвигов в cl команда shr сдвигает поле children в дальний правый край ax, подготавливая условия для увеличения этого значения с помощью inc. Если поле children уже находится в правом конце переменной — установите счетчик сдвигов равным 0 — команды сдвигов могут быть пропущены. Например, можно написать:

```
     mov     cl, children          ; Поместить счетчик сдвигов в cl
     or      cl, cl                , Счетчик = 0 ?
     jz      @@10                  , Переход , если да, cl = 0
     shr     ax, cl                , Иначе сдвиг ax, cl раз
@@10
     inc     ax                    ; Прибавить 1 к количеству детей
```

Здесь лучше использовать условную директиву IF, которая подробно рассматривается в главе 8. Она позволяет ассемблеру, а не программе, решать, нужен ли в данной ситуации сдвиг. После выполнения шагов 1 и 2, копирующих и маскирующих переменную записи, с помощью следующих команд результат сдвигается вправо только в том случае, если константа children больше 0:

```
IF   childrenGT 0
     mov     cl, children          , Переслать ненулевой счетчик в cl
     shr     ax, cl                , Сдвиг ax, cl раз
ENDIF
     inc     ax                    ; Прибавить 1 к числу детей
```

Если выражение в условном IF истинно, тогда Turbo Assembler ассемблирует код до следующей директивы ENDIF. Если выражение ложно, код пропускается. Этот метод устраняет ненужное сравнение, переход и команды сдвига из предыдущего подхода.

Вставка битовых полей

После выделения битового поля и обработки его значения необходимо вставить результат обратно в переменную записи. Предполагая, что результат находится в правом конце регистра, выполните следующих четыре шага

1. Сдвиньте регистр влево на константу имени поля.
2. Выполните операцию AND над регистром и маской поля.
3. Выполните операцию AND над первоначальным значением и маской поля, с наложенным на нее оператором NOT.
4. Примените операцию OR к первоначальному значению и регистру.

Шаг 1 сдвигает значение в нужную позицию, снова используя имя поля в качестве счетчика сдвигов, но в этот раз сдвиг выполняется не вправо, а влево. Шаг 2 является необязательным, ограничивающим для безопасности размер нового значения шириной поля в битах. Если вы уверены, что новое значение поля находится в допустимом диапазоне, тогда этот шаг можно пропустить. Однако любое значение, выходящее за допустимый диапазон, например случайно введенное количество — 45 детей, может изменить значения других полей. По этой причине желательно до возврата значения обратно в начальную переменную маскировать его следующим способом. Шаг 3 дополняет шаг 2, устанавливая в начальном значении все биты поля в 0, как бы проделывая отверстие, подобно форме для печенья, продавливающей круговые отверстия в тесте. Шаг 4 с помощью оператора OR помещает новое значение в подготовленное место, завершая процесс.

Иллюстрируя эти четыре шага в языке ассемблера, в следующем фрагменте значение поля children (находящееся в правом конце ax) возвращается обратно в переменную subject:

```
mov  cl, children                    ; Переслать счетчик сдвигов в cl
shl  ax, cl                          ; Шаг 1 - сдвиг в нужную позицию
and  ax, maskChildren                ; Шаг 2 - ограничение значения
and  [subject], NOT maskChildren     ; Шаг 3 - подготовить место для вставки
or   [subject], ax                   ; Шаг 4 - вставить новое значение
```

Так же как и при выделении битового поля, можно использовать условную директиву IF, пропускающую сдвиг, если children=0, и отражающую тот факт, что поле уже находится в правом конце переменной. Кроме того, можно устранить первую команду and, если результат не может быть большим 15 — максимального значения, которое может выражаться 4-битовым полем children.

Объединяя вместе выделение и вставку, в следующем примере поле age из нашей переменной subject увеличивается на 10:

```
mov  ax, [subject]           ; Скопировать переменную в ax
and  ax, maskAge             ; Выделить поле age
mov  cl, age                 ; Подготовить счетчик сдвига
shr  ax,cl                   ; Сдвинуть поле age вправо
add  ax, 10                  ; Прибавить 10 к age
shl  ax, cl                  ; Сдвинуть age назад в нужную позицию
and  ax, maskAge             ; Ограничить age максимальным диапазоном
and  [subject], NOT maskAge  ; Подготовить место в поле age (заполнить 0)
or   [subject], ax           ; Вставить новое значение для age
```

Многие программисты избегают использования битовых полей, вероятно, из-за того, что просто не понимают эти методы. Это видно из многих программ на языке ассемблера, в которых описываются константы для сдвига значений и маскирования, затрудняющие модификацию кода. Если вы потратите время на изучение того, как использовать RECORD и MASK, то, определяя, как показано здесь, упакованные записи, вы сможете писать программы, которые автоматически настраиваются на новые ситуации — изменение числа битов в поле school или использование зарезервированного однобитового поля xxx. Кроме того, можно изменять назначенные по умолчанию значения полей — без необходимости просмотра большого количества операторов и внесения в программу ненужных изменений! Только измените определения RECORD — и все готово. Такими же достоинствами обладают STRUCT и UNION, которые помогают упростить работу со сложными структурами данных.

Эффективные логические операции

Выражение "Всегда есть возможности для улучшения" особенно верно для языка ассемблера. Одно, часто не используемое, улучшение — это замена команд, работающих со словами на более короткие и потенциально более быстрые команды, выполняющие в определенных ситуациях аналогичные действия.

Например, при проверке битов в записи или при установке битов с помощью XOR, даже работая со значением 16-битового слова, можно использовать байтовые команды, если целевой бит находится в младшей части слова. Следующий пример поможет прояснить проблему и ее решение. Рассмотрим запись с битовыми полями:

```
RECORD      BitRec  b0 : 1, b1 : 4, b2 : 3, b3 : 7
```

Логические команды and, or, test и xor могут оперировать битами переменных записи, ссылаясь на метки b0, b1, b2 и b3. Например, можно установить бит b2 в регистре ax с помощью команды

```
or   ax, b2
```

При ассемблировании генерируется команда, основанная на слове и занимающая три байта машинного кода:

```
0D   07      00
```

Однако следующая команда, выполняющая те же действия и так же влияющая на флаги процессора, будет закодирована эффективнее:

```
or   al, b2
```

После ассемблирования эта команда будет занимать только два байта машинного кода:

```
0C 07
```

Несмотря на то что переменная расположена в 16-битовом регистре ax, 8-битовая команда, работающая с 8-битовым младшим байтом регистра al, приводит к тому же результату.

Автоматическое использование эффективных логических операций

Для автоматического использования эффективных логических команд Turbo Assembler версии 3.0 и выше пре доставляет четыре псевдокоманды: SETFLAG, MASKFLAG, TESTFLAG и FLITFLAG. С их помощью ассемблер может автоматически выбирать наиболее эффективную форму логической команды. Например, ассемблер заменяет команду

```
SETFLAG      ax, b2
```

более эффективной

```
or   al, 07
```

а не эквивалентной, но менее эффективной командой, которая может показаться здесь необходимой:

```
or   ax, b2
```

В следующем фрагменте показано, как использовать псевдокоманды. В комментариях представлен ассемблерный код. Например, команда SETFLAG в первой строке расшифровывается как байтовая логическая команда or. Эквивалентная ей, но потенциально менее эффективная команда находится во второй строке. Обратите внимание на то, что в этом примере не допускается замена логического and байтовой командой:

```
SETFLAG      ax, b2          ; or    al, 07          / 0C07
     or      ax, b2          ; or    ax, 0007        / 0D0700
MASKFLAG     ax, b2          ; and   ax, 0007        / 250700
     and     ax, b2          ; and   ax, 0007        / 250700
TESTFLAG     ax, b2          ; test  al, 07          / A807
     test    ax, b2          ; test  ax, 00007       / A90700
FLIPFLAG     ax, b2          ; xor   al, 07          / 3407
     xor     ax, b2          ; xor   ax, 0007        / 350700
```

Автоматические команды для работы с полями записи

Turbo Assembler 3.0 и следующих версий предоставляет две дополнительные псевдокоманды — SETFIELD и GETFIELD, которые существенно упрощают работу с битовыми полями записей. До их применения вы должны быть уже знакомы с дискуссией из этой главы о маскировании переменных записи для установки или восстановления значений битов, упакованных в байты или слова.

Несколько примеров покажут, как эти новые команды могут упростить процесс вставки и выделения полей в записи person. Чтобы не перелистывать страницы, повторим описание этой записи:

```
RECORD       person  sex  1, married  1, children  4, xxx  1, age  7, school : 2
```

Как вы знаете, для установки и восстановления значений полей записи person необходимо использовать комбинацию сдвигов, ротаций и логических команд, однако Turbo Assembler версии 3.0 и выше может сам создавать за вас необходимые команды. Например, первая команда подготавливает регистр для хранения записи person:

```
xor  ax, ax          ; Очистить запись person
```

Эта команда просто устанавливает 0 в регистре ax. Для того чтобы вставить значение в поле children, сначала поместите это значение в регистр (здесь bl) и используйте SETFIELD следующим образом:

```
mov          bl, 3                ; Поместить в bl количество детей
SETFIELD     children ax, bl      ; Установить в ax поле children
```

Во второй строке значение bl вставляется в ax, *не оказывая воздействия на другие биты* ax. Для этого Turbo Assembler помещает на место псевдокоманды SETFIELD следующие логические операции:

```
rol  bl, 02
or   ah, bl
```

Первая команда циклически сдвигает значение children влево на две позиции, вторая команда помещает это значение в ax, используя логическое OR. Ассемблер также выбирает более эффективную байтовую форму логического or, вместо того чтобы оперировать целым 16-битовым словом.

Аналогично можно использовать SETFIELD для вставки значений в любое поле записи, например в поле age:

```
mov         bl  43          , Записать age в bl
SETFIELD    age ax, bl      , Установить в ax поле age
```

При этом для вставки значения age в ax без воздействия на остальные поля записи генерируется другой набор циклических сдвигов и логических операций.

Для выделения из записи значения битового поля используется GETFIELD. Например, следующая команда устанавливает в bl значение children из записи person, содержащейся в регистре ax:

```
GETFIELD children bl, ax , Получить в bl значение children (bh разрушается!)
```

Так как предварительно была выполнена команда SETFIELD, в bl окажется значение 03. Вместо псевдокоманды GETFIELD Turbo Assembler записывает следующие команды:

```
mov  bl  ah
ror  bl  02
and  bl, 0F
```

В первой строке часть записи, содержащая требуемое значение битового поля (ah), пересылается в bl, во второй выполняется циклический сдвиг на две позиции вправо, в результате чего значение перемещается в правый конец bl, в третьей для выделения требуемого значения применяется литеральная маска 0Fh, которая в этом примере устанавливает в bl значение 03.

Аналогично можно использовать GETFIELD для выделения значения age из записи, содержащейся в ax:

```
GETFIELD    age bl, ax      , Получить age в bl (bh разрушается!)
```

Ассемблер генерирует при этом другой набор логических операций, чем при упаковке значения age в запись.

Одна неприятность с командой GETFIELD состоит в том, что она всегда использует целевой 16-битовый регистр целиком, даже в том случае, если вы задали только его младшую часть. В двух предыдущих примерах, как отмечено в комментариях, логические команды, генерируемые Turbo Assembler, могут разрушить старший значащий байт bh.

Для SETFIELD и GETFIELD можно использовать другие регистры и обращения к памяти, а не только ax и bl, как показано здесь. Полный синтаксис для обеих псевдокоманд имеет следующий вид:

```
SETFIELD    field_name destinat on_r/m  source_reg
GETFIELD    field_name destination_reg  source_r/m
```

Используйте эти правила при создании команд SETFIELD и GETFIELD. Каждая команда требует, чтобы за именем поля следовали определения назначения и источника. Назначение для SETFIELD может быть регистром или ссылкой на память. Источник должен быть регистром. Назначение для GETFIELD должно быть регистром, источник может быть регистром или ссылкой на память.

Замечание

Команды, генерируемые для SETFIELD и GETFIELD, зависят от значений операндов, т.е регистров и ссылок на память, а также от позиций значений в записи с битовыми полями. В предыдущих примерах была показана только одна из нескольких возможных последовательностей команд Для исследования других последовательностей создайте примеры команд SETFIELD, GETFIELD и просмотрите ассемблированный код в окне CPU Turbo Debugger.

Использование заранее указываемых макроопределений

Turbo Assembler предоставляет несколько заранее указываемых макроопределений, которые можно без описания использовать в качестве значений переменных программы. Эти макроопределения представлены в табл. 6.1, каждое из них начинается с двух знаков вопроса.

Программа 6.2, VERSION.ASM, показывает, как использовать эти макроопределения для автоматического создания при ассемблировании строки с информацией о версии программы. Выполните ассемблирование, компоновку и запустите программу на исполнение с помощью команд:

```
tasm        version
tlink       version,,, mta
version
```

Таблица 6.1. **Заранее указываемые макроопределения**

Символ	Значение
??Date	Текущая дата в стиле DOS
??Filename	Имя модуля или дискового файла программы
??Time	Текущее время в стиле DOS
??Version	Номер версии Turbo Assembler

VERSION использует модули STRIO и STRING из главы 5, поэтому при выполнении tlink подразумевается что в MTA.LIB уже содержится ассемблированный код этих модулей. Если вы хотите изучать программу в Turbo Debugger, добавьте опцию /zi к tasm и опцию /v к tlink, о которых вы, вероятно, уже знаете.

Программа 6.2. VERSION.ASM

```
1:    %TITLE "Automatic Program Version Demo -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:    cr    EQU   13                    ; ASCII-символ возврата каретки
9:    lf    EQU   10                    ; ASCII-символ прогона строки
10:
11:
12:         DATASEG
13:
14:   exCode db    0
15:
16:   ident db    cr, lf, ??FileName,' ', ??Date,' ', ??Time
17:               db                cr, lf, 0
18:
19:
20:         CODESEG
21:
22:   ;----- Из STRIO.OBJ
23:
24:         EXTRN StrWrite:proc
25:
26:   Start:
27:         mov   ax, @data           ; Поместить в DS адрес
28:         mov   ds, ax              ; сегмента данных
29:         mov   es, ax              ; Сделать es = ds
30:
31:         mov   di, OFFSET ident    ; Загрузить di адресом строки ident
32:         call  StrWrite            ; Вывести строку
33:
34:   Exit:
35:         mov   ah, 04Ch            ; Функция DOS : Выход из программы
36:         mov   al, [exCode]        ; Возвратить значение кода выхода
37:         int   21h                 ; Вызов DOS. Останов программы
38:
39:         END   Start               ; Конец программы / точка входа
```

Выполнение VERSION

В строках 16–17 создается ASCIIZ-строка, начинающаяся и заканчивающаяся символами возврата каретки и перехода на новую строку плюс нулевой байт-ограничитель. Внутри строки в директиве db используются предопределенные макроопределения ??FileName, ??Date и ??Time для создания строки, содержащей эти значения, разделенные несколькими пробелами. При выполнении программы выводится строка вида

```
version      02/15/95        08 : 13 : 23
```

Полезность создания такой автоматической строки состоит в том, что дата и время версии автоматически изменяются при каждом повторном ассемблировании программы. Это очень удобно для отслеживания изменений, вносимых в программу.

Преобразования чисел и строк

В языках высокого уровня можно непосредственно считывать и записывать числовые значения. Например, для ввода числа и отображения результата, подразумевая, что n является целым, можно использовать следующие операторы Pascal:

```
Write ( ' Enter a value . ' ) ;
ReadLn (n) ;
WriteLn ( ' Value is : ', n );
```

В чистом ассемблере такие возможности отсутствуют. Вместо этого нужно считывать и записывать строки, а затем преобразовывать их для обработки в двоичные значения и из двоичных значений, записывая на диск, и т.д. Естественно, это же автоматически осуществляется и в языках высокого уровня.

Программа 6.3, BINASC.ASM, является модулем, который вы можете использовать для упрощения этого процесса в своих программах. Модуль содержит процедуры, преобразующие 16-битовые значения в знаковые и беззнаковые десятичные и шестнадцатеричные значения, в двоичные ASCIIZ-строки и обратно. Ассемблируйте этот модуль для получения BINASC.OBJ и запишите его в файл MTA.LIB с помощью команд:

```
tasm /z binasc
tlib / E mta -+binasc
```

Так же как и с модулями из главы 5, не обращайте внимания на предупреждение о том, что BINASC не содержится в библиотеке. Оно появляется только при первой установке модуля в библиотеку. Кроме того, имейте в виду, что BINASC использует две процедуры из STRING; поэтому, пока эти два модуля не установлены в MTA.LIB, вы не можете компоновать свои программы с BINASC.

Программа 6.3. BINASC.ASM

```
1:    %TITLE "Binary to/from ASCII Conversion -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:
7:    ;----- Макроопределения
8:
9:    ASCnull     EQU              0       ; Нулевой ASCII-символ
10:
11:         DATASEG
12:
13:         CODESEG
14:
15:   ;----- Из STRINGS.OBJ
16:
17:         EXTRN StrLength:proc, StrUpper:proc
18:
19:         PUBLIC                HexDigit, ValCh, NumToAscii
20:         PUBLIC                BinToAscHex, SBinToAscDec, BinToAscDec, BinToAscBin
21:         PUBLIC                AscToBin
22:
23:   %NEWPAGE
```

```
;-------------------------------------------------------------
; HexDigit Преобразует 4-битовое значение в ASCII цифру
;-------------------------------------------------------------
; Вход:
; dl = значение в диапазоне 0..15
; Выход:
; dl = шестнадцатеричный эквивалент ASCII цифры
; Регистры:
; dl
;-------------------------------------------------------------
PROC HexDigit
      cmp   dl, 10              ; dl < 10 (т.е. шестнадцатеричного 'A')?
      jb    @@10                ; если да, то переход
      add   dl, 'A'-10          ; Иначе преобразовать в A, B, C, D, E или F
      ret                       ; Возврат в вызывающую программу
@@10:
      or    dl, '0'             ; Преобразовать цифры от 0 до 9
      ret                       ; Возврат в вызывающую программу
ENDP HexDigit
%NEWPAGE
;-------------------------------------------------------------
; ValCh Преобразует символ ASCII цифры в двоичное значение
;-------------------------------------------------------------
; Вход:
; dl = ASCII цифры '0'..'9'; 'A'..'F'
; bx = основание (2=двоичная, 10=десятичная, 16=шестнадцатеричная)
; Выход:
; cf = 0: dx = эквивалентное двоичное значение
; cf = 1: неверный символ для данного основания (значение в dx смысла не имеет)
; Регистры:
; dx
;-------------------------------------------------------------
PROC ValCh
      cmp   dl, '9'             ; Проверить на возможные шестнадцатеричные цифры
      jbe   @@10                ; В случае '0'..'9' переход
      sub   dl, 7               ; Скорректировать в диапазон 3A..3F
@@10:
      sub   dl, '0'             ; Преобразовать из ASCII в десятичную
      test  dl, 0f0h            ; Проверить 4 старших бита (установить cf=0)
      jnz   @@99                ; Переход, если ошибка (не цифра и не A - F)

      xor   dh, dh              ; Преобразовать байт из dl в слово dx
      cmp   dx, bx              ; Сравнить с основанием (cf=1, если все в порядке)
@@99:
      cmc                       ; Инвертировать cf /сбросить флаг ошибки
      ret                       ; Возврат в вызывающую программу
ENDP ValCh
%NEWPAGE
;-------------------------------------------------------------
; NumToASCII Преобразует беззнаковое двоичное значение в ASCII
;-------------------------------------------------------------
; Вход:
; ax = 16-битовое преобразуемое значение
; bx = основание результата (2=двоичный; 10=десятичный; 16=шестнадцатеричный)
; cx = минимальное количество выводимых цифр
; di = адрес строки с результатом
; Замечание: подразумевается, что результат должен поместиться в строку
; Замечание: результат получается полным, если cx меньше количества цифр.
; необходимых для его представления, или cx = 0
; Замечание: если cx=0 и ax=0, тогда длина строки будет равна 0
; установите cx=1, если хотите получить строку = '0' при ax=0
; Замечание: подразумевается (2<=bx<=16)
; Выход:
; отсутствует
; Регистры:
; ax, cx
;-------------------------------------------------------------
PROC NumToASCII                 ; Обычная точка входа
      push  dx                  ; Сохранить некоторые изменяемые регистры
      push  di
      push  si

; si = количество цифр в стеке
```

```
        xor   si, si                ; Установить счетчик цифр в 0
        jcxz  @@20                  ; Если cx = 0, переход для установки cx = 1
@@10:
        xor   dx, dx                ; Расширить ax до 32-битового dxax
        div   bx                    ; ax<-axdx div bx; dx<-остаток
        call  HexDigit              ; Преобразовать dl в ASCII-цифру
        push  dx                    ; Сохранить цифру в стеке
        inc   si                    ; Увеличить подсчитываемое количество цифр в стеке
        loop  @@10                  ; Возврат, пока не обработано минимальное число цифр
@@20:
        inc   cx                    ; Установить cx = 1, если не все цифры обработаны
        or    ax, ax                ; ax = 0? (все цифры обработаны)
        jnz   @@10                  ; Если ax <> 0, продолжить преобразование
        mov   cx, si                ; Установить в cx количество символов в стеке
        jcxz  @@40                  ; Пропустить следующий цикл, если cx = 0000
        cld                         ; Автоувеличение di для stosb
@@30:
        pop   ax                    ; Вытолкнуть в al следующую цифру из стека
        stosb                       ; Записать цифру в строку; увеличить di
        loop  @@30                  ; Цикл для вывода cx цифр
@@40:
        mov   [byte di], ASCnull    ; Дописать нуль к концу строки
        pop   si                    ; Восстановить сохраненные регистры
        pop   di
        pop   dx

        ret                         ; Возврат в вызывающую программу
ENDP NumToASCII
%NEWPAGE
;---------------------------------------------------------------
; BinToAscHex Преобразует двоичные значения в шестнадцатеричные ASCII-строки
;---------------------------------------------------------------
; Вход:
; ax = 16-битовое преобразуемое значение
; cx = минимальное количество выводимых чисел
; di = адрес строки с результатом
; Замечание: подразумевается, что результат умещается в строке
; Замечание: результат получается полным, если
; cx меньше количества цифр, необходимых для его представления
; Выход:
; отсутствует
; Регистры:
; ax, cx
;---------------------------------------------------------------
PROC BinToAscHex
        push  bx                    ; Сохранить bx в стеке
        mov   bx, 16                ; Установить основание = 16 (шестнадцатеричная)
        call  NumToAscii            ; Преобразовать ax в ASCII
        pop   bx                    ; Восстановить bx
        ret                         ; Возврат в вызывающую программу
ENDP BinToAscHex
%NEWPAGE
;---------------------------------------------------------------
; BinToAscDec Преобразует двоичные значения в десятичные ASCII-строки
;---------------------------------------------------------------
; Вход:
; Тот же, что и в BinToAscHex
; Выход:
; отсутствует
; Регистры:
; ax, cx (косвенно)
;---------------------------------------------------------------
PROC BinToAscDec
        push  bx                    ; Сохранить bx в стеке
        mov   bx, 10                ; Установить основание = 10 (десятичная)
        call  NumToAscii            ; Преобразовать ax в ASCII
        pop   bx                    ; Восстановить bx
        ret                         ; Возврат в вызывающую программу
ENDP BinToAscDec
%NEWPAGE
;---------------------------------------------------------------
; SBinToAscDec Преобразует знаковые двоичные значения в десятичные ASCII-строки
;---------------------------------------------------------------
; Вход:
```

```
172:  ; Тот же, что и в BinToAscHex (ax = знаковое 16-битовое значение)
173:  ; Выход:
174:  ; отсутствует
175:  ; Регистры:
176:  ; ax, cx
177:  ;--------------------------------------------------------------
178:  PROC SBinToAscDec
179:        push   bx                  ; Сохранить bx и di
180:        push   di
181:        cmp    ax, 0               ; ax < 0?
182:        jge    @@10                ; Переход, если ax >= 0
183:        neg    ax                  ; Построить в ax значение в дополнительном коде
184:        mov    [byte di],'-'       ; Вставить '-' в строку (минус)
185:        inc    di                  ; Увеличить указатель строки
186:  @@10:
187:        mov    bx, 10              ; Установить основание = 10 (десятичная)
188:        call   NumToAscii          ; Преобразовать ax в ASCII
189:        pop    di                  ; Восстановить bx и di
190:        pop    bx
191:        ret                        ; Возврат в вызывающую программу
192:  ENDP SBinToAscDec
193:  %NEWPAGE
194:  ;--------------------------------------------------------------
195:  ; BinToAscBin Преобразует двоичные значения в двоичные ASCII-строки
196:  ;--------------------------------------------------------------
197:  ; Вход:
198:  ; Тот же, что и в BinToAscHex
199:  ; Выход:
200:  ; отсутствует
201:  ; Регистры:
202:  ; ax, cx (косвенно)
203:  ;--------------------------------------------------------------
204:  PROC BinToAscBin
205:        push   bx                  ; Сохранить bx в стеке
206:        mov    bx, 2               ; Установить основание = 2 (двоичная)
207:        call   NumToAscii          ; Преобразовать ax в ASCII
208:        pop    bx                  ; Восстановить bx
209:        ret                        ; Возврат в вызывающую программу
210:  ENDP BinToAscBin
211:  %NEWPAGE
212:  ;--------------------------------------------------------------
213:  ; ChToBase Возвращает основание для строки
214:  ;--------------------------------------------------------------
215:  ; Замечание:
216:  ; Процедура, используемая AscToBin. Непосредственно не вызывается
217:  ; Вход:
218:  ; si = указатель на нулевой ограничитель в конце строки
219:  ; Замечание: подразумевается, что длина строки >= 1
220:  ; Выход:
221:  ; bx = 2(двоичная), 10(десятичная по умолчанию), 16(шестнадцатеричная)
222:  ; si = адрес последней возможной в строке цифры
223:  ; Регистры
224:  ; bx, dl, si
225:  ;--------------------------------------------------------------
226:  PROC ChToBase
227:        mov    dl, [byte si-1]     ; Взять последний символ строки
228:        mov    bx, 16              ; Установить основание 16 (шестнадцатеричная)
229:        cmp    dl, 'H'             ; Это шестнадцатеричная строка?
230         je     @@10                ; Переход, если шестнадцатеричная
231         mov    bx, 2               ; Установить основание 2 (двоичная)
232:        cmp    dl, 'B'             ; Это двоичная строка?
233:        je     @@10                ; Переход, если двоичная
234:        mov    bx, 10              ; Установить основание 10 (десятичная)
235:        cmp    dl, 'D'             ; Это десятичная строка?
236:        jne    @@20                ; Переход, если не десятичная
237:  @@10:
238:        dec    si                  ; Установить si на следующую цифру
239:  @@20:
240:        ret                        ; Возврат в вызывающую программу
241:  ENDP ChToBase
242:  %NEWPAGE
243:  ;--------------------------------------------------------------
244:  ; AscToNum Преобразует ASCII-символы в двоичные
245:  ;--------------------------------------------------------------
```

```
246:  ; Замечание:
247:  ; Процедура используется в AscToBin. Непосредственно не вызывается
248:  ; Вход:
249:  ; ax = начальное значение (0)
250:  ; bx = основание (2=двоичная, 10=десятичная, 16=шестнадцатеричная)
251:  ; di = адрес беззнаковой строки (любой формат)
252:  ; si = адрес последней допускаемой в строке цифры
253:  ; Выход:
254:  ; cf = 0 : ax = беззнаковое значение
255:  ; cf = 1 : неверный символ в строке (значение в ax смысла не имеет)
256:  ; Регистры:
257:  ; ax, cx, dx, si
258:  ;---------------------------------------------------------------
259:  PROC AscToNum
260:        mov   cx, 1               ; Инициализация множителя
261:  @@10:
262:        cmp   si, di              ; В начале строки?
263:        je    @@99                ; Выход, если в начале (cf = 0)
264:        dec   si                  ; Настроиться на следующий слева символ
265:        mov   dl, [byte si]       ; Загрузить символ в dl
266         call  ValCh               ; Преобразовать dl в значение в dx
267:        jc    @@99                ; Выход по ошибке (неверный символ)
268:        push  cx                  ; Сохранить cx в стеке
269:        xchg  ax, cx              ; ax = множитель; cx = частное
270:        mul   dx                  ; dxax <- значение цифры * множитель
271:        add   cx, ax              ; cx <- cx + ax (новое частное)
272         pop   ax                  ; Восстановить множитель в ax
273:        mul   bx                  ; dxax <- множитель * основание
274:        xchg  ax, cx              ; ax = частное; cx = новый множитель
275:        jmp   @@10                ; обработать следующую цифру
276:  @@99:
277.        ret                       ; Возврат в вызывающую программу
278   ENDP AscToNum
279:  %NEWPAGE
280:  ;---------------------------------------------------------------
281:  ; AscToBin Преобразует ASCII-строки в двоичные значения
282:  ;---------------------------------------------------------------
283:  ; Вход:
284:  ; di = ASCIIZ строка, преобразуемая в двоичную
285:  ; 'H' в конце строки = шестнадцатеричная
286:  ; 'B' в конце строки = двоичная
287:  ; 'D' или цифра в конце строки = десятичная
288:  ; '-' в s[0] означает отрицательное число
289:  ; Замечание: пробелы в строке не допускаются
290:  ; Выход:
291:  ; cf = 1 : неверный символ в строке (ax не определен)
292:  ; cf = 0 : ax = значение строки
293:  ; Замечание: символы строки преобразуются в прописные
294:  ; Замечание: нулевые строки устанавливают ax в 0
295:  ; Регистры:
296:  ; ax
297:  ;---------------------------------------------------------------
298:  PROC AscToBin
299:        push  bx                  ; Сохранить изменяемые регистры
300:        push  cx                  ; (некоторые из них изменяются
301:        push  dx                  ; в процедурах, вызываемых
302:        push  si                  ; этой процедурой)
303:
304:        call  StrUpper            ; Преобразовать символы строки в прописные
305:        call  StrLength           ; Установить в cx длину строки из di
306:        xor   ax, ax              ; Инициализировать результат 0 (cf = 0)
307:        jcxz  @@99                ; Выход, если длина = 0. ax = 0, cf = 0
308:        mov   si, di              ; Поместить в si адрес строки из di
309:        add   si, cx              ; Перевести si на нуль в конце строки
310:        cmp   [byte di],'-'       ; Проверить на знак минус
311:        pushf                     ; Сохранить результат сравнения
312:        jne   @@10                ; Перейти, если знак минус не найден
313:        inc   di                  ; Передвинуть di за знак минус
314:  @@10:
315:        call  ChToBase            ; Установить bx = основание; si на следующую цифру
316:        call  AscToNum            ; Преобразовать ASCII (основание bx) в число
317:        rcl   bx, 1               ; Сохранить cf путем сдвига в bx
318:        popf                      ; Восстановить флаги, получившиеся после проверки на знак
319:        jne   @@20                ; Перейти, если знак минус не найден
```

```
320:        neg   ax                ; Иначе, построить в ax значение в дополнительном коде
321:        dec   di                ; и восстановить di в голове строки
322:  @@20:
323:        rcr   bx, 1             ; Восстановить cf, получившийся после выполнения AscToNum
324:  @@99:
325:        pop   si                ; Восстановить регистры
326:        pop   dx
327:        pop   cx
328:        pop   bx
329:        ret                     ; Возврат в вызывающую программу
330:  ENDP AscToBin
331:
332:        END                     ; Конец модуля
```

Использование модуля BINASC

В модуле BINASC содержится восемь подпрограмм, которые можно вызывать из своих собственных програм (см. строки 19–21). Две другие подпрограммы используются процедурами модуля. Далее описывается каждая под программа модуля. Ниже приведены две программы, которые также иллюстрируют использование описанных здес процедур.

HexDigit (24–42)

HexDigit преобразует 4-битовое значение в регистре dl в эквивалентную шестнадцатеричную ASCII-цифру Скорее всего, вам не потребуется вызывать эту процедуру, хотя, конечно, ей можно найти применение. Другие про цедуры модуля вызывают HexDigit для преобразования длинных двоичных значений в ASCII-представление.

ASCII-цифрам от 0 до 9 соответствуют шестнадцатеричные значения от 030h до 039h. На основании этого удоб ного свойства добавление шестнадцатеричного значения 30h преобразует любую отдельную цифру в ASCII представление. Значение 04h равно 34h в ASCII, 08h равно 038h и т.д. Для преобразования ASCII-цифры в эквива лентное ей двоичное значение этот процесс просто выполняется в обратную сторону — вычитая 30h.

К сожалению, этот простой план не подходит для шестнадцатеричных символов 0–9 и A–F, требующих в строк 35 HexDigit проверки, является ли dl меньшим десятичного значения 10. Если нет, тогда в строке 37 выполняето преобразование, заменяя значения 0Ah, 0Bh, 0Ch, 0Dh, 0Eh и 0Fh правильными ASCII-символами — A–F. В про тивном случае команда or в строке 40 вставляет 30h в значение, преобразуя в ASCII десятичные цифры 0–9.

Замечание

HexDigit подразумевает, что четыре старших значащих бита равны 0. Другими словами, dl должен находиться в диапа зоне от 0 до 15, иначе результат будет неверным.

ValCh (44–70)

ValCh выполняет функции, обратные HexDigit, преобразуя ASCII цифры 0–9 и A–F в эквивалентные двоичны значения. Поскольку эта процедура используется для преобразования строк с разными основаниями, в ней выпол няется проверка на соответствие символа заданному в bx основанию. Для использования ValCh поместите симво цифры в dl, а основание системы в bx — 2 для двоичной, 10 для десятичной или 16 для шестнадцатеричной:

```
mov  dl, 'A'          ; Преобразуемый символ
mov  bx, 16           ; Основание = шестнадцать
call ValCh            ; Преобразовать dl в двоичное значение в dx
```

ValCh возвращает преобразованное значение в регистре dx. Если встречается неверный символ, флаг cf уст навливается в 1, при этом значение в dx будет неправильным. Обычно после ValCh нужно использовать условны переход, проверяющий эту ситуацию:

```
call ValCh            ; Преобразовать символ из dl в значение в dx
jc   Error            ; Перейти, если обнаружена неверная цифра
```

В процедуре использовано несколько методов, которые могут быть неочевидными при беглом чтении. В строк 57–59 шестнадцатеричные символы A–F преобразуются в ASCII-символы со значениями от 03Ah до 03Fh. (Вы може называть эти значения псевдошестнадцатеричными символами). После выполнения этого шага в dl содержится и неверный символ, или значение в диапазоне от 030h до 03Fh, что упрощает последующее преобразование.

Следующий шаг заключается в преобразовании значения из dl в двоичное значение с помощью вычитания 30h (строка 61). Как объясняется в комментариях к HexDigit, вычитание 30h преобразует символы в двоичные значения. В этом случае вычитание применимо также и для псевдошестнадцатеричных символов из предыдущих шагов.

Команды в строках 65–66 завершают преобразование, обнуляя верхнюю половину dx и используя обычный для 8086 метод xor. После этого dx сравнивается с основанием в bx. Если результат меньше основания, тогда значение находится в допустимом диапазоне, в противном случае начальный символ является неверным. К сожалению, при этом сравнении флаг cf остается в состоянии, противоположном необходимому, но проблема легко решается с помощью команды cmc в строке 68, которая инвертирует флаг переноса, изменяя 1 на 0 и 0 на 1. Это также необходимо, если при проверке в строке 62 оказывается, что значение ASCII-символа не попадает в диапазон от 030h до 03Fh.

NumToASCII (72–125)

NumToASCII является преобразователем двоичных чисел в ASCII общего назначения, который можно использовать для преобразования значений в ASCII строки по любому основанию от 2 до 16. Так как для правильного использования NumToASCII требуются значительные усилия, можно вызывать другие процедуры, такие как BinToASCHex и BinToAscDec, которые, в свою очередь, вызывают NumToASCII для выполнения этих преобразований. Я объясню эти процедуры чуть позже. Но для того чтобы понять, как они работают, вы по крайней мере должны изучить код NumToASCII.

В строках 75–85 NumToASCII приводятся описания регистров и несколько важных замечаний. Подразумевается, что преобразуемое значение содержится в регистре ax, bx равен основанию системы счисления (как объяснялось для ValCh), cx — минимальному вставляемому в строку количеству цифр, es:ds адресуют строковую переменную, достаточно большую, чтобы в ней уместился результат преобразования. Вам помогут разобраться в этом несколько советов.

- В целях безопасности для шестнадцатеричных значений используйте строковую переменную размером по меньшей мере 5 байт, для десятичных значений — 6 байт, для двоичных значений — 17 байт. Эта длина будет достаточной для размещения результата, включая 1 дополнительный байт для ограничителя строки.
- Если вы хотите, чтобы нулевое значение преобразовалось в '0', а не в пустую строку, установите cx в 1. Если cx и преобразуемое значение оба равны 0, в результате получится строка с нулевой длиной.
- Основание в bx не ограничивается 2, 10 и 16. Вы можете преобразовывать двоичные значения в восьмеричные, устанавливая в bx 8 или в значения по другим основаниям. Регистр bx должен находиться в диапазоне от 2 до 16.
- Обычные классифицирующие символы *b*, *d* и *h*, которые используются в конце строк типа 0FA9Ch, 01110010b или 12345d, процедурой не вставляются. При необходимости вы должны добавлять их сами.
- NumToASCII не может преобразовывать в строки отрицательные (в двоичном дополнительном коде) значения. Для этого вызывайте SBinToAscDec, которая создана для обработки целых значений со знаком.

BinToAscHex (127–148)

BinToAscDec (150–166)

SBinToAscDec (168–192)

BinToAscBin (194–210)

Эти четыре процедуры используют одинаковые параметры, поэтому я опишу их вместе. BinToAscHex преобразует 16-битовые беззнаковые значения в шестнадцатеричные строки, BinToAscDec — в десятичные строки; SBinToAscDec преобразует 16-битовые значения в двоичном дополнительном коде в десятичные строки. И наконец, BinToAscBin преобразует 16-битовые значения в двоичные строки.

Замечание

При преобразовании чисел в ASCII всегда выделяйте под результат достаточное пространство в строке. Осознайте, что двоичные значения могут иметь до 16 цифр. Не забывайте оставлять дополнительный байт для нулевого ограничителя. Для страховки лучше выделяйте большее количество пространства. При ускорении работы эти процедуры не предотвращают случайной перезаписи в памяти других переменных.

При использовании любого из этих преобразователей запишите соответствующее значение в ax. В cx установите минимальное допустимое количество цифр в результате — по меньшей мере 1, если вам необходимо, чтобы нуль преобразовывался в '0'. В es:di установите адрес строковой переменной, которая может быть неинициализированной. Например, для загрузки значения из памяти и преобразования в строку можно написать:

```
DATASEG
value       dw      1234            ; 16-битовое десятичное значение
string      db      20 DUP (?)      ; Более чем необходимое пространство
```

```
CODESEG
mov  ax, @data
mov  ds, ax                ; Инициализация ds и es
mov  es, ax                ; адресом сегмента данных программы
mov  ax, [value]           ; Получить преобразуемое значение
mov  cx, 1                 ; Допускается по меньшей мере одна цифра
mov  di, OFFSET string     ; Адрес строковой переменой
call BinToAscDec           ; Преобразовать ax в десятичную строку
```

Вместо BinToAscDec вы можете вызывать любую другую из этих трех процедур — оставшиеся шаги для них одинаковы. Напомню, что этот пример содержит шаги инициализации ds и es. BINASC вызывает процедуры из STRINGS, которые требуют, чтобы es был равен ds.

Замечание

При вызове SBinToAscDec оставляйте один дополнительный символ для знака минуса. Если минимальное количество цифр, заданное в cx, меньше или равно 6, тогда минимальная длина строки должна равняться 7.

ChToBase (212–241)

AscToNum (243–278)

Эти две процедуры являются собственностью модуля BINASC, и вы, вероятно, сможете найти им непосредственное применение. ChToBase возвращает в bx значение, равное основанию строки, заканчивающейся на D или 0-9 — для десятичной, H — шестнадцатеричной и B — двоичной. (Буквы должны быть прописными, строчные буквы **d, h** и **b** в данной процедуре не допускаются.) При входе в ChToBase si адресует нулевой ограничитель строки, при выходе в нем возвращается адрес последней возможной цифры строки. В остальном код очевиден.

AscToNum выполняет преобразование строки из ASCII в двоичное представление, вызывая в цикле процедуру ValCh. Для каждого символа, загруженного в dl, ValCh возвращает эквивалентное значение или показывает ошибку, устанавливая cf. Для лучшего понимания этого механизма выполните эту часть в Turbo Debugger, уделив повышенное внимание значениям в регистрах.

AscToBin (280–330)

Используйте AscToBin для преобразования строк в двоичные значения. Строка должна быть в формате ASCIIZ и может заканчиваться буквой *d* или цифрой — для десятичных значений, *h* — шестнадцатеричных, *b* — двоичных. В es:di установите адрес преобразуемой строки. После завершения AscToBin флаг cf, равный 0, показывает, что результат в ax верный, а флаг, равный 1, говорит о том, что в строке встретился недопустимый символ. В строке не допускаются пробелы, все символы преобразуются в прописные. (Если вы хотите сохранить первоначальную строку, используйте процедуру StrCopy из модуля STRING). Строки нулевой длины устанавливают ax в 0. Следующий пример иллюстрирует различные форматы строк, допускаемые AscToBin:

```
DATASEG
  s1   db      ' 12345 ', 0      ; Десятичная строка (по умолчанию)
  s2   db      ' 54321d ', 0     ; Десятичная строка, заканчивающаяся d
  s3   db      ' -9876 ', 0      ; Отрицательная десятичная строка
  s4   db      ' F19Ch ', 0      ; Шестнадцатеричная строка
  s5   db      ' 1010b ', 0      ; Двоичная строка
  CODESEG
  mov  di, OFFSET s1             ; Адрес строки s1 (или s2-s5)
  call AscToBin                  ; Преобразовать строку в значение в ax
  jc   Error                     ; Перейти, если ошибка, иначе продолжить
```

Как вы можете увидеть из этих примеров, здесь не требуется, как в программах на языке ассемблера, чтобы шестнадцатеричные числа начинались с десятичной цифры. Целые значения со знаком могут изменяться в диапазоне *от –32768 до +32767. Беззнаковые целые могут изменяться от 0 до 65535. Необычные значения в диапазоне от* 65535 до –32769 не допускаются, однако к ошибке не приводят. Эти значения, так же как и другие, вне допустимого диапазона "свертываются" в эквивалентные двоичные значения.

Использование BINASC

Два примера программ иллюстрируют использование BINASC для преобразования значений в строки. Программа 6.4, EQUIP.ASM, кроме этого, определяет переменную RECORD (строка 20), необходимую для извлечения битовых полей из системной переменной, показывающей параметры оборудования, присоединенного к компьютеру. Программа использует процедуры из BINASC и STRIO и косвенно из STRINGS, которые должны находиться в MTA.LIB. Выполните ассемблирование и компоновку программы с помощью команд:

```
tasm          equip
tlink         equip,, mta
equip
```

Замечание

Эту программу нужно запускать из DOS. Вследствие прогресса в современных персональных компьютерах и операционных системах EQUIP уже не такая привлекательная утилита, какой она была при первом издании книги в 1989 г. Однако эта программа все еще служит полезной демонстрацией работы модуля BINASC.

Программа 6.4. EQUIP.ASM

```
1.    %TITLE "Display PC Equipment Info -- by Tom Swan"
2
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:
9:    ;----- Макроопределения
10:
11:   EOS     EQU   0                         ; Ограничитель конца строки
12:   cr      EQU   13                        ; ASCII-символ возврата каретки
13:   lf      EQU   10                        ; ASCII-символ прогона строки
14:
15:
16:   ;----- Определение байтовой записи, с полями для информации об оборудовании
17:
18:   ; !! ЗАМЕЧАНИЕ: Строку 20 наберите в одну строку !!
19:
20:   RECORD Equip printers:2,x:1,game:1,ports:3,y:1,drives:2,mode:2,ram:2,z:1,disk:1
21:
22:
23:   ;----- Определения масок для выделения отдельных битовых полей
24:
25:   ;---------------------------------------
26:   ; Маски AND Поле
27:   ;---------------------------------------
28:   maskPrinters = MASK printers
29:   maskGame     = MASK game
30:   maskPorts    = MASK ports
31:   maskDrives   = MASK drives
32:   maskMode     = MASK mode
33:   maskDisk     = MASK disk
34:
35:
36:         DATASEG
37:
38:   exCode        db      0
39:
40:   welcome       db      cr,lf,'Определение оборудования'
41:                 db      cr,lf,' (C) 1995 Том Сван',cr,lf,lf,EOS
42:
43:   strPrinters   db      'Количество принтеров ........ ', EOS
44:   strGame       db      'Игровой I/O порт ............ ', EOS
45:   strPorts      db      'Количество RS232 портов ..... ', EOS
46:   strDrives     db      'Драйверы дисков (минус 1) ..... ', EOS
47:   strMode       db      'Начальный видеорежим......... ', EOS
48:   strDisk       db      'Имеется ли драйвер диска (1=да) .... ', EOS
49:
50:   string        db      40 DUP (?)          ; Рабочая строка
51:
52:
53:         CODESEG
54:
55:   ;----- Из STRIO.OBJ и BINASC.OBJ
56:
57:         EXTRN BinToAscDec:proc, StrWrite:proc, NewLine:proc
58:
59:   Start:
60:         mov   ax, @data             ; Поместить в DS адрес
```

```
61:         mov   ds, ax              ; сегмента данных
62:         mov   es, ax              ; Сделать es = ds
63:
64:         mov   di, OFFSET welcome  ; Адрес сообщения-приглашения
65:         call  StrWrite            ; Вывести сообщение
66:         int   11h                 ; Определение оборудования BIOS
67:         mov   bx, ax              ; Сохранить информацию в bx
68:
69:         mov   di, OFFSET strPrinters     ; Адресовать метку раздела
70:         mov   dx, maskPrinters    ; Назначить маску AND
71:         mov   cl, printers        ; Определить счетчик сдвигов
72:         call  ShowInfo            ; Вывести метку и информацию
73:
74:         mov   di, OFFSET strGame  ; Следующий раздел
75:         mov   dx, maskGame
76:         mov   cl, game
77:         call  ShowInfo
78:
79:         mov   di, OFFSET strPorts        ; Следующий раздел
80:         mov   dx, maskPorts
81:         mov   cl, ports
82:         call  ShowInfo
83:
84:         mov   di, OFFSET strDrives ; Следующий раздел
85:         mov   dx, maskDrives
86:         mov   cl, drives
87:         call  ShowInfo
88:
89:         mov   di, OFFSET strMode  ; Следующий раздел
90:         mov   dx, maskMode
91:         mov   cl, mode
92:         call  ShowInfo
93:
94:         mov   di, OFFSET strDisk  ; Следующий раздел
95:         mov   dx, maskDisk
96:         mov   cl, disk
97:         call  ShowInfo
98:
99:   Exit:
100:        mov   ah, 04Ch            ; Функция DOS : Выход из программы
101:        mov   al, [exCode]        ; Возвратить значение кода выхода
102:        int   21h                 ; Вызов DOS. Останов программы
103:  %NEWPAGE
104:  ;---------------------------------------------------------------
105:  ; ShowInfo Выводимые метки и значения, соответствующие оборудованию
106:  ;---------------------------------------------------------------
107:  ; Вход:
108:  ; bx = Данные об оборудовании из int 11h
109:  ; cl = Счетчик сдвига битовых полей
110:  ; dx = маска AND битового поля
111:  ; di = адрес строки
112:  ; Выход:
113:  ; Метка и значения данных
114:  ; Регистры:
115:  ; ax, cx
116:  ;---------------------------------------------------------------
117:  PROC ShowInfo
118:        mov   ax, bx              ; Поместить в ax значение, соответствующее оборудованию
119:        and   ax, dx              ; Выделить в ax битовое поле
120:        shr   ax, cl              ; Сдвинуть поле в правый край ax
121:        call  StrWrite            ; Вывести метку из di
122:        mov   di, OFFSET string   ; Адресация рабочей строки
123:        mov   cx, 1               ; Требуется по меньшей мере 1 цифра
124:        call  BinToAscDec         ; Преобразовать ax в ASCIIZ-строку
125:        call  StrWrite            ; Вывести строку
126:        call  NewLine             ; Переход на новую строку
127:        ret                       ; Возврат в вызывающую программу
128:  ENDP ShowInfo
129:
130:        END   Start               ; Конец программы / точка входа
```

Как работает EQUIP

Константы масок в строках 28–33 используются для выделения каждого поля из записи Equip, определенной в строке 20. Это выполняется в строках 104–128 процедурой ShowInfo, использующей под маску регистр dx. Большая часть программы связана с подготовкой к вызовам этой процедуры (см. строки 69–97). В строке 66 вызывается процедура BIOS (базовая система ввода/вывода) ROM через прерывание 11h, в результате чего в регистр ax загружается конфигурация системы. Процедура ShowInfo вызывает BinToAscDec для преобразования маскированного и сдвинутого значения из ax в строку, которая выводится с помощью вызова StrWrite (строка 125). На рис. 6.5 показан результат исполнения программы.

```
Определение оборудования
 (C) 1995 Том Сван
Количество принтеров ..............1
Игровой I/O порт..................0
Количество RS232 портов...........2
Драйверы дисков (минус 1)  .........0
Начальный видеорежим..............2
Жесткий диск (1 = да).............1
```

Рис. 6.5. Пример исполнения программы 6.4, EQUIP.*ASM*

Программирование преобразователя между различными системами счисления

Объединяя многие идеи из этой главы, программа 6.5, CONVERT.ASM, является полезной утилитой, которая может использоваться для преобразования значений между двоичной, десятичной и шестнадцатеричной системами счисления. Программа иллюстрирует использование многих процедур из модуля BINASC. На рис. 6.6 показан простой сеанс работы с CONVERT.

```
Преобразование двоичных, шестнадцатеричных, десятичных значений
(C) 1995 Том Сван
Для выхода нажмите <Enter>.

Преобразуемое значение ? 745

Двоичное..................... : 0000001011101001
Шестнадцатеричное............ : 02E9
Беззнаковое десятичное....... : 745
Знаковое десятичное.......... : 745

Преобразуемое значение ? face

**ОШИБКА : недопустимый символ в строке

Преобразуемое значение ? faceh

Двоичное..................... : 1111101011001110
Шестнадцатеричное............ : FACE
Беззнаковое десятичное....... : 64206
Знаковое десятичное.......... : -1330
```

Рис. 6.6. Пример исполнения программы 6.5, CONVERT.ASM

Поскольку большая часть основной работы выполняется модулями STRING, STRIO и BINASC, которые должны находиться в файле MTA.LIB, программа CONVERT в основном состоит из вызовов соответствующих процедур. Другие команды просто подготавливают регистры для этих вызовов. Поэтому при чтении программы и комментариев у вас могут возникнуть некоторые сложности в понимании, что делает каждая строка. Выполните ассемблирование, компоновку и запустите программу на исполнение с помощью команд:

```
tasm        convert
tlink       convert,,, mta
convert
```

```
%TITLE "Преобразование двоичных, десятичных и шестнадцатеричных значений"

        IDEAL

        MODEL small
        STACK 256

;----- Макроопределения

EOS     EQU     0       ; Конец строки
cr      EQU     13      ; ASCII-символ возврата каретки
lf      EQU     10      ; ASCII-символ прогона строки
maxLen  EQU     40      ; Максимальная длина входной строки


        DATASEG

exCode          db      0 ; код ошибки DOS

welcome         db      cr,lf,'Преобразование двоичных, шестнадцатеричных и десятичных чисел'
                db      cr,lf,'(C) 1995 Том Сван',cr,lf
                db      cr,lf,'Для выхода нажмите <Enter>.',cr,lf,EOS
prompt          db      cr,lf,lf,' Преобразуемое значение? ', EOS
error           db      cr,lf,'**ОШИБКА. недопустимый символ в строке',EOS
binary          db      cr,lf, 'Двоичное ..... : ',EOS
hex             db      cr,lf, 'Шестнадцатеричное  ... .. : ',EOS
decimal         db      cr,lf, 'Беззнаковое десятичное ..  : ',EOS
sdecimal        db      cr,lf, 'Знаковое десятичное ..... : ',EOS

value           dw      ?                       ; Результат AscToBin
response        db      maxLen+1 DUP (?)        ; Строка для ввода пользователя


        CODESEG

;----- Из STRINGS.OBJ & STRIO.OBJ

        EXTRN StrLength:proc, StrRead:proc
        EXTRN StrWrite:proc, NewLine:proc

;----- Из BINASC.OBJ

        EXTRN BinToAscHex.proc, SBinToAscDec.proc, BinToAscDec:proc
        EXTRN BinToAscBin:proc, AscToBin.proc

Start
        mov     ax, @data               ; Поместить в DS адрес
        mov     ds, ax                  ; сегмента данных
        mov     es, ax                  ; Сделать es = ds

        mov     di, OFFSET welcome      ; Вывести сообщение-приглашение
        call    StrWrite

;----- Приглашение к вводу преобразуемого значения

Again:
        mov     di, OFFSET prompt       ; Вывести строку приглашения
        call    StrWrite
        mov     di, OFFSET response             ; Получить ответ пользователя
        mov     cx, maxLen              ; Максимальная длина строки
        call    StrRead
        call    NewLine                 ; Перейти на новую строку
        call    StrLength               ; Нажал ли пользователь <Enter>?
        jcxz    Exit                    ; Выход, если да (cx = 0)

;----- Преобразование введенных символов в двоичное значение

        call    AscToBin                ; Преобразовать строку в ax
        mov     [value], ax             ; Сохранить результат в переменной
        jnc     Continue                ; Переход, если cf равно 0, нет ошибки
        mov     di, OFFSET error        ; Иначе вывести сообщение об ошибке
```


Резюм

Структу
структуры
меняться на
менной стр
определенн

По умол
ричные зна
тичные мог
помощью д

Turbo A
в интервала
но с помощ
ассемблера
один байт у
вычислений
запятой.

Несмотр
отдельных з
способов дл

```
72          call    StrWrite
73          jmp     Again               , Повторение ввода
74
75      ,----- Преобразование двоичного значения в различные числовые форматы
76
77      Continue
78          mov     di, OFFSET binary   , Вывести двоичное представление
79          call    StrWrite
80          mov     ax, [value]         , Получить преобразуемое значение
81          mov     cx, 16              , Минимальное количество цифр
82          mov     di, OFFSET response , Использовать эту же строку для результата
83          call    BinToAscBin         , Преобразовать в двоичные цифры
84          call    StrWrite            , Вывести результат
85
86          mov     di, OFFSET hex      , Вывести шестнадцатеричное представление
87          call    StrWrite
88          mov     ax, [value]         , Получить преобразуемое значение
89          mov     cx, 4               , Минимальное количество цифр
90          mov     di, OFFSET response , Использовать эту же строку для результата
91          call    BinToAscHex         , Преобразовать в шестнадцатеричные цифры
92          call    StrWrite            , Вывести результат
93
94          mov     di, OFFSET decimal  , Вывести десятичное представление
95          call    StrWrite
96          mov     ax, [value]         , Получить преобразуемое значение
97          mov     cx, 1               , Минимальное количество цифр
98          mov     di  OFFSET response         , Использовать эту же строку для результата
99          call    BinToAscDec         , Преобразовать в десятичные цифры
100         call    StrWrite            , Вывести результат
101
102         mov     di, OFFSET sdecimal         , Вывести знаковое десятичное представление
103         call    StrWrite
104         mov     ax, [value]         , Получить преобразуемое значение
105         mov     cx, 1               , Минимальное количество цифр
106         mov     di, OFFSET response , Использовать эту же строку для результата
107         call    SBinToAscDec        , Преобразовать в знаковое десятичное
108         call    StrWrite            , Вывести результат
109         jmp     Again               , Повторять до выхода
110     Exit
111         mov     ah, 04Ch            , Функция DOS   Конец программы
112         mov     al, [exCode]        , Возвратить значение кода выхода
113         int     21h                 , Вызов DOS  Останов программы
114
115         END     Start               , Конец программы / точка входа
```

Резюме

Структуры не являются переменными, они только основа для создания переменных с набором полей. Описание структуры начинается со STRUC и заканчивается ENDS. Значения полей, используемые по умолчанию, могут изменяться на значения, содержащиеся в описании структуры. Для обращения к полю структуры запишите имя переменной структуры и через точку — имя поля. Строковые поля в режиме Ideal заполняются начальными символами, определенными в описании структуры.

По умолчанию в программах на языке ассемблера используется десятичная система счисления. Шестнадцатеричные значения должны начинаться с десятичной цифры и заканчиваться h, двоичные заканчиваются на b, десятичные могут заканчиваться на d. Основание используемой по умолчанию системы счисления можно изменить с помощью директивы RADIX.

Turbo Assembler позволяет определять целые числа со знаком в диапазоне от –32678 до 65535, однако значения в интервалах от –32768 до –1 и от 32768 до 65535 одинаково представляются в двоичной системе счисления. Можно с помощью директивы dt описывать числа с плавающей запятой в формате IEEE, хотя использовать их в языке ассемблера достаточно сложно. Эта же директива может создавать двоично-десятичные (BCD) числа, в которых в один байт упаковывается две цифры и длина числа может достигать 20 цифр. BCD числа удобны для финансовых вычислений, так как они избегают ошибок округления, получающихся при вычислении выражений с плавающей запятой.

Несмотря на то что в ассемблере отсутствует встроенный механизм для создания массивов, для чтения и записи отдельных элементов массива могут использоваться базовый и индексный режимы адресации. Существует много способов для создания массивов в памяти с помощью директив LABEL и REPT; даже можно создать массив с ав-

томатически присвоенными значениями. Индексация массива преследует цель вычислить адрес его каждого отдельного значения. Проще всего это сделать, если элементы массива имеют размер 1 или являются степенями 2.

Внешне объединения аналогичны структурам, но описываются с помощью директивы UNION. Поля объединения накладываются одно на другое в переменной объединения, отличаясь этим от структуры, в которой поля располагаются отдельно. Комбинирование структур и объединений дает возможность создавать в языке ассемблера сложные структуры данных.

Директива RECORD позволяет описывать байты и слова с битовыми полями. Имена полей в записи являются константами, которые представляют число сдвигов, необходимое для перемещения значения поля в самый правый край регистра или переменной. Оператор MASK преобразует константу битового поля в двоичную маску, которая может использоваться логическими командами, такими как and и or, для выделения и соединения значений битовых полей.

Для автоматической генерации наиболее эффективных логических команд or, and, test и xor вместо них можно использовать соответственно следующие псевдокоманды: SETFLAG, MASKFLAG, TESTFLAG и FLIPFLAG.

Если вы имеете в распоряжении Turbo Assembler версии 3.0 или выше, то для вставки и извлечения из записи битовых полей можно пользоваться псевдокомандами SETFIELD и GETFIELD.

Заранее указываемые макроопределения Turbo Assembler помимо всего могут использоваться для создания строки, выводящей версию программы при каждом ее ассемблировании.

Модуль BINASC из этой главы обеспечивает преобразование знаковых и беззнаковых двоичных значений в ASCIIZ-строки, а также преобразует ASCII-строки по этим трем основаниям в двоичные значения. Эти процедуры на практике полезны для преобразования числовых ASCII-значений, введенных с клавиатуры в двоичные значения для последующей обработки.

Упражнения

6.1. Создайте структуру с именем Time и полями hourse, minutes и seconds.

6.2. Опишите переменные типа Time с заранее определенными значениями 10:30:45, 14:00:00, 16:30 и полночь

6.3. Создайте переменную с именем theTime типа Time из упражнения 6.1 и напишите команды языка ассемблера для установки времени 15:45:12, увеличения часа на 1, сброса времени в 00:00:00 и копирования переменной theTime в другую переменную oldTime.

6.4. Предположим, что по умолчанию используется шестнадцатеричная система счисления. Какие из следующих чисел являются десятичными: 00001011, 10000000b, 1234, 4321d, FACE, 00FF?

6.5. Создайте переменные для значений с плавающей точкой 2.5, 88.999 и 0.141. Определите двоично десятичные значения 125,000 и 1,250,500. Какое можно создать максимальное BCD значение?

6.6. Создайте массивы из 45 двухбайтовых слов, 100 четырехбайтовых значений (двойных слов), 1024 байт и 75 двоично-десятичных значений. Сколько занимает каждый из этих массивов в памяти?

6.7. Создайте индексную переменную и, используя ее значение, напишите команды для загрузки в bx адреса любого элемента из четырех массивов в упражнении 6.6.

6.8. Определите объединение, аналогичное представленному на рис. 6.3, однако с полями, которые допускают доступ к своим значениям как к байтам, словам и двойным словам. Приведите примеры команд для доступа к переменным каждого из этих трех типов.

6.9. Создайте упакованную запись с именем inventory с четырьмя битовыми полями (их размер показан в скобках): location (3), status (1), quantity (5) и vendor (4). Сколько байтов занимает эта переменная в памяти? В каких диапазонах может представлять значения каждое поле?

6.10. Напишите команды для выполнения следующих операций с записью inventory из упражнения 6.9: установите location в 3, status в 1, прибавьте 6 к quantity, загрузите поле vendor в dh, переключите поле status, обнулите все поля записи. Подсказка: для создания масок AND используйте оператор MASK.

6.11. Напишите программу ADDHEX.ASM для вывода суммы двух шестнадцатеричных значений, введенных с клавиатуры. Используйте при необходимости в своем ответе процедуры из модулей BINASC, STRINGS и STRIO.

6.12. Добавьте к своему ответу в упражнении 6.11 вывод строки с информацией о версии программы.

Задания

6.1. Напишите процедуры для упаковки и распаковки BCD чисел, преобразовывая стандартный dt формат (две цифры-на-байт) в 20-байтовую переменную, содержащую по одной цифре в каждом байте.

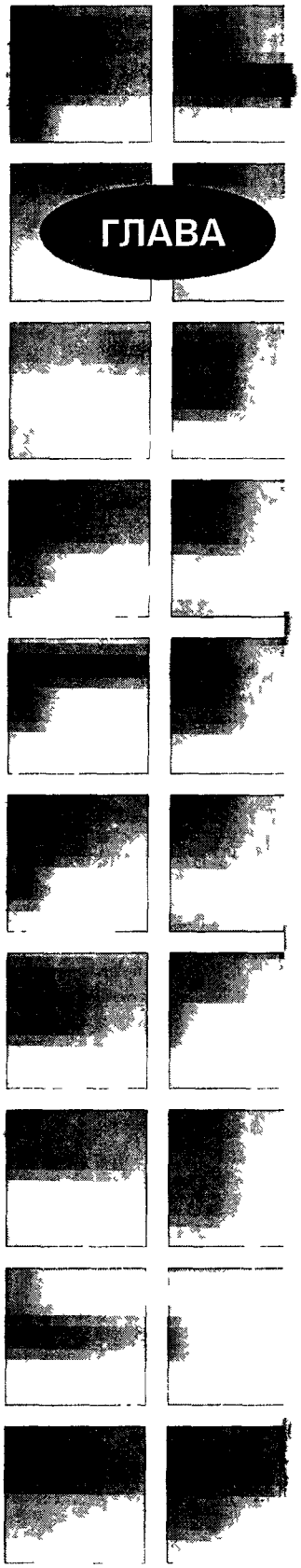
6.2. Напишите логический калькулятор для вывода результатов выполнения операций AND, OR, XOR, NOT, NEG, SHL и SHR над двоичными значениями. Пользователь должен иметь возможность ввести значения и команды с клавиатуры.

63. [Повышенной сложности] Создайте новую версию BINASC с именем BINASC32 для обработки 32-битовых десятичных целых значений.

64 Напишите программу для создания массива строковых записей. После этого напишите процедуры, позволяющие вводить и выводить значения полей для каждой записи. (Замечание: не занимайтесь сохранением ваших данных на диске, это является предметом главы 9.)

65. Создайте процедуры общего назначения для упаковки и распаковки битовых полей в слова переменной записи. Код должен работать как со значениями типа слово, так и с байтовыми значениями

66 Создайте калькулятор для вычисления индексов массива, который возвращает адрес смещения для каждого значения массива, имеющего любой размер.



7 ГЛАВА

Ввод и вывод

- **Стандартный ввод и вывод**
- **Написание DOS фильтров**
- **Печать текста**
- **Использование видеопамяти**
- **Модуль для работы с видеопамятью**
- **Модуль управления клавиатурой**
- **Резюме**
- **Упражнения**
- **Задания**

Стандартный ввод и вывод

Если вы хотите, чтобы ваша программа выполнялась на максимальном количестве DOS систем, а не только на IBM PC, нужно использовать стандартные методы для чтения ввода с клавиатуры и вывода на дисплей — не говоря уже о таких устройствах, как принтеры и плоттеры.

DOS обеспечивает несколько стандартных функций ввода-вывода, простейшие из которых читают и записывают за один раз по одному символу. Например, можно считать символ со стандартного устройства ввода в регистр al с помощью двух простых команд:

```
mov  ah, 1     , Специальная функция DOS ' Ввод символа '
int  21h       , Вызов DOS  Код символа записывается в al
```

Если стандартное устройство вывода является основной консолью, как это обычно и бывает, то при таком чтении символ каждой нажатой клавиши отображается на экране. Поскольку ввод-вывод DOS — перенаправляемый, нет никакой гарантии, что входные данные поступили с клавиатуры. Пользователь, не сообщив программе, мог выполнить команду, которая изменяет стандартный входной файл с клавиатуры на дисковый файл:

```
program <afile.txt
```

Преимущество использования функций DOS для чтения со стандартного входного файла состоит в том, что в программе не нужно выполнять каких-либо специальных действий, чтобы допускать возможность переназначения ввода и вывода. В

ьшинстве случаев программа абсолютно "не подозревает" о деталях ввода-вывода на физическом уровне. Если то-нибудь захочет вместо просмотра на экране распечатать выходные данные программы на принтере, это делает- я исключительно с помощью DOS. Аналогично, вывод символа на стандартное устройство вывода также требует ыполнения только нескольких простых команд:

```
mov  ah,2              ; Специальная функция DOS ' Вывод символа '
mov  dl, [thechar]     ; Пересылка символа в dl
int  21h               ; Вызов DOS
```

Код выводимого символа загружается в dl из байтовой переменной theChar (здесь не показанной). Снова, так как вывод для DOS функции 2 может быть перенаправлен, нет никакой гарантии, что при выполнении этого кода символ будет выведен на экран. Например, кто-либо мог выполнить команду, посылающую выводимые данные программы на последовательный порт, к которому могут присоединяться различные устройства:

```
program >com1
```

Прерывание выполнения программы

Функции DOS 1 и 2 проверяют, была ли набрана некоторое время назад команда прерывания <Ctrl+C>. Если да, тогда DOS вызывает прерывание 23h, которое прерывает дальнейшее исполнение программы. (Детально прерывания описываются в главе 10. Они аналогичны вызову подпрограммы.) Есть три способа избежать неожиданного рерывания программы при нажатии <Ctrl+C>:

- Использовать другую функцию DOS.
- Заменить код прерывания 23h собственным обработчиком <Ctrl+C>.
- Дать указание драйверу устройства игнорировать нажатия <Ctrl+C>.

бычно наилучшим является первый способ; второй потребует написания собственного обработчика прерыва- ий, на что, вероятно, уйдет слишком много времени; третий метод более трудоемкий (я объясню это позже, в данной главе), но в некоторых случаях может оказаться полезным. *Драйверы устройств* являются строго специализированными программами, обеспечивающими интерфейс с физическими устройствами, такими как клавиатура, принтер и дисплей. Они описаны во многих хороших руководствах по программированию в DOS.

Всегда помните, что обе стандартные функции 1 и 2 *ввода и вывода* символа выполняют проверку нажатия Ctrl+C> Если это происходит при вызове DOS функции ввода 1, то ваша программа никогда не получит Ctrl+C>. Если <Ctrl+C> определяется при вызове DOS функции вывода 2, то символ из dl передается в стандартный файл вывода *до проверки* на <Ctrl+C>.

Эти проверки на специальные символы называются *фильтрами*, так как они обеспечивают фильтрацию определенных нажатий клавиш и символов специального назначения. Помимо <Ctrl+C>, функции ввода-вывода 1 и 2 также фильтруют другие управляющие коды, выполняя действия, приведенные в табл. 7.1. Кроме <Ctrl+C>, Ctrl+P> и <Ctrl+S>, которые применяются только при выводе, остальные действия имеют место для обеих функций ввода и вывода 1 и 2.

Нефильтруемый ввод

Если вы не хотите фильтровать <Ctrl+C> и другие управляющие коды, то можно использовать одну из двух функций:

- DOS функция 6: прямой ввод-вывод на консоль (в основном — с клавиатуры на экран)
- DOS функция 7: нефильтруемый ввод без отображения

Функция 6 включена в DOS в основном для поддержки программ, конвертируемых из CP/M, имеющей аналогичную функцию для прямого ввода-вывода на консоль. Поскольку в DOS существуют другие, возможно лучшие, способы прямого доступа к устройствам ввода-вывода, редко имеет смысл использовать функцию 6. Вместо нее лучше применять функцию 7, читающую символ без отображения на стандартном устройстве вывода нажатой клавиши и без фильтрации <Ctrl+C>. Кроме номера функции, код аналогичен коду для функции 1:

```
mov  ah, 7             ; Специальная функция DOS ' Ввод без отображения
                       ; символа на экране '
int  21h               ; Вызов DOS. Символ возвращается в al
```

Этот метод не выполняет проверок нажатий <Ctrl+C> и <Ctrl+Break> и, следовательно, предотвращает преждевременное окончание программы. Другие управляющие коды из табл. 7.1 рассматриваются вашей программой как обычные нажатия на клавиатуре. Чтобы добавить при вводе фильтрацию без отображения символов на стандартном устройстве вывода, используйте функцию 8. Она генерирует прерыванием 23h сигнал к завершению программы, если DOS определил нажатие <Ctrl+C> или <Ctrl+Break>. В остальном функция 8 идентична функции 7.

Таблица 7.1. **Стандартные коды управления I/O**

Ctrl+клавиша	ASCII код	Действие
<Ctrl+C>	03	Генерация прерывания 23h
<Ctrl+G>	07	Звонок
<Ctrl+H>	08	Возврат курсора без удаления символа
<Ctrl+I>	09	Табуляция
<Ctrl+J>	10	Прогон строки (с возможной прокруткой)
<Ctrl+M>	13	Возврат каретки
<Ctrl+P>	16	Переключение устройства PRN вкл/выкл
<Ctrl+S>	19	Приостанов вывода до нажатия клавиши

Нефильтруемый вывод

Как уже объяснялось, с помощью DOS функции 9 можно записывать ASCII$-строки. Кроме странного необходимого формата строки со знаком доллара, функция 9 (так же, как и функция 2) определяет нажатие <Ctrl+C> и отвечает на другие управляющие коды из табл. 7.1. Если нужно использовать эти функции, то следует защититься от прерывания исполняемой программы с помощью вызова DOS функции 44 (управление драйвером устройства) или IOCT — доступной, начиная со второй версии DOS. Эти функции позволяют перепрограммировать драйвер устройства вывода, чтобы игнорировать нажатия <Ctrl+C> и <Ctrl+Break>. Сначала вызовите функцию 44h с al, равным 0, прочитав текущий управляющий бит устройства:

```
mov  ax, 4400h          ; Функция DOS 44h, раздел 00: получить
                        ; информацию об устройстве
mov  bx, 1              ; Определить стандартный вывод
int  21h                ; Вызов DOS. Возвратить данные в dx
```

Биты установок драйвера устройства теперь находятся в регистре dx. Бит 5 в установке драйвера устройства сообщает драйверу, будут ли обрабатываться все данные (бит=1) или нужно фильтровать нажатия <Ctrl+C> и <Ctrl+Break> (бит=0). Установка бита 5 отключает фильтрацию:

```
mov  ax, 4401h          ; Функция DOS 44h, al = 01: установить
                        ; информацию об устройстве
xor  dh, dh             ; dh должен быть равен 0 для этого вызова функции
or   dl, 20h            ; Установить бит 5
int  21h                ; Вызов DOS с данными в dx
```

При этом запрещается фильтрация <Ctrl+C>, <Ctrl+S> и <Ctrl+P>, причем не только для вашей программы, но и для всех других программ, включая DOS, которые вызывают функции 2 и 9 для посылки символов на стандартное устройство вывода. Например, после перепрограммирования драйвера устройства вы не остановите нажатием <Ctrl+C> вывод длинного каталога, начатого с помощью команды DIR. До окончания программы очистите бит 5 с помощью семи команд, аналогичных предыдущим, заменив or dl, 20h на and dl, 0dfh для восстановления проверки <Ctrl+C>.

Ожидание — тоже работа

Программы, использующие для ввода DOS функции 1, 7 и 8, могут попадать в бесконечные циклы, ожидая нажатия клавиши. (Собственно, так и происходит всегда, но на это не обращают внимания.) Часто вам будет необходимо, чтобы программа реагировала на нажатие клавиши, продолжая выполнение своих операций, если на вход ничего не поступало. Например, текстовый процессор может выполнять длительную операцию поиска-замены, пока вы не нажмете клавишу <Esc>; по введенным командам в реальном времени на экран могут выводиться результаты моделирования. Этого можно достичь двумя способами:

- Буферизированный ввод, управляемый через прерывания
- Использование промежуточного буфера

В первом методе при поступлении данных процессор выполняет специальный код, разработанный для записи входных данных в буферы памяти для последующей обработки. (Детально этот метод рассматривается в главе 11.)

Во втором методе программа периодически проверяет устройство ввода, читая его только после обнаружения там ожидающих данных. Если на входе ничего нет, программа продолжает выполнение других операций.

При использовании промежуточного буфера, чтобы избежать потери символов, нужно достаточно часто читать данные на входе. Например, если будут нажаты две клавиши до того, как вы проверите клавиатуру на предмет ввода нового символа, первое нажатие может быть потеряно. К счастью, процедуры IBM PC ROM BIOS автоматически обрабатывают нажатия клавиш, записывая их ASCII-коды в *буфер клавиатуры*. Когда DOS считывает данные с клавиатуры, в действительности происходит удаление символов из этого буфера. В результате — единственная опасность состоит в переполнении буфера до того момента, когда программа приступит к его чтению. Но эта опасность уменьшается с помощью автоматического звонка, который предупреждает о слишком быстром вводе.

Замечание

Помните, что буфер клавиатуры заполняется только при вводе с клавиатуры. Если ввод и вывод перенаправлены на другие устройства, символы не буферизуются, и, чтобы избежать потери данных, вы должны достаточно часто считывать их с устройства ввода. Это особенно касается последовательных портов ввода-вывода, которые DOS вызывает *вспомогательным* вводом-выводом. Если при связи с удаленным компьютером, в частности через модем, ваша программа не будет достаточно часто проверять поступление данных на вход, то часть их будет потеряна. Даже во время простой записи на диск некоторые символы могут быть пропущены. Таким образом, для последовательного ввода-вывода в DOS лучше использовать другие методы, в особенности на IBM PC, как объясняется в главе 11 при обсуждении обработки прерываний.

Проверка нажатия клавиш

Для проверки, имеются ли данные для считывания, используйте DOS функцию 11 "Получить статус ввода", которая в случае отсутствия данных возвращает al, равное 0, и равное 0ffh — если поступил символ для чтения. (Функция 11 возвращает только два значения, нуль и 0ffh, поэтому достаточно проверить, равняется ли нулю al) С помощью этого метода можно написать простой цикл с вызовом процедуры, обрабатывающей новые символы по мере их поступления:

```
@@10
     call    OtherStuff      , Код, исполняющийся до поступления символа
     mov     ah, 11          ; Функция DOS ' Получить статус ввода '
     int     21h             , Вызов DOS  Результат в al
     or      al, al          , al = 0 ?
     je      @@10            ; Переход, если al = 0  На входе пусто
     mov     ah, 7           ; Иначе прочитать символ без отображения на экране
     int     21h             ; Вызов DOS  Возврат символа в al
     call    ProcessChar     ; Обработать новые данные в al
     jmp     @@10            , Повторяем это снова
```

В этом фрагменте циклически вызывается процедура OtherStuff (не показана), пока функция 11 не сообщит, что поступил символ для считывания. Когда становится доступным новый символ (вероятно, в результате нажатия клавиши) он считывается функцией 7. Затем для выполнения соответствующих действий вызывается ProcessChar (также не показана), которая в ответ на нажатие <Esc> или другой клавиши может осуществлять выход из программы. Фактически этот простой пример вы можете использовать в качестве "основного цикла" любой программы, которая в процессе исполнения должна реагировать на нажатия клавиш. К сожалению, здесь не все так хорошо: функция 11 также определяет <Ctrl+C> и <Ctrl+Break>, завершая при нажатии этих клавиш программу с помощью прерывания 23h. Это полностью разрушает преимущества использования функции 7 для чтения нефильтрованного ввода В этой ситуации не поможет даже описанное ранее перепрограммирование драйвера устройства.

Решением является вызов вместо DOS функции BIOS 16h, проверяющей нажатие клавиш. При ah, равном 1, если буфер клавиатуры пуст, эта функция возвращает 1 во флаге нуля zf или 0, если в буфере находится по меньшей мере один символ. Вдобавок, при наличии ожидающего ввода символа, функция BIOS возвращает его значение в al и, кроме того, его скан-код (номер символа на клавиатуре) — в ah. Если первоначально ah равно 0, эта же функция устраняет и возвращает в ax один символ из буфера клавиатуры. Эти функции обеспечивают программе полностью нефильтрованный ввод-вывод. Предыдущий код принимает следующий вид:

```
@@10:
     call    OtherStuff      ; Код, исполняющийся до поступления символа
     mov     ah, 1           ; Выбор функции ' Статус ввода '
     int     16h             ; Вызов функции BIOS ввода-вывода с клавиатуры
     jz      @@10            ; Переход, если zf = 1  На входе пусто
     xor     ah, ah          ; Выбор функции ' Чтение символа '
     int     16h             ; Вызов функции BIOS ввода-вывода с клавиатуры
     call    ProcessChar     ; Обработка новых входных данных в al
     jmp     @@10            ; Снова сначала
```

При использовании этого метода ни одна последовательность нажатий клавиш не может привести к преждевременному завершению программы. Решив задачу для ввода, другая функция BIOS позволяет выводить символы без фильтрации <Ctrl+C> и <Ctrl+Break>. С помощью этой функции процедура ProcessChar может выводить символы, считанные в предыдущем примере:

```
PROC ProcessChar
     cmp     al, 27   , al = Escape клавиша ?
     je      Exit     ; Если да, выход из программы
     mov     bl, 15   ; Цвет переднего плана для графических дисплеев
     mov     ah, 14   ; Выбор функции ' Запись TTY '
     int     10h      ; Вызов функции BIOS для ввода-вывода на экран
     ret              ; Возврат в вызывающую программу
ENDP ProcessChar
```

Сначала al сравнивается с ASCII кодом для <Esc> (27), и если она нажата, то выполняется переход на метку Exit (не показана). (Нужно обеспечивать выход из программы, если вы не полагаетесь в этом на DOS, через <Ctrl+C> и <Ctrl+Break>.) Если <Esc> не нажималась, в bl устанавливается цвет переднего плана, необходимый для графических дисплеев. Затем в ah записывается десятичное значение 14, выбирающее функцию BIOS "Вывод на TTY". Она называется так, потому что обеспечиваемый ею простой вывод символов напоминает телетайп, другими словами, в нем отсутствуют возможности для позиционирования курсора, изменения атрибутов и цветов символов, очистка строк и т.д. Тем не менее прерывание 10h полезно для приемлемо быстрого вывода, особенно если вы хотите, чтобы программа полностью управляла вводом-выводом.

Замечание

Функция BIOS Вывод на TTY прерывания 10h фильтрует <Ctrl+G> (Звонок), <Ctrl+H> (Возврат), <Ctrl+J> (Прогон строки) и <Ctrl+M>, описанные в табл. 7.1. Остальные управляющие коды из табл. 7.1 выводятся как графические символы.

За вызов функций ввода-вывода ROM BIOS, как и за все хорошее в жизни, нужно платить. Как вы можете увидеть, в последних нескольких примерах устранены все вызовы DOS. Следовательно, теперь программа может исполняться только на IBM PC и совместимых с ними на 100% компьютерах, которые имеют соответствующие функции ROM BIOS. Код может не исполняться на обычной DOS или на другой операционной системе, которая эмулирует версии DOS. Однако, поскольку сегодня во всем мире в офисах установлены многие миллионы PC, это уже не является такой серьезной проблемой, как было в прошлом. Тем не менее, используя эти методы, вы должны по меньшей мере предупреждать о невозможности выполнения программы на несовместимых системах.

Более существенная проблема — это потеря возможности перенаправления ввода-вывода, что является одной из наиболее привлекательных сторон DOS. Вызовы функций DOS обеспечивают программе полное управление вводом-выводом символов. Это означает, что пользователи вашей программы больше не смогут перенаправлять ввод из текстового файла или посылать вывод на принтер. Многие программисты рассматривают эту потерю как преимущество, позволяя их программам полностью управлять выводом на принтер, дисплей и т.д. Однако для небольших программ и утилит перенаправление ввода-вывода является удобной возможностью, и в этих случаях можно использовать стандартные вызовы функций DOS.

Чтение функциональных клавиш

Набор ASCII-символов непосредственно представляет только 32 управляющих кода со значениями от 0 до 31, 95 символов со значениями от 32 до 126 плюс символ удаления со значением 127. Включая прописные и строчные буквы, пунктуацию и различные комбинации с <Ctrl>, <Shift> и <Alt>, этого недостаточно, чтобы покрыть все комбинации клавиш, введенных даже с небольшой 83- или 84-клавишной клавиатуры PC.

Замечание

Несмотря на то что PC расширяет обычный набор 128 ASCII-кодов значениями от 128 до 255, они зарезервированы для графических символов, с помощью которых можно рисовать рамки, выводить математические символы, греческие буквы, стрелки и другие символы. Чтобы ввести эти коды, нажмите клавишу <Alt> и, удерживая ее, наберите на числовой части клавиатуры ASCII значение необходимого символа.

Чтобы обработать специальную клавишу, методы ввода DOS, обсуждаемые в предыдущих разделах, возвращают два кода, представляющих функциональную клавишу. Всегда нулевой первый код называется *lead-in символом* Если какая-либо функция ввода с клавиатуры возвращает 0, то следующий символ укажет на нажатую функциональную клавишу. Эта схема приводит к коду:

```
mov  ah, 1              ; Специальная функция DOS ' Ввод символа '
int  21h                ; Вызов DOS. Символ возвращается в al
or   al, al             ; Проверка на lead-in, введенный с клавиатуры
jnz  NormalChar         ; Переход на обработку обычного символа
int  21h                ; Вызов DOS для следующего символа
jnz  FunctionKey        ; Переход на обработку функциональной клавиши
```

Как показано здесь, для определения и чтения функциональных клавиш, включая такие специальные клавиши, как <Ins>, <Del>, <PgUp> и <PgDn>, клавиши управления курсором и пронумерованные функциональные клавиши <F1>–<F10>, содержащиеся на всех клавиатурах PC, необходимы два вызова DOS функции 1. Обычные символы обрабатываются путем перехода на NormalChar (не показана), функциональные — переходом на FunctionKey (также не показана).

Замечание

В предыдущем примере ah устанавливается в 1 только перед первым вызовом DOS в int 21h. Нет необходимости делать это во второй раз, так как DOS сохраняет все регистры, кроме тех, которые специальным образом возвращаются различными функциями; поэтому можно считать, что неиспользуемые регистры остаются между вызовами DOS неизменными. Применяя этот прием, остерегайтесь неумышленного изменения номера функции в ah, что будет бедствием для вашей программы.

Многие программисты используют метод двойного вызова DOS, однако я нахожу его слишком громоздким для практики. Несмотря на то что можно определить функциональные клавиши, отсутствует простой способ для представления их подобно простым символам A или Q. По этой причине я отображаю (т.е. перевожу) значения функциональных клавиш в отдельные коды. Этот метод описывается дальше в данной главе вместе с текстом модуля для ввода с клавиатуры, который можно добавить в свою библиотеку.

Очистка буфера клавиатуры

При запросе подтверждения на выполнение разрушающих данные операций, таких как форматирование диска или удаление важного дискового файла, хорошей идеей является очистка буфера клавиатуры перед началом его чтения, что заставляет внимательнее подходить к ответам на серьезные вопросы программы. Существует два способа очистки буфера клавиатуры. Первый является наиболее очевидным и состоит в том, чтобы просто считать все введенные к этому моменту символы с клавиатуры:

```
@@10
     mov     ah, 1     ; Выбор функции ' Статус ввода '
     int     16h       ; Вызов функции BIOS ввода-вывода с клавиатуры
     jz      @@20      ; Переход, если zf = 1. На входе пусто
     xor     ah, ah    ; Выбор функции ' Чтение символа '
     int     16h       ; Чтение и прогон одного символа
     jmp     @@10      ; Перейти к повторению цикла
@@20                   ; Буфер клавиатуры пуст
```

Этот код аналогичен предыдущему примеру и вызывает для проверки наличия данных на входе прерывание BIOS 16h с ah, равным 0. Если таковые имеются (что показывает флаг zf=0), то ah устанавливается в 0 и снова вызывается прерывание 16h для считывания из буфера одного символа. Этот процесс повторяется, пока в буфере не останется ни одного символа.

Замечание

Для очистки буфера можно также вызывать одну из функций ввода символов DOS — 7 или 8. Помните, что это не будет работать, если ввод перенаправлен.

Еще одна возможность состоит в вызове специальной функции DOS, которая очищает буфер и выполняет затем другую команду ввода символа. Если программа должна работать на всех системах, поодерживаемых DOS, то нужно использовать именно этот метод. Сначала загрузите в ah номер функции 0Ch. Затем поместите в al номер одной из других команд ввода — 1, 6, 7, 8 или 0Ah. При использовании 0Ah команды "Получить строку" также установите в ds dx адрес буфера для размещения введенной строки. Вызовите DOS с помощью int 21h, в результате чего очистится буфер и затем выполнится функция, определенная в al. Например:

```
mov  ah, 0Ch            ; Выбрать ' Сбросить буфер ввода и выполнить '
mov  al, 7              ; Допустимы 1, 6, 7, 8 или 0Ah
int  21h                ; Вызов DOS для очистки буфера и
                        ; исполнения команды в al
```

Некоторые программисты на языке ассемблера применяют другой способ для очистки буфера, оперируя двумя указателями (адресами), определяющими начало и конец буфера. Эти указатели адресуют начало (голову) и конец (хвост) буфера клавиатуры, расположенного в памяти. Третий указатель определяет начальную точку буфера. По определению, если указатели на голову и хвост равны, то буфер пустой. Все три указателя расположены в сегменте данных BIOS по адресу 0040h, области, зарезервированной под системные переменные. Как показано в следующем фрагменте, можно использовать эту информацию для очистки буфера, установив указатели головы и хвоста равными начальному адресу буфера:

```
bufferStart   EQU 0080h          ; Указатель на начало буфера
head EQU 001Ah                   ; Указатель на голову
tail EQU 001Ch                   ; Указатель на хвост

mov  ax, 0040h                   ; Переслать адрес сегмента данных BIOS
mov  ds, ax                      ; в регистр ds
mov  ax, [bufferStart]           ; Получить начальный адрес буфера
cli                              ; Предупреждение случайных прерываний
mov  [head], ax                  ; Присвоить адрес указателю заголовка
mov  [tail], ax                  ; Заголовок = хвост, очистка буфера
sti                              ; Разрешить прерывания
```

Сначала в регистре ds устанавливается адрес сегмента данных BIOS, начинающийся с 0040h. Затем в ax загружается значение, записанное в [bufferStart] и содержащее адрес смещения буфера клавиатуры. Присвоение этого значения указателям на голову и хвост очищает буфер. Команда cli (Очистить прерывание) предотвращает прерывание клавиатуры на время корректировки указателей. Команда sti снова разрешает прерывания после очистки буфера.

Замечание

Затронутое здесь „прерывание клавиатуры" относится к аппаратным прерываниям. Всякий раз, когда вы нажимаете клавишу, это прерывание вызывает исполнение функции ROM BIOS, считывающей и записывающей нажатую клавишу в буфер клавиатуры, как это объяснялось ранее. Это может происходить в любой момент времени, независимо от исполнения другого кода. Поэтому прерывания запрещаются на время очистки буфера до полного ее завершения — для предотвращения нежелательного, но возможного события от нажатия клавиши.

Введение в дескрипторы DOS

Другой полезный способ для передачи данных в программу и из нее состоит в чтении и записи файлов, определяемых через значения, называемые *дескрипторами*. Слово "файл" может относиться как к дисковому файлу, так и к другим устройствам, таким как дисплей, клавиатура и принтер. Вместо того чтобы непосредственно писать код для доступа к этим различным устройствам, можно читать их и записывать в логические файлы, назначенные устройствам, применяя простой набор вызовов функций DOS для связи с широким классом аппаратного обеспечения. (Мы вернемся к этому в главе 9 при рассмотрении вопроса использования дескрипторов для чтения и записи дисковых файлов.)

Когда DOS загружает и исполняет программу, инициализируются некоторые стандартные файлы. В табл. 7.2 представлено пять дескрипторов, связанных с этими файлами, и показаны значения, которые могут использоваться в программах на языке ассемблера для взаимодействия с дисплеем, клавиатурой, принтером и одним последовательным каналом ввода-вывода.

Если вы направляете DOS команду перенаправления ввода-вывода, используя символ перенаправления "<" для определения нового входного устройства или файла и ">" для задания нового устройства вывода или файла, DOS закрывает дескрипторы 0 и 1, а затем снова открывает их, но уже для новых устройств, переключая таким образом ввод-вывод с обычного устройства CON, т.е. дисплея и клавиатуры. Это происходит до начала выполнения программы, поэтому все, что нужно делать, — это читать с дескриптора 0 и записывать на дескриптор 1, что обеспечивает полное управление вводом-выводом программы.

Дескриптор 2 чаще используется для вывода сообщений об ошибках. Поскольку перенаправление ввода-вывода оказывает влияние только на дескрипторы 0 и 1 и дескриптор 2 обычно указывает на консоль, при перенаправлении вывода на другое устройство запись на дескриптор 2 все равно приводит к выводу на экран. Это позволяет следить за ходом исполнения программы и выводить сообщения об ошибках, не заботясь о смешивании сообщений с другими выводимыми данными. (Вы можете записывать на дескриптор 2 любую информацию, а не только сообщения об ошибках.)

Дескриптор 3 приписан к первому последовательному порту, также известному под названием COM1. Однако, так как DOS плохо обрабатывает последовательный ввод-вывод, скорее всего, вы не будете использовать этот дескриптор для связи с удаленными системами через модемы или высокоскоростные интерфейсы RS-232.

Дескриптор 4 связан с принтером, который можно присоединить к параллельному или последовательному порту компьютера. Некоторые программисты на языке ассемблера используют функцию для работы с принтером ROM BIOS, прерывания 17h, подходящую только для параллельных принтеров. Хотя это и является стандартной конфигурацией для большинства PC систем, многие из систем тем не менее имеют последовательные принтеры. Вывод на стандартное устройство печати является лучшим способом для поддержки всевозможных конфигураций принтеров.

Таблица 7.2. **Стандартные дескрипторы DOS**

Указатель	Название устройства	Описание устройства
0	CON	Стандартное устройство ввода
1	CON	Стандартное устройство вывода
2	CON	Стандартное устройство вывода ошибки
3	AUX	Вспомогательное устройство (последовательный I/O)
4	PRN	Стандартное устройство печати (принтер)

Написание DOS-фильтров

Используя стандартные методы ввода-вывода DOS, можно создавать так называемые программы-фильтры, которые считывают стандартный входной файл, выполняют с этими поступившими данными некоторые операции и затем записывают модифицированные данные в стандартный файл вывода. Программы-фильтры могут объединяться с помощью специального символа, называемого конвейером и представляемого вертикальной чертой (|). Конвейер передает выход первого фильтра на вход следующего, который, в свою очередь, может направить свой вывод в третий фильтр, и т.д. Объединение различных фильтров, каждый из которых может иметь простое назначение, например сортировка строк текста и выделение данных по тому или иному критерию, позволяет создавать сложные команды для решения задач, для которых в противном случае потребовалось бы написание самостоятельных программ.

Наряду с другими утилитами, DOS предоставляет три стандартные программы-фильтры: FIND, MORE и SORT. (За информацией об использовании этих программ обратитесь к своему руководству по DOS). К этому основному набору можно добавить и свои фильтры. Чтобы помочь вам начать, программа 7.1, FILTER.ASM, представляет оболочку, описывающую большинство низкоуровневых деталей, связанных с написанием фильтров. FILTER является завершенным фильтром, считывающим данные со стандартного устройства ввода и записывающим их на стандартное устройство вывода. Программа представляет собой только оболочку, поэтому она не выполняет никаких полезных функций. После текста программы я объясню, как модифицировать оболочку, чтобы она делала нечто полезное. Сейчас, после ввода программы, можно выполнить ассемблирование FILTER с помощью команды tasm filter.

Замечание

Если вы попытаетесь исполнить FILTER без предоставления входного и выходного файлов, то компьютер „зависнет". Для восстановления нажмите <Ctrl+Z> (клавиша DOS „Конец файла") и <Enter>.

Программа 7.1. FILTER.ASM

```
1     %TITLE "Оболочка фильтра -- Copyright (c) 1989,1995 by Tom Swan"
2:
3:          IDEAL
4
5:          MODEL  small
6:          STACK  256
7:
8:
9     ;----- Макроопределения
10:
11    InputHandle  EQU 0              ; Стандартный дескриптор ввода
12.   OutputHandle                    EQU 1   ; Стандартный дескриптор вывода
13:   ErrOutHandle                    EQU 2   ; Стандартный дескриптор вывода ошибки
14:   bell         EQU 07             ; ASCII-звонок
15.   cr           EQU 13             ; ASCII-возврат каретки
16    lf           EQU 10             ; ASCII-прогон строки
```

```
eof             EQU 26                  ; DOS-символ конца файла (^Z)

        DATASEG

exCode          DB 0                    ; Код ошибки ввода-вывода

;----- Сообщения об ошибках

errMessage   DB bell, cr, lf, 'ОШИБКА ФИЛЬТРА: '
lenErrMessage = $-errMessage

codeAccess   EQU 5
errAccess    DB 'доступ запрещен', cr, lf
lenErrAccess  = $-errAccess

codeNotOpen  EQU 6
errNotOpen   DB 'неверный дескриптор или файл не открыт', cr, lf
lenErrNotOpen                   = $-errNotOpen

codeDiskFull                    EQU 29
errDiskFull  DB 'диск полный', cr, lf
lenErrDiskFull = $-errDiskFull

errGeneral   DB 'Неизвестная причина', cr, lf ; Code = ?
lenErrGeneral = $-errGeneral


;----- Буфер ввода

oneChar         DB                  ?       ; Один входной символ


        CODESEG

Start:
        mov     ax, @data           ; Поместить в DS адрес
        mov     ds, ax              ; сегмента данных
        mov     es, ax              ; установить es = ds (необязательно)

Repeat:
        call    ReadChar            ; Прочитать следующий символ
        jz      Done                ; Конец цикла при достижении конца файла

;----- Здесь обработать [oneChar]

        call    WriteChar           ; Записать обработанный символ
        jnz     Repeat              ; Продолжать, пока есть место на диске
        mov     [exCode], codeDiskFull      ; Установить код ошибки
        jmp     Exit                ; и пропустить запись eof
Done:
        mov     [oneChar], eof      ; Записать символ конца файла
        call    WriteChar           ; в стандартное устройство вывода. Не проверяйте
                                    ; в этом месте диск на наличие места!
Exit:
        cmp     [exCode], 0         ; Проверка на возможные ошибки
        jz      @@99                ; Перейти, если ошибка не обнаружена
        call    DisplayError        ; иначе вывести сообщение об ошибке
@@99:
        mov     ah, 04Ch            ; Функция DOS: Выход из программы
        mov     al, [exCode]        ; Возвратить значение кода выхода
        int     21h                 ; Вызов DOS. Остановка программы

%NEWPAGE
;---------------------------------------------------------------
; ReadChar Чтение одного символа из стандартного устройства ввода
;---------------------------------------------------------------
; Вход:
; Отсутствует
; Выход:
; zf = 0 : al = следующий входной символ (0..255)
; zf = 1 : на входе нет больше символов
; Регистры:
```

```
91:   ; ax
92:   ;---------------------------------------------------------------
93:   PROC ReadChar
94:         push  bx                  ; Сохранить изменяемые регистры
95:         push  cx
96:         push  dx
97:
98:         mov   ah, 03Fh            ; Номер функции чтения с устройства
99:         mov   bx, InputHandle     ; Определить дескриптор вывода
100:        mov   cx, 1               ; Число считываемых символов
101:        mov   dx, offset oneChar  ; Записать вывод в ds:dx
102:        int   21h                 ; Вызов DOS. Ввести данные
103:        jnc   @@10                ; Переход, если нет ошибок,
104:        mov   [exCode], al        ; иначе сохранить код ошибки
105:        jmp   Exit                ; и досрочно выйти из программы
106.  @@10:
107:        or    ax, ax              ; Установить/очистить флаг нуля (zf)
108:
109:        pop   dx                  ; Восстановить регистры
110:        pop   cx
111:        pop   bx
112:        ret                       ; Возврат в вызывающую программу
113:  ENDP ReadChar
114:  %NEWPAGE
115:  ;---------------------------------------------------------------
116:  ; WriteChar Записать один символ на стандартное устройство вывода
117:  ;---------------------------------------------------------------
118:  ; Вход:
119:  ; [oneChar] = выводимый символ
120:  ; Выход:
121:  ; zf = 0 : символ записан на стандартное устройство вывода
122.  ; zf = 1 : переполнение устройства вывода (при выводе на диск)
123:  ; Регистры:
124:  ; ax
125:  ;---------------------------------------------------------------
126:  PROC WriteChar
127:        push  bx                  ; Сохранить изменяемые регистры
128:        push  cx
129:        push  dx
130:
131:        mov   ah, 040h            ; Номер функции записи на устройство
132:        mov   bx, OutputHandle    ; Определить дескриптор вывода
133:        mov   cx, 1               ; Число выводимых символов
134:        mov   dx, offset oneChar  ; Взять данные из ds:dx
135:        int   21h                 ; Вызов DOS. Вывести данные
136:        jnc   @@10                ; Переход, если нет ошибок
137:        mov   [exCode], al        ; иначе сохранить код ошибки
138:        jmp   Exit                ; и досрочно выйти из программы
139:  @@10:
140:        or    ax, ax              ; Установить/очистить флаг нуля (zf)
141:
142:        pop   dx                  ; Восстановить регистры
143:        pop   cx
144:        pop   bx
145:        ret   ; Возврат в вызывающую программу
146.  ENDP WriteChar
147:  %NEWPAGE
148:  ;---------------------------------------------------------------
149:  ; DisplayError Вывести сообщение об ошибке
150:  ;---------------------------------------------------------------
151:  ; Вход:
152:  ; [exCode] = ненулевой код ошибки
153:  ; Выход:
154:  ; отсутствует: на стандартное устройство вывода посылается сообщение
155:  ; Регистры:
156:  ; ax, bx, cx, dx
157:  ;---------------------------------------------------------------
158:  PROC DisplayError
159:        mov   cx, lenErrMessage   ; Длина общей строки
160:        mov   dx, offset errMessage      ; Адрес общей строки
161:        call  DisplayString       ; Вывести первую часть сообщения
162:
163:        cmp   [exCode], codeAccess       ; Проверка на ошибку codeAccess
164:        jne   @@10                ; Переход, если код ошибки отличен
```

```
165:        mov   cx, lenErrAccess    ; Установить длину строки
166:        mov   dx, offset errAccess        ; Установить адрес строки
167:        jmp   DisplayString       ; Вывести строку
168:  @@10:
169:        cmp   [exCode], codeNotOpen
170:        jne   @@20
171:        mov   cx, lenErrNotOpen
172:        mov   dx, offset errNotOpen
173:        jmp   DisplayString
174:  @@20:
175:        cmp   [exCode], codeDiskFull
176:        jne   @@30
177:        mov   cx, lenErrDiskFull
178:        mov   dx, offset errDiskFull
179:        jmp   DisplayString
180:  @@30:
181:        mov   cx, lenErrGeneral   ; Другие значения ошибок
182:        mov   dx, offset errGeneral
183:
184:  DisplayString:
185:        mov   ah, 040h            ; Номер функции вывода на устройство
186:        mov   bx, ErrOutHandle    ; Определить дескриптор для вывода ошибки
187:        int   21h                 ; Вызов DOS. Осуществить вывод
188:        ret   ; Возврат в вызывающую программу
189:
190:  ENDP DisplayError
191:
192:        END Start                 ; Конец программы/точка входа
```

Как работает FILTER

FILTER использует дескрипторы DOS для чтения и записи символов со стандартных устройств ввода и вывода Программа также корректно обрабатывает ошибки, включая проверку на переполнение диска, отсутствующую во многих аналогичных программах, и иллюстрирует несколько других приемов, которые можно использовать в своих собственных программах.

Идентификаторам макроопределений в строках 11–13 присваиваются значения трех стандартных дескрипторов DOS (см. табл. 7.2). Позднее они передаются в соответствующие функции DOS для чтения и записи символов Строки 27–43 иллюстрируют различные способы описания строк символов. В отличие от ASCII$ и ASCIIZ методов, описанных ранее, эти строки являются неограниченными. Первая строка, errMessage, в строке 27 начинается со звонка, возврата каретки и символов перехода на новую строку. При записи этой строки раздается звонок и с новой строки выводятся символы "ОШИБКА ФИЛЬТРА:" Строка 28 показывает, как ассемблировать числовое макроопределение, равное длине строки. Ниже приводится простой пример:

```
DATASEG
dumbJoke     db       "My Texas fleas have dogs "
LenString    =        $ - dumbJoke
```

Знак доллара ($) называется *счетчиком положения*. Turbo Assembler заменяет $ текущим адресом смещения в этом месте программы — в этом случае относительно сегмента данных, хотя можно также использовать этот символ в любом другом сегменте. Поскольку адрес смещения является обычным значением, таким же как метка dumbJoke, вычитанием dumbJoke из счетчика положения определяется длина строки. Этот же прием можно использовать с любой другой меткой для вычисления размеров массива или структуры, либо даже для нахождения количества байтов между двумя точками в кодовом сегменте.

Замечание

В режиме MASM в выражениях, включающих счетчик положения $, можно использовать как директиву EQU, так и знак равенства. В режиме Ideal вы должны использовать только знак равенства — директива EQU в этом случае не подходит. Причина состоит в том, что в режиме Ideal EQU утверждения записываются в виде текста, а выражения вычисляются позже, при использовании соответствующего идентификатора макроопределения. Макроопределения со знаком равенства вычисляются в месте описания. Поэтому, чтобы символ $ имел правильное значение, выражение должно вычисляться при описании, а не позднее — при его использовании!

В строках 27–43 FILTER расположены строки, представляющие собой сообщения об ошибках, связанные со значениями, описанными в директивах EQU. Например, codeAccess является кодом ошибки для строки errAccess, имеющей длину LenErrAccess. Между прочим, использование таких имен — это хороший способ при организации программы, особенно когда в ней имеется большое количество символов.

В строках 58–67 выполняется ввод и вывод путем повторных вызовов двух подпрограмм, ReadChar и WriteChar, которые читают один символ со стандартного устройства ввода и записывая его в переменную oneChar (строка 48). В строке 62 вы можете вставить свой собственный текст для обработки символа до вызова WriteChar в строке 64, посылающей oneChar на стандартное устройство вывода.

В строках 68–70 к концу выходного файла добавляется символ конца файла ASCII 26 (<Ctrl+Z>). Для одних программ он необходим, другие обходятся без него. Поэтому для надежности лучше всегда добавлять этот маркер.

В конце FILTER.ASM проверяется переменная exCode, которая до этого момента не использовалась. В данной программе код ошибки может записываться или в ReadChar, или в WriteChar. В этом случае третья подпрограмма DisplayError посылает соответствующее сообщение на стандартный дескриптор устройства вывода ошибки номер 2, после чего программа заканчивается с помощью функции DOS 04Ch, снова возвращающей значение exCode в al (строки 77–79).

Код в строках 58–75 предназначен для обработки всевозможных ошибок ввода-вывода. Если ReadChar возвращает установленным флаг zf, это означает, что вводить больше нечего и в строке 60 осуществляется переход на метку Done, в которой записывается маркер конца файла. Если WriteChar возвращает установленным флаг zf, это означает, что файл вывода является дисковым файлом и на диске нет больше места — ситуация, которую по странным причинам DOS не считает ошибочной. Во многих программах отсутствуют эти важные шаги проверки диска на наличие свободного места, выполняемые здесь в строках 64–67.

В остальном FILTER состоит из трех подпрограмм, которые вы можете вызывать из своих собственных программ. Как это делается, описано в следующем разделе.

ReadChar (82–113)

ReadNhar показывает, как читать один символ со стандартного устройства ввода (дескриптор 0). DOS функция 03Fh, "Чтение из файла или устройства", требует, чтобы в bx содержался номер дескриптора, в cx — максимальное количество считываемых символов, а в ds:dx — указатель на место, в котором DOS хранит вводимые данные. При обнаружении ошибки эта процедура возвращает установленный флаг cf, код ошибки (5 или 6) записывается в строке 104 в exCode, после чего осуществляется переход на метку Exit, сразу заканчивающую программу в случае появления ошибки. Команда or в строке 107 устанавливает либо очищает zf. Если ax равен 0, значит, на входе нет больше данных; в противном случае ax равно числу прочитанных символов.

WriteChar (115–146)

WriteChar вызывает DOS функцию 040h, "Запись в файл или устройство", для записи одного символа в стандартное устройство вывода (дескриптор 1). Снова bx равно номеру дескриптора, cx — числу символов и ds:dx — адресу выводимых данных. Если функция DOS 040h возвращает установленный флаг cf, то в строках 137–138 код ошибки из al записывается в переменную exCode и выполняется переход на метку Exit. В строке 140, как описано выше, устанавливается или очищается флаг zf.

DisplayError (148–190)

DisplayError показывает, как использовать для вывода ошибки или другого сообщения из программ-фильтров ту же DOS функцию 040h, что и в подпрограмме WriteChar. В этом случае в строке 186 в bx записывается стандартный дескриптор для вывода ошибки, cx равно длине строки, а ds:dx адресует строковую переменную. Поскольку используется дескриптор 2, даже если стандартный вывод перенаправлен, сообщения все равно будут появляться на экране.

Доработка FILTER

Поскольку FILTER читает символы со стандартного устройства ввода и записывает их на стандартное устройство вывода, можно использовать при выполнении программы символы перенаправления ввода-вывода (< и >) и конвейера (|). Для изменения программы, чтобы она выполняла нечто полезное, сначала скопируйте FILTER.ASM в LC.ASM и замените в копии строку 62 на код с рис. 7.1.

Добавив новые строки, выполните ассемблирование и компоновку с помощью команд:

```
tasm        lc
tlink       lc
```

Теперь у вас есть программа-фильтр LC, которая преобразует в строчные все символы из текстовых файлов. LC можно использовать для преобразования в строчные символы текстов свободно распространяемых программ на языке ассемблера, многие из которых написаны прописными буквами. До обработки нужного вам файла ис-

пытайте программу на его *копии* или другом текстовом файле. Если ваш файл называется OLDFILE.TXT, введите команду

```
lc <oldfile.txt > newfile.txt
```

для преобразования текста из OLDFILE.TXT в строчные буквы и записи результата в новый файл с именем NEWFILE.TXT. При этом OLDFILE.TXT останется неизменным.

```
        mov     al, [oneChar]    ; Загрузить в al введенный символ
        cmp     al, ' A '        ; Проверить al > ' A '
        jb      @@10             ; Перейти, если al < ' A '
        cmp     al, ' Z '        ; Проверить al < ' Z '
        ja      @@10             ; Перейти, если al > ' Z '
        add     al, ' a ' - ' A ' ; Преобразовать A-Z в a-z
        mov     [oneChar], al    ; Сохранить измененные символы
@@10:
```

Рис. 7.1. Код, заменяющий строку 62 в программе 7.1 и преобразующий оболочку FILTER.ASM в LC.ASM

Замечание

Единственная опасность при перенаправлении ввода-вывода в программах-фильтрах состоит в том, что вы не получаете никакого предупреждения о том, что существующий файл будет перезаписан новым выходным файлом. Будьте внимательны, чтобы не удалить важный файл, набирая имя выходного файла после символа перенаправления „>". Всегда храните резервные копии ваших файлов!

Другой способ использования программ-фильтров, подобных LC, состоит в направлении вывода из одного фильтра на вход другому. Например, чтобы вывести строчными буквами отсортированный дисковый каталог, используйте команду

```
dir|lc|sort|more
```

DIR, конечно, является командой DOS; LC — фильтр из этой главы; MORE — стандартная программа-фильтр DOS, которая приостанавливает вывод при полном заполнении экрана; SORT — другой стандартный фильтр, сортирующий строки текста. Поскольку дисплей — стандартное устройство вывода, в этом случае нет необходимости перенаправлять вывод. Если вы хотите перенаправить вывод с конвейера, например, напечатать строчными буквами каталог на принтере, используйте команду

```
dir|lc >prn
```

Печать текста

Принтер является просто другим устройством вывода, поэтому проще всего напечатать текст, записав его на стандартный дескриптор устройства печати с номером 4 (см. табл. 7.2). Например, можно распечатать строку с помощью следующего кода:

```
DATASEG
string    DB            ' This string is printed '
LenString =             $ - string
CODESEG
mov     ah, 040h                ; Функция DOS ' Записать в файл или устройство '
mov     bx, 4                   ; Номер дескриптора стандартного устройства печати
mov     cx, LenString           ; Записать длину строки
mov     dx, offset string       ; Записать адрес строки в ds.dx
int     21h                     ; Вызов DOS для печати строки
```

После выполнения этого кода, если флаг cf не установлен, то в регистре ax содержится количество напечатанных символов, в противном случае ax равно коду ошибки, возможно, 5 (доступ запрещен) или 6 (неверный дескриптор или файл не открыт). Если cf не установлен, возможно, что значение в ax будет меньше значения cx, эти показывается, что успешно распечатана только часть символов. Вы можете при желании обработать эту ситуацию, однако в большинстве случаев в этом нет необходимости и можно просто написать:

```
jnc     Continue                ; Нет ошибки - продолжить
mov     [errorCode], ax         ; Иначе запомнить код ошибки
```

```
    jmp     Error                   ; Выход из программы
Continue
```

Простой способ печатать одиночные символы — это использовать функцию DOS 5, которая посылает символ из dl на стандартное устройство печати, связанное с дескриптором 4:

```
    mov     ah, 5                   ; DOS вывод на принтер
    mov     dl, [anyChar]           ; Занести символ в dl
    int     21h                     ; Вызов DOS для печати одного символа
```

Этот и предыдущий методы обеспечивают компактность и работают с любыми конфигурациями принтер/интерфейс, которые только могут встретиться вашей программе. Как уже упоминалось, можно печатать символы через вызов прерывания 17 ROM BIOS, хотя этот метод не работает с последовательными принтерами:

```
    mov     ah, 0                   ; Выбор функции печати прерывания 17h
    mov     al, [anyChar]           ; Занесение символа в al
    mov     dx, 0                   ; Номер принтера 0, 1 или 2
    int     17h                     ; Вызов ROM BIOS для печати одного символа
```

Если после исполнения этого кода значение в ah равно 1, значит, символ напечатан не был — вероятно, из-за неготовности принтера или его отсутствия. Используйте этот метод только тогда, когда вы уверены, что ваша программа будет управлять принтером, присоединенным к компьютеру через параллельный интерфейс, и что в системе установлен IBM-совместимая BIOS.

Выбор параметров печати

Все современные принтеры "понимают" разнообразные управляющие коды, устанавливающие соответствующие режимы печати, такие как подчеркивание, печать полужирным и др. Выбор параметра осуществляется посылкой "на печать" *подходящей последовательности управляющих кодов. Когда принтер получает такую последовательность*, он рассматривает эти значения как команды, а не как распечатываемые ASCII-коды. Например, для выбора сжатой печати для большинства Epson-совместимых принтеров можно написать:

```
    mov     ah, 5               ; DOS вывод на принтер
    mov     dl, 14              ; Управляющий код сжатия текста
    int     21h                 ; ' Распечатать ' команду
```

Некоторые команды состоят из двух или более последовательных кодов, обычно начинающихся с символа Esc (ASCII 27). Наверно, лучше всего для обработки этих кодов написать небольшую подпрограмму, выводящую на печать один символ:

```
PROC PrintChar
    mov     ah, 5               ; DOS вывод на принтер
    int     21h                 ; Печать символа
    ret                         ; Возврат в вызывающую программу
ENDP PrintChar
```

После этого поместите распечатанное значение в dl и вызовите PrintChar. Для задания режима подчеркивания можно написать:

```
    mov     dl, 27
    call    PrintChar
    mov     dl, 45
    call    PrintChar
    mov     dl, 1
    call    PrintChar
```

При этом на принтер посылается последовательность 27, 45, 1, в результате которой дальнейший текст будет распечатываться с подчеркиванием. (Для устранения подчеркивания замените 1 на 0.) В табл. 7.3 представлены чаще всего используемые последовательности управляющих кодов, понимаемые многими принтерами. Другие коды вы найдете в руководстве к своему принтеру.

Таблица 7.3. **Типичные управляющие последовательности принтера**

ASCII-код	Десятичные значения	Действие
BELL	7	Звонок
HT	9	Горизонтальная табуляция (вперед)
LF	10	Прогонка строки
VT	11	Вертикальная табуляция
FF	12	Прогонка страницы
CR	13	Возврат каретки
SO	14	Двойная ширина текста установлена*
SI	15	Установить сжатие текста
DC2	18	Отменить сжатие текста
DC4	20	Отменить двойную ширину текста
CAN	24	Очистить буфер принтера
ESC,-,NUL	27,45,0	Установить подчеркивание
ESC,-,SOH	27,45.1	Отменить подчеркивание
ESC,E	27,69	Установить выделение текста
ESC,F	27,70	Отменить выделение текста
ESC,W,NUL	27,87,0	Отменить двойную ширину текста
ESC,W,SOH	27,87,1	Установить двойную ширину текста

*SO отменяется с помощью CR, LF или DC4

Использование видеопамяти

Перефразируя хорошо известного автора, имя которого сходно с моим (но заканчивается нехорошим Вольфом (волк), вместо замечательного Свана (лебедь)), программисты на языке ассемблера любят улучшать свой код, достигая максимальной скорости вывода на экран персонального компьютера, записывая символы непосредственно в видеопамять PC.

Несмотря на то что существует несколько различных типов видеоадаптеров и систем, применяемых на IBM PC и совместимых с ними компьютерах, все они используют две специальные области памяти для вывода на экран текста. Эти области, называемые видео- или *regen*-буферами, начинаются с сегментного адреса 0B000h для монохромных и мониторов Hercules, с 0B800h для графических систем, включая стандарты CGA, EGA и VGA. Каждое слово в буфере определяет значение символа от 0 до 255 из расширенной ASCII-таблицы плюс второй байт с атрибутами, такими как полужирный и подчеркивание на монохромных системах или цвета фона и переднего плана на цветных мониторах. Хотя можно использовать различные режимы и особенности этих стандартов, если вы собираетесь выводить текст, записывая его непосредственно в видеобуфер, этот процесс относительно прост.

Причина наличия двух буферов состоит в том, что IBM PC позволяет одновременное использовать как монохромный, так и цветной графические адаптеры. Хотя большинство людей сегодня используют единственный CRT и карту адаптера, очевидно, что такое двойное применение требует наличия двух буферов для содержания данных экрана. Таким образом, сначала нужно определить, имеет ли система монохромный или цветной адаптер — или какой из них является активным в системе с двумя установленными видеоадаптерами. Для этого вызовите прерывание 10h ROM BIOS с ah, равным 15 в десятичной системе:

```
DATASEG
vBASE       dw      ?               ; Основной адрес видеобуфера
CODESEG
     mov    [vBASE], 0B800h         ; Инициализация адреса сегмента по умолчанию
     mov    ah, 15                  ; Номер функции ROM BIOS ' Получить видеорежим '
     int    10h                     ; Вызов прерывания BIOS для обслуживания видео I/O
     cmp    al, 7                   ; Результат: монохромный?
     jne    @@10                    ; Перейти, если не монохромный
     mov    [vBASE], 0B000h         ; Иначе изменить адрес сегмента
@@10:
```

В этих командах с помощью int 10h вызывается видеофункция BIOS и проверяется результат, возвращаемый в al. Если al равно 7 — это монохромная система (включая популярный адаптер Hercules); в противном случае система имеет графическую карту некоторого типа. Соответственно в переменной vBASE устанавливается правильный адрес сегмента для использования в других подпрограммах.

После этого для вывода символа на экран его ASCII значение и 8-битовый код атрибута помещаются в память со смещением в сегменте, определенным в vBASE. Для этого существует несколько способов, но я считаю, что проще всего загрузить в es адрес сегмента, а в di — смещение:

```
mov es, [vBASE]          ; Поместить адрес сегмента видеобуфера в es
mov di, 0                ; Поместить адрес смещения в di
```

После этого загрузите ASCII значение в al, атрибут или значение цвета — в ah и выполните stosw для вывода символа:

```
mov     al, [anyChar]          ; Загрузить выводимый символ в al
mov     ah, [attribute]        , Загрузить атрибут в ah
stosw                          , Запомнить ax в es di
```

Если вы собираетесь с помощью этого метода последовательно выводить ряд символов, выполните до первого stosw команду cld для подготовки автоувеличения di. При выводе одного символа не имеет значения, увеличивается di или уменьшается, так что этот шаг можно пропустить.

Рис. 7.2 показывает, что символы в буфере видеопамяти монохромного и цветного дисплея состоят из символа и байта атрибутов. Рис. 7.3 представляет формат байта атрибутов символа, одинаковый как для цветных, так и для монохромных адаптеров. Естественно, цвета можно увидеть только на цветных дисплеях. На монохромных системах "цвета" представляются с помощью подчеркивания, выделения и негативно отображения (черное на светлом).

В буфере видеопамяти байты символа записываются по четным адресам, байты атрибутов — по нечетным. При чтении и записи значений символов и атрибутов в 16-битовый регистр помните, что процессор 8086 записывает байты слова в обратном порядке. Следовательно, предполагая, что значение в di четное, при выполнении следующих двух команд значение символа загружается в al, атрибута — в ah:

```
lodsw                    , al <- символ, ah <- атрибут
mov     ax, [es di]      ; То же самое, но di не изменяется
```

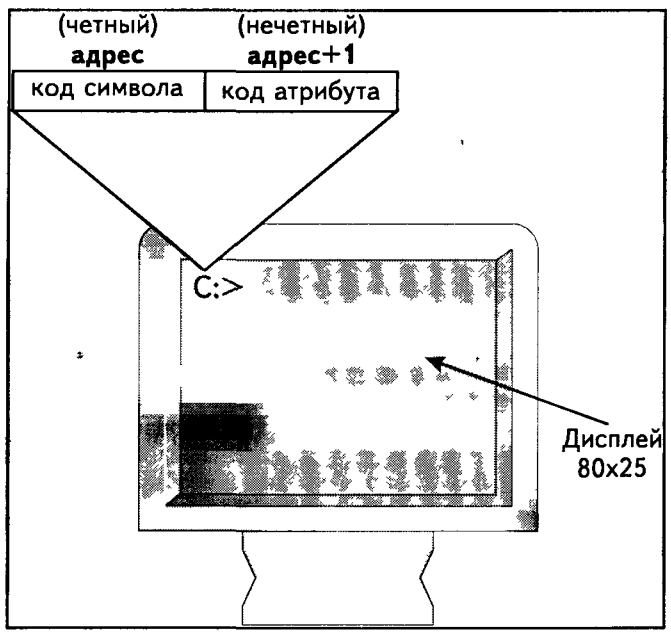


Рис. 7.2. Позиции экрана и видеобуферы

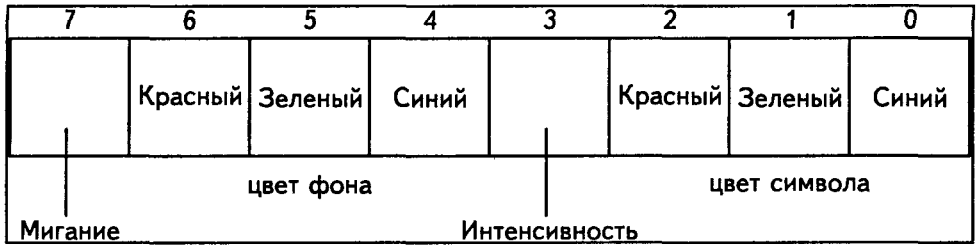


Рис. 7.3. Байт атрибутов для цветного и монохромного дисплея

Координаты курсора

Для помещения курсора в заданную позицию вызывайте прерывание BIOS 10h с ah, равным 2, dh — номеру строки, dl — номеру колонки. Позиция (0,0) — это верхний левый угол экрана, поэтому на обычном дисплее размером 80×25 символов максимальный номер колонки — 79, строки — 24. Некоторые видеосистемы могут осуществлять вывод на разные страницы, поэтому вы должны указать в bh номер страницы. Обычно можно ограничиться заданием стандартной страницы 0, позиционируя курсор с помощью следующих команд:

```
xor     bh, bh          ; Выбор страницы 0 (по умолчанию)
mov     ah, 2           ; Определить номер функции установки курсора
mov     dh, [row]       ; Загрузить строку в dh
mov     dl  [column]    ; Загрузить столбец в dl
int     10h             ; Вызов функции BIOS, обслуживающей видеосервис
```

Если программа использует другие номера страниц или вы изменили страницу с помощью команд:

```
mov     ah  5           ; Определить номер функции изменения страницы
mov     al, 1           ; Определить номер страницы 1 (вторая страница)
int     10h             ; Вызов функции BIOS, обслуживающей видеосервис
```

то до изменения положения курсора нужно определить номер текущей страницы. Это делается с помощью вызова прерывания 10h с десятичным значением 15 в ah:

```
mov     ah, 15          ; Задать номер функции для получения видеорежима
int     10h             ; Вызов функции BIOS, обслуживающей видеосервис
```

В результате в bh загружается номер текущей видеостраницы, в ah устанавливается ширина дисплея (обычно 80) и, как описано выше, в al также устанавливается текущий видеорежим. Имея в bh номер страницы, теперь можно позиционировать курсор, не заботясь о том, что это делается на неверной странице — ошибка, которая иногда встречается даже в коммерческих программах. (Если в используемой вами программе курсор иногда исчезает или ведет себя странно, вероятно, вы столкнулись именно с этой ошибкой.)

Замечание

Если вы изменяли видеостраницы, то перед выходом из программы снова вернитесь на страницу 0.

Эффект "снега"

Снег — замечательное явление, но не тогда, когда он "дрейфует" по экрану компьютера. К сожалению, при непосредственной записи в видеопамять в текстовом режиме CGA можно получить "снег" вследствие интерференции со временем ответной реакции микросхем на запись или *восстановление* экрана. (Эта проблема не возникает с монохромными, Hercules, EGA- и VGA-адаптерами дисплеев.) Это восстановление выполняется автоматически около 60 раз в секунду, создавая иллюзию неподвижности.

CGA-адаптеры редко используются на современных PC, и эта проблема не столь актуальна, как раньше. Но, несмотря на это, если вы хотите, чтобы ваша программа для DOS на языке ассемблера работала на всех PC, нужно обеспечить ее работу на более старых системах. Кроме того, методы, описанные в этом разделе, полезны и для других компьютерных систем, где могут потребоваться аналогичные подходы для создания устойчивого изображения.

Технология устранения "снега" состоит в обращении к видеопамяти только в те моменты времени, когда по этим же адресам не происходит чтения данных. Наиболее подходящее время для этого — период *вертикального возврата*, когда CRT-луч невидимо перемещается снизу вверх экрана после окончания полного цикла восстановления. При записи в буфер видеопамяти в течение этого времени гарантируется отсутствие интерференции с собст-

венными временными параметрами CGA. Для определения периода вертикального возврата необходимо с помощью команды in читать значение регистра в CRT-контроллере Motorola 6845, которая вместе с соответствующей командой out имеет такой вид:

```
in аккумулятор, порт
out порт, аккумулятор
```

Аккумулятором может быть al (для ввода байта) или ax (для ввода слова). *Порт* определяет физический адрес устройства, с которого осуществляется чтение, и должен быть числом от 0 до 255 или значением в dx от 0 до 65535. Команда in считывает из порта байт или слово. Команда out записывает байт или слово в порт. Для некоторых портов простая запись или чтение правильных адресов вызывает определенные действия, и в этом случае передача в них данных бессмысленна.

Для устранения "снега" на CGA команда in читает регистр статуса контроллера 6845 по адресу 03dah. Если бит 3 результата в al равен 1, значит, в данный момент происходит вертикальный возврат и можно быстро поместить символ в память. Это делается с помощью кода:

```
M6845 EQU 03dah          ; Адрес CGA 6845 CRT контроллера

      mov     dx, M6845  ; Установить в dx для адреса порта ввода
@@10
      in      al, dx     ; Прочесть статус 6845
      test    al, 08h    ; Проверить, если бит 3=1
      je      @@10       ; Повторить, если бит 3=0
```

Непосредственно после этого можно спокойно записывать символ и атрибут в видеобуфер. Вы можете использовать любой из методов адресации, описанных в этой книге, но проще всего применять команду stosw. Предположив, что es di адресует видеобуфер и что в cx содержится символ, а в ch — его атрибут, можно продолжить предыдущий код следующим образом:

```
mov ax, cx               ; Записать символ/атрибут в ax
stosw                    ; Запомнить ax в es di
```

К сожалению, в первую очередь все эти попытки предотвращения "снега" в текстовом режиме CGA приводят к потере скорости, получаемой при непосредственной записи в видеобуферы. Это еще хуже, так как теперь программа перед выводом каждого символа должна проверять, нужно ли "управлять снегом"; вывод на другие типы дисплеев также значительно замедляется. По этим причинам вы, возможно, захотите создать два библиотечных модуля, один с "управлением снегом", другой — без него. Имейте также в виду, что некоторые пользователи готовы мириться со "снегом" ради достижения более быстрого вывода на экран, так что вы всегда должны предоставлять возможность отключить устранение "снега". К сожалению, некоторые авторы обзоров и компьютерные журналисты считают, что "снег" всегда необходимо устранять, не указывая во многих случаях, что это приводит к значительной потери скорости вывода. Многие же предпочитают максимальную скорость, даже если при этом они будут видеть временами "снегопад" на экране.

Еще о портах ввода-вывода

Как видно из предыдущего раздела, чтение и запись портов с помощью команд in и out является одним из самых низкоуровневых и аппаратно-зависимых действий. Адреса портов заложены в компьютере и интерфейсных микросхемах, и вы не можете изменять их в программе. Некоторые интерфейсы позволяют выбирать адреса портов с помощью переключателей или установки перемычек. Можно также создавать интерфейсные карты с программируемыми адресами портов, однако на практике это делается редко. Большинство адресов портов фиксируется.

Поскольку адреса портов могут меняться от компьютера к компьютеру, прямой доступ к портам ввода-вывода приводит к тому, что программа будет исполняться только на определенной модели компьютера. Некоторые адреса, такие как последовательные порты ввода-вывода (рассматриваемые в главе 10), всегда устанавливаются в то или иное значение, другие добавляются разработчиками компьютеров для управления определенными параметрами. Например, следующие команды переключают одну из моих старых компьютерных систем (ALR 386/2) между обычным и турбо режимом:

```
                    ; Отключить турбо режим
mov al, 0EAh        , Присвоить значение al
out 64h, al         ; Вывести al в порт 64h
                    , Включить турбо режим
mov al, 0E5h        ; Присвоить значение al
out 64h, al         ; Вывести al в порт 64h
```

К сожалению, эти же команды не работают на других системах, так что не испытывайте их, если у вас нет именно такой модели компьютера. Если вы используете такой аппаратно зависимый код, тогда нужно предоставить возможность пользователю изменять назначения адресов портов, выбирать альтернативный код (может быть, вызывать функции DOS, если система не обладает определенными особенностями) или вообще пропускать эти команды.

Модуль для работы с видеопамятью

Программа 7.2, SCREEN.ASM, содержит ряд процедур, в которых воплощены идеи этой главы. Так же как и STRINGS, STRIO и BINASC, программа создана в виде библиотечного модуля и поэтому до исполнения требует компоновки с главной программой. (Полноценный пример приводится после этого раздела.) Несколько новых подходов, использованных в SCREEN.ASM, описаны далее. Однако все применяемые в программе команды 8086 были введены в этой или предыдущих главах, и вы должны без особого труда понять большую часть кода. Выполните ассемблирование и запишите SCREEN в вашу библиотеку MTA.LIB с помощью команд:

```
tasm /zi screen
tlib /E mta -+screen
```

Если вы позже будете изменять SCREEN, то повторите эти же команды. (Как объяснялось для других модулей, не обращайте внимание на возможное предупреждение об отсутствии SCREEN в библиотеке.) Если вы не планируете исполнять ассемблированную программу в Turbo Debugger, то для уменьшения размера получаемого файла можете устранить опцию /zi.

Программа 7.2. SCREEN.ASM

```
1:    %TITLE 'Memory-Mapped Video - Copyright (c) 1989,1995 -- by Tom Swan'
2:
3:
4:    ;----- До использования процедур из этого пакета вы должны вызвать ScInit
5:    ;
6:
7:
8:          IDEAL
9:          MODEL small
10:
11:   MaxRow        EQU 25              ; Максимальное количество выводимых строк
12:   MaxCol        EQU 80              ; Максимальное количество выводимых столбцов
13:   MonoBASE      EQU 0b000h          ; Адрес сегмента RAM для монохромного адаптера
14:   DefaultBASE   EQU 0b800h          ; Адрес сегмента RAM для других адаптеров
15:
16:
17:   ;----- Байт атрибутов символа & маски AND
18:
19:   RECORD attrByte Blink:1, Background:3, Intensity:1, Foreground:3
20:
21:   BlinkMask     EQU MASK Blink
22:   BackMask      EQU MASK Background
23:   IntensityMask                     EQU MASK Intensity
24:   ForeMask      EQU MASK Foreground
25:
26:
27:
28:         DATASEG
29:
30:   attribute     attrByte <0,0,7>    ; Атрибут, начальное значение
31:   vBASE DW      DefaultBASE         ; Адрес буфера видеопамяти
32:
33:
34:   ;----- ScRow: Массив смещений (от vBASE) в видеобуфере RAM
35:
36:   BytesPerRow = MaxCol * 2
37:   row = 0
38:   LABEL ScRow Word
39:   REPT MaxRow
40:         DW ( row * BytesPerRow )
41:         row = row + 1
42:   ENDM
43:
44:         CODESEG
```

```
45:
46:         PUBLIC ScGotoXY, ScReadXY, ScPokeChar, ScPokeStr, ScClrRect
47:         PUBLIC ScSetBack, ScSetFore, ScBright, ScDim, ScBlink
48:         PUBLIC ScNoBlink, ScGetAttribute, ScSetAttribute, ScInit
49:
50:    %NEWPAGE
51:    ;---------------------------------------------------------------
52:    ; SetVidAddr Подготовить адрес видеопамяти
53:    ;---------------------------------------------------------------
54:    ; Замечание:
55:    ; Процедура используется в ScPokeChar и ScPokeStr
56:    ; Вход:
57:    ; dh = строка (0 - верхняя строка)
58:    ; dl = column (0 - самый левый столбец )
59:    ; Выход:
60:    ; es:di = адрес видеобуфера RAM
61:    ; Замечание: dh и dl не проверяются!!
62:    ; Регистры:
63:    ; изменяются bx, dx, di, es
64:    ;---------------------------------------------------------------
65:    PROC SetVidAddr
66:         mov    es, [vBASE]          ; Установить в es адрес видеосегмента
67:         xor    bh, bh               ; Обнулить верхнюю половину bx
68:         mov    bl, dh               ; Поместить строку в bl
69:         shl    bx, 1                ; Умножить строку (bx) на 2
70:         mov    di, [scRow+bx]       ; Установить в di адрес строки видеобуфера
71:         xor    dh, dh               ; Преобразовать столбец в 16-битовое слово
72:         shl    dx, 1                ; Умножить столбец (dx) на 2
73:         add    di, dx               ; Прибавить смещение столбца к адресу строки
74:         ret                         ; Возврат в вызывающую программу
75:    ENDP SetVidAddr
76:    %NEWPAGE
77:    ;---------------------------------------------------------------
78:    ; ScGotoXY Установить позицию курсора
79:    ;---------------------------------------------------------------
80:    ; Вход:
81:    ; dh = строка (0 - верхняя строка)
82:    ; dl = столбец (0 - самый левый столбец)
83:    ; Выход:
84:    ; Курсор помещается в позицию (строка, столбец) на текущей странице
85:    ; Регистры:
86:    ; Не используются
87:    ;---------------------------------------------------------------
88:    PROC ScGotoXY
89:         push   ax                   ; Сохранить изменяемые регистры
90:         push   bx
91:         mov    ah, 15               ; Поместить в bh номер видеостраницы
92:         int    10h                  ; Вызов видеофункции BIOS
93:         mov    ah, 2                ; Номер функции BIOS
94:         int    10h                  ; Вызов BIOS--установить позицию курсора
95:         pop    bx                   ; Восстановить регистры
96:         pop    ax
97:         ret                         ; Возврат в вызывающую программу
98:    ENDP ScGotoXY
99:    %NEWPAGE
100:   ;---------------------------------------------------------------
101:   ; ScReadXY Получить позицию курсора
102:   ;---------------------------------------------------------------
103:   ; Вход:
104:   ;      нет
105:   ; Выход:
106:   ;      dh = строка (0 - верхняя строка)
107:   ;      dl = столбец (0 - самый левый столбец)
108:   ; Регистры:
109:   ;      изменяется dx
110:   ;---------------------------------------------------------------
111:   PROC ScReadXY
112:        push   ax                   ; Сохранить изменяемые регистры
113:        push   bx
114:        push   cx
115:        mov    ah, 15               ; Поместить в bh номер видеостраницы
116:        int    10h                  ; Вызов видеофункции BIOS
117:        mov    ah, 3                ; Номер функции BIOS
118:        int    10h                  ; Вызов BIOS--получить позицию курсора
```

```
119:        pop    cx                  ; Восстановить регистры
120:        pop    bx
121:        pop    ax
122:        ret                        ; Возврат в вызывающую программу
123:  ENDP ScReadXY
124:  %NEWPAGE
125:  ;---------------------------------------------------------------
126:  ; ScPokeChar Вывести символ на дисплей
127:  ;---------------------------------------------------------------
128:  ; Вход:
129:  ;     al = код ASCII символа
130:  ;     dh = строка (0 - верхняя строка)*
131:  ;     dl = столбец (0 - самый левый столбец)*
132:  ; Выход:
133:  ;     Символ из al выводится в позиции (строка, столбец)
134:  ;     * Замечание: значения Строка и Столбец не проверяются !!
135:  ; Регистры:
136:  ;     изменяются ax, bx, dx, di
137:  ;---------------------------------------------------------------
138:  PROC ScPokeChar
139:        push   es                  ; Сохранить сегментный регистр es
140:        call   SetVidAddr          ; Подготовить es:di
141:        mov    ah, [attribute]     ; Поместить атрибут в ah
142:        stosw                      ; Вывести атрибут и символ
143:        pop    es                  ; Восстановить регистр es
144:        ret                        ; Возврат в вызывающую программу
145:  ENDP ScPokeChar
146:  %NEWPAGE
147:  ;---------------------------------------------------------------
148:  ; ScPokeStr Вывести строку на дисплей
149:  ;---------------------------------------------------------------
150:  ; Вход:
151:  ;     cx = число выводимых символов
152:  ;     dh = строка (0 - верхняя строка)*
153:  ;     dl = столбец (0 - самый левый столбец)*
154:  ;     ds:si = адрес ASCII строки (любой формат)
155:  ; Выход:
156:  ;     * Замечание: значения Строка и Столбец не проверяются!!
157:  ;     Замечание: любой ограничитель строки игнорируется
158:  ; Регистры:
159:  ;     изменяются ax, bx, cx, dx, di, si
160:  ;---------------------------------------------------------------
161:  PROC ScPokeStr
162:        push   es                  ; Сохранить адрес сегмента es
163:        call   SetVidAddr          ; Подготовить es:di
164:        mov    ah, [attribute]     ; Поместить атрибут в ah
165:        cld                        ; Автоувеличение si, di
166:  @@10:
167:        lodsb                      ; Взять следующий символ в al
168:        stosw                      ; Вывести атрибут и символ
169:        loop   @@10                ; Цикл по cx
170:        pop    es                  ; Восстановить адрес сегмента es
171:        ret                        ; Возврат в вызывающую программу
172:  ENDP ScPokeStr
173:  %NEWPAGE
174:  ;---------------------------------------------------------------
175:  ; ScClrRect Очистить прямоугольную область на экране
176:  ;---------------------------------------------------------------
177:  ; Вход:
178:  ;     ch, cl = строка & столбец верхнего левого угла
179:  ;     dh, dl = строка & столбец нижнего правого угла
180:  ; Выход:
181:  ;     Очищенная текущими атрибутами прямоугольная область
182:  ;     с координатами ch,cl & dh,dl
183:  ; Регистры:
184:  ;     ax
185:  ;---------------------------------------------------------------
186:  PROC ScClrRect
187:        mov    ah, 6               ; Выбрать функцию прокрутки BIOS
188:        mov    al, 0               ; Указать на необходимость очистки области
189:        mov    bh, [attribute]     ; Используемые атрибуты
190:        int    10h                 ; Вызов видеофункции BIOS
191:        ret                        ; Возврат в вызывающую программу
192:  ENDP ScClrRect
```

```
193:  %NEWPAGE
194:  ;---------------------------------------------------------------
195:  ; ScSetBack Установить цвет фона (атрибут)
196:  ;---------------------------------------------------------------
197:  ; Вход:
198:  ; al = цвет фона
199:  ; Выход:
200:  ; Устанавливается цвет фона для ScPokeChar и ScPokeStr
201:  ; Регистры:
202:  ; al
203:  ;---------------------------------------------------------------
204:  PROC ScSetBack
205:  IF Background GT 0
206:        push  cx                  ; Если фон не в lsbs,
207:        mov   cl, Background      ; тогда сдвинуть биты в
208:        shl   al, cl              ; позицию ORing в байте атрибутов
209:        pop   cx                  ;
210:  ENDIF
211:        and   al,BackMask         ; Выделить биты в al
212:        and   [attribute], NOT BackMask  ; Обнулить биты фона
213:        or    [attribute], al     ; Установить фон
214:        ret                       ; Возврат в вызывающую программу
215:  ENDP ScSetBack
216:  %NEWPAGE
217:  ;---------------------------------------------------------------
218:  ; ScSetFore Установить цвет переднего плана
219:  ;---------------------------------------------------------------
220:  ; Вход:
221:  ; al = цвет переднего плана
222:  ; Выход:
223:  ; Устанавливается цвет переднего плана для ScPokeChar и ScPokeStr
224:  ; Регистры:
225:  ; al
226:  ;---------------------------------------------------------------
227:  PROC ScSetFore
228:  IF Foreground GT 0
229:        push  cx                  ; Если цвет переднего плана не в lsbs,
230:        mov   cl, Foreground      ; тогда сдвинуть биты в
231:        shl   al, cl              ; в позицию ORing в
232:        pop   cx                  ; байте атрибутов
233:  ENDIF
234:        and   al, ForeMask        ; Выделить биты в al
235:        and   [attribute], NOT ForeMask  ; Обнулить биты переднего плана
236:        or    [attribute], al     ; Установить цвет переднего плана
237:        ret                       ; Возврат в вызывающую программу
238:  ENDP ScSetFore
239:  %NEWPAGE
240:  ;---------------------------------------------------------------
241:  ; ScBright   Включить бит интенсивности
242:  ; ScDim      Выключить бит интенсивности
243:  ; ScBlink    Включить бит мерцания
244:  ; ScNoBlink  Выключить бит мерцания
245:  ;---------------------------------------------------------------
246:  ; Вход:
247:  ;      нет
248:  ; Выход:
249:  ;      В атрибуте изменяются биты интенсивности и мерцания
250:  ; Регистры:
251:  ;      не используются
252:  ;---------------------------------------------------------------
253:  PROC ScBright
254:        or    [attribute], IntensityMask
255:        ret
256:  ENDP ScBright
257:
258:  PROC ScDim
259:        and   [attribute], NOT IntensityMask
260:        ret
261:  ENDP ScDim
262:
263:  PROC ScBlink
264:        or    [attribute], BlinkMask
265:        ret
266:  ENDP ScBlink
```

```
PROC ScNoBlink
        and     [attribute], NOT BlinkMask
        ret
ENDP ScNoBlink
%NEWPAGE
;---------------------------------------------------------------
; ScGetAttribute Получить текущее значение атрибута
;---------------------------------------------------------------
; Вход:
;       нет
; Выход:
;       dl = текущее значение атрибута
; Регистры:
;       dl
;---------------------------------------------------------------
PROC ScGetAttribute
        mov     dl, [attribute]     ; Получить байт атрибута
        ret                         ; Возврат в вызывающую программу
ENDP ScGetAttribute
%NEWPAGE
;---------------------------------------------------------------
; ScSetAttribute Изменить значение атрибута
;---------------------------------------------------------------
; Вход:
;       al = новое значение атрибута
; Выход:
;       нет: атрибут записывается для последующего использования
; Регистры:
;       нет
;---------------------------------------------------------------
PROC ScSetAttribute
        mov     [attribute], al     ; Установить байт атрибута
        ret                         ; Возврат в вызывающую программу
ENDP ScSetAttribute
%NEWPAGE
;---------------------------------------------------------------
; ScInit Инициализация пакета SCREEN
;---------------------------------------------------------------
; Вход:
; нет
; Выход.
; Инициализируется vBASE
; Регистры:
; нет
;---------------------------------------------------------------
PROC ScInit
        push    ax                  ; Сохранить изменяемые регистры
        push    bx
        mov     ah, 15              ; Номер функции BIOS
        int     10h                 ; Получить в al видеорежим
        cmp     al, 7               ; Режим монохромный?
        jne     @@10                ; Если нет, переход
        mov     [vBASE], MonoBASE   ; Присвоить адрес для монохромного адаптера
@@10:
        pop     bx                  ; Восстановить регистры
        pop     ax
        ret                         ; Возврат в вызывающую программу
ENDP ScInit

        END                         ; Конец модуля
```

Пример использования SCREEN

В качестве модели программы для экспериментирования с новым модулем SCREEN рассмотрим небольшой пример. Программа 7.3, CHARS.ASM, выводит на экран диаграмму атрибутов и цветов вашей видеосистемы. Программа также показывает, как бесконфликтно комбинировать модуль STRIO из главы 5 с видеопроцедурами из SCREEN, несмотря на то что оба этих модуля имеют одноименные процедуры. Выполните ассемблирование, компоновку и запустите CHARS на исполнение с помощью следующих команд:

```
tasm /zi chars
tlink /v chars,,, mta
chars
```

Программа 7.3. CHARS.ASM

```
%TITLE "Display Character/Attribute Ref -- by Tom Swan"

        IDEAL

        MODEL small
        STACK 256


cr EQU 13       ; ASCII-возврат каретки
lf EQU 10       ; ASCII-прогон строки
ChartRow EQU 7 ; Строка для диаграммы атрибута


        DATASEG

exCode DB 0
welcome DB 'Character attributes -- by Tom Swan',cr,lf
        DB 'Rows=background, Columns=foreground',cr,lf
        DB 'First char is dim, second char is bright',0
template DB ' 00 01 02 03 04 05 06 07',cr,lf
        DB '00',cr,lf,'01',cr,lf,'02',cr,lf,'03',cr,lf
        DB '04',cr,lf,'05',cr,lf,'06',cr,lf,'07',0
blinkString DB 'This line should be blinking.', 0


        CODESEG

;----- Из STRINGS.OBJ, STRIO.OBJ
        EXTRN StrLength:proc, StrWrite:proc

;----- Из SCREEN.OBJ
        EXTRN ScInit:proc, ScGotoXY:proc, ScClrRect:proc
        EXTRN ScPokeChar:proc, ScSetBack:proc, ScSetFore:proc
        EXTRN ScPokeStr:proc, ScDim:proc, ScBright:proc
        EXTRN ScBlink:proc, ScNoBlink:proc

Start:
        mov ax, @data               ; Поместить в DS адрес
        mov ds, ax                  ; сегмента данных
        mov es, ax                  ; Установить es = ds

        call ScInit                 ; Инициализировать пакет SCREEN
        call Setup                  ; Установить дисплей
        call Attributes             ; Вывести диаграмму атрибута
        call Blinking               ; Вывести мерцающие символы

        mov dh, 23                  ; Позиционировать курсор на следующей
        mov dl, 0                   ; после последней выведенной строки
        call ScGotoXY               ; до завершения программы

Exit:
        mov ah, 04Ch                ; Функция DOS: Выход из программы
        mov al, [exCode]            ; Возвратить значение кода выхода
        int 21h                     ; Вызов DOS. Останов программы


;----- SETUP: Инициализация дисплея

PROC SetUp
        mov ch, 0                   ; Очистить экран
        mov cl, 0
        mov dh, 24
        mov dl, 79
        call ScClrRect
        mov dh, 1                   ; Вывести сообщение-приглашение
        mov dl, 0
        call ScGotoXY
```

```
68:         mov di, offset welcome
69:         call StrWrite
70:         mov dh, ChartRow          ; Вывести шаблон диаграммы
71:         mov dl, 0
72:         call ScGotoXY
73:         mov di, offset template
74:         call StrWrite
75:         ret
76:   ENDP Setup
77:
78:
79:   ;----- ATTRIBUTES: Вывести диаграмму атрибута
80:
81:   UDATASEG
82:   row DB ?                        ; Неинициализированные переменные
83:   column DB ?
84:   background DB ?
85:   foreground DB ?
86:
87:   CODESEG
88:   PROC Attributes
89:         mov [row], ChartRow       ; Инициализировать строку
90:         mov [background], 0       ; Инициализировать фон
91:   @@10:
92:         inc [row]                 ; Следующая строка
93:         mov al, [background]      ; Установить атрибут фона
94:         call ScSetBack
95:         mov [column], 1           ; Инициализировать столбец
96:         mov [foreground], 0       ; Инициализировать цвет переднего плана
97:   @@20:
98:         add [column], 3           ; Перейти на следующий столбец
99:         mov al, [foreground]      ; Установить атрибут переднего плана
100:        call ScSetFore
101:        call ScDim                ; Первый символ - тусклый
102:        call OneChar
103:        inc [column]
104:        call ScBright             ; Следующий символ - яркий
105:        call OneChar
106:        inc [foreground]          ; Повторить для всех цветов переднего плана
107:        cmp [foreground], 7
108:        jbe @@20
109:
110:        inc [background]          ; Повторить для всех цветов фона
111:        cmp [background], 7
112:        jbe @@10
113:
114:        ret
115:  ENDP Attributes
116:
117:
118:  ;----- ONECHAR: Подпрограмма, используемая в ATTRIBUTES
119:
120:  PROC OneChar
121:        mov dh, [row]             ; Получить номер строки
122:        mov dl, [column]          ; Получить номер столбца
123:        mov al, 'A'               ; Выводимый символ
124:        call ScPokeChar           ; Вывести символ
125:        ret
126:  ENDP OneChar
127:
128:
129:  ;----- BLINKING: Выводит мерцающий/немерцающий текст
130:
131:  PROC Blinking
132:        mov al, 0
133:        call ScSetBack            ; Установить черный фон
134:        mov al, 7
135:        call ScSetFore            ; Установить белый цвет переднего плана
136:        call ScBright             ; Сделать его еще более белым
137:        call ScBlink              ; Включить мерцание
138:        mov di, offset blinkString      ; Поместить в di адрес строки
139:        call StrLength            ; Установить в cx длину строки
140:        mov dh, 19                ; Задать позицию в dh, dl
141:        mov dl, 0
```


142:
143:
144:
145:
146:
147:
148:

Исполь

В моду
модуля. Вы
сывается,
собственны
ма 7.3). Е
SCREEN.A

Замечан

Самое ва
SCREEN
вы забудет

SetVidAdd

SetVid
Метод, пр
строки и с
бой позиц
те, если вы
значений в
дур из SCR

Сущест
ных номер
ределения
волов и ат
цей просмо

```
BytesPe
row = 0
LABEL S
REPT Ma
    dw
    ro
ENDM
```

Результ
сколькими
сива, разде
итерации ц
тов в одно
одной стро
символ, ка
ждого след

В резул
димой стро
жая сначал
di здресует
чтобы приб
строке. Это
and. Умно

```
142.          mov si, offset blinkString      ; Поместить адрес строки в si
143.          call ScPokeStr                  ; Вывести строку
144.          call ScNoBlink                  ; Выключить мерцание
145.          ret
146.    ENDP Blinking
147.
148.          END Start                       ; Конец программы / точка входа
```

Использование модуля SCREEN

В модуле SCREEN содержится 14 процедур общего назначения плюс одна, используемая другими процедурами модуля. Вы можете вызывать в своих программах любую из этих процедур общего назначения. В этом разделе описывается, как работает каждая из них, и указываются интересные методы, которые можно использовать в своих собственных проектах. Читая эти описания, для полноценного примера обращайтесь к CHARS.ASM (программа 7.3). *Если обратное не оговорено, то используемые здесь номера строк соответствуют номерам строк в* SCREEN.ASM, программа 7.2.

Замечание

Самое важное правило, которое следует помнить: перед использованием описываемых далее процедур из модуля SCREEN нужно вызывать ScInit. При этом vBASE инициализируется правильным адресом сегмента видеобуфера. Если вы забудете вызвать ScInit, ваши программы не будут работать на системах с монохромными адаптерами дисплеев.

SetVidAddr (51–75)

SetVidAddr вызывается другими процедурами SCREEN, поэтому вам никогда не придется вызывать ее прямо. Метод, применяемый в подпрограмме, пригоден для решения широкого класса задач. Процедура берет номера строки и столбца в dh и dl, затем возвращает в es:di адрес сегмента и смещения соответствующего символа из любой позиции экрана. Для повышения скорости никакие проверки задаваемых значений не выполняются. В результате, если вы попробуете задать координаты, выходящие за допустимый диапазон, это может привести к изменению значений в некотором месте памяти. Вы можете предотвратить эти разрушения проверкой dh и dl до вызова процедур из SCREEN, если не уверены, что значения находятся в подходящем диапазоне.

Существует несколько хорошо известных методов для вычисления адреса смещения в видеобуфере при заданных номерах строки и столбца позиции на экране. Обычно используется сложная формула, аналогичная методу определения значений в массивах, описанному в главе 6. (Видеобуфер, кроме всего, является массивом значений символов и атрибутов.) Однако существует лучший способ, который использует структуру данных (называемую *таблицей просмотра*), создаваемую в строках 36–42 и для удобства повторенную ниже:

```
BytesPerRow = MaxCol * 2
row = 0
LABEL ScRow Word
REPT MaxRow
     dw ( row * BytesPerRow )
     row = row + 1
ENDM
```

Результат работы этой конструкции аналогичен автоинициализации массивов, введенной в главе 6, но с несколькими новыми особенностями. Директива LABEL присваивает метке ScRow типа Word стартовый адрес массива, раздел REPT...ENDM повторяется количество раз, заданное в MaxRow (определяется в строке 11). На каждой итерации цикла директива dw определяет слово со значением, равным номеру строки, умноженному на число байтов в одной строке буфера, используя вычисляемое ранее числовое макроопределение BytesPerRow. Число байтов в одной строке буфера равно числу выводимых колонок (MaxCol), умноженному на 2, так как каждый выводимый символ, как вы помните, состоит из байта символа и байта атрибутов. После записи каждого слова в памяти для каждого следующего цикла row увеличивается на единицу.

В результате создается таблица слов, соответствующих адресам смещений самых левых символов каждой выводимой строки — (0,0), (0,1), (0,2), ..., (0,79). SetVidAddr получает новый правильный адрес из этой таблицы, умножая сначала номер строки на 2 (строки 67–69) и загружая затем адрес из таблицы в di (строка 70). С этого момента di адресует строку, содержащую символ и атрибут в позиции, определенной dh и dl. Последний шаг состоит в том, чтобы прибавить к di умноженный на 2 номер столбца, продвигая таким образом указатель на нужный адрес в этой строке. Это выполняется в строках 71–73 с помощью двух логических команд (xor и shl), следующих за командой and. Умножением на 2 учитывается наличие в каждой позиции байта символа и байта атрибута.

Путем использования логических команд и таблицы просмотра во избежание повторяющихся вычислений достигается большая скорость. В своих собственных программах, когда вам требуется вычислять значения, зависящие от параметров, находящихся в известных диапазонах (как здесь номера строк и столбцов), вычисляйте эти значения предварительно и записывайте их в таблицы просмотра. Это может значительно увеличить скорость программы — особенно для процедур типа SetVidAddr, которые будут вызываться тысячи раз при обычном исполнении программы.

ScGotoXY (77-98)

ScReadXY (100-123)

Поскольку эти две процедуры дополняют одна другую, имеет смысл рассматривать их вместе. ScGotoXY позиционирует курсор в позиции, определенной в dh (строка) и dl (столбец), вызывая функцию BIOS 10h, как описывалось ранее в этой главе. ScReadXY возвращает в эти же регистры текущую позицию курсора. Обе процедуры также устанавливают в bh номер текущей видеостраницы — важный шаг, который игнорируется в процедурах позиционирования курсора многих программ.

При использовании в своих программах процедур прямого вывода в видеопамять помните, что вы отвечаете за управление курсором и решаете, куда выводить текст.

ScPokeChar (125–145)

ScPokeStr (147–172)

Эти две процедуры — короткие и очень быстрые. ScPokeChar выводит символ из al, который может представляться расширенным ASCII-кодом от 0 до 255, в строку и столбец, заданные в dh и dl. Если существует вероятность, что эти значения могут выходить за допустимые пределы, поместите перед вызовом ScPokeChar и ScPokeStr код, подобный следующему:

```
        cmp     dh, 24          ; dh (строка) <= 24 ?
        jbe     @@10            ; Перейти, если dh <= 24
        mov     dh,24           ; Иначе установить dh = 24
@@10:
        cmp     dl, 79          ; dl (столбец) <= 79 ?
        jbe     @@20            ; Перейти, если dl <= 79
        mov     dl, 79          ; Иначе установить dl = 79
@@20:
```

После этого можно спокойно вызывать ScPokeChar для вывода оного символа, не беспокоясь, что этим можно изменить другие значения в памяти. Конечно, для достижения максимальной скорости, и если вы уверены, что номера строк и столбцов всегда находятся в допустимом диапазоне, можно отказаться от этих проверок. Например, при выполнении следующего кода в конце каждой выводимой строки помещается знак плюс:

```
        mov     dh, 24          ; Поместить в dh максимальное количество строк
        mov     dl, 79          ; Поместить в dl максимальное количество столбцов
@@10:
        mov     al, ' + '       ; Выводимый символ
        push    dx              ; Сохранить dx - изменяемый ScPokeChar
        call    ScPokeChar      ; Вывести один символ
        pop     dx              ; Восстановить dx
        dec     dh              ; Вычесть 1 из номера строки
        jns     @@10            ; Перейти, если dh >= 0
```

Обратите внимание на то, как в этом фрагменте, пока результат положителен или равен 0, в цикле уменьшается номер строки в dh. Если dh становится меньшим 0, флаг знака sf устанавливается в 1 и переход по команде jns больше не выполняется.

Для ускорения исполнения в эти процедуры не включены команды проверки на эффект "снега", описанные раньше. Если вы используете текстовый экран CGA и у вас возникают проблемы со "снегом", можно изменить обе процедуры, выполняя запись в видеобуфер в период вертикального возврата.

ScPokeChar и ScPokeStr выводят текст, используя текущие установки атрибутов, которые могут изменяться другими процедурами SCREEN. (Например, см. ScSetBack и ScSetFore.) Программа CHARS.ASM является хорошим примером того, как выводить символы во всевозможных комбинациях. Обе процедуры вызывают также SetVidAddr для инициализации es:di правильным адресом в видеобуфере, который соответствует требуемым строке и столбцу.

ScPokeStr выводит целую строку, которая может быть как в ASCIIZ-формате, так и в других форматах. Для использования этой процедуры нужно установить в cx количество выводимых символов, в dh и dl — номер строки и столбца, начиная с которых будет выводиться строка, и в ds:si — адрес первого символа строки. Если строка пред-

ставлена в ASCIIZ-формате, можно вызывать процедуру StrLength из модуля STRINGS, инициализируя cx до вызова ScPokeStr.

Замечание

Вывод текста с помощью ScPokeChar и ScPokeStr никогда не вызывает прокрутки экрана. Это означает, что можно выводить символ в правый нижний угол в позицию (79, 24), не оказывая влияния на остальной текст, находящийся на экране. Эти две процедуры выводят символы с любым расширенным ASCII-кодом от 0 до 255, включая возвраты каретки, прогоны строк, звонки и другие управляющие коды.

ScClrRect (174–192)

ScClrRect очищает прямоугольник, определенный регистрами ch, cl (верхние левые строка и столбец) и dh, dl (нижние правые строка и столбец). Убедитесь до вызова ScClrRect, что значения регистров находятся в допустимых диапазонах, так как она не выполняет соответствующих проверок. Процедура вызывает прерывание 10h ROM BIOS с ah, равным 6 (номер команды прокрутки). Если al равно 0, эта процедура очищает определенную область экрана, используя атрибуты, заданные в bh (см. строку 189).

Некоторые программисты разрабатывают свои сверхбыстрые процедуры очистки экрана, что, конечно, можете сделать и вы, используя описанные ранее методы для записи в видеобуфер. Например, можно просто очистить весь видеобуфер, используя повторяющуюся команду stosw для установки каждого символа в пробел (ASCII 20h) и каждого атрибута в определенный цвет фона (обычно в 0 для черного). Однако для большинства случаев вполне достаточно стандартного метода, используемого в ScClrRect.

ScSetBack (194–215)

ScSetFore (217–238)

Применяйте эти процедуры для изменения установок атрибутов цвета фона и переднего плана, используемых при следующих вызовах ScPokeChar, ScPokeStr и ScClrRect. Вызывайте ScSetBack с al, равным новому цвету фона, и значением от 0 до 7. Вызывайте ScSetFore с al, равным новому цвету переднего плана (самого символа), и значением также от 0 до 7. В табл. 7.4 представлены значения цветов для дисплеев CGA, EGA и VGA. (Для получения цвета переднего плана повышенной интенсивности следует вызывать описанные далее процедуры ScBrigth и ScDim. Цвета фона усиливаться не могут.) В табл. 7.5 представлены эквивалентные значения и связанные с ними эффекты для монохромных дисплеев. Кроме этого, можно вызывать ScBrigth, ScDim, ScBlink и ScNoBlink для дополнительных эффектов. Также допускаются другие значения цветов фона и переднего плана в интервале от 0 до 7, но они приводят к тем же видимым эффектам, что и значения в таблице.

ScSetBack и ScSetFore используют метод упакованных битовых полей для изменения отдельных значений в байтах атрибутов, определенных в записи attrByte в строке 19, в соответствии с рис. 7.3. Обратите внимание, как условные операторы IF/ENDIF в строках 205–210 и 228–233 предотвращают ассемблирование ненужного кода, если поля Foreground и Background уже находятся в правом конце байта. В этом случае, так как формат байта атрибутов изменять не надо, дополнительные операторы IF/ENDIF, возможно, и не нужны. Но, несмотря на это, данные команды показывают, как писать процедуры, допускающие изменения форматов записей.

Таблица 7.4. **Значения цветов переднего плана и фона**

Значение	Цвет	Интенсивность (только для переднего плана)
0	Черный	Темно-серый
1	Синий	Светло-синий
2	Зеленый	Светло-зеленый
3	Голубой	Светло-голубой
4	Красный	Светло-красный
5	Фиолетовый	Светло-фиолетовый
6	Коричневый	Желтый
7	Белый	Ярко-белый

Таблица 7.5. **Значение атрибута для монохромного дисплея**

Фоновая	Передний план	Эффект
0	0	Не выводится
0	1	Подчеркивание
0	7	Обычный текст
7	0	Негативный текст

ScBringht (240–256)

ScDim (258–261)

ScBlink (263–266)

ScNoBlink (268–271)

Эти четыре процедуры изменяют биты мерцания и интенсивности в переменной атрибута. Для установки и очистки этих битов команды используют маски and и or, модифицируя значения, установленные ScSetBack, ScSetFore и используемые затем при следующих вызовах ScPokeChar, ScPokeStr и ScClrRect. Названия и назначение этих процедур должны быть очевидными.

Замечание

Вследствие аппаратных ограничений мигать могут только цвета на переднем плане. Фоновые цвета не мигают. Более того, на цветных дисплеях некоторые „тусклые" цвета на самом деле ярче, чем цвета с повышенной интенсивностью. Мне кажется, что лучше рассматривать „интенсивные" цвета как цвета, смешанные с белой краской, а не как „более яркие".

ScGetAttribute (273–286)

ScSetAttribute (288–301)

Вместо вызовов ScSetFore, ScSetBack, ScBrigth, ScBlink и ScNoBlink можно вызывать ScSetAttribute с 8-битовым значением атрибута. При последующих вызовах ScPokeChar, ScPokeStr и ScClrRect это новое значение будет использоваться для всего выводимого текста. В большинстве случаев это быстрее, чем вызов для выбора атрибутов цвета различных комбинаций других процедур. Используя эту процедуру совместно с ScGetAttribute, если вы захотели временно изменить цвет, можно сохранять и восстанавливать текущий атрибут. Например, для вывода бросающегося в глаза красного сообщения об ошибке можно воспользоваться следующим кодом:

```
call ScGetAttribute   ; Записать текущий атрибут в dl
push dx               ; Сохранить значение в стеке
mov al, 4             ; Присвоить красный цвет al
call ScSetFore        ; Изменить цвет переднего плана на красный
call ScBringht        ; Увеличить интенсивность цвета
call ScBlink          ; Установить мерцание цвета переднего плана
;
;-----------------вывод сообщения об ошибке с новыми атрибутами
;
pop  ax               ; Вытолкнуть атрибут из стека
call ScSetAttribute   ; Вернуть атрибуту начальное значение
```

Другой полезный прием состоит в том, чтобы задать значения атрибутов с помощью вызовов ScSetFore и ScSetBack, после чего запомнить их в переменной для последующего использования. Например, можно сделать это в установочной утилите, которая позволяет настраивать цвета в главной программе:

```
DATASEG
customColor  db              0
CODESEG
mov  al, 6                   ; Присвоить желтый цвет al
call ScSetFore               ; Изменить цвет переднего плана на желтый
call ScBringht               ; Увеличить интенсивность цвета
mov  al, 1                   ; Присвоить голубой цвет al
call ScSetBack               ; Изменить цвет фона на голубой
call ScGetAttribute          ; Получить составной атрибут
mov  [customColor], dl       ; Сохранить атрибут на будущее
```

Для использования атрибута нужно загрузить [customColor] в al и вызвать ScSetAttribute, больше ничего делать не надо.

ScInit (303–325)

ScInit — последняя процедура в модуле SCREEN, и вы должны вызывать ее в начале программы перед использованием для вывода текста ScPokeChar или ScPokeStr. Поскольку предварительно vBASE инициализируется адресом сегмента цветного дисплея (см. строку 31), то, если вы забудете вызвать ScInit, ваша программа не будет работать на системах с монохромными (включая Hercules) адаптерами.

Модуль управления клавиатурой

Практически всегда методы, описанные в начале этой главы, обеспечивают языку ассемблера адекватные возможности для ввода с клавиатуры. Однако существуют ситуации, когда использование стандартных функций DOS становится неудобным. Например, вы можете захотеть запретить пользователю перенаправлять ввод. Кроме того, DOS делает сложной обработку функциональных и специальных клавиш, требуя вызова двух функций для чтения одной ключевой последовательности.

В ответ на это программа 7.4, KEYBOARD.ASM, содержит две функции, которые я считаю полезными. Все нажатия клавиш, включая ASCII-символы, управляющие и функциональные клавиши, читаются с помощью одного вызова подпрограммы. После текста программы следует пример, объясняющий, как это работает. Выполните ассемблирование KEYBOARD и запишите его в файл библиотеки MTA.LIB с помощью команд:

```
tasm /zi keyboard
tlib /E mta -+keyboard
```

Как всегда, не обращайте внимание на возможное предупреждение о том, что KEYBOARD отсутствует в библиотеке, и для уменьшения размера получаемого файла не используйте опцию /zi, если вы не планируете исполнять главную программу в Turbo Debugger.

Программа 7.4. KEYBOARD.ASM

```
1:    %TITLE "Keyboard Input Routines - Copyright (c) 1989,1995 -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:
7:
8:          CODESEG
9:
10.         PUBLIC KeyWaiting, GetCh
11:
12:   %NEWPAGE
13:   ;---------------------------------------------------------------
14:   ; KeyWaiting Проверка на нажатие клавиатуры
15:   ;---------------------------------------------------------------
16:   ; Вход:
17:   ; нет
18:   ; Выход:
19.   ; zf = 0 : (JNZ) Есть символ для чтения
20:   ; zf = 1 : (JZ) Нет символа для чтения
21:   ; Регистры:
22:   ; нет (только флаги)
23:   ;---------------------------------------------------------------
24:   PROC KeyWaiting
25:         push  ax                ; Сохранить изменяемые регистры
26:         mov   ah, 1             ; Функция проверки буфера BIOS
27:         int   16h               ; Вызов функции обработки клавиатуры BIOS
28:         pop   ax                ; Восстановить регистры
29:         ret                     ; Возврат в вызывающую программу
30:   ENDP KeyWaiting
31:   %NEWPAGE
32:   ;---------------------------------------------------------------
33:   ; GetCh      Возвращает значение ASCII-символа, управляющей или функциональной клавиши
34:   ;---------------------------------------------------------------
35:   ; Вход:
36:   ; нет
```

```
37:    ; Выход:
38:    ; zf = 0 (ah = 1) : (JNZ) al = ASCII-символ
39:    ; zf = 1 (ah = 0) : (JZ) al = ASCII-значение управляющей или функциональной клавиши
40:    ; Регистры:
41:    ; ax
42:    ;--------------------------------------------------------------
43:    PROC GetCh
44:          xor   ah, ah              ; Функция чтения клавиши BIOS
45:          int   16h                 ; Вызов функции обработки клавиатуры BIOS
46:          or    al, al              ; ASCII коде = 0?
47:          jnz   @@10                ; Если нет, переход (не специальная клавиша)
48:          xchg  ah, al              ; Иначе установить ah <- 0, al <- скан-код
49:          add   al, 32              ; Сделать скан-код >= 32
50:          jmp   short @@20          ; Переход на выход
51:    @@10:
52:          xor   ah, ah              ; Инициализировать ah нулем
53:          cmp   al, 32              ; ASCII-код < 32 (т.е. Ctrl)?
54:          jb    @@20                ; Если да, переход (al = управляющая клавиша)
55:          inc   ah                  ; Иначе установит ah = 1 (al = ASCII-символ)
56     @@20
57:          or    ah, ah              ; Установить или очистить флаг результата zf
58:          ret                       ; Возврат в вызывающую программу
59:    ENDP GetCh
60:
61:          END                       ; Конец модуля
```

Пример использования KEYBOARD

Программа 7.5, KEY.ASM, показывает, как использовать модуль KEYBOARD. Запустив программу на исполнение, нажмите любую клавишу и вы увидите ее тип и числовое значение. (Замечание: вы можете заметить, что значения функциональных клавиш отличаются от соответствующих значений в других программах. Причина этого расхождения будет объяснена позже.) Для окончания программы нажмите <Esc>. Полагая, что в библиотеке содержатся модули из этой и предыдущих глав, выполните ассемблирование, компоновку и запустите KEYS на исполнение с помощью команд:

```
tasm /zi keys
tlink /v keys,,, mta
keys
```

Программа 7.5. KEYS.ASM

```
1:     %TITLE "Display Key Values -- Copyright (c) 1989,1995 -- by Tom Swan"
2:
3:           IDEAL
4:
5:           MODEL small
6:           STACK 256
7:
8:     cr EQU 13                       ; ASCII-возврат каретки
9:     lf EQU 10                       ; ASCII-прогон строки
10:
11:
12:          DATASEG
13:
14:    exCode       DB                 0
15:    charKey      DB                 'Character key : ', 0
16:    funcKey      DB                 'Function key : ', 0
17:    numString    DB                 7 DUP (0)
18:    welcome      DB                 cr,lf,'Display Key Values -- by Tom Swan'
19:                 DB                 cr,lf,'Press any key, or press Esc to quit'
20:                 DB                 cr,lf,lf,0
21:
22:
23:          CODESEG
24:
25:    ;----- Из BINASC.OBJ
26:          EXTRN BinToAscDec:proc
27:
```

```
28:   ;----- Из STRIO.OBJ
29:         EXTRN StrWrite:proc, NewLine:proc
30:
31:   ;----- Из KEYBOARD.OBJ
32:         EXTRN Keywaiting:proc, Getch:proc
33:
34:   Start:
35:         mov   ax, @data           ; Поместить в DS адрес
36:         mov   ds, ax              ; сегмента данных
37:         mov   es, ax              ; Установить es = ds
38:
39:         mov   di, offset welcome  ; Вывести сообщение-приглашение
40:         call  StrWrite
41:
42    Repeat:
43:         call  KeyWaiting          ; Ожидание нажатия клавиши
44:         jz    Repeat              ; Повторять до нажатия клавиши
45:         call  GetCh               ; Считать клавишу
46:         mov   di, offset charKey  ; Адрес строки charKey
47:         jnz   @@10                ; Переход, если введен символ
48:         cmp   al, 27              ; Нажата ли клавиша <Escape>?
49:         je    Exit                ; Если да, выход
50:         mov   di, offset funcKey  ; Адрес funcKey
51:   @@10:
52:         call  StrWrite            ; Вывести тип нажатой клавиши
53:         xor   ah, ah              ; Преобразовать al в 16-битовое представление
54:         mov   cx, 1               ; Минимальное количество цифр
55:         mov   di, offset numString      ; Адрес строки цифр
56:         call  BinToAscDec         ; Преобразовать число в строку
57:         call  StrWrite            ; Вывести значение клавиши
58:         call  NewLine             ; Перейти на новую строку
59:         jmp   Repeat              ; Перейти к обработке нажатия следующей клавиши
60:
61:   Exit.
62:         mov   ah, 04Ch            ; Функция DOS : выход из программы
63:         mov   al, [exCode]        ; Возвратить значение кода выхода
64:         int   21h                 ; Вызов DOS. Останов программы
65:
66:         END   Start               ; Конец программы / точка входа
```

Использование модуля KEYBOARD

Замечание

Используемые номера строк, если не оговорено обратное, соответствуют номерам в программе 7.4.

KeyWaiting (13–30)

Если символ ожидает считывания в буфере клавиатуры, KeyWaiting возвращает очищенным флаг zf (равным 0). Если флаг zf установлен (равен 1), значит, символа нет. Используйте KeyWaiting в цикле следующим образом:

```
@@10:
    call    AnyProcedure          ; Код, исполняющийся во время ожидания
    call    KeyWaiting            ; Проверка на нажатие клавиши
    jz      @@10                  ; Перейти, если клавиша не нажата
    call    GetCh                 ; Прочесть символ с клавиатуры
```

GetCh (32–59)

GetCh — это мой личный ответ на дилемму о чтении функциональных клавиш PC. "Обычный" метод состоит в двукратном вызове функции ввода DOS — один раз для считывания первого нулевого символа (ASCII 0), другой — для получения значения функциональной клавиши. Поэтому, чтобы избежать вывода этих специальных значений в виде текста, все программы должны уметь определять функциональные клавиши. (Вы, вероятно, встречали программы, которые не делают этого и выводят "K" или другие странные буквы при нажатии стрелок или функциональных клавиш.)

Возвращаемый GetCh флаг нуля zf показывает, является ли получаемый в ah символ обычным ASCII-символом (zf=0) либо управляющей или функциональной клавишей (zf=1). Значение ASCII-символа изменяется от 32 до 255. Значения функциональных и управляющих клавиш изменяются от 0 до 255. Для обработки любой ключевой строки вы

просто должны вызвать GetCh. В табл. 7.7 представлены дополнительные значения для клавиш с обычными ASCII-значениями в двух первых столбцах (zf=0) и различные комбинации <Ctrl>, <Alt> и <Shift+Ctrl> для этих же клавиш в других колонках (zf=1). Недоступные значения отмечены черточкой. Комбинации клавиш, возвращающие те же значения, что и другие комбинации, показаны в скобках. Все значения в обеих таблицах — десятичные.

Использовать GetCh просто. Вызовите подпрограмму и для различения обычных ASCII и функциональных или управляющих клавиш проверьте состояние флага zf:

```
call GetCh                  ; Получить символ с клавиатуры
jz   FunctionKey            ; Вызвать программу для обработки
                            ; функциональных/управляющих клавиш
jnz  ASCIIKey               ; Вызов программы для обработки обычных ASCII клавиш
```

GetCh работает, вызывая прерывание 16h ROM BIOS с ah, равным 0, читая следующую нажатую клавишу или получая ее значение из буфера клавиатуры. Прерывание BIOS возвращает скан-код клавиши (число, представляющее позицию клавиши) в ah и ASCII-значение в al. Если al равно 0, тогда ah представляет функциональную клавишу; в противном случае — это обычный ASCII-символ. В строках 48–50 к значениям функциональных клавиш прибавляется 32, чтобы предотвратить конфликт с управляющими кодами из диапазона 0–31. По этой причине значения, возвращаемые GetCh, не совпадают со значениями соответствующих функций в языках высокого уровня. Для определения клавиши, которая произвела данное значение, используйте программу KEYS вместе с табл. 7.6 и 7.7. Остальные команды в GetCh устанавливают ah в 1 для ASCII-символов и в 0 — для функциональных и управляющих клавиш. В строке 57, если ah равно 0, флаг zf устанавливается в 1.

Таблица 7.6. **Значения функции GetCh и управляющих клавиш**

Клавиша	Обычная	+Shift	+Ctrl	+Alt
F1	91	116	126	136
F2	92	117	127	137
F3	93	118	128	138
F4	94	119	129	139
F5	95	120	130	140
F6	96	121	131	141
F7	97	122	132	142
F8	98	123	133	143
F9	99	124	134	144
F10	100	125	135	145
Ins	114	(114)	-	-
Del	115	(115)	-	-
Home	103	(103)	151	-
PgUp	105	(105)	164	-
PgDn	113	(113)	150	-
Up	104	(104)	-	-
Down	112	(112)	-	-
Left	107	(107)	147	-
Right	109	(109)	148	-
End	111	(111)	149	-
Esc	27	(27)	(27)	-

Таблица 7.7. **Дополнительные значения клавиш GetCh**

Клавиша	Обычная	+Shift	+Ctrl	+Alt
A	97	65	1	62
B	98	66	2	80
C	99	67	3	78

Окончание таблицы 7.7

Клавиша	Обычная	+Shift	+Ctrl	+Alt
D	100	68	4	64
E	101	69	5	50
F	102	70	6	65
G	103	71	7	66
H	104	72	8	67
I	105	73	9	55
J	106	74	10	68
K	107	75	11	69
L	108	76	12	70
M	109	77	13	82
N	110	78	14	81
O	111	79	15	56
P	112	80	16	57
Q	113	81	17	48
R	114	82	18	51
S	115	83	19	63
T	116	84	20	52
U	117	85	21	54
V	118	86	22	79
W	119	87	23	49
X	120	88	24	77
Y	121	89	25	53
Z	122	90	26	76
0	48	41	-	161
1	49	33	-	152
2	0	64	35	153
3	51	35	-	154
4	52	36	-	155
5	53	37	-	156
6	54	94	30	157
7	55	38	-	158
8	56	42	-	159
9	57	40	-	160
]	93	125	29	-
[	91	123	27	-
-	45	95	31	162

Резюме

Стандартные методы ввода-вывода DOS могут быть не очень привлекательными, но они позволяют программе исполняться на максимально возможном количестве различных систем. Одно из преимуществ использования стандартного ввода-вывода DOS состоит в том, что пользователь имеет возможность перенаправить ввод и вывод, причем программа (или вы) об этом даже не будете знать.

Буфер клавиатуры заполняется вводимыми символами независимо от других действий в программе. При каждом нажатии клавиши вызывается прерывание, получающее значение клавиши и записывающее ее в память. Когда стандартным устройством ввода является клавиатура, как это обычно и бывает, при вызове функций ввода DOS

значения клавиш удаляются из буфера клавиатуры. Очистить буфер клавиатуры можно очень просто, приравняв два указателя, которые отмечают первый и последний символы буфера.

Дескрипторы являются значениями, связанными с логическими файлами, обеспечивающими интерфейс между программой и периферийными устройствами. DOS инициализирует пять дескрипторов, которые могут использоваться в программе для вывода на экран, чтения с клавиатуры, вывода сообщений об ошибках, доступа к портам и печати текста. Дескрипторы удобно использовать для создания простых программ-фильтров, которые могут объединяться в конвейер для выполнения сложных операций.

Знак доллара ($) в Turbo Assembler является счетчиком положения, равным текущему адресу в любом месте программы. Этот символ особенно полезен для определения размеров переменных, в частности строк. В режиме Ideal макроопределения, использующие счетчик положения, должны быть описаны с помощью знака равенства.

Проще всего вывод на принтер осуществляется с помощью записи на стандартное устройство печати DOS, используя один из предопределенных дескрипторов. Вызывать для печати текста ROM BIOS нецелесообразно, так как эти функции не работают с принтерами, присоединенными к последовательному порту.

Самый быстрый способ вывода текста — это запись символов непосредственно в видеобуферы. В буферах памяти содержатся вместе символы и значения атрибутов, которые устанавливают цвета и такие характеристики, как подчеркивание и негативное изображение на монохромных системах. Использование этих методов на старых текстовых дисплеях CGA может приводить к появлению "снега". Эта проблема может быть устранена с помощью синхронизации программы и сигнала вертикального возврата луча дисплея, однако это приводит к значительной потере скорости вывода.

Упражнения

7.1. Какие три функции DOS можно использовать для вывода единичных символов? Напишите команды языка ассемблера для вызова этих функций.

7.2. Напишите программу для считывания одного символа с клавиатуры, преобразования его в прописной (независимо от того, нажаты ли клавиши <Caps Lock> или <Shift>, и направьте измененные символы в стандартный файл вывода. Программа должна завершаться по нажатии <Esc>.

7.3. Напишите подпрограмму, которая возвращает установленным флаг нуля (zf=1), если нажата клавиша <Esc>. Подпрограмма должна возвращать флаг нуля очищенным (zf=0), если: а) клавиши не нажимались, б) нажимались клавиши, отличные от <Esc>. Подпрограмма должна возвращать zf=1 только в том случае, если была нажата клавиша <Esc>. Подпрограмма не должна останавливаться для ввода и должна сохранять все регистры. (ASCII-значение <Esc> равно 27 в десятичной системе.)

7.4. Пересмотрите свой ответ на Упражнение 7.3 так, чтобы zf=1 возвращался, если была нажата клавиша <F1>. Не используйте для этого GetCh из модуля KEYBOARD.

7.5. Что такое дескриптор? Как использовать дескрипторы? Сколько дескрипторов описано в DOS?

7.6. Почему полезны программы-фильтры? Назовите по меньшей мере один фильтр, поставляемый с DOS.

7.7. Создайте макроопределение, которому автоматически присваивается значение длины строки "I hate meeses to pieces."

7.8. Напишите подпрограмму для заполнения экрана любым одиночным символом, передавая его как параметр через регистр al. Используйте в своем ответе модуль SCREEN.

7.9. Выведите строку "ERROR: Dumb mistake detected" яркими белыми мигающими буквами на красном фоне цветного экрана. Используйте в своем ответе модуль SCREEN. (Замечание: на монохромных дисплеях красный фон отображается черным цветом. Под управлением Microsoft Windows, в зависимости от типа и режима дисплея, мигающие символы могут не допускаться.)

7.10. Какую программу необходимо вызывать из модуля SCREEN при использовании монохромных дисплеев?

7.11. Напишите подпрограмму, которая возвращает установленным флаг нуля (zf=1), если пользователь нажал клавишу <Y>. При нажатии любой другой клавиши флаг должен очищаться (zf=0). Сохраняйте все регистры. Используйте в своем ответе модуль KEYBOARD.

Задания

7.1. Создайте объектный модуль с такими функциями, как очистка экрана, очистка конца строки, очистка до конца экрана, позиционирование курсора и звонок. Модуль должен использовать вызовы стандартных функций DOS.

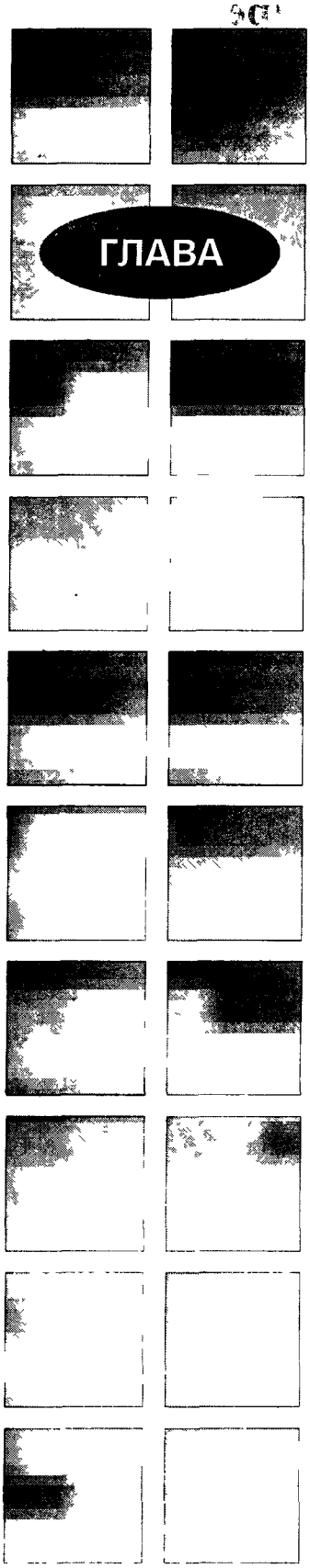
7.2. Напишите подпрограмму, вставляющую последовательности символов (желательно ASCIIZ-строки) в буфер клавиатуры. Как можно использовать эту подпрограмму?

7 3. Напишите фильтр, преобразующий символы табуляции в тексте в пробелы. Напишите другой фильтр для преобразования пробелов в символы табуляции.

7.4 Напишите программу для выбора всех (или большинства) специальных режимов печати вашего принтера. Сделайте программу простой для настройки на другие модели принтеров.

7 5. Измените модуль SCREEN для устранения "снега" на текстовых дисплеях CGA.

7.6. [Повышенной сложности] Напишите объектный модуль для прокрутки экрана вверх, вниз, влево и вправо, не используя для этого функции BIOS.



ГЛАВА 8

Макросы и условное ассемблирование

- Что такое макрос
- Преимущества и недостатки макросов
- Создание макросов
- Удаление макросов
- Подстановка параметров
- Макросы и переменные
- Макросы и код
- Условная компиляция
- Запуск библиотеки макросов DOS
- Резюме
- Упражнения
- Задания

Что такое макрос

По мере накопления опыта в программировании на языке ассемблера вы, несомненно, будете повторяться, переписывая снова и снова одни и те же последовательности команд. Чтобы уменьшить количество повторов в программе, можно хранить одну или больше команд в именованном *макросе* и потом использовать его имя всякий раз, когда вам необходим этот код. Когда Turbo Assembler встречает макроимя, он заменяет его последовательностью команд из макроса. Вдобавок можно передавать параметры в макросы, изменяя согласно новым требованиям ассемблируемые команды.

С помощью макросов, в соответствии с вашими вкусами, вы вводите в привычный Turbo Assembler новые команды.

В дополнение к широкому выбору макрооператоров и директив Turbo Assembler предоставляет набор *директив условного ассемблирования*, которые часто используются в макросах. Эти директивы позволяют писать программы, которые ассемблируются по-разному, основываясь на различных условиях, содержащихся обычно в начале модуля или программы. Например, можно писать программы, которые ассемблируют специальный код для отладки, а затем удалить его из конечной версии.

Преимущества и недостатки макросов

Некоторые программисты никогда не используют макросов. Другие создают обширные библиотеки сложных макросов, и в их программах оказывается больше идентификаторов макросов, чем обыкновенных мнемоник языка ассемблера. При этом макросы становятся персональными, позволяя программистам проявить свою индивидуальность в Turbo Assembler. В групповых проектах макросы могут помогать обеспечивать преемственность. Например, компания, разрабатывающая программное обеспечение, может создать макробиблиотеку, сокращающую количество ошибок, совершаемых из-за простой невнимательности. Макросы могут создаваться для управления таким специальным аппаратным обеспечением, как CRT контроллеры и плоттеры. Тогда эти макросы могли бы использовать члены рабочей группы для всех процессов ввода-вывода на устройства, обеспечивая ассемблирование для этих специальных операций правильных последовательностей команд.

Макросы также могут помочь прояснить логику программы, заменяя зашифрованные мнемоники языка ассемблера на макроимена, такие как GetValue и RingBell. Хороший набор макроимен может сделать программы на языке ассемблера похожими на программы Pascal или C.

Но, несмотря на эти и другие преимущества, макросы имеют несколько недостатков. В отличие от отдельных объектных модулей, которые можно вставить в библиотечный файл для непосредственной компоновки с программами, макросы хранятся в виде текста и, следовательно, должны переассемблироваться для каждого отдельного модуля. По этой причине обширная библиотека макросов может увеличивать время ассемблирования, особенно если используются только несколько из большого количества макросов, содержащихся в библиотеке. Упрощая и проясняя язык ассемблера, макросы также могут легко скрывать результаты работы индивидуальных команд. Хорошим примером может служить макрокоманда, которая изменяет значение регистра, что вовсе неочевидно из простого чтения текста программы. Так же как и подпрограммы, макросы требуют подробного документирования использования регистров, флагов и переменных.

Создание макросов

Вы можете определить макрос в любом месте программы, но самое общее и, возможно, наилучшее место для макроса — в начале файла, вместе с другими макросами, записями и структурами. Простейший макрос начинается с ключевого слова MACRO, за которым следует одна или несколько команд, и заканчивается ENDM:

```
MACRO Terminate
    mov     ah,04ch         ;; Функция "Выход из программы"
    mov     al,[exCode]     ;; Загрузить exCode в al
    int     21h             ;; Вызов DOS. Останов программы
ENDM Terminate
```

Вы, возможно, узнали эти команды, они используются в большинстве программ данной книги для возврата управления COMMAND.COM после завершения программы. Если вставить этот макрос в программу — лучше перед директивой DATASEG, то можно завершить программу, просто написав Terminate. Во время ассемблирования Turbo Assembler заменяет имя макроса командами из этого макроса; этот процесс называется *макроразворачиванием*. Конечно, если вы используете Terminate только однажды, едва ли стоит хранить команды в макросе. Тем не менее понятно, что означает Terminate, а внесение в программу дополнительной ясности уже само по себе играет важную роль.

Обратите внимание, что комментарии в этом макросе начинаются с двух знаков "точка с запятой", хотя обычно они начинаются с одного такого знака. В макросах допускаются оба типа комментариев, но если при ассемблировании вы используете опцию /l, то комментарии с одним знаком "точка с запятой" будут выводиться в файл листинга. Если в программе многократно используются одни и те же макросы, то вывод повторяющихся комментариев может удлинять время печати. В этом случае можно удалить комментарии из макрорасширений, используя двойные точки с запятой.

Удаление макросов

После чтения макроса Turbo Assembler запоминает макроимя и последовательность команд на время всей программы. При ассемблировании больших программ с обширными библиотеками макросов, если ваша система имеет ограничения по памяти, ассемблеру может не хватить места для новых символов. Если такое случается и во время ассемблирования вы получаете ошибку, связанную с недостатком памяти, можно удалить ненужные вам макросы, освобождая дополнительную память для других целей. Чтобы удалить макрос, используйте ключевое слово PURGE вместе с его именем:

```
PURGE Terminate
```

После удаления Terminate Turbo Assembler больше не распознает этого имени. Другая причина удаления макроса — временная замена макроса, находящегося в библиотеке, новой последовательностью команд. Это может быть полезно, когда вам требуется проверить версию макроса, перед тем как окончательно включить его в библиотеку. Например, чтобы заменить Terminate последовательностью кодов, которые перезапускают программу, можно записать:

```
PURGE  Terminate
MACRO  Terminate
       jmp Start
ENDM   Terminate
```

Директива PURGE устраняет старое определение Terminate, после чего создается новый макрос с тем же именем. Если вы сделаете это в начале программы, то в каждом месте, в котором Terminate прежде заканчивал программу, теперь будет осуществляться переход в начало программы на метку Start: (здесь не показана). Вы можете сделать это для создания демонстрационной версии вашей программы, которая нормально запускается, но никогда не заканчивается.

Замечание

Turbo Debugger считает макросы обычными командами языка ассемблера. При нажатии <F7> или <F8>, когда курсор находится на имени макроса, выполняются все связанные с ним команды. Чтобы пошагово выполнить отдельные команды макроса, вызовите окно CPU и нажмите <Ctrl+M> для выбора предпочитаемого вами стиля вывода. Нажимая теперь <F7> или <F8>, вы будете перемещаться по отдельным командам макроса.

Подстановка параметров

Добавление параметров к макросам дает вам возможность изменять их при ассемблировании. В макросах могут использоваться три типа параметров:

- символьные параметры
- числовые параметры
- строковые параметры.

Символьные параметры ссылаются на имена регистров, мнемонических команд и другие ключевые слова и идентификаторы языка ассемблера. *Числовые параметры* являются знаковыми или беззнаковыми целыми числами либо выражениями. *Строковые параметры* — это обычные неограниченные строки символов. Тип параметра определяется при его использовании, поэтому некоторые параметры могут представлять более чем один тип данных. Называйте свои параметры аналогично другим идентификаторам, размещая их в первой строке макроса:

```
MACRO  Swap16 v1, v2
       push   [word v1]
       push   [word v2]
       pop    [word v1]
       pop    [word v2]
ENDM . Swap 16
```

Этот макрос, с именем Swap 6, определяет два параметра — v1 и v2, называющихся *подставляемыми параметрами*, или, более точно, *формальными параметрами*. В действительности в программе не существует никаких переменных с именами v1 и v2 — эти два идентификатора принадлежат исключительно макросу. Параметры разделяются запятыми. В макросе Swap16 имена параметров используются в командах push и pop, сначала сохраняя слова v1 и v2 в стеке, а затем немедленно выталкивая из стека эти же два параметра в противоположном порядке. В результате две переменные обмениваются в памяти своими значениями. Чтобы использовать макрос с параметрами, наберите после его имени действительные значения для обработки. Например, если вы объявили две переменных типа слова countA и countB, можно использовать для перестановки их значений предыдущий макрос так:

```
DATASEG
countA        dw       100
countB        dw       200
CODESEG
Swap16        countA, countB
```

CountA и countB называются *действительными параметрами*, потому что они предоставляют действительные значения для вычислений. При обработке макроса Turbo Assembler заменяет *формальные параметры* v1 и v2 действительными параметрами countA и countB, ассемблируя макрос Swap16 так, как если бы вы написали:

```
push [word countA]
push [word countB]
pop  [word countA]
pop  [word countB]
```

Вы также можете передавать имена регистров в Swap16, содержащие указатели на переставляемые данные. Если бы в bx содержался адрес countB, то вы могли бы написать:

```
Swap16              bx, countA        ; переставить значения слов [bx] и [countA]
```

Вместо запятых можно также разделять параметры с помощью пробелов. Например, следующая команда тождественна предыдущей:

```
Swap16        bx countA          ; разделение параметров запятыми или пробелами
```

Поскольку формальные параметры можно заменить символами, такими как bx, или метками, такими как countA (которые представляют адреса смещений значений), в этом примере v1 и v2 одновременно являются как числовыми, так и символьными параметрами. Действительный тип зависит от того, как используются параметры.

Как показывают эти примеры, параметры позволяют создавать макросы общего назначения, которые можно перепрограммировать для удовлетворения новым требованиям. Узнав, как объявлять и использовать параметры, вы можете написать эффективные макросы, которые делают больше, чем простое повторение общих последовательностей команд. В следующих разделах детально исследуются все типы параметров макросов.

Замечание

Имена параметров, такие как v1 и v2 в предыдущих примерах, являются локальными для макросов. Можно спокойно использовать эти же имена в качестве меток в других частях программы.

Символьные параметры

Символьные формальные параметры могут заменяться любыми символами, мнемоническими названиями, директивами и ключевыми словами, используемыми в ассемблере. Одно из применений символьных параметров состоит в определении новых имен для команд. Например, можно дать процессору урок орфографии, заменив Move на mov:

```
MACRO         Move     aSym,bSym
     mov ,    aSym,bSym
ENDM Move
```

В этом примере aSym и bSym являются символьными формальными параметрами, которые при использовании макроса заменяются именами регистров или другим текстом:

```
Move ax,bx              ; Заменяется на mov ax bx
Move [value],cx         ; Заменяется на mov [value],cx
mov  cx,dx              ; Ассемблируется обычным образом
```

Теперь использование mov и Move приводит к тождественными результатам. (Этот макрос может быть полезным при наборе программы, особенно если вы, как и я, постоянно ошибочно печатаете *move* с буквой *e* вместо *ov*.) Можно также создать символьные параметры, которые при обработке макросов становятся новыми метками. Чтобы показать, как это работает, ниже приведен макрос, резервирующий пространство под переменную типа слова, имя которой передается в макрос через параметр:

```
MACRO         DeclareWord      vName, vValue
vName         dw       vValue
ENDM          DeclareWord
```

Макрос DeclareWord преобразуется в директиву dw, выделяя одно слово памяти с меткой vName и инициализированное значением vValue. Чтобы создать переменную типа слова с начальным значением 100, можно написать:

```
DeclareWord  TheCount  100
```

Конечно, в этом случае проще непосредственно использовать директиву dw. Аналогичный, но более практический пример показывает, как писать макросы, которые автоматически помечают переменные в соответствии с их начальными значениями. Для этого необходимо использовать *оператор подстановки* &, который информирует Turbo Assembler о том, что текст после & является именем формального параметра и ничем другим. Это легче увидеть в следующем примере:

```
MACRO       Aword   vNum
Word&vNum   dw      vNum
ENDM Aword
```

Макрос Aword объявляет один формальный параметр vNum. В директиве dw метка Word&vNum показывает Turbo Assembler, что vNum ссылается на формальный параметр этого имени. Без & ассемблер не смог бы понять, что vNum в WordvNum является формальным параметром. При использовании макроса Aword автоматически создаются имена переменных типа слова:

```
Aword1
Aword2
Aword3
```

В результате создаются три 16-битовые переменные с именами Word1, Word2 и Word3 так, как если бы вы написали.

```
Word1dw     1
Word2dw     2
Word3dw     3
```

Заметьте, при внесении одного изменения с помощью этого макроса модифицируется как значение, так и соответствующая ему метка. Например, при замене 3 на 8 будет создана переменная типа слова Word8, с начальным значением 8. Чтобы сделать это без макроса, нужно было бы изменять две цифры.

Числовые параметры

Как вы могли увидеть из некоторых предыдущих примеров, символьные параметры иногда трактовались как числа Например, vNum в макросе Aword является символом, если используется как часть метки, и числом, когда используется для инициализации переменной типа слова Соответствующий тип данных определяется в контексте применения параметра. Параметр является числовым только тогда, когда этого требует следующая за ним команда Рассмотрим другой макрос, который использует как символьный, так и числовой параметры:

```
MACROShiftLeft      destination, count
    push    cx
    mov     cl  count
    shl     destination, cl
    pop     cx
ENDM ShiftLeft
```

Макрос ShifLeft определяет два параметра — назначение, представляющее сдвигаемый регистр или переменную в памяти, и счетчик, определяющий количество сдвигов влево в целевом значении. Можно также написать похожие макросы для других команд сдвигов и ротаций. Команды в макросе сохраняют cx в стеке, помещают числовой параметр count в cl, сдвигают destination влево заданное количество раз и затем восстанавливают cx. Чтобы использовать макрос, напишите следующие команды.

```
ShiftLeft           ax, 5
ShiftLeft           [value] 3
ShiftLeft           <[word bx]> 2
```

Первая строка пять раз сдвигает значение ax влево. Во второй строке переменная value (не показанная здесь) сдвигается три раза влево. В третьей строке иллюстрируется ситуация с параметрами, которые имеют пробелы. Если вы попытаетесь написать:

```
ShiftLeft           [word bx], 2
```

то получите ошибку, указывающую, что типы операндов не соответствуют макросу. Это получается из-за того, что пробелы или запятые используются для разделения нескольких действительных параметров. (Формальные параметры в макросах должны разделяться запятыми.) Чтобы решить эту дилемму, используйте скобки < и >, взяв в них параметры, содержащие пробелы:

```
ShiftLeft           <[byte si]>, 4
ShiftLeft           <[word bx+di]>, 2
```

Передавая через числовые параметры макросов выражения, вы должны решить, в каком месте они будут вычисляться. Обычно параметры передаются в макросы в форме текста, например в виде MySize*MyCount, вычисляемо-

то внутри макроса. Чтобы выражение вычислялось до обработки макроса, начните его со знака процента %, "Оператора вычисления выражения". Например:

```
ShiftLeft           ax, MySize*MyCount
```

В этом месте проявляется особенно неприятный аспект использования числовых параметров макросов. К сожалению, Turbo Assembler игнорирует часть выражения *MyCount, предполагая, что эти символы являются просто дополнительными параметрами. Чтобы передать выражение в макрос, нужно использовать угловые скобки, как объяснялось выше:

```
ShiftLeft           ax, < MySize*MyCount>
```

Однако это решение не является совершенным, потому что выражения MySize*MyCount передается в макро в качестве текста. Чтобы вычислить и передать *результат* выражения, начинайте его со знака процента, как показано ниже:

```
ShiftLeft           ax, %MySize*MyCount
```

Знак процента заставляет ассемблер вычислять выражение и передавать в макрос окончательный результат этого вычисления. Это может быть полезным, если формальный параметр макроса (count в этом примере) используется более одного раза. Если вы передаете выражение как текст, то ассемблер должен при каждом использовании вычислять его значение. Если же вы передадите результат выражения, то вычисление будет происходить только один раз.

Строковые параметры

Как и в предыдущих примерах, вы должны заключать строковые параметры в скобки < и >, которые показывают ассемблеру, что текст в скобках является литералом (в том числе пробелы и знаки препинания, обычно используемые для разделения отдельных идентификаторов). Следующий полезный макрос использует этот подход, чтобы описать строковую переменную в ASCIIZ-формате:

```
MACROASCIIZ  name, chars
name         db      '&shars',0       ;,Строка + нулевой ограничитель
name&len     dw      $-name-1         ;; Длина строки
ENDM ASCIIZ
```

Макрос ASCIIZ определяет два параметра — name, который используется для создания двух меток, и chars, определяющий символы, из которых создается строка. Внутри макроса директива db создает ограниченную нулем строку, используя name в качестве метки. Оператор & показывает Turbo Assembler, что chars является именем параметра. Это необходимо, чтобы предотвратить создание строки из пяти символов — 'c', 'h', 'a', 'r ','s', что могло получиться, если бы в макросе не использовались кавычки. Директива dw записывает после строки ее длину (минус нулевой ограничитель) как переменную типа слова. Хотя эта величина не является частью стандартного ASCIIZ-формата, она обеспечивает программам быстрый путь определения длины строки — по крайней мере для строковых констант, которые не изменяют своей длины. Обратите внимание на то, как другой оператор & создает метку, начинающуюся с name и заканчивающуюся на "len". Например, если name — это MyString, то слово с ее длиной будет иметь метку MyStringlen. Чтобы использовать макрос ASCIIZ, заключите символы строки в угловые скобки:

```
ASCIIZ              s1, <Any old string will do>
ASCIIZ              s2, <Commas, and periods, work too >
```

Когда Turbo Assembler обрабатывает эти строки, создается две строки — одна с меткой s1, другая с s2. К тому же ассемблер хранит длины строк в s1Len и s2Len. В результате для вычисления длины строки можно использовать или процедуру StrLength из модуля STRINGS, или загружать длины непосредственно, как показано в следующих примерах:

```
mov  di, offset s1      ; Адрес строки s1 в di
call StrLength          ; Вычислить cx = длина строки
mov  cx, [s1len]        ; То же, что и в предыдущих двух командах
```

Если длина строки изменяется, можно вызвать Strlength и затем сохранить результат в s1len. Предполагая, что в cx содержится новая длина строки, можно написать:

```
mov  [s1len], cx ; Сохранить новую длину строки
```

Замечание

Для исследования меток и значений, созданных макросом ASCIIZ, используйте команду Turbo Debugger View:Variables.

Чтобы создавать строки, содержащие специальные для макросов символы, используйте восклицательный знак (!), "Оператор цитирования". Например, чтобы включить угловую скобку в качестве символа , можно использовать строку

```
ASCIIZ              s3, <Couldn''t locate--!>>
```

В результате создается переменная s3, равная строке "Couldn't locate-->", которую вы, возможно, будете использовать совместно со второй строкой, содержащей имя файла, который не может быть найден на диске.

Оператор цитирования вставляет в строку угловую скобку (>) в качестве символа. Обратите также внимание на двойные апострофы, необходимые здесь, чтобы вставить одиночный апостроф, так как макрос ASCIIZ использует этот же символ в качестве разделителя строк.

Макросы и переменные

Макросы удобно использовать для добавления в язык ассемблера новых типов данных, подобных макросу ASCIIZ. Как описывалось в предыдущих главах, в макросах может быть использовано любое сочетание директив, таких как dw и db. Вместе с оператором DUP это упрощает написание макросов для создания массивов:

```
MACRO       WordArray aName, aSize, aValue
aName&count           dw      aSize
aName                 dw      aSize DUP (aValue)
ENDM        WordArray
```

WordArray имеет три параметра: идентификатор метки (aName), количество слов в массиве (aSize) и начальные значения, назначенные каждому слову (aValue). В теле макроса первая директива dw создает переменную, равную количеству слов в массиве, помечая ее именем массива плюс "count". Вторая директива dw определяет значения элементов массива, используя оператор DUP для резервирования пространства под величины aSize значений, инициализируемых aValue. Два следующих примера показывают, как можно использовать этот макрос:

```
WordArray     a1, 10, 0
WordArray     a2, 100, ?
```

При выполнении этих макрокоманд создаются два массива: первый — с меткой a1 с десятью словами со значением 0 и второй — с меткой a2 и со 100 неинициализированными словами. При этом также создаются две переменные — a1count и a2count, значения которых равны количествам слов в каждом массиве. Эти переменные могут использоваться в программах для определения количества элементов, содержащихся в массивах:

```
mov  cx, [a1count]    , Установить cx = количество слов в массиве a1
mov  cx, [a2count]    ; Установить cx = количество слов в массиве a2
```

Макросы, повторяющие команды

Для создания макросов, которые повторяют команды, обычно каждый раз с разными параметрами, могут быть использованы три директивы — IRP. IRPC и REPT. Чтобы создать мощные новые команды, эти директивы можно использовать как отдельно, так и внутри макросов. Как и обычные макросы, повторяющие макросы завершаются с помощью ENDM. Ранее вы научились использовать директиву REPT для создания автоматически инициализируемых массивов. (Например, см. строки 36–42 в программе 7.2, SCREEN.ASM.) IRP работает аналогично, но берет аргументы, находящиеся в угловых скобках и разделенные запятыми:

```
IRP  register, <ax,bx,cx,dx>
     inc     register
ENDM
```

При ассемблировании этого макроса создается четыре команды inc, по одной для каждого из четырех регистров, находящихся в скобках:

```
inc  ax
inc  bx
inc  cx
inc  dx
```

Директива IRPC работает аналогично IRP, но вместо использования аргументов в скобках она повторяет команды для каждой буквы в строке. (Буква C в IRPC означает Character.) Как показывает следующий пример, можно использовать IRPC, чтобы создавать строки, каждый символ которых хранится в слове, а не в байте, как это обычно делается директивой db:

```
LABELchars WORD
IRPC nextChar, ABCDEFG
    dw ' &nextChar'
ENDM
```

Директива LABEL в этом случае является необходимой, так как ассемблер не позволяет помещать метку прямо в той же строке, что и конструкцию IRPC. Формальный параметр nextChar последовательно берет символы из строки ABCDEFG, которая не требует для себя кавычек. При каждом повторении директива dw создает двухсимвольную переменную, состоящую из пробела и ASCII значения nextChar на этой итерации. Обратите внимание, как & отождествляет nextChar с именем параметра. В результате получается то же самое, как если бы вы написали:

```
charsdw     ' A'
    dw      ' B'
    dw      ' C'
    dw      ' D'
    dw      ' E'
```

Замечание

Чтобы увидеть эти символы в Turbo Debugger, используйте команду View.Variables, нажмите <TAB>, поместите курсор на chars, после чего вызовите окно View:Dump.

Можно также использовать IRP, IRPC, REPT и внутри макроса, что позволит вам давать повторяющимся конструкциям имена. Типичный пример использования IRP состоит в проталкивании в стек регистров в начале процедуры:

```
MACRO       PushReg registers
 IRP reg, <registers>
    push reg
 ENDM
ENDM
```

Обратите внимание на то, что в данном случае требуется две директивы ENDM — одна для окончания IRP, другая — для окончания макроса. Соответственно следующий макрос, скорее всего в конце процедуры, выталкивает регистры из стека:

```
MACRO       PopReg registers
 IRP reg, <registers>
    pop reg
 ENDM
ENDM
```

В каждом макросе через формальный параметр с именем registers список регистров передается в IRP. В параметр reg цикла IRP помещаются последовательные значения из этого списка, и пока список не пуст, для каждого значения reg ассемблируется одна команда — push или pop. Вместе PushReg и PopReg упрощают составление программ, избавляя от необходимости писать последовательности команд:

```
push ax
push bx
push cx
push dx
```

Вместо этого, чтобы сохранить в стеке эти же четыре регистра, можно просто написать:

```
pushreg     <ax,bx,cx,dx>
```

Четыре регистра внутри угловых скобок преобразуются в четыре команды push, по одной для каждого регистра. В конце процедуры можно использовать PopReg, чтобы восстановить эти регистры в обратном порядке. С этими двумя макросами можно писать ваши процедуры в следующей общей форме:

```
PROC Subroutine
     PushReg <ax, bx, cx, dx, si, di>
     :
     :-----Команды подпрограммы
     :
     PopReg <di, si, dx, cx, bx, ax>
     ret
ENDP Subroutine
```

Замечание

Не обвиняйте меня в непрактичности, просто стараясь не применять технические приемы до их введения и учитывая, что макросы необязательны, в программах из этой книги я не использую, как предложено здесь, макросы в процедурах для сохранения и восстановления регистров. Вы, конечно, можете видоизменять программы, используя PushReg и PopReg, которые упрощают набор и способствуют устранению ошибок, помещая списки сохраняемых и восстанавливаемых регистров в легко сравниваемые одиночные строки.

Макросы и код

Как уже упоминалось, макросы дают возможность изобретать новые команды, расширяющие возможности отдельных команд языка ассемблера. Используемые таким образом, макросы являются типом подпрограмм, которые непосредственно вставляются в строку с другими командами, не требуя для своей активизации команды call. Фактически один из путей оптимизации программ по скорости состоит в замене вызовов подпрограмм макросами, выполняющими соответствующие действия. Это может повысить производительность программы благодаря устранению команд call и ret. Например, предположим, что у вас есть процедура:

```
PROC DecReg
     dec     ax
     dec     bx
     dec     cx
     dec     dx
     ret
ENDP DecReg
```

После отладки программы вы можете захотеть *развернуть* команды подпрограммы, т.е. вставить их непосредственно туда, где они требуются. Проще всего для этого создать макрос:

```
MACRO        DecReg registers
 IRP reg, <registers>
     dec reg
 ENDM
ENDM
```

Конечно, существуют более простые способы для написания этого макроса, но, собираясь вводить его в код, можно сделать макрос максимально универсальным. После создания DecReg можно затем использовать глобальный поиск и замену в вашем текстовом редакторе (или утилитную программу), чтобы перевести все команды call DecReg в следующую команду:

```
DecReg       <ax, bx  cx, dx>
```

Если DecReg вызывается в программе часто, возможно из основного цикла, то развернутый код выполняется быстрее благодаря устранению необходимости выполнять многочисленные команды call и ret. К тому же DecReg даже больше полезен в качестве макроса, чем подпрограммы, потому что, в отличие от нее, позволяет оперировать любой комбинацией регистров.

Замечание

Макросы также могут быть вложенными, т.е. вы можете использовать имя одного макроса внутри другого. Такие макросы очень мощные, но они разворачиваются в большое количество строк кода.

Сохранение регистров

Потенциальная опасность скрыта в тех случаях, когда макросы изменяют значения одного или более регистров. Поскольку имена регистров в исходном коде не встречаются, вы легко можете пропустить этот факт и полагать, что в регистре неизменно содержится какое-либо важное значение. Некоторые программисты создают макросы, сохраняющие все регистры, с помощью команд push и pop:

```
MACRODispChar ch
    push    ax              ;, Сохранить ax
    push    dx              ,, Сохранить dx
    mov     ah,2            ,, Загрузить номер функции в ah
    mov     dl, '&ch'       ,, Загрузить выводимый символ
    int     21h             ,, Вызов DOS-вывести символ ch
    pop     dx              ,, Восстановить сохраненный dx
    pop     ax              ,, Восстановить сохраненный ax
ENDM DispChar
```

Макрос DispCar определяет единственный параметр ch, который помещается в регистр dl, снова с использованием оператора &, указывающего ассемблеру, что ch — это имя параметра, а не две буквы c и h, взятые в кавычки. Затем номер стандартной функции вывода DOS (2) записывается в ah, после чего int 21h вызывает DOS для записи символа из dl в стандартный файл вывода Две пары команд push и pop сохраняют и восстанавливают используемые в макросе значения регистров. В программе, для вывода символа, можно использовать этот макрос следующим образом:

```
DispChar <Q>
```

Если макрос DispChar не сохраняет используемые регистры, можно легко забыть, что при его вызове изменяются значения в ah и dl. Конечно, это имеет и обратную сторону — при каждом вызове DispChar снова и снова восстанавливаются и сохраняются одни и те же регистры, причем даже тогда, когда в этом нет необходимости. Нельзя с уверенностью сказать, должны вы или нет сохранять регистры в своих макросах — выбор принадлежит вам Если вы не сохраняете их, тогда аккуратно опишите все регистры, используемые вашим макросом, и приведите все в порядок, иначе вам предстоит долгое общение с Turbo Debugger, чтобы определить, почему ваши программы не работают.

Использование директивы включения

Несмотря на то что можно объявить собственные макросы в начале вашей программы, лучше размещать макросы в отдельном текстовом файле и затем загружать этот файл во время ассемблирования. Чтобы сделать это, вставьте директиву INCLUDE следующим образом:

```
INCLUDE "MACROS ASM" , Прочитать библиотеку макросов
```

Замечание

В режиме Ideal при ассемблировании директив INCLUDE вы обязаны брать имена файлов в кавычки. В режиме MASM кавычки не требуются, но при этом вы не можете заканчивать строку, как показано здесь, так как ассемблер будет рассматривать комментарий как часть имени файла.

Можно также использовать INCLUDE для включения файлов, содержащих другой текст на языке ассемблера, а не только для загрузки макросов. Текст включаемого файла вставляется в программу и ассемблируется так, как если бы два файла были одним. Обычно программисты хранят макросы в отдельных файлах, включая их по мере необходимости в один или больше модулей. Директива INCLUDE может появляться в любом месте в тексте программы и использоваться для загрузки макросов, переменных и команд языка ассемблера. Вы также можете вкладывать директивы INCLUDE одну в другую, при этом включаемый файл может содержать текст, включающий в него другие файлы, и т.д.

На практике, вероятно, лучше не использовать INCLUDE для вставки переменных и команд в программы. Лучше писать для них отдельные объектные модули и затем компоновать полученный код с программой, используя методы, описанные в этой книге, например для модулей SCREEN и STRINGS. Помните, что включаемый текст снова и снова ассемблируется вместе с другими командами программы, тогда как отдельные объектные модули сразу готовы для компоновки.

Локальные метки

Чтобы создавать автоматически нумерующиеся локальные метки, используйте внутри макроса директиву LOCAL. При этом ассемблер будет автоматически создавать подходящие метки, избавляя вас от ненужной работы по придумыванию уникальных меток для циклов и переходов.

Вставлять директиву LOCAL необходимо после строки, открывающей макрос:

```
MACRO       AnyMacro
LOCAL       @@yonder,@@ponder


ENDM        AnyMacro
```

Turbo Assembler заменяет метки youder и pouder числовыми локальными метками, такими как @@0001, @@0002, @@0003. Символы youder и pouder используются только при написании макроса и не появляются в метках, которые создает ассемблер.

Ниже приводится практический пример использования LOCAL. Следующий макрос применяет директиву LOCAL для создания цикла:

```
MACRO CallOn register, subroutine
LOCAL @@restart, @@exit
 push register                ,, сохранить регистр
 or   register register       ,, регистр равен 0?
 jz   @@exit                  ,, если да, то выход, иначе продолжить
@@restart
 call         subroutine      ,, вызов подпрограммы
 dec  register                ,, вычесть из регистра 1
 jnz  @@ restart              ,, перейти, если регистр отличен от 0
@@exit
 pop  register                ,, восстановить регистр
ENDM CallOn
```

Макрос CallOn вызывает подпрограмму определенное в аргументе register количество раз. Директива LOCAL создает две локальные метки: @@restart и @@exit. Команды в макросе сохраняют и восстанавливают заданный регистр, вызывают подпрограмму, уменьшают значение в регистре и выполняют переход на локальные метки в зависимости от значения регистра.

Для использования макроса напишите вызываемую подпрограмму. Простейший пример приведен ниже:

```
PROC AnyProc
     ret
ENDP AnyProc
```

Затем инициализируйте регистр-счетчик и вызовите подпрограмму с помощью следующих команд:

```
mov dx  4              , Поместить счетчик в dx
CallOn dx, AnyProc     , Вызвать AnyProc четыре раза
```

Ассемблер преобразует эту макрокоманду CallOn в следующий код:

```
     push    dx
     or      dx,dx
     jz      @@0001
@@0000
     call    AnyProc
     dec     dx
     jns     @@0000
@@0001
     pop     dx
```

Ассемблер заменяет две локальные метки @@restart и @@exit числовыми метками @@0000 и @@0001. Наиболее важно то, что если вы *снова* используете тот же самый макрос в пределах действия этих меток, ассемблер создает две *новые* метки — @@0002 и @@0003. Таким образом, можно многократно использовать макрос CallOn и это не приведет к введению в программу конфликтующих меток.

Условная компиляция

Директивы условной компиляции образуют мини-язык, встроенный в Turbo Assembler. Используя условные директивы, можно изменить ход ассемблирования программы, основываясь на различных условиях, обычно определенных в начале программного модуля (или записанных в отдельном INCLUDE файле) и относящихся к идентификаторам, называемым *условными символами*. Например, можно определить условный символ с именем DisplayType, показывающий имеющийся у компьютера тип адаптера дисплея. Чтобы модифицировать программу для нового аппаратного видеообеспечения, вы просто изменяете DisplayType на правильное значение и снова ассемблируете программу. Некоторые компании, разрабатывающие программное обеспечение, вставляют сотни таких символов в свои программы, предоставляя возможность программистам быстро создавать необходимые покупателям приложения простой заменой нескольких символов в том или ином месте.

В табл. 8.1 приведен список условных директив компиляции Turbo Assembler, ни одна из которых непосредственно не создает какого-либо машинного кода. Директивы, зависящие от проходов, такие как ERRIF1 и

ERRIF2, включены для совместимости с MASM, который обрабатывает программы на языке ассемблера в два прохода Поскольку Turbo Assembler является однопроходным ассемблером, то эти директивы в нем не используются

Определение условных символов

Условные символы определяются подобно другим числовым именованным идентификаторам Например, чтобы определить условный символ с именем DisplayType, можно написать

```
DisplayType  =       1
```

Для определения условных символов можно также использовать EQU, но обычно следует применять знак равенства, который создает числовой символ Так как "1" в этом примере имеет не слишком большой смысл, можно определять другие значения для ваших условных символов Например, вы можете установить четыре символа, представляющих различные типы дисплея

```
CGAAdapter   =       0
MonoAdapter  =       1
EGAAdapter   =       2
VGAAdapter   =       3
```

Конкретные значения в этом примере роли не играют — они только делают программу более читаемой, позволяя создавать абсолютно ясные присваивания типа

```
DisplayType  =       EGAAdapter
```

Таблица 8 1 **Директивы условной компиляции**

Директива	Значение
ELSE	Ассемблировать следующие строки, если предыдущии IF ложныи
ELSEIF	Конец директивы ELSE Начинает новыи IF
ENDIF	Конец директивы IF
ERR	Заставляет ассемблер выдавать сообщение об ошибке
ERRIF	Ошибка, если выражение истинно
ERRIF1	Ошибка на проходе 1*
ERRIF2	Ошибка на проходе 2*
ERRIFB	Ошибка, если аргумент пропущен (отсутствует)
ERRIFDEF	Ошибка, если аргумент определен
ERRIFDIF	Ошибка, если аргументы различаются
ERRIFDIFI	Ошибка, если аргументы различаются (различия прописных и строчных букв игнорируются)
ERRIFE	Ошибка, если выражение ложное (равно нулю)
ERRIFIDN	Ошибка, если аргументы тождественны
ERRIFIDNI	Ошибка, если аргументы тождественны (различия прописных и строчных букв игнорируются)
ERRIFNB	Ошибка, если аргумент не пропущен
ERRIFNDEF	Ошибка, если аргумент не определен
EXITM	Немедленная остановка развертки макроса
GOTO target	Продолжать макроразворачивание по заданнои метке
IF	Ассемблировать, если выражение истинно
IF1	Ассемблировать на проходе 1*
IF2	Ассемблировать на проходе 2*
IFB	Ассемблировать, если аргумент пропущен
IFDEF	Ассемблировать, если символ определен
IFDIF	Ассемблировать, если аргументы различаются
IFDIFI	Ассемблировать, если аргументы различаются (различия прописных и строчных букв игнорируются)

Окончание таблицы 8.1

Директива	Значение
IFE	Ассемблировать, если выражение ложное (равно 0)
IFIDN	Ассемблировать, если аргументы тождественны
IFIDNI	Ассемблировать, если аргументы тождественны (различия прописных и строчных букв игнорируются)
IFNB	Ассемблировать, если аргумент не пропущен
IFNDEF	Ассемблировать, если аргумент не определен

*Примечание: Условные директивы 1 и 2 проходов включены для совместимости с MASM и не используются в Turbo Assembler.

С этой точки программы символ DisplayType считается *определенным* независимо от значения, которое он имеет. Символ определен, как только ему присвоено какое-либо значение. Символ *не определен*, если ему не присвоено никакого значения. Нужно понимать разницу между значением символа и фактом его определенности или неопределенности. Следующие положения разъясняют это различие.

- Символ определен, как только вы приравниваете ему любое значение. Конкретное значение при этом роли не играет.
- Символ не определен, если ему не присвоено никакого значения.
- Проверка, определен ли символ, — не то же самое, что проверка, имеет ли символ заданное значение.
- Для определения условных символов, которые должны быть числовыми, лучше использовать знак равенства. Это также позволит вам позднее, по мере необходимости, переопределять одни и те же символы.

Создавая условные символы, помните, что имена символов представляют простые значения. Это важно, так как условные директивы, в частности IF и IFE, работают только с выражениями и аргументами, которые приводятся к целым значениям.

Использование условных символов

Самое характерное использование условных символов состоит в выборе того, какая из двух или более частей кода будет в действительности ассемблироваться. Например, предположим, что вам необходимы две версии программы — одна для отладки, а другая для окончательной рабочей версии. Отладочная версия может включать специальные команды для показа использования стека, дампа важных переменных на принтер и т.д. Естественно, вы не хотите включать такие компоненты в конечную рабочую версию. Директивы условной компиляции позволяют легко ассемблировать каждую версию, просто определяя в начале программного модуля несколько условных символов:

```
False       = 0       ; Значение "ложь"
True = 1              ; Значение "истина"
Debugging   = True    ; Ложно для конечной версии
```

Теперь у вас есть способ указать ассемблеру на то, какая версия будет создаваться, в зависимости от установки Debugging. Это дает возможность изменять переменные, вставлять код, вызывать отладочные процедуры, модифицировать другие особенности программы, и все это с помощью установки Debugging в True или False. В сегменте данных можно проверить Debugging в условной директиве, чтобы соответствующим образом изменить строку, описывающую программу:

```
DATASEG
IF Debugging
programID db 'Chess v1.0 (TEST MODEL)',0
ELSE
programID db 'Chess v1.0',0
ENDIF
```

Когда Turbo Assembler обрабатывает эту директиву, если Debugging равно True (равно любому ненулевому значению), то ассемблируется строка TEST MODEL; в противном случае ассемблируется строка, характеризующая конечную версию. В окончательный код всегда включается только одна строка, даже в том случае, если в тексте программы многократно повторяется строка Chess v1.0. Если Debugging — False, то первая директива db *полностью пропускается во время ассемблирования*. Помните, что условные директивы являются командами Turbo Assembler — IF, ELSE и ENDIF не генерируют никакого кода и не являются командами, выполняемыми во время исполнения программы.%%

Другая проверка Debugging может выполняться позже в сегменте кода программы. Например, предположим, что программа должна вызывать специальную процедуру инициализации значений, необходимых только во время отладки. Это можно осуществить следующим образом:

```
IF Debugging EQ TRUE
 call DebugInit          , Инициализация для отладки
ENDIF
```

Если Debugging равно True, то ассемблируется команда вызова DebugInit. В противном случае ассемблер полностью пропускает этот вызов

В другой части программы только в том случае, если Debugging равно True, может вставляться процедура отладки

```
IF Debugging
PROC DebugInit

 --------- Подпрограмма инициализации для отладки

 ret                     , возврат в вызывающую программу
ENDP DebugInit
ENDIF
```

Вызов процедуры и сама подпрограмма добавляются к конечному коду только в случае True. Если Debugging равно False, программа ассемблируется так, как будто этих строк вообще не существует.

Вы могли заметить в этих примерах, что одна директива IF использует выражение IF Debugging EQ TRUE, тогда как другая просто имеет вид IF Debugging Обе формы правильные и приводят к одинаковому результату до тех пор, пока вы придерживаетесь соглашения, что любое ненулевое значение (обычно 1 или –1) представляет True и 0 представляет False. Оператор EQ в первой условной директиве является одним из нескольких приведенных в табл. 8 2 операторов, которые можно использовать в аналогичных условных выражениях.

Таблица 8 2 **Операторы константных выражений**

Оператор	Значение
AND	Логическое И
EQ	Равно
GE	Больше или равно
GT	Больше чем
LE	Меньше или равно
LT	Меньше чем
MOD	Деление по модулю (деление нацело)
NE	Не равно
NOT	Логическое отрицание (переключение битов)
OR	Логическое ИЛИ
SHL	Сдвиг влево
SHR	Сдвиг вправо
XOR	Логическое исключающее ИЛИ

Директивы IF должны заканчиваться ENDIF, отмечающими конец блока условий. Между ними можно вставлять необязательные ELSE, которые в случае, если выражение ложно, приводят к ассемблированию альтернативных команд Это позволяет использовать IF так:

```
IF Debugging
, код для debugging = TRUE
ENDIF
```

Или с ELSE для выбора альтернативных команд:

```
IF Debugging
, код для debugging = TRUE
ELSE
, код для debugging = FALSE
ENDIF
```

Определять или не определять

Вместо использования IF для проверки истинности выражения (не равно 0) или IFE для проверки ложности (равно 0) можно использовать директивы IFDEF и IFNDEF, проверяющие, определен ли символ. Как вы помните, символ определен, как только он получает какое-либо значение. Если вы напишите в программе

```
IFDEF Debugging
call DebugInit
ENDIF
```

вызов подпрограммы будет ассемблироваться только в том случае, если Debugging приписано какое-нибудь значение. Чтобы определить Debugging, можно поместить в начале программы строку

```
Debugging = 1 ; Определение Debugging
```

Для отмены определения символа просто удалите эту строку или вставьте слева от нее точку с запятой, превратив строку в комментарий. Можно также проверить неопределенность символа с помощью команд:

```
IFNDEF Debugging
 welcome      db       'Production version 5 01',0
ENDIF
```

IFDIF и IFNDEF наиболее полезны при использовании вместе с опцией /d Turbo Assembler, которую вы можете применять для определения символов в командной строке DOS. Для ассемблирования программы с именем Banana при отладке можно использовать команду

```
tasm /dDebugging=1 Banana
```

/dDebugging=1 определяет при ассемблировании программы символ Debugging — поэтому нет надобности добавлять в первоначальный текст программы строку с определением Debugging. Заметьте, что между /d и именем символа нет пробела. Позже, когда отладка больше не требуется, обычное ассемблирование отменяет определение debugging, устраняя из окончательной версии проверочный код:

```
tasm Banana
```

Обработка условных ошибок

Можно создавать многочисленные условия с помощью IF, ELSE и ELSEIF, заканчивая их с помощью ENDIF. Например, чтобы определить строку в соответствии с указанными раньше типами дисплея, вы можете написать:

```
DATASEG
IF DisplayType EQ      CGAAdapter
displayID     db       'CGA Adapter',0
ELSEIF DisplayType     EQ MonoAdapter
displayID     db       'Monochrome Adapter',0
ELSEIF DisplayType     EQ EGAAdapter
displayID     db       'EGA Adapter',0
ELSEIF DisplayType     EQ VGAAdapter
displayID     db       'VGA Adapter',0
ENDIF
```

В зависимости от установки DisplayType в окончательном коде определяется только одна строка. Этот пример не является полным, так как он не допускает возможности, что для DisplayType может быть определено неизвестное значение. Для обработки этого условия можно заменить ENDIF директивами:

```
ELSE
displayID db 'Unknown adapter type',0
ENDIF
```

Чтобы предотвратить ассемблирование при неизвестном условии, можно также вывести ошибку, заменив первоначальный ENDIF следующими инструкциями:

```
ELSE
ERR
DISPLAY " **Error** Unknown Display Type value"
ENDIF
```

При ассемблировании, если DisplayType неизвестен, директива ERR заставляет Turbo Assembler выдавать сгенерированное пользователем сообщение об ошибке. Директива DISPLAY также показывает ошибочную строку, указывая в этом примере, что произошла ошибка. При ассемблировании программы на экране появляется следующий текст:

```
Assembling file: TEST ASM
**ERROR** Unknown DisplayType value
**ERROR** TEST.ASM(102) User generated error
Error messages  1
Warning messages: None
Remaining memory: 331k
```

Окончание развертывания макроса

Директива EXITM используется для немедленного окончания развертывания макроса. Она часто бывает полезной для создания отладочных макросов, которые вы хотите удалить из окончательной ассемблированной версии программы, но при этом оставив их в исходном тексте на языке ассемблера. Например, следующий макрос сохраняет четыре регистра и приостанавливает программу с помощью бесконечно повторяющейся команды перехода:

```
MACRO PauseMac
LOCAL @@here
     IFNDEF DEBUGGING
     EXITM
     ENDIF
     push    ax
     push    bx
     push    cx
     push    dx
@@here  jmp   @@here ;; Пауза
ENDM PauseMac
```

Вставьте макрос PauseMac для сохранения ax, bx, cx, dx и выполнения последующего бесконечного цикла по метке @@here:. Это приводит к остановке программы, поэтому вам следует выполнять этот код *только* под управлением отладчика, который позволяет выйти из непрерывного цикла через нажатие клавиш (например <Ctrl+Break>).

Макрос использует условную директиву IFNDEF для проверки на определенность символа DEBUGGING. Если он не определен, EXITM немедленно прекращает выполнение макроса и, следовательно, получается тот же результат, что и при полном удалении команд макроса из программы. Если DEBUGGING определен, EXITM пропускается и вставляются команды push и jmp. Определяйте DEBUGGING с помощью следующей строки либо превращайте его в комментарий или удаляйте из программы, чтобы отменить его определение:

```
DEBUGGING EQU 1
```

Директива GOTO

В руководстве по Turbo Assembler содержится информация по директиве GOTO, которая переводит макроразворачивание в другое место. Я считаю, что эта директива имеет сомнительную ценность, однако можно использовать ее следующим образом:

```
MACRO AnyMac
........
GOTO location
........

location:

........

ENDM AnyMac
```

Встретив GOTO, ассемблер продолжает разворачивать макрос по заданной целевой метке — location в этом примере.

В соответствии с руководством пользователя по Turbo Assembler вы не должны использовать GOTO внутри условной директивы для изменения разворачивания макроса:

```
MACRO AnyMac
IFDEF DEBUGGING
 GOTO location
ENDIF
location:
 .........
ENDM AnyMac
```

Этот пример работать не будет, так как он приводит к пропуску директивы ENDIF. Следовательно, ассемблер заканчивает работу по ошибке "Открытое условие", и программа не ассемблируется.

Но не волнуйтесь! Этот пример не показателен, так как другие условные директивы, в частности IFDEF, IFNDEF, ELSEIF и EXITM, предоставляют все необходимые средства для управления макроразворачиванием. Честно говоря, я считаю непрактичным применение директивы GOTO, но если вы не согласны со мной, пожалуйста, — можете использовать ее.

Тем временем обратно в макрос

Еще одна новая директива, WHILE, дает возможность расширять макрос заданное количество раз. Рассмотрим простой макрос, который сохраняет в стеке регистр-аккумулятор ax:

```
MACRO PushAX
 push ax
ENDM
```

Чтобы повторить макрос, вы, конечно, можете написать его несколько раз:

```
 PushAX
 PushAX
 PushAX
 PushAX
```

Но с помощью WHILE можно создать цикл, который повторяет макрос, *пока* выражение остается истинным. Вот один из способов использования WHILE для повторения предшествующего макроса заданное количество раз:

```
count = 4
 WHILE count GT 0
 PushAX
count = count-1
 ENDM
```

Первая строка определяет числовой символ, count, инициализированный значением 4. (Эта строка может находиться в другом файле или в начале программы.) Директива WHILE повторяет макрос PushAX, *пока* count больше (GT) ноля. Внутри директивы WHILE после каждого повторения PushAX count уменьшается на единицу. Обратите внимание на то, что вся эта конструкция заканчивается с помощью ENDM — директива WHILE сама по себе является предопределенным макросом.

Сохранение и восстановление состояния ассемблера

Для сохранения и восстановления операционного состояния Turbo Assembler используйте директивы PUSHSTATE и POPSTATE. Эти директивы особенно полезны в макросах, которые изменяют различные опции ассемблера, такие как текущая система счисления, или используют директивы SMART, NOSMART и другие значения. Вы можете, конечно, для сохранения и восстановления состояния ассемблера в любое время применять PUSHSTATE и POPSTATE и вне тела макроса.

Вставляя PUSHSTATE в программу, вы сохраняете следующие опции и установки. Вставляя директиву POPSTATE, вы восстанавливаете последние сохраненные значения установок:

- Текущая установка VERSION (например, T400)
- Операционный режим (например, IDEAL или MASM)
- Состояние переключателей, включая EMUL, NOEMUL, MULTERRS, NOMULTERRS, SMART, NOSMART, JUMPS, NOJUMPS, LOCALS и NOLOCALS
- Генерируемый код (например, P8086 или P386)
- Текущая система счисления
- Текущий префикс локальной метки (например, LOCALS @@)

PUSHSTATE и POPSTATE особенно полезны в макросах, используемых при различных условиях. Вы можете применять эти директивы в любом месте программы следующим образом:

```
PUSHSTATE
radix 2
NOJUMPS
.........
POPSTATE
```

В этом примере после PUSHSTATE сохраняется текущее состояние ассемблера, программа выбирает двоичную систему счисления и определяет переключатель NOJUMPS. Многоточие указывает на то место, где вставляются команды, требующие этих установок. После того как эта часть программы заканчивается, POPSTATE восстанавливает предыдущие установки.

Можно также использовать эти директивы для создания макроса, который сохраняет и восстанавливает состояние ассемблера. Просто начните и завершите макрос следующим образом:

```
MACRO AnyMac
PUSHSTATE
radix 2
  .......
POPSTATE
ENDM AnyMac
```

В примере макроса AnyMac PUSHSTATE сохраняет операционное состояние ассемблера перед тем, как макрос устанавливает двоичную систему счисления. Многоточие показывает, где вставляются другие команды макроса, требующие этой установки системы счисления. Перед самым концом тела макроса POPSTATE восстанавливает предыдущие значения операционного состояния.

Замечание

В соответствии с документацией Borland, Turbo Assembler поддерживает 16-уровневый стек для использования с этими директивами. Поэтому вкладывать один в другой PUSHSTATE больше, чем 16 раз, не рекомендуется, хотя проверка показывает, что при этом ассемблер не сообщает об ошибке. Более того, нужно следить за тем, чтобы каждому PUSHSTATE соответствовал POPSTATE. Неточность здесь тоже не рассматривается как ошибка, так же как и включение большего количества POPSTATES, чем PUSHSTATES. Все эти условия, казалось бы, создают проблемы для ассемблера, но так как вы не получаете предупреждений или сообщений об ошибках на этот счет, следует использовать эти директивы чрезвычайно аккуратно.

Запуск библиотеки макросов DOS

Многие программы на языке ассемблера тратят большое количество времени на вызовы процедур DOS, каждая из которых имеет свои требования, например получение значений в определенных регистрах.

Макросы DOS из этого раздела могут облегчить вам написание программ благодаря упрощению записей для вызова процедур DOS и документированию назначений регистров и других требований.

Не ассемблируйте программу 8.1, DOSMACS.ASM. Вместо этого запишите ее текстовый файл на диск и добавьте их в вашу программу, включив в ее начало (предпочтительно перед директивой DATASEG) следующую строку:

```
INCLUDE "DOSMACS.ASM"
```

Программа 8.1. DOSMACS.ASM

```
1:    ; DOS макросы для режима Ideal -- by Tom Swan
2:    %NOLIST
3:
4:    ;---------------------------------------------------------------
5:    ; MS_DOS Вызов любой функции DOS
6:    ;---------------------------------------------------------------
7:    ; Вход:
8:    ; functionNumber = номер функции DOS
9:    ; Выход:
10:   ; зависит от определенной функции
11:   ; Регистры:
```

```
; зависит от определенной функции
;---------------------------------------------------------------
MACRO MS_DOS functionNumber
      mov   ah, functionNumber ;; Задать номер функции
      int   21h                ;; вызов DOS
ENDM MS_DOS functionNumber

;---------------------------------------------------------------
; (01h) DOS_GetChar Получить символ с отображением
;---------------------------------------------------------------
; Вход:
; нет
; Выход:
; al = следующий символ, полученный со стандартного устройства ввода
; Регистры:
; ax
;---------------------------------------------------------------
MACRO DOS_GetChar
      mov   ah, 1              ;; Задать номер функции DOS
      int   21h                ;; Вызов DOS
ENDM DOS_GetChar

;---------------------------------------------------------------
; (02h) DOS_PutChar Записать символ на стандартное устройство вывода
;---------------------------------------------------------------
; Вход:
; dl = ASCII-символ (0-255)
; Выход:
; нет
; Регистры:
; ah
;---------------------------------------------------------------
MACRO DOS_PutChar
      mov   ah, 2              ;; Задать номер функции DOS
      int   21h                ;; Вызов DOS
ENDM DOS_PutChar

;---------------------------------------------------------------
; (05h) DOS_PrintChar Послать символ на печать
;---------------------------------------------------------------
; Вход:
; dl = ASCII-символ (0-255)
; Выход:
; нет
; Регистры:
; ah
;---------------------------------------------------------------
MACRO DOS_PrintChar
      mov   ah, 5              ;; Задать номер функции DOS
      int   21h                ;; Вызов DOS
ENDM DOS_PrintChar

;---------------------------------------------------------------
; (07h) DOS_GetRawChar Получить нефильтрованный символ без отображения
;---------------------------------------------------------------
; Вход:
; нет
; Выход:
; al = следующий символ со стандартного устройства ввода
; Регистры:
; ax
;---------------------------------------------------------------
MACRO DOS_GetRawChar
      mov   ah, 7              ;; Задать номер функции DOS
      int   21h                ;; Вызов DOS
ENDM DOS_GetRawChar

;---------------------------------------------------------------
; (08h) DOS_GetCharNoEcho Получить фильтрованный символ без отображения
;---------------------------------------------------------------
; Вход:
; нет
; Выход:
; al = следующий символ со стандартного устройства ввода
```

```
86:   ; Регистры:
87:   ; ax
88:   ;-------------------------------------------------------------
89:   MACRO DOS_GetCharNoEcho
90:         mov   ah, 8               ;; Задать номер функции DOS
91:         int   21h                 ;; Вызов DOS
92:   ENDM DOS_GetCharNoEcho
93:
94:   ;-------------------------------------------------------------
95:   ; (09h) DOS_PutString Вывести ASCII$-строку на стандартное устройство вывода
96:   ;-------------------------------------------------------------
97:   ; Вход:
98:   ; string = метка ASCII$-переменной
99:   ; Выход:
100:  ; нет
101:  ; Регистры:
102:  ; ah, dx
103:  ;-------------------------------------------------------------
104:  MACRO DOS_PutString string
105:        mov   ah, 9               ;; Задать номер функции DOS
106:        mov   dx, offset string   ;; Поместить адрес строки в ds:dx
107:        int   21h                 ;; Вызов DOS
108:  ENDM DOS_PutString
109:
110:  ;-------------------------------------------------------------
111:  ; (0Bh) DOS_Keypressed Проверка нажатия клавиш
112:  ;-------------------------------------------------------------
113:  ; Вход:
114:  ; нет
115:  ; Выход:
116:  ; zf = 0 : (jnz) Символ ожидает считывания
117:  ; zf = 1 : (jz) Нет ожидающего символа
118:  ; Регистры:
119:  ; ax
120:  ;-------------------------------------------------------------
121:  MACRO DOS_Keypressed
122:        mov   ah, 0Bh             ;; Задать номер функции DOS
123:        int   21h                 ;; Вызов DOS
124:        or    al, al              ;; Установить/очистить zf
125:  ENDM DOS_Keypressed
126:
127:  ;-------------------------------------------------------------
128:  ; (0Eh) DOS_SetDrive Изменить текущий диск
129:  ;-------------------------------------------------------------
130:  ; Вход:
131:  ; dl = номер диска (0 = A:, 1 = B:, 2 = C:, ..., 25 = Z:)
132:  ; Замечание: от F: до Z: требуется указывать LASTDRIVE = Z в файле CONFIG.SYS
133:  ; Выход:
134:  ; al = количество имеющихся дисков
135:  ; Регистры:
136:  ; ax
137:  ;-------------------------------------------------------------
138:  MACRO DOS_SetDrive
139:        mov   ah, 0Eh             ;; Задать номер функции DOS
140:        int   21h                 ;; Вызов DOS
141:  ENDM DOS_SetDrive
142:
143:  ;-------------------------------------------------------------
144:  ; (19h) DOS_GetDrive Получить текущий номер диска
145:  ;-------------------------------------------------------------
146:  ; Вход:
147:  ; нет
148:  ; Выход:
149:  ; al = номер диска (0 = A:, 1 = B:, 2 = C:, ..., 25 = Z:)
150:  ; Регистры:
151:  ; ax
152:  ;-------------------------------------------------------------
153:  MACRO DOS_GetDrive
154:        mov   ah, 19h             ;; Задать номер функции DOS
155:        int   21h                 ;; Вызов DOS
156:  ENDM DOS_GetDrive
157:
158:  ;-------------------------------------------------------------
159:  ; (25h) DOS_SetVector Установить вектор прерывания
```

```
160:  ;---------------------------------------------------------------
161:  ; Вход:
162:  ; interrupt = номер прерывания (0-255)
163:  ; address = метка начала процедуры прерывания
164:  ; Выход:
165:  ; нет
166:  ; Регистры:
167:  ; ax, dx
168:  ;---------------------------------------------------------------
169:  MACRO DOS_SetVector interrupt, address
170:        push  ds                  ;; Сохранить текущий регистр ds
171:        mov   ax, SEG address     ;; Поместить адрес сегмента
172:        mov   ds, ax              ;; устанавливаемого прерывания в ds
173:        mov   dx, OFFSET address  ;; Поместить адрес смещения в dx
174:        mov   ah, 025h            ;; Задать номер функции DOS
175:        mov   al, interrupt       ;; Поместить номер прерывания в al
176:        int   21h                 ;; Вызов DOS
177:        pop   ds                  ;; Восстановить сегментный регистр ds
178:  ENDM DOS_SetVector
179:
180:  ;---------------------------------------------------------------
181:  ; (35h) DOS_GetVector Получить вектор прерывания
182:  ;---------------------------------------------------------------
183:  ; Вход:
184:  ; interrupt = номер прерывания
185:  ; Выход:
186:  ; es:bx = сегмент:смещение прерывания
187:  ; Регистры:
188:  ; ax, bx, es
189:  ;---------------------------------------------------------------
190:  MACRO DOS_GetVector interrupt
191:        mov   al, interrupt       ;; Поместить номер прерывания в al
192:        mov   ah, 35h             ;; Задать номер функции DOS
193:        int   21h                 ;; Вызов DOS
194:  ENDM DOS_GetVector
195.
196:  ;---------------------------------------------------------------
197:  ; (3Bh) DOS_ChDir Изменить текущий каталог
198:  ;---------------------------------------------------------------
199:  ; Вход:
200:  ; dirName = метка ASCIIZ строки в сегменте данных ds
201:  ; Выход:
202:  ; cf = 0 : (jnc) Успешное изменение
203:  ;
204:  ; cf = 1 : (jc) Неудача при изменении
205:  ; ax = код ошибки (3=каталог не найден)
206:  ; Регистры:
207:  ; ax, dx
208:  ;---------------------------------------------------------------
209:  MACRO DOS_ChDir dirName
210:        mov   ah, 3Bh             ;;Задать номер функции DOS
211:        mov   dx, OFFSET dirName  ;; Поместить адрес строки в ds:dx
212:        int   21h                 ;; Вызов DOS
213:  ENDM DOS_ChDir
214:
215:  ;---------------------------------------------------------------
216:  ; (3Ch) DOS_CreateFile Создать новый файл
217:  ;---------------------------------------------------------------
218:  ; Вход:
219:  ; fileName = метка ASCIIZ строки в сегменте данных ds
220:  ; cx = атрибут файла
221:  ; 00 = обычный файл
222:  ; 01 = только для чтения (для чтения/записи доступ запрещен)
223:  ; 02 = скрытый (DIR имя не показывает)
224:  ; 04 = системный файл
225:  ; Выход:
226:  ; cf = 0 : (jnc) Файл создан
227:  ; ax = дескриптор файла для последующего использования
228:  ;
229:  ; cf = 1 : (jc) Файл не создан
230:  ; ax = код ошибки
231:  ; 3 = путь не найден
232:  ; 4 = нет свободных дескрипторов
233:  ; 5 = доступ запрещен
```

```
234:  ; Регистры:
235:  ; ax, dx
236:  ;-------------------------------------------------------------
237:  MACRO DOS_CreateFile fileName
238:        mov   ah, 3Ch              ;; Задать номер функции DOS
239:        mov   dx, OFFSET fileName        ;; Поместить имя по адресу ds:dx
240:        int   21h                  ;; Вызов DOS
241:  ENDM DOS_CreateFile
242:
243:  ;-------------------------------------------------------------
244:  ; (3Dh) DOS_OpenFile Открыть файл для ввода-вывода
245:  ;-------------------------------------------------------------
246:  ; Вход:
247:  ; fileName = метка ASCIIZ строки в сегменте данных ds
248:  ; Выход:
249:  ; cf = 0 : (jnc) Файл открыт
250:  ; ax = дескриптор файла для последующего использования
251:  ;
252:  ; cf = 1 : (jc) Файл не открыт
253:  ; ax = код ошибки
254:  ; 2 = файл не найден
255:  ; 3 = путь не найден
256:  ; 4 = нет свободных дескрипторов
257:  ; 5 = доступ запрещен
258:  ; Регистры:
259:  ; ax, dx
260:  ;-------------------------------------------------------------
261:  MACRO DOS_OpenFile fileName
262:        mov   ah, 3Dh              ;; Задать номер функции DOS
263:        mov   al, 02               ;; Открыть для чтения/записи
264:        mov   dx, OFFSET fileName        ;; Поместить имя в ds:dx
265:        int   21h                  ;; Вызов DOS
266:  ENDM DOS_OpenFile
267:
268:  ;-------------------------------------------------------------
269:  ; (3Eh) DOS_CloseFile Закрыть предварительно открытый файл
270:  ;-------------------------------------------------------------
271:  ; Вход:
272:  ; bx = дескриптор файла DOS_CreateFile или DOS_OpenFile
273:  ; Выход:
274:  ; cf = 0 : (jnc) Файл закрыт
275:  ;
276:  ; cf = 1 : (jc) Файл не закрыт
277:  ; ax = код ошибки
278:  ; 6 = неверный дескриптор или файл не открыт
279:  ; Регистры:
280:  ; ax
281:  ;-------------------------------------------------------------
282:  MACRO DOS_CloseFile
283:        mov ah, 3Eh                ;; Задать номер функции DOS
284:        int 21h                    ;; Вызов DOS
285:  ENDM DOS_CloseFile
286:
287:  ;-------------------------------------------------------------
288:  ; (3Fh) DOS_ReadFile Читать из файла или устройства
289:  ;-------------------------------------------------------------
290:  ; Вход:
291:  ; bx = дескриптор файла из DOS_CreateFile или DOS_OpenFile
292:  ; cx = количество считываемых байтов
293:  ; buffer = метка буфера в сегменте данных ds
294:  ; Замечание: Буфер должен иметь длину не меньшую cx!
295:  ; Выход :
296:  ; cf = 0 : (jnc) Чтение успешное
297:  ; ax = количество считанных байтов (0 = конец файла)
298:  ;
299:  ; cf = 1 : (jc) Чтение неудачное
300:  ; ax = код ошибки
301:  ; 5 = доступ запрещен
302:  ; 6 = неверный дескриптор или файл не открыт
303:  ; Регистры:
304:  ; ax, dx
305:  ;-------------------------------------------------------------
306:  MACRO DOS_ReadFile buffer
307:        mov   ah, 3Fh              ;; Задать номер функции DOS
```

```
308:        mov    dx, OFFSET buffer   ;; Поместить адрес буфера в ds:dx
309:        int    21h                 ;; Вызов DOS
310:  ENDM DOS_ReadFile
311:
312:  ;---------------------------------------------------------------
313:  ; (40h) DOS_WriteFile Запись в файл или устройство
314:  ;---------------------------------------------------------------
315:  ; Вход:
316:  ; bx = дескриптор файла из DOS_CreateFile или DOS_OpenFile
317:  ; cx = количество записываемых символов
318:  ; buffer = метка буфера источника в сегменте данных ds
319:  ; Выход:
320:  ; cf = 0 : (jnc) Запись успешная
321:  ; ax = количество записанных байтов (0 = диск полон)
322:  ;
323:  ; cf = 1 : (jc) Запись не успешна
324:  ; ax = код ошибки
325:  ; 5 = доступ запрещен
326:  ; 6 = неверный дескриптор или файл не открыт
327:  ; Регистры:
328:  ; ax, dx
329:  ;---------------------------------------------------------------
330:  MACRO DOS_WriteFile buffer
331:        mov    ah, 40h             ;; Задать номер функции DOS
332:        mov    dx, OFFSET buffer   ;; Поместить буфер в ds:dx
333:        int    21h                 ;; Вызов DOS
334:  ENDM DOS_WriteFile
335:
336:  ;---------------------------------------------------------------
337:  ; (42h) DOS_Seek Изменить положение для следующего чтения/записи
338:  ;---------------------------------------------------------------
339:  ; Вход:
340:  ; bx = Дескриптор файла из DOS_CreateFile или DOS_OpenFile
341:  ; cx = старшее слово 32-битового смещения байта
342:  ; dx = младшее слово 32-битового смещения байта
343:  ; Выход:
344:  ; cf = 0 : (jnc) Поиск удачный
345:  ; dx = старшее слово 32-битового смещения позиции после поиска
346:  ; ax = младшее слово 32-битового смещения позиции после поиска
347:  ;
348:  ; cf = 1 : (jc) Поиск неудачен
349:  ; ax = код ошибки
350:  ; 6 = неверный указатель или файл не открыт
351:  ; Регистры:
352:  ; ax
353:  ;---------------------------------------------------------------
354:  MACRO DOS_Seek
355:        mov    ah, 42h             ;; Задать номер функции DOS
356:        xor    al, al              ;; Поиск позиции в cx,dx
357:        int    21h                 ;; Вызов DOS
358:  ENDM DOS_Seek
359:
360:  ;---------------------------------------------------------------
361:  ; (47h) DOS_GetDir Получить имя текущего каталога
362:  ;---------------------------------------------------------------
363:  ; Вход.
364:  ; string = адрес 64-байтовой (минимально) переменной
365:  ; Выход:
366:  ; имя каталога, вставленное в строку ASCIIZ-формата
367:  ; Регистры:
368:  ; ax, dl, si
369:  ;---------------------------------------------------------------
370:  MACRO DOS_GetDir string
371:        mov    ah, 47h             ;; Задать номер функции DOS
372:        xor    dl, dl              ;; 0 определяет текущий диск
373:        mov    si, OFFSET string   ;; Поместить адрес строки в ds:si
374:        int    21h                 ;; Вызов DOS
375:  ENDM DOS_GetDir
376:
377:  ;---------------------------------------------------------------
378:  ; (4Ch) DOS_Terminate Конец программы
379:  ;---------------------------------------------------------------
380:  ; Вход:
381:  ; code = [метка] или значение, передаваемое в DOS либо в родительский процесс
```

```
382:   ; Выход:
383:   ; нет
384:   ; Регистры:
385:   ; ax
386:   ;-----------------------------------------------------------------
387:   MACRO DOS_Terminate code
388.         mov    ah, 4Ch              ;; Задать номер функции DOS
389.         mov    al, code             ;; Задать возвращаемый код
390.         int    21h                  ;; Вызов DOS
391    ENDM DOS_Terminate
392.
393    %LIST
```

Использование DOSMACS.ASM

Большинство макросов в DOSMACS не нуждаются в объяснениях — просто прочитайте комментарии, предшествующие каждому из них, чтобы разобраться во всех требованиях, выходных данных и изменяемых регистрах. Файл DOSMACS.ASM начинается с команды %NOLIST, предотвращающей распечатку макросов, даже если вы определите при ассемблировании опцию печати /l. Это сокращает длину листинга вашей программы, без повторения одного и того же текста для всех модулей, содержащих макросы. Для справки, в табл. 8.3 представлен каждый макрос, вместе со связанным с ним номером функции в шестнадцатеричной системе.

Таблица 8.3. **Макросы DOSMACS**

Номер	Названия и параметры макросов
-	MS_DOS functionNumber
01h	DOS_GetChar
02h	DOS_PutChar
05h	DOS_PrintChar
07h	DOS_GetRawChar
08h	DOS_GetCharNoEcho
09h	DOS_PutString string
0Bh	DOS_Keypressed
0Eh	DOS_SetDrive
19h	DOS_GetDrive
25h	DOS_SetVector interrupt, address
35h	DOS_GetVector interrupt
3Bh	DOS_ChDir dirName
3Ch	DOS_CreateFile fileName
3Dh	DOS_OpenFile fileName
3Eh	DOS_CloseFile
3Fh	DOS_ReadFile buffer
40h	DOS_WriteFile buffer
42h	DOS_Seek
47h	DOS_GetDir string
4Ch	DOS_Terminate code

Замечание

DOSMACS содержит в себе только подмножество функций DOS. Хорошо было бы расширить DOSMACS до полного набора DOS; но имейте в виду, что это приведет к увеличению времени ассемблирования программ, включающих в себя макросы.

Вы также можете вызывать функции DOS по номеру, используя для этого макрос MS_DOS вместо того, чтобы загружать ah и выполнять int 21h. Помните, что при этом изменяется ah. Чтобы вывести символ, помещенный в dl, можно написать:

```
mov dl, 'A'             ; Выводимый символ
MS_DOS 2                ; Вызов функции вывода DOS
```

Чтобы использовать макро, для которого определяются параметры, прочитайте комментарии, загрузите регистр или выделите пространство для переменной и используйте метку идентификатора в качестве параметра. Для записи ASCII$-строк в стандартный файл вывода используйте следующие команды:

```
DATASEG
Welcome db 'Welcome to my program','$'
CODESEG
DOS_PutString Welcome ; Вывод приглашения
```

Некоторые макросы возвращают результаты в регистры и флаги. Например, чтобы проверить, вводились ли данные с клавиатуры, можно написать:

```
@@10: DOS_Keypressed  ; Была ли нажата клавиша?
  jz continue         ; Переход, если нет
  DOS_GetRawChar      ; Иначе получить символ (без отображения)
  call Process char   ; Вызвать процедуру обработки символа
Continue:
```

Если DOS_Keypressed установил флаг zf, тогда не вводился ни один символ и программа продолжается с метки Continue:. Если zf сброшен, тогда второй макрос DOS_GetRawChar читает символ и вызывает процедуру ProcessChar (не показанную), обрабатывающую ввод. Макросы помогают документировать программу, превращая номера функций DOS в хорошо понятные имена.

Замечание

Если получена странная ошибка, например "Неопределенный символ", в ответ на использование таких известных ключевых слов, как OFFSET, проверьте, определили ли вы все требуемые параметры. Попробуйте также взять параметры в угловые скобки, например <OFFSET CodeLabel>. Если все еще невозможно определить, что приводит к ошибке, вставьте %MACS в начало программы и выполните ассемблирование с опцией /l, чтобы создать листинг, показывающий вызовы вашего макроса вместе с развернутыми командами. Читая этот листинг, вы должны понять, что в вашей программе работает неправильно.

Резюме

Записывая основные последовательности команд в макросы, вы добавляете в Turbo Assembler свои новые команды. Макросы могут упрощать язык ассемблера, уменьшая размер текста программы, и помогают обеспечивать совместимость, особенно в групповых проектах. Макросы имеют несколько недостатков, такие как необходимое для каждого модуля переассемблирование библиотеки макросов и незаметное изменение значений регистров.

Макроопределение начинается с MACRO и заканчивается ENDM. При удалении макроса с помощью PURGE он убирается из памяти, экономя память, и дает вам возможность заменять отдельные макросы, возможно для пересмотра и проверки.

Существует три типа параметров макросов: символьные, числовые и строковые. Формальные параметры записываются в макросах. Действительные параметры указываются при использовании макроса. Когда Turbo Assembler встречает в программе имя макроса, он разворачивает его, заменяя имя макроса на команды из этого макроса, и подставляет вместо формальных параметров действительные. Параметры дают возможность писать программируемые макросы, которые могут изменяться в соответствии с новыми требованиями.

Макросы могут использоваться для определения новых типов данных с применением таких директив, как dw и db. Макросы можно также использовать вместо процедур, заменяя команды call на разворачивающийся код, что является важным приемом оптимизации, позволяющим увеличить скорость выполнения программы. Повторяющиеся макросы могут создавать команды, использующие списки значений регистров и символов.

Для создания автоматически нумерующихся локальных меток используйте внутри макроса директиву LOCAL. Для немедленной остановки разворачивания макроса применяйте директиву EXITM. Используйте WHILE для повторения макроса определенное количество раз или до тех пор, пока некоторое другое условие остается истинным.

Для сохранения состояния ассемблера, включая множество его опций и установок, вставьте в любом месте программы директиву PUSHSTATE.

Чтобы восстановить последнее сохраненное состояние ассемблера, вставьте директиву POPSTATE. Можно также использовать PUSHSTATE и POPSTATE внутри макросов.

Условные символы и директивы позволяют писать программы, которые ассемблируются по-разному, основываясь на условиях, определенных в начале программы. Условный символ является числовым макроопределением. По определению символ считается определенным, когда вы присваиваете ему значение. Условные директивы, такие как IF и IFE, могут проверять значения символов и выражений. Другие директивы, такие как IFDEF и IFNDEF, проверяют, определен ли символ.

Многочисленные макросы обычно хранятся в текстовых файлах и затем загружаются в модули с помощью директивы INCLUDE. Эта глава содержит пример библиотеки макросов, DOSMACS.ASM, с несколькими макросами для вызова общеупотребительных функций DOS.

Упражнения

8.1. Каковы преимущества и недостатки использования макросов?

8.2. Напишите макрос с именем StartUp, инициализирующий в начале программы регистры es и ds.

8.3. Какое значение или значения должны иметь условия true и false? Какое значение или значения обычно используются для представления true?

8.4. Для чего необходимы двойные точки с запятой ;; ?

8.5. Как можно выйти из макроса?

8.6. Как вы определяете типы параметров в макросах?

8.7. Напишите макросы str и clz для установки и очистки флага нуля zf. Макросы не должны воздействовать на любые другие флаги и должны сохранять все значения регистров.

8.8. Напишите макрос для присваивания литерального значения любому сегментному регистру. Покажите, как использовать ваш макрос для установки в es адреса видеобуфера цветного дисплея 0B800h.

8.9. Какую команду или команды вы будете использовать для добавления файлов библиотеки макросов FLOAT.MAX, BIOSMAC.TXT и CUSTOM.MAX в программу?

8.10. Создайте условный символ с именем HasFastCrt, устанавливаемый в true или false в начале программы и показывающий, имеет ли система memory-mapped видеодисплей, как все PC, или замедленный "немой" терминал, который мог бы находиться на main-frames и более старых PC. Используйте ваш символ в процедуре, которая выводит символ, соответственно выбирая для этого или процедуру ScPokeChar из модуля SCREEN (см. главу 7), или аналогичную функцию вывода DOS.

8.11. Процедура должна работать одинаково во всех отношениях, независимо от выбранного аппаратного обеспечения. В своем ответе можете использовать DOSMACS.ASM.

Задания

8.1. Примените метод из упражнении 8.10 ко всем процедурам в модуле SCREEN, создав модуль, который можно ассемблировать для PC с memory-mapped видео или для систем, использующих в качестве главной консоли медленный "немой" терминал.

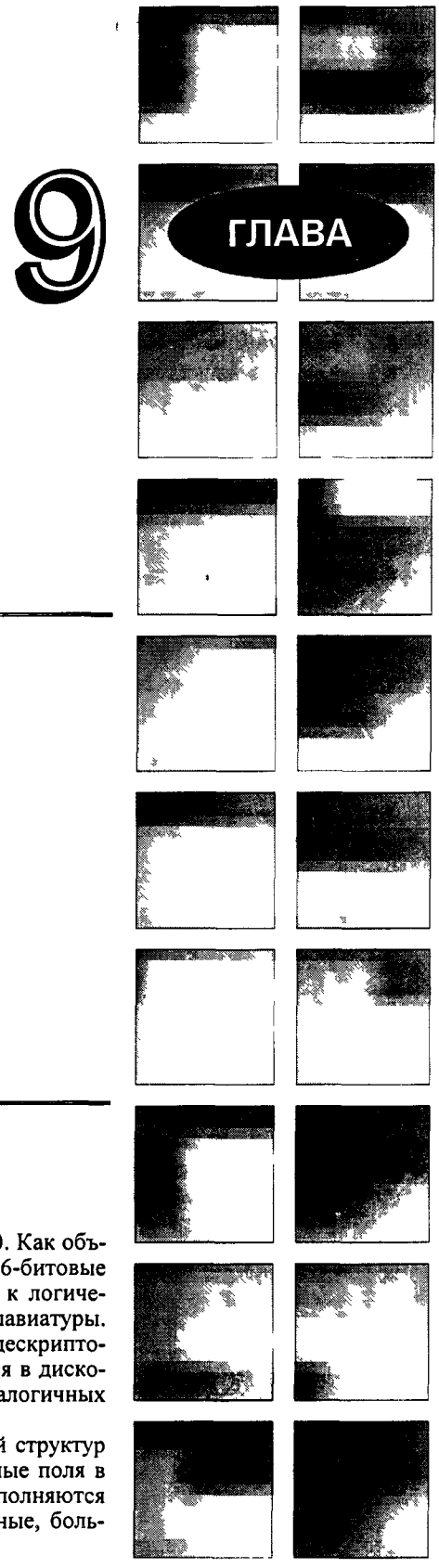
8.2. Напишите модуль для выбора режимов некоторых разновидностей принтеров, выбирая в соответствии с условиями код для переключения на жирную печать, подчеркивание и другие опции. Создайте код так, чтобы он позволял печатать текст на простых принтерах без таких возможностей.

8.3. Создайте библиотеку макросов BIOSMAC.ASM, аналогичную DOSMACS.ASM из этой главы. Ваши процедуры должны упростить вызов функций ROM BIOS, представленных в руководстве по PC.

8.4. Найдите свободно распространяемый текст программы на языке ассемблера (или возьмите один из примеров этой книги), в котором используются повторяющиеся вызовы процедур. Замените процедуры на макросы, вставляя код непосредственно в строку вместе с другими командами. Проверьте, как это отражается на скорости программы и размере исполняемого файла.

8.5. Создайте библиотеку файлов макросов и объектных модулей, которые упрощают стандартные методы отладки программы. Включите в нее процедуры для вывода (или печати) используемого процедурами стека, списка значений ключевых переменных и проверки других значений, например диапазонов индексов массивов.

8.6. Напишите макросы, которые используют условные директивы для создания переменных в форматах ASCIIZ и ASCII$ с автоматическими переменными длины и без них.



ГЛАВА 9

Работа с дисковыми файлами

- Понятие о дескрипторах файлов
- Общие представления о дисках и файлах
- Максимальное количество файлов
- Открытие и закрытие файлов
- Заполнение файловых буферов
- Закрытие файлов
- Обработка дисковых ошибок
- Создание новых файлов
- Чтение командной строки DOS
- Чтение и запись текстовых файлов
- Чтение и запись файлов данных
- Чтение каталога диска
- Резюме
- Упражнения
- Задания

Понятие о дескрипторах файлов

Понятие дескриптора файлов было введено, начиная с DOS версии 2.0. Как объяснялось в главе 7, в дескрипторах нет ничего загадочного. Это простые 16-битовые беззнаковые целые, используемые DOS и программами для обращения к логическим файлам, которые относятся к таким устройствам, как принтеры и клавиатуры. В данной главе затрагивается эта тема и показывается, как применяются дескрипторы в программах на языке ассемблера для обработки данных, хранящихся в дисковых файлах — включая файлы на гибких и жестких дисках и других аналогичных устройствах.

До версии DOS 2.0 файловый ввод-вывод ограничивался поддержкой структур данных, называемых *файловыми управляющими блоками* (FCB). Различные поля в FCB содержат информацию о позициях, в которых последовательно выполняются операции чтения или записи, о размерах записей в файле и другие данные, боль-

шинство из которых необходимы DOS, но редко используются (если вообще это происходит) непосредственно в программе. Дескрипторы файлов упрощают файловый ввод-вывод, устраняя необходимость создавать и хранить записи FCB, но не жертвуя при этом какими-либо операционными возможностями. После создания нового файла или открытия уже существующего файла на диске необходимо только указать на этот файл с помощью дескриптора файла, даже в том случае, если вы собираетесь выполнять самые сложные файловые операции. По этим причинам и так как Microsoft прекратила использование старых вызовов функций FCB, эта глава ориентирована исключительно на самые новые методы дескрипторов файлов.

Общие представления о дисках и файлах

Перед написанием программ, считывающих и записывающих данные в дисковые файлы, важно усвоить несколько универсальных понятий, связанных с файловым вводом-выводом.

1. Необходимо открыть файл, перед тем как читать данные из него или записывать новые данные на диск. При открытии существующих файлов сохраняется информация, записанная там ранее.
2. При создании нового файла он также открывается для чтения и записи, но при этом, если на диске уже существует файл под таким же именем, вся информация в нем удаляется.
3. Данные, которые вы записываете в дисковые файлы, временно хранятся в буфере памяти DOS. Не думайте, что операция записи на диск действительно сразу их записывает.
4. При закрытии файла все данные из буфера записываются на диск, обеспечивая при этом сохранность всех прежде сохраненных в буфере данных.
5. При закрытии файла также пополняется каталог диска и файловый дескриптор освобождается для дальнейшего использования.
6. Текущая позиция — это указатель на место в файле, где будет начинаться следующая операция считывания или записи. DOS сохраняет для вас этот указатель. Можно перемещать текущую позицию внутри файла, что обеспечивает доступ к данным, находящимся в разных местах файла, но при этом существует только один указатель, связанный с конкретным открытым файлом.

Максимальное количество файлов

Каждая программа может одновременно иметь максимум 20 открытых файлов; суммарное число открытых файлов для всех активных программ составляет 255. Когда одна программа запускает другую через вызов DOS Exec функции 04Bh, DOS выделяет новой программе по максимуму 20 дескрипторов файлов, до тех пор пока не будет достигнута сумма в 255 дескрипторов, допустимых для всех одновременно выполняющихся программ. При завершении программы функция DOS 04Ch закрывает все активные дескрипторы, освобождая их для использования в других программах. Из 20 допустимых дескрипторов файлов для каждой активной программы DOS резервирует пять дескрипторов от 0 до 4 для стандартных устройств ввода-вывода (см. главу 7). Таким образом, в программах обычно допускается открытие только 15 файлов. Для увеличения этого предела можно закрыть один или более стандартных дескрипторов. Например, программы, которые не вызывают функции DOS управления принтером и последовательными портами ввода-вывода, могут увеличивать это число на 2 после выполнения следующего кода:

```
mov  ah, 03Eh          ; Номер функции DOS закрытия файла
mov  bx, 3             ; Установить в bx номер дескриптора AUX (3)
int  21h               ; Вызов DOS для закрытия файла
inc  bx                ; Установить в bx номер дескриптора PRN (4)
int  21h               ; Вызов DOS для закрытия файла
```

Открытие и закрытие файлов

Открытие файла для чтения и записи подобно открытию двери для вноса мебели внутрь и выноса наружу. После открытия файла можно читать и записывать туда данные с любой частотой, не делая при этом ошибок. Чтобы открыть дисковый файл на языке ассемблера, пошлите адрес имени файла в формате ASCIIZ строки функции DOS 03Dh, как в следующем примере:

```
DATASEG
Filename DB 'C:\TASM\TEST.ASM' ,0
CODESEG
mov  ax, @data                 ; Поместить в ds адрес
mov  ds, ax                    ; сегмента данных
mov  dx, offset filename       ; Поместить адрес filename в ds:dx
mov  ah, 03Dh                  ; Номер функции DOS открытия файла
```

```
mov  al, 0              ; 0 = доступ только на чтение
int  21h                ; Вызов DOS для открытия файла
jc   Error              ; Обработка ошибок
```

В имени файла может определяться диск и название подкаталога, как показано в этом примере. После инициализации сегментного регистра ds (что вы должны делать во всех программах) используйте ds:dx для адресации на имя файла при вызове функции DOS 03Dh. Вдобавок, регистр al устанавливается в 0, указывая DOS, что файл открывается только для операции чтения. Под DOS 2.0 и более поздними версиями al может иметь одно из трех значений:

- al=0 – только операции чтения
- al=1 – только операции записи
- al=2 – операции чтения и записи

Под DOS 3.0 и более поздними версиями доступны дополнительные значения для разделяемых файлов в сетевых системах. (См. документацию по DOS, где описываются эти значения.) После вызова DOS для открытия файла флаг переноса cf показывает, была ли операция успешной. Как демонстрирует предыдущий пример, это дает возможность использовать условные переходы, такие как jc, для перехода на процедуру обработки ошибок при неудачной операции, возможно, вследствие того, что файл не был найден. В этом случае ax содержит один из кодов ошибок, приведенных в табл. 9.1. Если ошибки не произошло, то в ax содержится дескриптор файла, который можно использовать для последующих операций. Обычно хорошо сразу сохранять этот дескриптор в переменной, освобождая ax для других целей:

```
DATASEG
 handle DW ? ; Переменная под дескриптор файла
CODESEG
;
; открыть файл с помощью функции DOS 03Dh
;
mov [handle], ax ; Сохранить дескриптор файла на будущее
```

Таблица 9.1 **Коды ошибок открытия файлов**

Code	Значение
01	Разделение файла не допускается
02	Файл не существует
03	Путь или файл не существует
04	Нет больше свободных дескрипторов
05	Доступ запрещен (ошибочный атрибут файла)
0Ch	Неверное значение для доступа в регистре al

Заполнение файловых буферов

Файловый буфер — это часть памяти, которая служит в качестве промежуточной станции при путешествии данных к диску и от диска. Ваша программа также может создать собственные файловые буферы для записи данных. Имейте в виду, что DOS имеет свои собственные файловые буферы, управляемые с помощью команды BUFFERS=n в вашем файле CONFIG.SYS. Большинство источников рекомендуют устанавливать n в 20, чтобы обеспечить по крайней мере один буфер для каждого из максимального количества файлов, которые могут использоваться программой.

Когда вы пишете данные в файл, они на время попадают в файловый буфер, а не сразу непосредственно на диск. Позже, когда программа читает другие данные из файла, открывает новый файл или выполняет другие операции над файлами, DOS может сбросить модифицированные буферы на диск, чтобы освободить место в памяти для новых данных. Всегда помните об этой задержке. Данные, которые вы записывали на диск, могут появляться там не сразу. Чтобы записать данные из буфера на диск, скопируйте дескриптор файла с помощью функции DOS 45h, после чего закройте эту копию, оставляя начальный дескриптор файла открытым:

```
mov  ah, 45h            ; Номер функции копирования дескриптора
mov  bx, [handle]       ; Копируемый дескриптор
int  21h                ; Вызов DOS
jc   error              ; Переход в случае ошибки (cf = 1)
mov  bx, ax             ; Поместить копию дескриптора в bx
mov  ah, 3Eh            ; Номер функции закрытия файла
int  21h                ; Вызов DOS
jc   error              ; Переход в случае ошибки (cf = 1)
```

Закрытие файлов

Закрывать файл достаточно просто — только передайте через регистр bx дескриптор любого открытого файла в функцию 03Eh. Закрывая файл, DOS записывает на диск все данные, содержащиеся в буферах памяти, дополняет запись файла в каталоге, записывая его размер, дату и время создания. Предполагая, что вы открыли файл, как это уже описано, и сохранили дескриптор файла, закройте его следующим образом:

```
mov  bx, [handle]     ; Поместить дескриптор в bx
mov  ah, 03Eh         ; Номер функции DOS закрытия файла
int  21h              ; Вызов DOS для закрытия файла
jc   error            ; Переход в случае ошибки
```

После вызова функции DOS 03Eh проверьте флаг переноса, как предложено здесь с помощью команды jc. Если cf=1, тогда ax содержит код ошибки, возможно 6, показывая, что дескриптор — неправильный (может быть, вы поместили в bx неверное значение дескриптора) или файл до этого не был открыт.

При закрытии файлов их дескрипторы освобождаются для дальнейшего использования. Хотя на практике лучше закрывать все открытые файлы до окончания программы, функция DOS 04Ch также закрывает все открытые дескрипторы файлов. Это значит, что можно открывать несколько файлов, считывать и записывать туда данные и заканчивать вашу программу функцией 04Ch с уверенностью, что DOS сохранит на диске любые изменения данных в памяти.

Обработка дисковых ошибок

Работая с файлами, вы должны уметь обнаруживать и обрабатывать всевозможные ошибочные ситуации. Это особенно важно в программировании на языке ассемблера, в котором отсутствует встроенный механизм обработки ошибок, обычно имеющийся в Pascal и Basic. На вас лежит ответственность за определение ошибок, вывод соответствующих предупреждений и сообщений и принятие надлежащих действий при переполнении диска и других возникающих проблемах.

Во всех случаях флаг переноса указывает на успех (cf=0) или на неудачу (cf=1) файловой операции; таким образом, всегда нужно проверять его после каждого вызова функции, работающей с файлом. Что вы будете делать после этого — решаете вы. Проще всего, можно заканчивать программу всякий раз, когда появляется ошибка. (Помните, что при этом закрываются все открытые файлы.) Можно возвращаться в известное место — главное меню программы, например, позволяя пользователям повторить неудачную операцию. Для получения более детальной информации можно вызвать функцию 059h, которая запрашивает у DOS добавочные сведения об ошибке. (Можно делать это после каждого вызова 21h. Эта функция применяется не только для файловых операций.)

Программа 9.1, DISKERR.ASM, использует этот метод в процедуре для получения расширенной информации об ошибке от DOS и выводит соответствующее сообщение. Программа написана как библиотечный модуль, который можно скомпоновать с вашей собственной программой (и с другими в этой главе) в качестве части блока, обрабатывающего ошибки. Выполните ассемблирование модуля и добавьте объектный код к вашему файлу библиотеки MTA.LIB с помощью команд:

```
tasm /zi diskerr
tlib /E mta-+diskerr
```

Повторите эти шаги, если вы позже измените DISKERR.ASM, и не обращайте внимания на обычное предупреждение о том, что DISKERR отсутствует в библиотеке при первом выполнении команды tlib. Чтобы уменьшить размер файла, уберите опцию /zi , необходимую только для запуска программ в Turbo Debugger.

Программа 9.1. DISKERR.ASM

```
1:    %TITLE " -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:
7:          DATASEG
8
9:    errString   DB                '** ERROR: ', 0
10:
11:   err00 DB    'Неизвестная причина', 0
12:   err01 DB    'Неправильный номер функции, 0
13:   err02 DB    'Файл не найден', 0
```

```
14:   err03 DB    'Путь не найден', 0
15:   err04 DB    'Слишком много открытых файлов', 0
16:   err05 DB    'Доступ запрещен', 0
17:   err06 DB    'Неверный дескриптор файла', 0
18:   err07 DB    'Разрушен блок управления памятью', 0
19:   err08 DB    'Недостаточно памяти для операции', 0
20:   err09 DB    'Неправильный адрес блока памяти', 0
21:   err0A DB    'Неправильное окружение', 0
22:   err0B DB    'Неверный формат', 0
23:   err0C DB    'Неверный код доступа', 0
24:   err0D DB    'Неверные данные', 0
25:   err0E DB    'Неизвестная причина', 0
26:   err0F DB    'Неправильная буква в названии диска', 0
27:   err10 DB    'Удаление текущего каталога недопустимо', 0
28:   err11 DB    'Устройство отлично от необходимого', 0
29:   err12 DB    'Нет больше файлов', 0
30:   err13 DB    'Диск защищен от записи', 0
31:   err14 DB    'Неизвестный модуль', 0
32:   err15 DB    'Диск не готов, 0
33:   err16 DB    'Неизвестная команда', 0
34:   err17 DB    'Ошибка данных (CRC) ', 0
35:   err18 DB    'Неправильная длина структуры', 0
36:   err19 DB    'Ошибка поиска', 0
37:   err1A DB    'Неизвестный тип носителя', 0
38:   err1B DB    'Сектор не найден', 0
39:   err1C DB    'В принтере нет бумаги', 0
40:   err1D DB    'Ошибка записи на диск', 0
41:   err1E DB    'Ошибка чтения с диска', 0
42:   err1F DB    'Общий сбой', 0
43:
44:   errorsDW    err00, err01, err02, err03, err04, err05, err06, err07
45:               DW                 err08, err09, err0A, err0B, err0C, err0D, err0E, err0F
46:               DW                 err10, err11, err12, err13, err14, err15, err16, err17
47:               DW                 err18, err19, err1A, err1B, err1C, err1D, err1E, err1F
48:
49:         CODESEG
50:
51:   ;----- Из STRIO.OBJ
52:         EXTRN NewLine:proc, StrWrite:proc
53:
54:         PUBLIC DiskErr
55:
56:   %NEWPAGE
57:   ;---------------------------------------------------------------
58:   ; DiskErr Вывести сообщение об ошибке на стандартное устройство вывода
59:   ;---------------------------------------------------------------
60:   ; Вход:
61:   ; нет (cf=1 из файловой операции DOS )
62:   ; Выход:
63:   ; нет (выводится сообщение об ошибке)
64:   ; Регистры:
65:   ; изменяются ax, bp, bx, cx, dx, di, si
66:   ;---------------------------------------------------------------
67:   PROC DiskErr
68:         push  ds                  ; Сохранить сегментные регистры,
69:         push  es                  ; изменяемые функцией DOS 59h
70:         mov   ah, 59h             ; Номер функции DOS получения расширенной информации
71:         xor   bx, bx              ; Должно равняться 0
72:         int   21h                 ; Получить расширенную информацию об ошибке
73:         pop   es                  ; Восстановить сегментные регистры
74:         pop   ds
75:
76:         cmp   ax, 1Fh             ; ax > 1Fh?
77:         jbe   @@10                ; Переход, если ax <= 1Fh
78:         xor   ax, ax              ; Использовать сообщение "Неизвестная причина"
79:   @@10:
80:         shl   ax, 1               ; Умножить ax на 2
81:         mov   bx, ax              ; Скопировать ax в bx
82:         mov   di, [errors+bx]     ; Получить адрес строки
83:         push  di                  ; Временно сохранить di
84:         call  NewLine             ; Перейти на новую строку
85:         mov   di, offset errString      ; Адрес первой части сообщения
86:         call  StrWrite            ; Вывести сообщение об ошибке
87:         pop   di                  ; Восстановить адрес сообщения
```

```
88:         call   StrWrite          ; Вывести сообщение на стандартное устройство вывода
89:         call   NewLine           ; Перейти на новую строку
90:         ret                      ; Возврат в вызывающую программу
91:   ENDP DiskErr
92:
93:         END                      ; Конец модуля
```

Использование DiskErr

Для использования модуля DISKERR добавьте в вашу программу команду EXTRN DiskErr:proc. Затем, полагая, что ваша программа называется MYSTUFF.ASM, выполните ассемблирование и компоновку с вашим файлом библиотеки с помощью команд:

```
tasm mystuff
tlink mystuff ,,, mta
```

В коде вашей программы, после обнаружения ошибки, полученной из функции DOS, вызовите для вывода соответствующего сообщения на экран DiskErr. После этого следует либо закончить программу, либо вернуться в меню, как предлагалось раньше. DiskErr не делает ничего, чтобы разрешить причину ошибки, он просто вызывает DOS для дополнительной информации и выводит сообщение. Далее в этой главе вы найдете примеры работы DiskErr (см. программу 9.4, строку 156).

Как работает DiskErr

Кроме выполнения полезной операции, процедура DiskErr демонстрирует интересный технический прием языка ассемблера для выбора элементов из массива объектов переменной длины, в данном случае — из массива ASCIIZ-строк. Сначала сообщения описываются в строках 11–42, давая каждой строке уникальную метку: err01, err02 и т.д. Затем, в строках 44–47, создается второй массив, состоящий из меток каждой строки. Помните, что метки являются адресами, поэтому массив errors просто список 16-битовых адресов смещений каждой строки символов.

Каждый элемент в массиве errors указывает на строку с ошибкой, связанную со значением кода ошибки (0–1Fh), используемого в качестве индекса значения в errors (рис. 9.1.). После получения расширенной информации об ошибке от DOS (строки 68–74) значение кода ошибки умножается на 2 (так как каждый элемент errors является двухбайтовым словом), после чего в строке 82 в di загружается адрес нужной строки. В заключение процедура выводит строку, начиная ее текст с "**ERROR:".

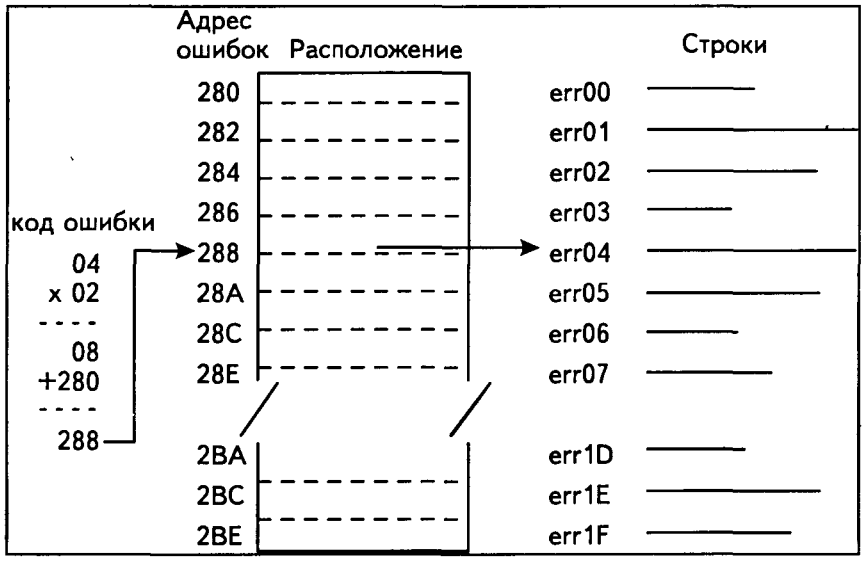


Рис. 9.1. Массив адресов смещений (в центре) определяет индексированные элементы (слева) из массива строк переменной длины (справа). Программа 9.1 использует этот подход в процедуре *DiskErr* для выбора нужной строки из массива ASCIIZ-строк

Создание новых файлов

Создание нового файла аналогично открытию уже существующего файла. Поместите адрес ASCIIZ строки, содержащей имя файла, в ds:dx и установите cx в одно из значений, приведенных в табл. 9.2. Это значение помещается в байт атрибутов файла в каталоге диска, влияя на дальнейшие операции над файлом. Обычно cx устанавливается в 0. После завершения инициализации для создания файла вызовите функцию 03Ch :

```
DATASEG
filename      DB      'C:\newfile.txt',0
handle        DW      ?
CODESEG
mov  ax, @data                ; Поместить в ds
mov  ds, ax                   ; адрес сегмента данных
mov  dx, offset filename      ; Адрес строки в ds:dx
mov  ax, 03Ch                 ; Номер функции DOS создания файла
xor  cx, cx                   ; Определить обычные атрибуты файла
int  21h                      ; Вызов DOS для создания файла
jc   error                    ; Переход в случае ошибки
mov  [Handle], ax             ; Сохранить дескриптор на будущее
```

Как обычно, флаг переноса показывает успех или неудачу при выполнении функции 03Ch. Если cf=1, тогда ax содержит код ошибки — 3, 4 или 5, как показано в табл. 9.1, в противном случае возвращается дескриптор файла, сохраняемый в этом примере в глобальной переменной handle.

Замечание

Одна неприятность, связанная с созданием новых файлов, заключается в том, что DOS не проверяет, есть ли уже файл с таким же именем. Если вы создаете файл с уже существующим именем, содержание старого файла уничтожается или усекается, как написано в некоторых руководствах по DOS. По этой причине нужно проверить, нет ли уже такого файла до вызова функции DOS 03Ch, которая создает новый файл и может разрушить существующие данные. Позже в этой главе будут рассмотрены примеры того, как сделать это на языке ассемблера.

Таблица 9.2. **Атрибуты создания файла**

Значение	Содержание
00	Обычный файл (большинство файлов данных)
01	Только чтение (операции записи не выполняются)
02	Скрытый (невидимый для команды DIR)
04	Системный файл (лучше использовать скрытые файлы)

Чтение командной строки DOS

Традиционная программа DOS дает вам возможность вводить в командной строке одно или более имен файлов, опции и другие данные. Иными словами, DOS позволяет пользователю вводить такие команды, как:

```
c>textsort /d file1.txt file2.txt
```

Вероятно, эта гипотетическая команда запускает программу сортировки текста, которая получает file1.txt, записывает окончательный результат в file2.txt и использует опцию /d для выбора сортировки сверху вниз. Большинство языков высокого уровня обеспечивают подобные этим методы для считывания параметров, расположенных после имени файла и разделенных пробелами. Но в языке ассемблера нет аналогичных встроенных механизмов, и считывание параметров командной строки DOS — более сложный процесс. В этой главе вы выполните ассемблирование программы, которая добавляет эту существенную возможность к вашим программам на языке ассемблера.

Когда command.com загружает exe-файл, в памяти создается 256-байтовый блок, называемый *префиксом программного сегмента* (PSP), который содержит среди другой информации текст, введенный в командной строке DOS после имени программы. Эти символы называются *хвостом команды*. До начала выполнения exe-программ и ds, и es адресуют PSP, 128 байт которого служат для хранения хвоста команды. К сожалению, эта же область — со смещением от 80h до FFh — служит временным дисковым буфером для некоторых функций DOS; поэтому первая задача состоит в том, чтобы скопировать текст из PSP для сохранности в переменную.

Действительное число символов в хвосте команды хранится по смещению 0080h в PSP. Первый символ (если таковой существует) находится в 0081h. Последний символ всегда является символом возврата каретки (0Dh). Программа 9.21, PARAMS.ASM, использует это для извлечения из PSP параметров командной строки и сохраняет параметры в виде прописных ASCIIZ-строк в 128-байтовом буфере в сегменте данных программы. Как и другие программные модули из этой книги, PARAMS требует наличия главной программы для своего исполнения. Ниже я приведу пример такой программы. А сейчас выполните ассемблирование PARAMS и поместите его в ваш файл библиотеки MTA.LIB с помощью команд:

```
tasm /zi params
tlib /E mta -+ params
```

Повторите эти же команды, если будете изменять эту программу. Для экономии пространства на диске и если вы не планируете исполнять программу под Turbo Debugger, удалите опцию /zi.

Программа 9.2. PARAMS.ASM

```
1:    %TITLE "Чтение командной строки DOS -- Copyright (c) 1989,1995 - by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:
7:
8:    TailLen      EQU 0080h          ; Смещение байта с длиной строки параметров
9:    CommandTail  EQU 0081h          ; Смещение параметров
10:
11:
12:         DATASEG
13:
14:   numParams    DW ?               ; Число параметров
15:   params       DB 128 DUP (?)     ; 128-байтовый блок для строки
16.
17:
18:         CODESEG
19:
20:         PUBLIC ParamCount, GetParams, GetOneParam
21:
22:   %NEWPAGE
23:   ;---------------------------------------------------------------
24:   ; Separators Внутренняя процедура для проверки на пробелы, табуляцию и возврат каретки
25:   ;---------------------------------------------------------------
26:   ; Вход:
27:   ; ds:si адрес проверяемого символа
28:   ; Выход:
29:   ; zf = 1 (je) = символ является пробелом, табуляцией или возвратом каретки
30:   ; zf = 0 (jne) = символ не является разделителем
31:   ; Регистры:
32:   ; al
33:   ;---------------------------------------------------------------
34:   PROC Separators
35:         mov   al, [si]           ; Взять символ из ds:si
36:         cmp   al, 020h           ; Символ является пробелом?
37:         je    @@10               ; Если да, то переход
38:         cmp   al, 009h           ; Символ является табуляцией?
39:         je    @@10               ; Если да, то переход
40:         cmp   al, 00Dh           ; Символ является возвратом каретки?
41:   @@10:
42:         ret                      ; Возврат в вызывающую программу
43:   ENDP Separators
44:   %NEWPAGE
45:   ;---------------------------------------------------------------
46:   ; ParamCount Возвращает число параметров
47:   ;---------------------------------------------------------------
48:   ; Вход:
49:   ; нет
50:   ; Выход:
51:   ; dx = число параметров командной строки
52:   ; Замечание: при вызове GetOneParam cx должно быть меньше,
53:   ; чем значение, возвращаемое в dx ParamCount
54:   ; Регистры:
```

```
55:   ; dx
56:   ;---------------------------------------------------------------
57:   PROC ParamCount
58:         mov   dx, [numParams]     ; Получить значение переменной
59:         ret                       ; Возврат в вызывающую программу
60:   ENDP ParamCount
61:   %NEWPAGE
62:   ;---------------------------------------------------------------
63:   ; GetParams Получить параметры командной строки DOS
64:   ;---------------------------------------------------------------
65:   ; Вход:
66:   ; ds = Префикс сегмента программы (PSP)
67:   ; es = Сегмент данных программы
68:   ; Замечание: если вы не изменяли ds, то он адресует PSP
69:   ; в начале выполнения всех .EXE-программ
70:   ; Выход:
71:   ; Глобальные параметры, помещенные в ASCIIZ-строки
72:   ; [numParams] = количество параметров
73:   ; ds = Сегмент данных программы (es не изменяется)
74    ; Регистры:
75:   ; al, bx, dx, si, di, ds
76:   ;---------------------------------------------------------------
77:   PROC GetParams
78:
79:   ;----- Инициализация счетчика (cx) и индексных регистров (si,di)
80:
81:         xor   ch, ch              ; Обнулить верхнюю половину cx
82:         mov   cl, [ds:TailLen]    ; cx = длина параметров
83:         inc   cx                  ; Включить в конец символ возврата каретки
84:         mov   si, CommandTail     ; Поместить адрес параметров в si
85:         mov   di, offset params   ; Поместить адрес назначения в di
86:
87:   ;----- Пропустить начальные пробелы и табуляцию
88:
89:   @@10:
90:         call  Separators          ; Пропустить пробелы и табуляцию
91:         jne   @@20                ; Переход, если пробелов и табуляции нет
92:         inc   si                  ; Пропустить этот символ
93:         loop  @@10                ; Цикл, пока не закончена обработка или cx = 0
94:
95:   ;----- Скопировать параметр строки в глобальную переменную
96:
97:   @@20:
98:         push  cx                  ; Сохранить cx на будущее
99:         jcxz  @@30                ; Пропустить movsb, если count = 0
100:        cld                       ; Автоувеличение si и di
101:        rep   movsb               ; скопировать cx байтов из ds:si в es:di
102:
103:  ;----- Преобразовать пробелы в нули и установить numParams
104:
105:  @@30:
106:        push  es                  ; Сохранить es в стеке
107:        pop   ds                  ; Сделать ds = es
108:        pop   cx                  ; Восстановить длину в cx
109:        xor   bx, bx              ; Инициализировать счетчик параметров
110:        jcxz  @@60                ; Пропустить цикл, если длина = 0
111:        mov   si, offset params   ; Поместить адрес параметров в si
112:  @@40:
113:        call  Separators          ; Проверить на пробелы, табуляцию и возврат каретки
114:        jne   @@50                ; Перейти, если это не разделитель
115:        mov   [byte ptr si], 0    ; Заменить разделитель на нуль
116:        inc   bx                  ; Увеличить число параметров
117:  @@50:
118:        inc   si                  ; Указать на следующий символ
119:        loop  @@40                ; Цикл, пока cx не равно 0
120:  @@60:
121:        mov   [numParams], bx     ; Сохранить число параметров
122:        ret                       ; Возврат в вызывающую программу
123:  ENDP GetParams
124:  %NEWPAGE
125:  ;---------------------------------------------------------------
126:  ; GetOneParam Получить адрес параметра по номеру
127:  ;---------------------------------------------------------------
128:  ; Вход:
```

```
; cx = номер параметра (0 = первый)
; Замечание: cx должно всегда быть меньше значения,
; возвращаемого в dx ParamCount
; Выход:
; di = смещение ASCIIZ-строки с этим параметром
; Регистры:
; al, cx, di
;-------------------------------------------------------------
PROC GetOneParam
        xor     al, al              ; Инициализировать искомое значение 0
        mov     di, offset params   ; Адрес параметров строки
        jcxz    @@99                ; Если number = 0, то выход
        cmp     cx, [numParams]     ; Сравнить number с максимальным значением
        jae     @@99                ; Выход, если number > максимального значения
        cld                         ; Автоувеличение di
@@10.
        scasb                       ; Поиск нулевого ограничителя
        jnz     @@10                ; Повторять, пока не найден
        loop    @@10                ; Повторять для счетчика в cx
@@99:
        ret                         ; Возврат в вызывающую программу
ENDP GetOneParam

        END                         ; Конец модуля
```

Пример выполнения PARAMS

Для понимания того, как работает модуль PARAMS, полезно ассемблировать и исполнить тестовую программу. После этого я объясню, как использовать PARAMS в ваших собственных программах. Сохраните программу 9.3 под именем SHOWPARM.ASM, выполните ассемблирование, компоновку и запустите программу на исполнение с помощью команд:

```
tasm /zi showparm
tlink /v showparm,,, mta
showparm param1 param2 param3
```

Замечание

При выполнении tlink подразумевается, что в MTA.LIB содержатся модули из предыдущих глав — PARAMS, BINASC, STRING, STRIO.

Программа 9.3. SHOWPARM.ASM

```
%TITLE "Вывод параметров командной строки DOS -- Copyright (c) 1989,1995 by Tom Swan"

        IDEAL

        MODEL small
        STACK 256

        DATASEG

exCode          DB 0
string          DB 20 DUP (0)
s1          DB 'Число параметров = ', 0

        CODESEG

;----- Из PARAMS.OBJ
        EXTRN ParamCount:Proc, GetParams:Proc, GetOneParam:Proc

;----- Из BINASC.OBJ, STRINGS.OBJ, STRIO.OBJ
        EXTRN BinToAscDec:Proc, NewLine:Proc, StrWrite:Proc
        EXTRN BinToAscHex:Proc, StrUpper:Proc

Start:
        mov     ax, @data           ; Установить в ax адрес сегмента данных
        mov     es, ax              ; Поместить в es адрес сегмента данных
```

```
26:          call   GetParams          ; Получить параметры, ds = PSP
27:                                    ; Замечание: ds не равно es
28:
29:     ;----- Вывести число параметров
30:
31:          call   NewLine            ; Перейти на новую строку
32:          mov    di, offset s1      ; Адрес строки
33:          call   StrWrite           ; Вывести строку
34:          call   ParamCount         ; Получить число параметров
35:          mov    ax, dx             ; Поместить счетчик в ax
36:          mov    cx, 1              ; Определить по меньшей мере одну цифру
37:          mov    di, offset string  ; Адрес рабочей строки
38:          call   BinToAscDec        ; Преобразовать ax в десятичную цифру
39:          call   StrWrite           ; Вывести число
40:          call   NewLine            ; Перейти на новую строку
41:
42:          xor    cx, cx             ; Инициализировать счетчик нулем
43:     @@10:
44:          call   ParamCount         ; Получить число параметров
45:          cmp    cx, dx             ; Сравнить счетчик с числом параметров
46:          je     Exit               ; Выход, если cx = dx
47:          push   cx                 ; Сохранить cx в стеке
48:          call   GetOneParam        ; Получить адрес одного параметра
49:          call   StrUpper           ; Преобразовать в прописные буквы
50:          call   StrWrite           ; Вывести строку параметров
51:          call   NewLine            ; Перейти на новую строку
52:          pop    cx                 ; Восстановить сохраненное значение cx
53:          inc    cx                 ; Перейти к следующему параметру
54:          jmp    @@10               ; Повторить
55:     Exit:
56:          mov    ah, 04Ch           ; Функция DOS : Выход из программы
57:          mov    al, [exCode]       ; Возвратить значение кода выхода
58:          int    21h                ; Вызов DOS. Останов программы
59:
60:          END    Start              ; Конец программы / точка входа
```

Использование PARAMS

Модуль PARAMS (программа 9.3) содержит три процедуры — ParamCount (45–60), GetParams (62–123) и GetOneParam (125–150), которые можно вызывать для получения параметров из командной строки. Как показано в SHOWPARM (программа 9.3) в строках 24–26, начинайте вашу программу с установки в es сегмента данных программы и сразу вызывайте GetParams:

```
mov  ax, @data        ; Установить в ax адрес сегмента данных
mov  es, ax           ; Установить в es адрес сегмента данных
call GetParams        ; Получить параметры с ds = PSP
```

Обратите внимание на то, что это отличается от обычного начала программы, регистр ds не инициализируется. Так как ds в начале выполнения программы адресует PSP, вы не должны изменять его до вызова GetParams; в противном случае процедура не сможет найти символы хвоста команды. Как дополнительное удобство, GetParams присваивает значение es регистру ds, так что после вызова этой процедуры нет необходимости инициализировать ds.

Замечание

Вследствие эффекта, который GetParams оказывает на ds, вы никогда не должны вызывать эту процедуру больше одного раза в начале программы.

После такой инициализации параметры становятся доступными в виде ASCIIZ-строк. Вызовите ParamCount для установки в dx числа строк в памяти. Так как первый параметр имеет номер 0, то максимальное число параметров всегда на единицу меньше значения, возвращаемого ParamCount в dx, если только dx не равняется 0, что указывает вообще на отсутствие параметров. Для использования отдельных строк параметров поместите номер параметра в cx и вызовите GetOneParam, как это показано в SHOWPARM (строка 48). При этом адрес ASCIIZ-строки этого параметра помещается в di и можно передавать его в любые процедуры, работающие с ASCIIZ-строками. Например, для открытия файла, введенного в качестве первого параметра, можно начать ваш кодовый сегмент так:

```
mov  ax, @data          ; Установить в ax адрес сегмента данных
mov  es, ax             ; Установить в es адрес сегмента данных
call GetParams          ; Получить параметры с ds = PSP
call ParamCount         ; Получить число параметров (dx)
or   dx, dx             ; Число = 0 ?
jz   Exit               ; Выход, если параметры не введены
```

В этой точке программа заканчивается, если ни один параметр не был введен. Если имеется по крайней мере один параметр, программа продолжает свою работу, сначала определяя адрес строки параметров с номером 0, затем передавая этот адрес в функцию DOS 03Dh для открытия файла и обрабатывая при этом возможные ошибки:

```
xor cx, cx              ; Определить параметр с номером 0
call GetOneParam        ; Получить адрес параметра
mov  dx,di              ; Поместить адрес ASCIIZ-строки в ds dx
mov  ah, 03Dh           ; Выбрать функцию DOS 03Dh
int  21h                ; Вызвать DOS для открытия файла
jc   Error              ; Перейти, если обнаружена ошибка
mov  [handle] ax        ; Иначе сохранить дескриптор на будущее
```

Можно также вызвать GetOneParam для получения параметра строки и передачи его адреса в любую из процедур для работы с ASCIIZ-строками из модулей STRINGS и STRIO. Например, для преобразования всех символов параметров в прописные выполните следующий код:

```
        call    ParamCount      ; Получить число параметров
@@10
        or      dx, dx          ; Число параметров = 0 ?
        jz      @@20            ; Перейти, если да
        dec     dx              ; Иначе вычесть 1 из числа
        mov     cx, dx          ; Переслать число параметров в cx
        call    GetOneParam     ; Получить адрес строки параметра
        call    StrUpper        ; Преобразовать символы строки в прописные
        jmp     @@10            ; Повторять до конца
@@20
```

Если вы не сделаете этого, параметры будут записываться вперемежку строчными и прописными буквами, точно так, как вы вводили их в командной строке. Можно извлечь из этого пользу, делая буквы опций чувствительными к верхнему или нижнему регистру. Например, строчная буква в опции /s может приводить к другому результату, чем прописная буква в /S.

Как работает PARAMS

GetParams из модуля PARAMS (программа 9.2, строки 62–123) копирует символы хвоста команды в глобальную переменную params, описанную в строке 15. Перед этим процедура пропускает все пробелы и табуляции (строки 89–93), введенные после имени файла. В этот момент значение в регистре cx равно числу символов в блоке параметров. Если это число равно 0, строка 99 пропускает операцию копирования, выполняемую повторяющейся строковой командой в строке 101. В остальной части процедуры в скопированных символах ищутся разделители параметров — пробелы, табуляции и возвраты каретки, которые преобразуются в нули, превращая таким образом параметры в ASCIIZ-строки.

Замечание

Поскольку GetParams преобразует два соседних пробела, символы табуляции и возврата каретки в нуль, можно случайно ввести параметр нулевой длины, набрав между параметрами в командной строке DOS несколько пробелов. Это не страшно — просто игнорируйте любую нулевую строку параметра, возвращаемую GetOneParam.

GetOneParam (125–150) сканирует блок параметров, находит ASCII-нули и устанавливает в регистре di адрес требуемой строки. В первой части процедуры проверяется, принадлежит ли номер параметра в cx допустимому диапазону. (Если вы зададите номер параметра, выходящий за допустимый диапазон, процедура возвратит адрес первого параметра, если таковой существует). Код в строках 143–147 демонстрирует важный метод языка ассемблера для сканирования объектов переменной длины. Для удобства этот код приводится ниже:

```
        cld             ; Автоувеличение di
@@10
        scasb           ; Поиск нулевого ограничителя
        jnz     @@10    ; Повторять, пока не найден
        loop    @@10    ; Повторять для счетчика в cx
```

Сначала df очищается командой cld, так что scasb автоматически увеличивает di при каждом проходе цикла (Подразумевается, что регистр di адресует первую проверяемую строку параметра.) Команда scasb сравнивает байт в [es:di] со значением в al, предварительно инициализированным в 0 (строка 138). В результате, если сравниваемые байты совпадают, scasb устанавливает флаг нуля zf. Если совпадение не найдено, команда jnz повторяет выполнение scasb; в противном случае программа продолжается по команде цикла. В этой точке cx равняется количеству строк, которое осталось проверить в блоке параметров. Команда loop вычитает 1 из cx; если при этом cx не становится равным 0, выполняется переход на метку @@10: и начинается проверка следующей строки. Если cx становится равным 0, то di адресует первый символ требуемой строки.

Чтение и запись текстовых файлов

При изучении работы с файлами данных в любом новом языке программирования всегда полезно начать с простой программы, которая посимвольно (или побайтово) копирует один файл в другой. Имея такую базовую оболочку, остается только вставить в программу код, модифицирующий эти символы. Можно использовать эту же конструкцию для написания программ, преобразующих символы в прописные или строчные, подсчитывающих количество слов в файле, защищающих данные паролем и выполняющих другие полезные операции.

Программа 9.4, KOPY.ASM, требует от вас введения в командной строке двух имен файлов. Программа открывает и читает первый файл, создает новый файл со вторым именем и копирует каждый байт из первого файла во второй. Если файл со вторым именем уже существует, программа спрашивает разрешения на его удаление. Если вы не ввели строго два параметра, то программа выводит соответствующие инструкции. Это минимальный сервис, которым, вероятно, должны обладать программы. Выполните ассемблирование и скомпонуйте KOPY с помощью следующих команд:

```
tasm /zi kopy
tlink /v kopy,,, mta
```

Не используйте опции /zi и /v, если вы не собираетесь проверять KOPY в Turbo Debugger. Наберите в командной строке DOS KOPY и нажмите <Enter> для вывода инструкций, либо укажите два имени файла для обработки. Например, чтобы скопировать файл ORIGINAL.TXT в новый файл с именем NEWTEXT.TXT, введите:

```
kopy original.txt newtext.txt
```

Программа 9.4. KOPY.ASM

```
1:    %TITLE "Копирование ввода в вывод -- Copyright (c) 1989,1995 by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:    cr            EQU           13      ; ASCII-символ возврата каретки
9:    lf            EQU           10      ; ASCII-символ прогона строки
10:
11:
12:         DATASEG
13:
14:   exCode        DB            0
15:
16:   inFile        DW            0       ; дескриптор входного файла
17:   outFile       DW            0       ; дескриптор выходного файла
18:   oneByte       DB            0       ; байтовая переменная для ввода-вывода
19:
20:   prompt        DB            cr,lf,'Стереть этот файл? (y/n) ', 0
21:   diskFull      DB            cr,lf,'**Ошибка: Диск полон ', 0
22:
23:   notes         DB            cr,lf,'KOPY копирует все байты из одного файла в другой'
24:                 DB            cr,lf,'в качестве демонстрации методов чтения и записи файлов '
25:                 DB            cr,lf,'в языке ассемблера. Программа может быть изменена для'
26:                 DB            cr,lf,'обработки данных в выходном файле,'
27:                 DB            cr,lf,'в то время как эта версия не вносит никаких изменений'
28:                 DB            cr,lf,'в информацию во входном файле. Используйте программу,'
29:                 DB            cr,lf,'указывая два имени файла: первое имя - файл, '
30:                 DB            cr,lf,'который вы хотите прочитать, второе - новый файл,'
31:                 DB            cr,lf,'который вы хотите создать с помощью KOPY :',cr,lf
32:                 DB            cr,lf,'KOPY <входной файл> <выходной файл>',cr,lf, 0
33:
```

```
34:
35:             CODESEG
36:
37:     ;----- Из STRIO.OBJ
38:             EXTRN StrWrite:Proc, NewLine:Proc
39:
40:     ;----- Из DISKERR.OBJ
41:             EXTRN DiskErr:Proc
42:
43:     ;----- Из PARAMS.OBJ
44:             EXTRN GetParams:Proc, ParamCount:Proc, GetOneParam:Proc
45:
46:     Start:
47:
48:     ;----- Инициализация и ввод сообщения, если не введены параметры
49:
50:             mov     ax, @data           ; Установить в ax адрес сегмента данных
51:             mov     es, ax              ; Установить в es адрес сегмента данных
52:             call    GetParams           ; Получить параметры с ds = PSP
53:             call    ParamCount          ; Получить число параметров (dx)
54:             cmp     dx, 2               ; Число = 2?
55:             je      @@10                ; Продолжить, если число параметров = 2
56:             mov     di, offset notes    ; Поместить адрес текста в di
57:             call    StrWrite            ; Вывести информацию
58:             jmp     Exit                ; Конец программы
59:
60:     ;----- Попытка открыть входной файл
61:
62:     @@10:
63:             xor     cx, cx              ; Задать параметр с номером 0
64:             call    GetOneParam         ; Получить адрес строки параметра
65:             mov     dx, di              ; Адресовать имя файла в ds:dx
66:             xor     al, al              ; Разрешить только чтение
67:             mov     ah, 3Dh             ; Функция открытия файла DOS
68:             int     21h                 ; Открыть входной файл
69:             jnc     @@20                ; Продолжить, если нет ошибки
70:             jmp     Errors              ; Иначе обработать ошибку
71:
72:     ;----- Проверка существования выходного файла
73:
74:     @@20:
75:             mov     [inFile], ax        ; Сохранить дескриптор входного файла
76:             mov     cx, 1               ; Задать параметр с номером 1
77:             call    GetOneParam         ; Получить адрес строки параметра
78:             mov     dx, di              ; Поместить адрес имени файла в ds:dx
79:             call    FileExists          ; Существует ли выходной файл?
80:             jc      @@30                ; Переход, если файл не существует
81:             call    StrWrite            ; Вывести имя файла
82:             call    Confirm             ; Иначе получить подтверждение на удаление
83:             je      @@30                ; Продолжить, если разрешение получено
84:             jmp     Exit                ; Иначе закончить программу
85:
86:     ;----- Попытка создать новый файл
87:
88:     @@30:
89:             mov     cx, 1               ; Задать параметр с номером 1
90:             call    GetOneParam         ; Получить адрес строки параметра
91:             mov     dx, di              ; Поместить адрес имени файла в ds:dx
92:             xor     cx, cx              ; Задать обычные атрибуты
93:             mov     ah, 3Ch             ; Функция DOS создания файла
94:             int     21h                 ; Создать выходной файл
95:             jnc     @@40                ; Продолжить, если нет ошибки
96:             jmp     Errors              ; Иначе обработать ошибку
97:     @@40:
98:             mov     [outFile], ax       ; Сохранить дескриптор выходного файла
99:
100:    ;----- С этого момента входной и выходной файлы открыты и
101:    ; их дескрипторы записаны в inFile и outFile. Следующий шаг -
102:    ; чтение из входного файла и побайтовая запись в выходной файл
103:    ;
104:
105:    @@50:
106:            mov     ah, 3Fh             ; Функция DOS чтения файла
107:            mov     bx, [inFile]        ; Поместить в bx дескриптор входного файла
```

```
108:        mov   cx, 1               ; Задать для чтения один байт
109:        mov   dx, offset oneByte  ; Поместить адрес переменной в ds:dx
110:        int   21h                 ; Вызов DOS для чтения из файла
111:        jnc   @@60                ; Переход, если нет ошибки
112:        jmp   Errors              ; Иначе обработать ошибку
113:  @@60:
114:        or    ax, ax              ; Проверка конца входного файла
115:        jz    @@80                ; ax = 0 = конец файла; переход
116:        mov   ah, 40h             ; Функция DOS записи в файл
117:        mov   bx, [outFile]       ; Поместить в bx дескриптор выходного файла
118:        mov   cx, 1               ; Определить один байт для записи
119:        mov   dx, offset oneByte  ; Поместить адрес переменной в ds:dx
120:        int   21h                 ; Вызов DOS для записи в файл
121:        jnc   @@70                ; Переход, если нет ошибки
122:        jmp   Errors              ; Иначе обработать ошибку
123:  @@70:
124:        or    ax, ax              ; Проверка на переполнение диска
125:        jnz   @@50                ; Повторять для следующего байта
126:
127:  ;----- Обработка в случае переполнения диска
128:
129:        mov   di, offset diskFull ; Адрес сообщения о переполнении диска
130:        call  StrWrite ; Вывести сообщение
131:
132:  ;----- Закрыть входной и выходной файлы. Это делать необязательно, так как при
133:  ; окончании программы с помощью функции 04Ch закрываются все открытые
134:  ; файлы. Замечание: ошибки обрабатываются только после закрытия выходного
135:  ; файла.
136:
137:  @@80:
138:        mov   bx, [inFile]        ; Получить дескриптор входного файла
139:        mov   ah, 3Eh             ; Функция DOS закрытия файла
140:        int   21h                 ; Закрыть входной файл
141:        mov   bx, [outFile]       ; Получить дескриптор выходного файла
142:        mov   ah, 3Eh             ; Функция DOS закрытия файла
143:        int   21h                 ; Закрыть выходной файл
144:        jnc   Exit                ; Выход, если нет ошибки
145:        jmp   Errors              ; Иначе обработать ошибку
146:  Exit:
147:        mov   ah, 04Ch            ; Функция DOS : Выход из программы
148:        mov   al, [exCode]        ; Возвратить значение кода выхода
149:        int   21h                 ; Вызов DOS. Останов программы
150:
151:  ;----- Сюда выполняются переходы при ошибках ввода-вывода, что вызывает
152:  ; завершение программы после вывода сообщения
153:
154:  Errors:
155:        mov   [exCode], al        ; Сохранить код ошибки
156:        call  DiskErr             ; Вывести сообщение об ошибке
157:        jmp   Exit                ; Выход из программы
158:
159:  %NEWPAGE
160:  ;---------------------------------------------------------------
161:  ; FileExists Проверка на существование файла
162:  ;---------------------------------------------------------------
163:  ; Вход:
164:  ; ds:dx = адрес ASCIIZ строки с именем файла
165:  ; Выход:
166:  ; cf = 0 (jnc) = Файл с таким именем существует
167:  ; cf = 1 (jc) = Файл с таким именем не существует
168:  ; Регистры: ax, bx
169:  ;---------------------------------------------------------------
170:  PROC FileExists
171:        xor   al, al              ; Разрешить только чтение
172:        mov   ah, 3Dh             ; Функция DOS открытия файла
173:        int   21h                 ; Вызов DOS для открытия файла
174:        jc    @@99                ; Выход - файл не существует
175:        mov   bx, ax              ; Скопировать дескриптор в bx
176:        mov   ah, 3Eh             ; Функция DOS закрытия файла
177:        int   21h                 ; Закрыть файл
178:        clc   ; Очистить флаг переноса (файл существует)
179:  @@99:
180:        ret   ; Возврат в вызывающую программу
181:  ENDP FileExists
```

```
182:  %NEWPAGE
183:  ;---------------------------------------------------------------
184:  ; Confirm Получить подтверждение Yes/No от пользователя
185:  ;---------------------------------------------------------------
186:  ; Вход:
187:  ; нет
188:  ; Выход:
189:  ; zf = 0 (jnz) = пользователь набрал N или n
190:  ; zf = 1 (jz) = пользователь набрал Y или y
191:  ; Registers: ax, cx, di
192:  ;---------------------------------------------------------------
193:  PROC Confirm
194:        mov   di, offset Prompt   ; Адрес строки подсказки
195:        call  StrWrite            ; Вывести сообщение
196:        mov   ah, 1               ; Функция DOS GetChar
197:        int   21h                 ; Получить ответ пользователя
198:        cmp   al, 'Y'             ; Сравнить с Y
199:        je    @@99                ; Выход, если символ = Y
200:        cmp   al, 'y'             ; Сравнить с y
201:        je    @@99                ; Выход, если символ = y
202:        cmp   al, 'N'             ; Сравнить с N
203:        je    @@20                ; Обработать ответ No
204:        cmp   al, 'n'             ; Сравнить с n
205:        jne   Confirm             ; Повторить, если не Y, y, N, n
206:  @@20:
207:        cmp   al, '@'             ; Сбросить флаг нуля (zf = 0)
208:  @@99:
209:        ret   ; Возврат в вызывающую программу
210:  ENDP Confirm
211:
212:        END   Start               ; Конец программы / точка входа
```

Как работает KOPY.ASM

KOPY.ASM иллюстрирует посимвольную обработку файлов, копируя содержание одного дискового файла в другой. Так как для этого требуются многочисленные вызовы DOS, программа работает медленнее, чем команды DOS copy и xcopy, выполняющие аналогичные действия. Хотя вы и можете использовать KOPY в качестве утилиты, программа полезнее как оболочка для написания новых программ, обрабатывающих символы в файле. Например, скопируйте KOPY.ASM в новый файл с именем UPCASE.ASM и вставьте между строками 115 и 116 код, модифицирующий значение, содержащееся в переменной oneByte:

```
      mov     al, [oneByte]     ; Получить введенный байт
      cmp     al, ' a '         ; Байт >= ' a ' ?
      jb      @@Continue        ; Перейти, если байт < ' a '
      cmp     al, ' z '         ; Байт <= ' z ' ?
      ja      @@Continue        ; Перейти, если байт > ' z '
      sub     al, 32            ; Преобразовать строчный символ в прописной
      mov     [oneByte], al     ; Снова запомнить символ в переменной
@@Continue:
```

Возможно, вы захотите изменить инструкции к программе, расположенные по метке notes (строки 23–32). После внесения этих изменений выполните ассемблирование и компоновку программы с помощью команд:

```
tasm upcase
tlink upcase,,, mta
```

Строки 74–84 показывают, как проверять, существует ли файл, предотвращая неприятности, которые могут произойти, если вы случайно зададите неправильное имя выходного файла. Подпрограмма FileExists (строки 160–181) пытается открыть файл, возвращая, если не происходит ошибки, очищенным флаг переноса. В противном случае устанавливается флаг переноса, показывающий, что файл не может быть найден. Процедура с закрытием файла аккуратна в случае успешной операции открытия (строки 176–177). Если этого не сделать, повторяющиеся вызовы FileExists могут при известных обстоятельствах привести к тому, что для DOS не останется свободных дескрипторов.

Другая подпрограмма, Confirm (строки 183–210), выводит сообщение и ожидает вашего ответа — Y для Yes и N для No, подтверждающего, хотите вы или нет стереть существующий файл.

После предварительных шагов — получения параметров с именами файлов, проверки файла на существование и получения вашего разрешения на уничтожение старых данных, шаги, занимающие большую часть программы (в

строках 105–125), выполняют собственно копирование, вызывая функцию DOS 03Fh для чтения из входного файла и функцию 040h для записи в выходной файл. Внимательно изучите эту часть на предмет обработки ошибок с помощью вызова DiskErr (строка 156) из модуля DISKERR. Обратите также внимание на то, как в строках 124–130 обрабатывается сложная ситуация с переполнением диска.

Чтение и запись файлов данных

Конечно, текстовые файлы являются просто разновидностью файлов данных, которые могут содержать любую информацию — записи с именами и адресами, статистические данные и т.д. В языке программирования ассемблера содержимое файлов не определяется, и вы ответственны за написание программ, выбирающих нужные методы для чтения и записи данных в различных форматах. Но, даже несмотря на это, для обработки любых файлов, независимо от типов находящихся в них данных, можно использовать обсуждавшиеся ранее функции DOS.

Тем не менее существует большая разница между побайтовым или посимвольным чтением и записью файла и обработкой файлов, содержащих многобайтовые записи. Во многих случаях программы должны иметь возможность читать и записывать такие записи в произвольном порядке, например, позволять редактировать запись с номером 1068 из 3277 записей, содержащихся на диске, без необходимости копировать весь файл в новое место. В общем случае это требует двух новых концепций ввода-вывода, добавляемых к списку в начале главы.

- Операция поиска позиционирует внутренний указатель позиции на первом байте записи в файле.
- Чтение и запись определенного количества байтов после операции поиска влияет только на одну запись в файле, оставляя другие записи неизменными.

Поиск в файле просто означает позиционирование внутреннего указателя файла DOS, который указывает DOS, откуда читать и куда записывать данные в каждом открытом файле. При обработке файлов в языке ассемблера важно помнить правило о том, что DOS всегда ищет позицию байта, независимо от того, сколько байтов занимает каждая запись. Поэтому для позиционирования указателя файла на начало многобайтовой записи прежде всего нужно умножить размер записи на ее номер. (Первая запись в файле имеет номер 0.) Подразумевая, что размер записи содержится в переменной с именем recSize, а номер — в ax, начните так:

```
mov  cx, [recSize]          ; cx = размер записи в байтах
mul  cx                     ; ax:dx <- ax * cx
```

Команда mul перемножает номер записи в ax на размер записи из cx, записывая 32-битовый результат в ax (нижняя половина) и dx (верхняя половина). Эти значения затем должны преобразовываться в cx (верхняя половина) и dx (нижняя половина) для удовлетворения требований функции поиска DOS 042h:

```
mov  cx, dx                 ; cx <- старшее слово результата
mov  dx, ax                 ; dx <- младшее слово результата
mov  ah, 042h               ; Функция DOS поиска в файле
mov  al, 0                  ; Поиск от начала файла
mov  bx, [handle]           ; Поместить дескриптор файла в bx
int  21h                    ; Позиционировать указатель файла
jz   Error                  ; Обработка ошибки
```

После выполнения этих шагов можно читать или записывать в новую позицию файла. Для чтения в переменную с именем buffer можно выполнить следующий код:

```
mov  ah, 03Fh               ; Функция DOS чтения файла
mov  bx, [handle]           ; Поместить дескриптор файла в bx
mov  cx, [recSize]          ; cx = количество считываемых байтов
mov  dx, offset Buffer      ; ds:dx = адрес назначения
int  21h                    ; Прочитать cx байтов из файла
jc   Error                  ; Обработка ошибки
```

Поскольку при чтении (и записи) указатель файла переходит на следующую запись, вам не нужно выполнять повторный поиск для чтения нескольких последовательных записей, начиная с определенной позиции. Запись отдельных записей выполняется аналогично предыдущему примеру, но вместо функции 03Fh вызывается функция 040h. Кроме того, некоторые из показанных здесь шагов на практике могут быть необязательными. Например, bx после операции поиска уже равняется дескриптору файла, так что нет необходимости снова перегружать этот регистр.

Вы также можете изменить работу функции поиска DOS 042h. Если al=0, как это было в предыдущем примере, тогда значение позиции байта в cx:dx рассматривается как абсолютное — другими словами, относительно начала файла. Если al=1, тогда значение позиции представляет смещение относительно текущей позиции. Можно использовать это для перехода на следующую запись:

```
xor  cx, cx             ; Обнуление значения старшего полуслова
mov  dx, [recSize]      ; cx:dx = размер записи в байтах
```

```
mov  ah, 042h          ; Функция DOS поиска в файле
mov  al, 1             ; Поиск от текущей позиции
mov  bx, [handle]      ; Поместить дескриптор файла в bx
int  21h               ; Позиционировать указатель файла
jc   Error             ; Обработка ошибки
```

Если al=2, то поиск выполняется от конца файла. Это позволяет легко позиционировать указатель на конец файла, может быть, для подготовки к добавлению в конец новых данных:

```
xor  cx, cx            ; Обнулить значение старшего полуслова
xor  dx, dx            ; Нулевое значение младшего полуслова
mov  ah, 042h          ; Функция DOS поиска в файле
mov  al, 2             ; Поиск конца файла
mov  bx, [handle]      ; Поместить дескриптор файла в bx
int  21h               ; Позиционировать указатель файла
jc   Error             ; Обработка ошибки
```

Чтение каталога диска

Две функции DOS упрощают чтение каталога. Основной план состоит в том, чтобы вызвать первую функцию для начала сканирования каталога, а затем повторно вызывать вторую функцию для просмотра оставшейся части каталога, находя совпадения с *шаблонами* типа *.*, *.PAS или MYFILE.???, аналогичными именам файлов, и шаблонами, которые можно использовать в команде DOS-dir. В программах на языке ассемблера эти строки для удобства хранятся в ASCIIZ-формате.

Программа 9.5, DR.ASM, показывает, как читать каталог диска, выводя имена файлов в стиле команды dir /w. Так же как и DIR, эта программа позволяет вам пользоваться шаблонами. Например, набрав dr *.asm, вы получите список всех файлов .ASM в текущем каталоге. Просто dr выводит все файлы. Выполните ассемблирование и компоновку DR.ASM с помощью команд:

```
tasm /zi dr
tlink /v dr,,, mta
```

Команда tlink подразумевает, что в файле библиотеки MTA.LIB содержатся объектные модули из предыдущих глав — PARAMS, STRINGS и STRIO.

Программа 9.5. DR.ASM

```
1:  %TITLE "Вывод каталога диска -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:  FileName EQU 30                                ; Смещение имени файла в dirData
9:
10:
11:         DATASEG
12:
13:   exCode       DB          0
14:
15:   defaultSpec  DB          '*.*', 0           ; Стандартный ASCIIZ шаблон
16:   DTAseg       DW          ?                  ; Сегмент для DTA
17:   DTAofs       DW          ?                  ; Смещение для DTA
18:   dirData      DB          43 DUP (?)         ; Место для записи содержимого каталога
19:
20:
21:         CODESEG
22:
23:   ;----- Из PARAMS.OBJ
24:         EXTRN GetParams:Proc, GetOneParam:Proc, ParamCount:Proc
25:
26:   ;----- Из STRINGS.OBJ, STRIO.OBJ
27:         EXTRN StrLength:Proc, StrWrite:Proc, NewLine:Proc
28:
29:   Start:
30:         mov   ax, @data          ; Установить в ax адрес сегмента данных
31:         mov   es, ax             ; Установить в es адрес сегмента данных
```

```
32:         call   GetParams           ; Получить параметры с ds = PSP
33:         call   NewLine             ; Перейти на новую строку
34:         call   ParamCount          ; Получить число параметров (dx)
35:         mov    di, offset defaultSpec     ; Адрес стандартной строки для поиска
36:         or     dx, dx              ; dx = 0?
37:         jz     @@10                ; Переход, если dx (число параметров) = 0
38:         xor    cx, cx              ; Иначе определить параметр #0
39:         call   GetOneParam         ; Получить адрес параметра
40:   @@10:
41:         mov    bx, offset Action   ; Адрес обрабатывающей подпрограммы
42:         call   DirEngine           ; Просмотреть каталог
43:   Exit:
44:         call   NewLine             ; Перейти на новую строку
45:         mov    ah, 04Ch            ; Функция DOS : Выход из программы
46:         mov    al, [exCode]        ; Возвратить значение кода выхода
47:         int    21h                 ; Вызов DOS. Останов программы
48:
49:   %NEWPAGE
50:   ;---------------------------------------------------------------
51:   ; DirEngine Просмотр каталога "engine"
52:   ;---------------------------------------------------------------
53:   ; Вход:
54:   ; cs:bx = адрес подпрограммы
55:   ; ds:di = адрес ASCIIZстроки для поиска (т.е. *.ASM)
56:   ; Выход:
57:   ; при каждом совпадении вызывается процедура по адресу cs:bx
58:   ; Регистры:
59:   ; ax, cx, dx + другие, изменяемые процедурой в cs:bx
60:   ;---------------------------------------------------------------
61:   PROC DirEngine
62:
63:   ;----- Получить текущий Disk Transfer Address (DTA)
64:
65:         push   es                  ; Сохранить регистры, изменяемые
66:         push   bx                  ; функцией DOS 2Fh
67:         mov    ah, 2Fh             ; Функция DOS получения DTA
68:         int    21h                 ; Получить текущий DTA
69:         mov    [DTAseg], es        ; Сохранить адрес сегмента
70:         mov    [DTAofs], bx        ; Сохранить адрес смещения
71:         pop    bx                  ; Восстановить регистры
72:         pop    es
73:
74:   ;----- Установить новый DTA в глобальной 43-байтовой переменной dirData
75:
76:         mov    dx, offset dirData  ; Поместить адрес переменной в ds:dx
77:         mov    ah, 1Ah             ; Функция DOS установки DTA
78:         int    21h                 ; Установить новый DTA
79:
80:   ;----- Проверить каталог на совпадение со строкой из ds:dx
81:
82:         mov    ah, 4Eh             ; Функция DOS первого поиска
83:         mov    cx, 10h             ; Атрибут--files + subdirs
84:         mov    dx, di              ; Поместить адрес строки в ds:dx
85:         jmp    short @@20          ; Пропустить следующее присваивание ah
86:   @@10:
87:         mov    ah, 4Fh             ; Функция DOS продолжения поиска
88:   @@20:
89:         int    21h                 ; Найти первое/следующее вхождение
90:         jc     @@99                ; Выход по ошибке или окончанию
91:         call   bx                  ; Вызов процедуры Action
92:         jmp    @@10                ; Повторить
93:
94:   ;----- Восстановить первоначальный адрес DTA
95:
96:   @@99:
97:         push   ds                  ; Сохранить текущий ds
98:         mov    ds, [DTAseg]        ; Присвоить старый адрес DTA
99:         mov    dx, [DTAofs]        ; ds:dx
100:        mov    ah, 1Ah             ; Функция DOS установки DTA
101:        int    21h                 ; Вернуться к старому DTA
102:        pop    ds                  ; Восстановить ds
103:        ret                        ; Возврат в вызывающую программу
104:  ENDP DirEngine
105:  %NEWPAGE
```

```
106:  ;-------------------------------------------------------------
107:  ; Action Вызывается для каждого члена каталога
108:  ;-------------------------------------------------------------
109:  ; Вход:
110:  ; dirData = каталог (в стиле DOS)
111:  ; Выход:
112:  ; выводится имя файла
113:  ; Регистры:
114:  ; ah, dl, cx, di
115:  ;-------------------------------------------------------------
116:  PROC Action
117:        mov   di, offset dirData + FileName
118:        call  StrWrite
119:        call  StrLength
120:        sub   cx, 16
121:        neg   cx
122:  @@10:
123:        mov   ah, 2
124:        mov   dl, ' '
125:        int   21h
126:        loop  @@10
127:        ret                        ; Возврат в вызывающую программу
128:  ENDP Action
129:
130:        END   Start                ; Конец программы / точка входа
```

Как работает DR

DR иллюстрирует несколько новых методов программирования на языке ассемблера. В строке 41 в регистр bx помещается адрес смещения подпрограммы, передаваемый в DirEngine (строки 50–104). Затем в строке 91 DirEngine вызывает эту подпрограмму с помощью команды

```
call bx              ; Вызов программы из cs:bx
```

Такой вызов подпрограммы по своему результату не отличается от более известного способа, при котором вы определяете метку подпрограммы как непосредственное значение. Однако этот метод с bx дает возможность передавать в подпрограмму адреса различных других подпрограмм. В данной программе такой способ позволяет корректировать дальнейшие действия при совпадении в каталоге. Этот пример показывает, что написание подпрограмм, которые в качестве входных параметров позволяют использовать адреса других процедур, является полезным методом программирования на языке ассемблера.

Большая часть подпрограммы DirEngine связана с сохранением и установкой Disk Transfer Address (DTA), позиции в памяти, которую DOS применяйте для некоторых недескрипторных файловых операций. При чтении каталога DOS копирует отдельные составляющие каталога в 43-байтовый DTA, который вы должны ему предоставить. Изучив комментарии к DirEngine, вы должны разобраться в том, как сохраняется текущий DTA, что в принципе необязательно делать в этом примере, так как по окончании программы можно не восстанавливать первоначальный DTA. Тем не менее в больших программах желательно сохранять DTA, как это показано здесь.

Подпрограмма Action (строки 106–128) выводит одно имя файла из DTA, заполненного DirEngine. На рис. 9.2 показан формат полей каталога в этой 43-байтовой переменной. В нашем случае программе требуется только одно поле со смещением 30 (в десятичной системе), указывающее на первый байт ASCIIZ строки, содержащей имя файла Для вывода строки необходимо установить в di адрес смещения в dirData переменной DTA, вычисляемый сложением известного смещения имени файла (30) с адресом смещения dirData:

```
mov di, offset dirData + FileName
```

Затем имя файла выводится с помощью StrWrite (из модуля STRIO.OBJ). Для выравнивания столбцов три следующие команды вычисляют число пробелов, которое необходимо вставить между последним символом каждого имени файла и началом следующей колонки:

```
call StrLength       ; Определить длину строки имени файла
sub  cx, 16          ; Вычесть 16
neg  cx              ; Определить абсолютное значение
```

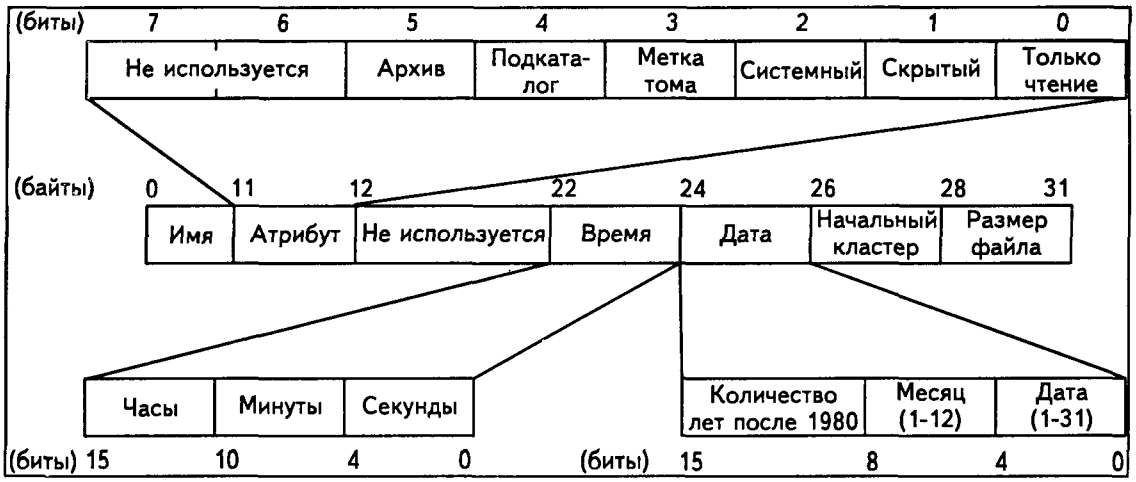


Рис. 9.2. Формат записи каталога

Существуют и другие способы установки в cx необходимого числа пробелов, которые нужны для выравнивания колонок переменной длины, но данный прием срабатывает всегда. Сначала отнимите длину объекта переменной длины (длину имени файла в данном случае) от фиксированной ширины колонки — здесь 16. Принимая во внимание, что длина имени файла меньше 16, в результате получаем отрицательное число в дополнительном коде. Путем отрицания этого результата находится абсолютное значение — число пробелов, выводимых для выравнивания следующей справа колонки.

Резюме

Дескрипторы файлов впервые появились в версии DOS 2.0, заменив прежние и больше не рекомендуемые методы FCB для ввода-вывода дисковых файлов. Дескрипторы упрощают обработку файлов, устраняя необходимость создания и поддержки записей FCB, содержащих информацию, редко непосредственно используемую в программах.

Чтобы данные, содержащиеся в файлах, стали доступны программам, файлы должны открываться. При создании нового файла стираются все данные, хранящиеся в файле под тем же именем. Данные, записываемые и читаемые с диска, временно содержатся в буферах памяти — вы не должны считать, что операция записи на диск действительно помещает на него данные. При закрытии файла все данные из буфера записываются на диск, пополняется дисковый каталог и освобождается дескриптор файла. Текущая позиция показывает на место в файле, где будет выполняться следующая операция чтения или записи. Это — важные и универсальные понятия файлового ввода-вывода.

Программа может открывать до 20 файлов, пока не будет достигнуто суммарное число, определенное командой files=n в файле CONFIG.SYS, вплоть до максимального количества в 255 дескрипторов для всех активных программ. Так как DOS резервирует дескрипторы от 0 до 4 для стандартного ввода-вывода, программа обычно ограничена одновременным открытием 15 файлов. Можно несколько увеличить этот предел, закрыв один или более из пяти стандартных дескрипторов.

Поскольку данные записываются на диск через буферы памяти, единственный метод, гарантирующий сохранение всей информации на диске, — это закрытие файла. Команда DOS "Очистить буфер" для этих целей непригодна. По окончании программы с помощью функции DOS 04Ch автоматически закрываются все открытые файлы; поэтому можно спокойно заканчивать программу, оставляя открытые файлы.

В языке ассемблера нужно аккуратно обрабатывать ошибки диска, так как в нем, в отличие от языков высокого уровня, отсутствуют встроенные возможности определения ошибок и выполнения в этих случаях соответствующих действий. При записи на диск важно проверять условие переполнения, которое DOS не считает ошибкой. Доступна также расширенная информация об ошибке — либо через использование модуля DISKERR из этой главы, либо через непосредственный вызов DOS. В модуле DISKERR показано также, как создавать массивы из объектов переменной длины, таких как строки символов.

Традиционные программы DOS позволяют набирать в командной строке параметры, передавая в нее опции, имена файлов и другую информацию. Для преобразования параметров в удобные для использования ASCIIZ строки можно использовать модуль PARAMS из этой главы.

Посимвольная обработка текстовых файлов осуществляется простым вызовом функций DOS для чтения ввода и записи вывода. Эти же функции можно использовать для обработки многобайтовых записей в файлах данных иных типов. С помощью функции поиска DOS вы можете оперировать отдельными записями, не оказывая при этом влияния на другие данные файла.

Другая пара функций DOS позволяет читать каталоги диска, находя имена файлов, соответствующие шаблонам типа *.TXT. Каждый член каталога загружается DOS в область памяти, называемую DTA, откуда можно извлечь информацию о каталоге.

Упражнения

9.1. К чему приводит закрытие файла?

9.2. К чему приводит открытие файла?

9.3. Напишите подпрограмму, которая запрашивает имя файла и по нажатию <Enter> открывает его (если существует).

9.4. Напишите подпрограмму, которая сбрасывает любые данные из памяти на диск. На вход подпрограммы должны поступать имя файла и дескриптор файла.

9.5. Напишите подпрограмму для считывания по номеру записи из n байтов из открытого файла данных.

9.6. Напишите подпрограмму, которая возвращает следующую после текущей запись из n байт из открытого файла данных.

9.7. Напишите подпрограмму, возвращающую установленным флаг нуля, если среди параметров, введенных в командной строке DOS, имеются опции -d или /Z.

9.8. Напишите программу, которая разделяет имя файла от его расширения, возвращая единственную строку длиной строго в 12 символов. (Для примера этого формата наберите DIR /w в строке DOS.) Измените DR, чтобы использовать эту новую подпрограмму для вывода имен файлов в новом формате.

9.9. Какие команды можно вставить в программу-оболочку KOPY.ASM, чтобы удалить из текстового файла все управляющие символы (кроме возврата каретки и перехода на новую строку)? Можно заменить управляющие коды пробелами.

9.10. Модифицируйте копию DR.ASM, чтобы выводить все файлы .COM и .EXE из текущего каталога диска.

Задания

9.1. Перепишите модуль PARAMS так, чтобы из хвоста команды удалялись строки параметров нулевой длины.

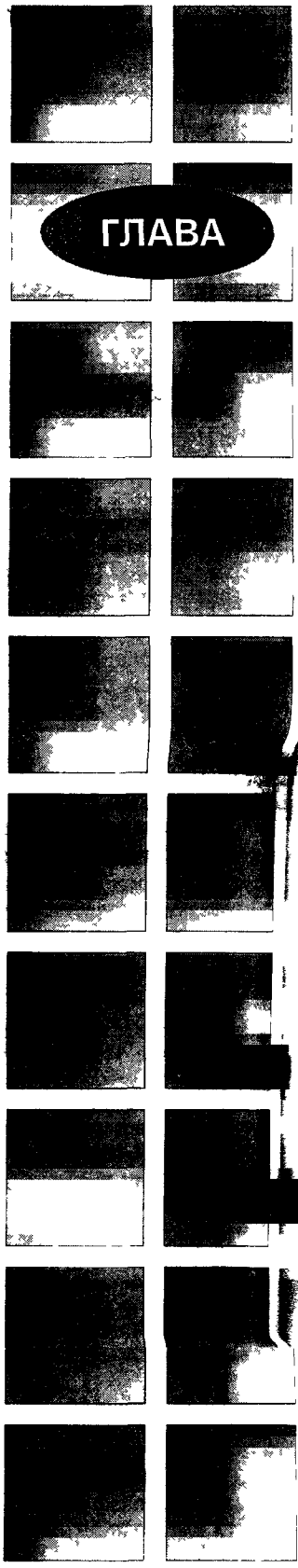
9.2. Напишите новую версию KOPY.ASM, которая читает n байтов из файла в большую переменную программы подходящего размера, например длиной в 256 или 512 байт. Затем создайте подпрограмму, возвращающую символы из вашего буфера. Как это повлияло на скорость KOPY?

9.3. Опишите, как можно создать программу, работающую одновременно более чем с 15 файлами или большим количеством файлов, разрешенных DOS. Какие структуры данных и переменные необходимы этой программе? Какие потребуются подпрограммы?

9.4. Преобразуйте DR.ASM в библиотечный модуль, который может использоваться любой программой.

9.5. Поскольку параметры командной строки обычно короткие, 128-байтовый буфер param из программы 9.2, PARAMS.ASM, редко заполняется полностью. Разработайте план для ограничения размера этого буфера только пространством, необходимым для записи хвоста команды, уменьшив этим бесполезное пространство в конце буфера.

9.6. Напишите подпрограммы для чтения и записи за один раз ASCIIZ-строк, распознавая в текстовом файле коды возврата каретки и перехода на новую строку в качестве разделителей строк и преобразуя их в ASCIIZ-строки.



ГЛАВА 10

Обработка прерываний

- Мы прерываем эту программу...
- Написание процедур обработки прерываний
- Маскируемые и немаскируемые прерывания
- Векторы прерываний и микросхема 8259
- Почему команда hlt не останавливает программу
- Обработка прерываний
- Внедрение в прерывание часов PC
- Команда окончания прерывания
- Прерывания и стеки
- Использование команд int и into
- Отлавливание прерывания ошибки деления
- Исправление ошибки деления
- Установка обработчика ошибки деления
- Установка кода TSR в памяти
- Прерывания, управляющие последовательным обменом
- Отладка с прерываниями
- Пошаговое выполнение
- Резюме
- Упражнения
- Задания

Мы прерываем эту программу...

Прерывание — это процесс, который временно останавливает работающую программу, выполняет подпрограмму — *процедуру обработки прерываний* (interrupt service routine — ISR), а затем запускает остановленную программу, как будто ничего не произошло. Это действие похоже на прерывание телевизионной программы для "важного сообщения", показ которой возобновляется после того, как диктор прочитает новости.

В компьютерном программировании прерывания помогают устранить повторяющийся *опрос* периферийных устройств, таких как клавиатура, принтер, световое перо, чтобы узнать, ожидают ли они ввода, и поступили ли на них данные, требую-

щие обработки. Вместо этого такие устройства могут генерировать сигнал прерывания, который автоматически вызывает соответствующее ISR, обслуживающее требования данного устройства. Таким образом устройства могут использовать прерывания для запуска своих собственных программ, независимо от работы остального программного обеспечения. При программировании процессора 8086 это классическое определение распространяется на два вида сигналов прерывания:

- внешние прерывания
- внутренние прерывания.

Внешние прерывания происходят, когда устройство, присоединенное к процессору, само генерирует сигнал прерывания. *Внутренние прерывания* исходят от процессора в двух случаях: в результате выполнения программной команды int или при определенных условиях, таких как деление на 0 при выполнении команды div, которая генерирует какой-то сигнал прерывания для этого типа ошибки. В добавление к этому, внутренние прерывания int — называемые также *программными прерываниями* — могут эмулировать прерывания внешние. Это полезный метод для отладки внешних прерываний.

Написание процедур обработки прерываний

ISR может делать все то же самое, что и любая другая процедура, написанная на ассемблере. ISR — не что иное, как специальный вид подпрограммы, вызываемый прерыванием так, как это выше описано. Отложив на время рассмотрение некоторых тонких моментов, сформулируем четыре основных правила, которых стоит придерживаться при написании своих собственных процедур обработки прерываний:

- Сохраняйте регистры в начале процедуры
- Выполняйте sti для генерирования прерываний вне данного ISR
- Восстанавливайте регистры в конце процедуры
- Последней командой выполняйте iret

Внешние прерывания могут возникнуть в любой момент времени; поэтому особенно важно, чтобы внешние ISR не вносили никаких изменений в состояние регистров. Заранее неизвестно, какие регистры могут использоваться, когда произойдет внешнее прерывание; следовательно, забывчивость при сохранении и восстановлении значений регистров, измененных внутри обрабатывающей процедуры, скорее всего, приведет к гибельным последствиям для работы остального программного обеспечения. Внутренние IRS могут менять значения регистров, поскольку программы имеют больший контроль над ними, когда возникает прерывание этого типа. (Внутренние ISR работают аналогично подпрограммам.) Выполняйте команду sti, устанавливающую флаг возможности прерывания (if), если вы хотите, чтобы другие прерывания имели возможность прерывать данную процедуру обработки. Иначе новые прерывания не будут вызываться до тех пор, пока ваша процедура не выполнит команду iret (возврат из прерывания), которой должна оканчиваться любая процедура обработки прерываний.

Замечание

Несмотря на то что прерывания могут возникать в любое время, они обрабатываются процессором только между другими командами. Иными словами, если прерывание возникло в процессе выполнения команды mul, которая может потребовать для выполнения 139 машинных циклов, mul завершится перед тем, как прерывание будет распознано и вызвано. Как результат этой потенциальной задержки, а также из-за того, что большинство инструкций требует разного числа циклов для выполнения, даже наиболее регулярные сигналы прерываний, скорее всего, будут обрабатываться в нерегулярные промежутки времени. Повторяемые строковые команды, такие как rep movsb, могут быть прерваны между повторениями.

Маскируемые и немаскируемые прерывания

Процессоры семейства 8086 имеют два входных контакта (или входные линии), к которым могут быть подключены устройства, генерирующие внешние прерывания:

- Маскируемые прерывания (INTR)
- Немаскируемые прерывания (NMI)

Линия INTR используется большинством устройств, генерирующих прерывания, для того чтобы сигнализировать процессору о необходимости обслуживания устройства. Команды cli и sti воздействуют на прерывания, поступающие по этой линии. Выполнение cli предотвращает (*маскирует*) распознавание процессором прерываний INTR. Выполнение sti вновь разрешает процессору распознавать сигналы прерываний INTR. Обе эти команды никак не влияют на вторую линию прерываний NMI, которая не может быть заблокирована. Обычно NMI зарезервирована для контроля за исключительными событиями и выполняет некоторые команды, когда обнаружено падение напряжения, система "зависает" при возникновении ошибки в памяти и т.п. Первоначальная конструкция IBM PC предполагала NMI для обработки ошибки четности в памяти, которая возникает, когда детектируется неисправный бит памяти. В настоящее время и другие устройства задействуют NMI, усложняя тем самым обработку прерываний NMI.

Команды sti и cli никак не действуют на программные прерывания — те, которые генерируются командой int в программе или при возникновении ошибки деления и других подобных обстоятельствах. В независимости от установки флага if, вы всегда можете выполнить int для того, чтобы заставить работать процедуру обработки прерывания.

Замечание

Некоторые программисты заблуждаются, веря в то, что NMI может быть заблокирована Не может! Однако в IBM PC возможно заблокировать другие каналы, которые генерируют сигналы прерывания процессору по линии NMI, что предотвращает возникновение прерываний NMI На машинах IBM XT и полностью с ними совместимых можно маскировать NMI записав 00h (запрещено) или 080h (разрешено) в порт 0A0h Тем не менее это может не дать желаемого эффекта, поскольку не запретит другим программам открыть NMI после вашего запрета Имейте также в виду, что некоторые каналы внешнего интерфейса используют NMI в своих собственных целях

Векторы прерываний и микросхема 8259

Имея всего только две линии прерываний — INTR и NMI, можно подумать, что возможности прерываний процессора 8086 жестко ограничены Но с помощью другой микросхемы, *программируемого контроллера прерываний* (PIC) Intel 8259, IBM PC может обслуживать до восьми устройств, генерирующих прерывания. (В IBM AT добавлен второй PIC, что позволяет обслуживать еще большее число устройств. Большинство современных PC имеет похожую конструкцию.) Каждое устройство приписывается одному уровню PIC (от 0 до 7 или 15 в AT), причем меньшие номера имеют большие приоритеты. Это означает, что если два прерывания возникают одновременно, контроллер 8259 первым обслуживает устройство с меньшим номером. В табл. 10.1 перечислены устройства, связанные с каждым уровнем PIC. Уровень 2 служит как канал к подчиненной 8259 на компьютерах PC. Так как NMI также генерируется внешним образом, оно представлено в таблице, хотя эта линия не подсоединена к контроллеру 8259.

Таблица 10.1 **Прерывания внешних устройств**

Уровень PIC	Номер прерывания	Устройство
0	08h	Таймер (программные часы)
1	09h	Клавиатура
2	0Ah	К подчиненной 8259
3	0Bh	Второи последовательный I/O (COM2)
4	0Ch	Первый последовательный I/O (COM1)
5	0Dh	Жесткии диск
6	0Eh	Дисковод
7	0Fh	Параллельный принтер
8	070h*	Системные часы
9	071h*	К главной 8259 Уровень 2
10	072h*	-
11	073h*	-
12	074h*	-
13	075h*	Числовои сопроцессор
14	076h*	Жесткий диск
15	077h*	-
NMI	02h	Четность памяти

*Только для IBM AT и совместимых с ними

Как можно видеть из табл. 10.1, каждый уровень PIC связан со второй величиной, называемой *номером прерывания* (или типом прерывания, уровнем прерывания), которая принимает значения от 08h до 0Fh на компьютерах PC, PcJr и XT с дополнительными восьмью уровнями на AT. Эта система двойных наименований для внешних прерываний приводит в замешательство многих. Помните, что уровень PIC в *действительности ссылается на контакт контроллера* 8259, к которому устройство подключено. Номер прерывания обозначает ISR, которая выполняется, когда устройство требует обслуживания. Программируя, можно игнорировать уровень PIC и вместо него обращаться к прерываниям по их номерам.

В табл. 10.2 представлен весь набор номеров прерываний, определенный в обычном компьютере типа PC/XT За исключением первых восьми внешних прерываний из табл. 10.1, которые повторяются в этой таблице, большинство прерываний из этого полного набора представляет программное множество. Независимо от вида прерывания, каждый

номер прерывания связан с уникальным вектором прерывания, который хранится в памяти по некоторому адресу. Эти адреса перечислены в середине табл. 10.2.

Таблица 10.2. **Номера программных прерываний и их векторы**

Номер прерывания	Местонахождение вектора	Назначение
000h	0000h	Деление на нуль
001h	0004h	Перевод в пошаговый режим
002h	0008h	Немаскируемое прерывание
003h	000Ch	Точка останова
004h	0010h	Переполнение регистров
005h	0014h	Печать копии экрана
006h	0018h	*
007h	001Ch	*
008h	0020h	Таймер (программные часы)
009h	0024h	Клавиатура
00Ah	0028h	*
00Bh	002Ch	Второй последовательный I/O (COM2)
00Ch	0030h	Первый последовательный I/O (COM1)
00Dh	0034h	Жесткий диск
00Eh	0038h	Дисковод
00Fh	003Ch	Параллельный принтер
010h	0040h	Видеоадаптер
011h	0044h	Запрос списка подсоединенного оборудования
012h	0048h	Запрос размера памяти
013h	004Ch	Управление жестким диском
014h	0050h	RS-232 I/O
015h	0054h	Магнитофон
016h	0058h	Клавиатура
017h	005Ch	Принтер
018h	0060h	Basic из ПЗУ
019h	0064h	Перезапуск системы
01Ah	0068h	Текущее время/дата
01Bh	006Ch	Обработка <Ctrl-Break>
01Ch	0070h	Пользовательская процедура таймера
01Dh	0074h	Параметры инициализации видеоадаптера
01Eh	0078h	Параметры дисковода**
01Fh	007Ch	Адрес расширенной таблицы символов**
020h-03Fh	0080h-00FCh	Зарезервировано DOS
040h-06Fh	0100h-01BCh	Различное
070h	01C0h	Системные часы
071h	01C4h	*
072h	01C8h	*
073h	01CCh	*
074h	01D0h	*
075h	01D4h	Числовой сопроцессор
076h	01D8h	Жесткий диск
077h	01DCh	*
078h-0FFh	01E0h-03FCh	Различное

* *Зарезервированы или не используются*
** Не являются процедурами обработки прерывания

Вектор прерывания — это просто указатель, 32-битовый (4-байтовый) адрес значений сегмента и смещения, который хранится в нижних адресах памяти, начиная с 0000:0000 до 0000: 03FF. Каждый вектор задает адрес начала процедуры обработки прерывания, связанной с соответствующим номером прерывания, меняющимся от 00 до FFh, определяя одно из 256 программных и аппаратных прерываний для типичного PC. Когда сигнал внешнего прерывания генерируется одним из устройств, перечисленных в табл. 10.1, контроллер 8259 активизирует линию INTR процессора, ожидает подтверждения (генерируемого автоматически) и затем посылает соответствующий номер прерывания процессору. Процессор использует этот номер для того, чтобы выбрать нужный вектор из нижней памяти и вызвать ISR. Подобные действия совершаются, когда процедура вызывает программное прерывание по команде int либо когда в результате деления на 0 или при других аналогичных обстоятельствах генерируется внутреннее прерывание. Как в случае внешних, так и внутренних прерываний происходит несколько событий после того, как процессор получил номер прерывания:

- Флаги проталкиваются (сохраняются) в стек
- Флаги if и tf очищаются
- Регистры ip и cs проталкиваются в стек
- Вектор прерывания копируется в cs:ip

Последний шаг этого процесса приводит к тому, что начинает выполняться программа обработки прерывания, хранимая в памяти по адресу вектора прерывания, соответствующего номеру прерывания (см. табл. 10.2). Изменяя один такой вектор, можно вставить свою собственную процедуру обработки прерывания на место установленного кода, который обслуживает прерывания в вашей системе, а также связать вашу обработку прерывания с существующей ISR. Этим приемом можно воспользоваться для перехватывания нажатия некоторых клавиш, которые задействованы как команды активизации, оставляя другие клавиши нетронутыми. Когда ISR заканчивает обработку прерывания, она выполняет команду iret, что приводит к таким действиям:

- Регистры cs и ip выталкиваются из стека
- Флаги выталкиваются из стека

Первое из этих действий обеспечивает нормальное выполнение прерванной программы. На втором шаге восстанавливаются значения всех флагов, которые могли меняться командами ISR. Благодаря тому, что флаги автоматически сохраняются и восстанавливаются таким способом, а также тому, что прерывания от устройств обрабатываются, только если установлен флаг if (командой sti, например), вам не нужно самому выполнять sti внутри ISR, чтобы после окончания работы ISR разрешить обработку новых прерываний (общее заблуждение). Изначальные значения флагов загружаются в стек перед тем, как if и tf очищаются процессором; поэтому, если if установить заранее, он будет вновь установлен после выполнения iret. Необходимо выполнять sti внутри вашей процедуры обработки только в том случае, когда вы хотите, чтобы прерывания обнаруживались в *процессе выполнения* ISR.

Когда вам нужно, чтобы ISR возвращала значения флагов — например, как это часто осуществляет функция ДОС команды int 21h — у вас есть два выбора: изменить значения флагов в стеке до выполнения iret или извлечь флаги из стека и выполнить ret вместо iret. Помните, что процедура обработки прерывания — это всего только подпрограмма специального вида; поэтому, чтобы предотвратить обратное изменение флагов в процедуре, можно использовать такой код:

```
retf 2 ; Вернуться и отбросить 2 байта стека
```

Это приведет к возврату из ISR с последующим удалением 2 байт из стека. Эти два байта содержат значения флагов, запомненных в стеке, когда была активизирована ISR. Поступайте так только со внутренними прерываниями, которые программы вызывают как подпрограммы. Отбрасывая флаги, сохраненные в стеке процессором после определения прерывания, вы в действительности превращаете ISR просто в подпрограмму, которая может заканчиваться ret Можно затем использовать команду call для выполнения того же кода, начиная, конечно, с другой точки входа. Несмотря на то что вам не понадобится часто применять этот фокус, полезно понимать, что ISR это всего только подпрограмма и вам решать, что этот код делает и как он возвращает управление вызывающему процессу.

Почему команда hlt не останавливает программу

Тесно связанная с программированием прерываний, команда hlt ведет себя не так, как вы могли бы подумать Выполняя hlt, процессор 8086 делает паузу, совершенно останавливая программу в этом месте. В это время, если прерывания разрешены, сигнал прерывания по линии INTR распознается процессором, как обычно, заставляя выполняться процедуру обработки прерывания и, таким образом, нарушая состояние остановки. Когда ISR завершается, работа продолжается командой, следующей за hlt. Другими словами, hlt на самом деле не останавливает процесс — система ожидает возникновения прерывания. Если прерывание не разрешено, hlt в действительности может заблокировать систему компьютера, препятствуя распознаванию сигналов INTR. Поэтому, чтобы поставить 8086 "на колени", вы могли бы выполнить команды:

```
cli  ; Запрещение прерываний, очищая if
hlt  ; Ожидание прерывания, которое не может возникнуть!
```

После этих двух команд только два события могут разблокировать процессор: RESET или NMI, которые оба игнорируют установку if. (RESET — входная линия процессора, которая, может быть, и не подсоединена к кнопке перезапуска в вашей системе. Многие ранние PC не имели кнопки перезапуска.)

На практике hlt в основном используется для синхронизации программ и внешних событий, при этом делается пауза до тех пор, пока сигнал прерывания не придет от определенного устройства. Основной смысл этого состоит в том, что команда sti, которая устанавливает флаг if, разрешает определять прерывания INTR. Однако это случится только *после следующей* за sti командой; поэтому для синхронизации программы и внешнего прерывания никогда не пишите:

```
sti  ; Разрешение на возникновение прерывания
cli  ; Запрещение прерывания
```

Из-за того, что прерывания распознаются только после команды, следующей за sti и если это команда запрещения прерывания, то даже самому "пронырливому" сигналу не хватит времени, чтобы проскользнуть. Корректный способ синхронизации программы и внешнего прерывания представлен следующим программным кодом:

```
cli  ; Запрещение прерывания
sti  ; Разрешение прерываний в следующей команде
hlt  ; Пауза до возникновения прерывания
cli  ; Снова запрещение прерывания (необязательно)
```

Если прерывание уже запрещено, первая cli не нужна. Вторая cli нужна, только если вы хотите предотвратить возникновение дополнительных прерываний. Поставив hlt следом за sti, вы гарантируете продолжение работы программы только при получении сигнала INTR, генерируемого, например, нажатием клавиши или получением символа в последовательный порт ввода

Обработка прерываний

Программный код ISR следует одной и той же базовой схеме как для внешних, так и для внутренних прерываний В этой части демонстрируется, как написать ISR для обработки прерываний, а также изучаются несколько тонкостей обработки прерывания в языке ассемблера 8086.

Программа 10.1, SLOWMO.ASM, включается в прерывание автономных часов для того, чтобы добавить в программу постоянно повторяющиеся паузы, которые замедляют выполнение кода до черепашьего шага. Это может оказаться полезным для отладки быстрых программ, когда проносящееся действие слишком мелькает, чтобы его рассмотреть. Программа также демонстрирует корректный способ обработки прерываний, приходящих от микросхемы PIC 8259. Когда прерывание происходит внутри прерывания часов PC, необходимо соблюдение особой осторожности для того, чтобы избежать разладки системных часов. SLOWMO служит базой для иллюстрации этого вопроса. Выполните ассемблирование и компоновку SLOWMO с вашим файлом MTA.LIB, используя команды:

```
tasm slowmo
tlink slowmo,,, mta
```

Программа 10 1. SLOWMO ASM

```
1   %TITLE "Slow Motion Interrupt -- Copyright (c) 1989, 1995 -- by Tom Swan
2
3         IDEAL
4
5         MODEL small
6         STACK 256
7
8   delay EQU   0010h                   ; Величина задержки
9   cr    EQU   13                      , ASCII-символ возврата каретки
10  lf    EQU   10                      ; ASCII- символ новой строки
11  BIOSData    EQU             040h    ; Адрес сегмента данных BIOS
12  LowTimer    EQU             006Ch   ; Адрес меньшего слова таймера
13  PIC8259     EQU             0020h   , Адрес порта микросхемы PIC
14  EOI   EQU   0020h                   , Значение конца прерывания
15
16        DATASEG
17
18  exCode      DB              0
19  string      DB              'This is a test of the timer', cr, lf
20              DB              ' slow-mo interrupt handler', cr, lf, 0
21  timerSeg    DW              ?       ; Сохранить первоначальный вектор прерывания
22  timerOfs    DW              ?       , Int 1Ch ISR
23
24
```

```
25:          CODESEG
26:
27:     ;----- Из STRIO.OBJ, KEYBOARD.OBJ
28:     EXTRN StrWrite:proc, KeyWaiting:proc
29:
30:     Start:
31:          mov    ax, @data           ; Инициализировать адрес
32:          mov    ds, ax              ; сегмента данных
33:          mov    es, ax              ; Установить es = ds
34:
35:          mov    [word cs:difference],delay; Установить время задержки
36:
37:          push   es                  ; Сохранить регистр es
38:          mov    ax, 351Ch           ; Получить вектор прерывания 1C
39:          int    21h                 ; Вызов DOS по этому вектору
40:          mov    [timerSeg], es      ; Сохранить значение сегмента
41:          mov    [timerOfs], bx      ; Сохранить значение смещения
42:          pop    es                  ; Восстановить es
43:
44:          push   ds                  ; Сохранить регистр ds
45:          mov    ax, 251Ch           ; Установить вектор прерывания 1C
46:          push   cs                  ; Установить ds = cs адресу
47:          pop    ds                  ; новой ISR, поместив полный
48:          mov    dx, offset SlowMo   ; адрес в ds:dx
49:          int    21h                 ; Установить новый вектор прерывания
50:          pop    ds                  ; Восстановить ds
51:
52:          mov    di, offset string   ; Строка тестирования адреса
53:     @@10:
54:          call   StrWrite            ; Выдаваемая строка
55:          call   KeyWaiting          ; Проверка нажатия клавиши
56:          jz     @@10                ; Цикл до нажатия клавиши
57:
58:          push   ds                  ; Сохранить ds, изменяемый ниже
59:          mov    ax, 251Ch           ; Установить вектор прерывания 1C
60:          mov    dx, [timerOfs]      ; Получить сохраненную величину смещения
61:          mov    ds, [timerSeg]      ; Получить сохраненную величину сегмента
62:          int    21h
63:          pop    ds                  ; Восстановить ds
64:     Exit:
65:          mov    ah, 04Ch            ; Функция DOS . Выход из программы
66:          mov    al, [exCode]        ; Вернуть значение кода возврата
67:          int    21h                 ; Вызвать DOS. Завершить программу
68:
69:     %NEWPAGE
70:     ;---------------------------------------------------------------
71:     ; SlowMo-Slow Motion Timer Interrupt Service Routine
72:     ;---------------------------------------------------------------
73:     ; Вход:
74:     ; нет
75:     ; Выход:
76:     ; нет (ожидает разницу во времени)
77:     ; Регистры:
78:     ; нет
79:     ;---------------------------------------------------------------
80:
81:     ;----- Переменные определяются внутри сегмента кода,
82:     ; где их легко можно будет найти во время выполнения ISR
83:
84:     inProgress   DB                 0       ; Флаг In-progress (0 = Нет, 1 = Да)
85:     difference   DW                 0       ; Относительное время паузы
86:
87:     PROC SlowMo
88:
89:     ;----- Проверить флаг inProgress, который показывает, были ли предыдущие
90:     ; копии SlowMo раньше запущены. Это необходимо предотвратить
91:     ; или система зависнет
92:
93:          cmp    [byte cs:inProgress], 0    ; Проверить флаг in-progress
94:          jne    @@99                       ; Переход, если SlowMo выполняется
95:          inc    [byte cs:inProgress]       ; В противном случае установить флаг = 1
96:
97:          sti    ; Разрешить возникновение прерываний
98:          push   ax                  ; Сохранить модифицируемые регистры
```

```
99:         push  ds
100:        push  dx
101:
102:        mov   al, EOI              ; al = значение конца-прерывания
103:        out   PIC8259, al          ; Выдать конец прерывания
104:
105:        mov   ax, BIOSData         ; Адресовать область данных BIOS
106:        mov   ds, ax               ; через ds
107:        mov   ax, [word LowTimer]        ; Получить младшее слово значения таймера
108:  @@10:
109:        mov   dx, [word LowTimer]        ; Получить новое значение таймера в dx
110:        sub   dx, ax               ; Вычесть старое значение таймера
111:        cmp   dx, [cs:difference]        ; Проверить, есть ли разница
112:        jb    @@10                 ; Цикл до тех пор, пока не будет разницы
113:
114:  ;----- Запретить прерывания, пока мы не очистим все и не выйдем после паузы
115:
116:        cli   ; Запретить прерывания
117:        dec   [byte cs:inProgress]       ; Переустановить флаг in-progress
118:        pop   dx                   ; Восстановить сохраненные регистры
119:        pop   ds
120:        pop   ax
121:  @@99:
122:        iret                       ; Возврат из прерывания
123:  ENDP SlowMo
124:
125:  END Start                        ; Конец программы / точка входа
```

Внедрение в прерывание часов PC

Все IBM PC (и даже менее, чем на 100% совместимые) содержат системные часы, которые генерируют сигнал прерывания приблизительно 18.2 раза или "тика" в секунду. В ROM BIOS прерывание 08h обрабатывает эти сигналы, которые проходят по линии 0 PIC 8259 (см. табл. 10.1.) Это задает прерыванию от часов наивысший приоритет. До тех пор пока прерывания разрешены, ISR часов будет выполняться первым, если более чем один сигнал прерывания, возникнет в одно и то же время.

ROM BIOS ISR часов выполняет две базовые функции. Первая функция — программа увеличивает 32-битовую величину, считая таким образом общее число тиков часов, которые прошли с того времени, как система была включена. (Эта величина обнуляется каждые 24 часа — не обязательно в полночь.) Вторая функция — уменьшается другой счетчик, который контролирует, как долго мотор дисковода остается включен. Когда эта величина становится равной 0, мотор дисковода выключается (если он был включен), тем самым оставляя дискету во вращающемся состоянии достаточно долго для того, чтобы увеличить скорость чтения и записи. (Каждый раз, когда дискета начинает вращаться, требуется время, чтобы ось набрала скорость. Если бы мотор выключался сразу после окончания каждого акта чтения и записи, то эти паузы замедлили бы работу с дискетой.) Как можно видеть, ISR часов — это бьющееся в PC сердце, и так же как с любым сердцем, большая задержка в выполнении его обязанностей может привести к проблемам; поэтому разумно не выключать прерывания командой cli на более чем 1/18.2 (около 0.05) секунд перед запуском sti для переключения прерываний обратно.

ISR часов выполняет третью функцию, которая позволяет вам постоянно следить за работой компьютера. В каждый тик часов эта процедура выполняет программное прерывание под номером 01Ch, которое обычно ничего не делает. Установив свое собственное 01Ch ISR, ваш код будет выполняться около 18.2 раз в секунду вдобавок к другим функциям часов. SLOWMO.ASM использует эту особенность для того, чтобы добавить паузы в исполняемую программу.

Фокусы с "тиканьем" часов

Для того чтобы перехватить прерывание часов PC, на первом шаге надо сохранить текущий вектор прерывания 01Ch, как это делается в строках 37–42 программы 10.1, вызвав функцию 035h для получения адреса вектора из регистров es:bx, который запоминается в переменных timerSeg и timerOfs. Далее, в строках 44–50 вызывается функция DOS 025h с адресом нового вектора прерывания, равным смещению в сегменте кода процедуры SlowMo, начинающейся в строке 87. Так заменяется исходный вектор адресом нового ISR. Можно также выключить прерывания и вставить адрес прямо в соответствующий сегмент нижней памяти, но вызов процедуры DOS, написанной для этих целей, проще. Обратите внимание, как регистр ds устанавливается в соответствии с сегментом кода:

```
push cs      ; Протолкнуть cs в стек
pop  ds      ; Вытолкнуть cs в ds
```

Запомните этот полезный трюк и избегайте приписывания значения третьему регистру (ax, например) только затем, чтобы записать эту величину по назначению. При установке вашего собственного ISR, если вы используете код, аналогичный тому, который представлен в строках 37–50, для замещения существующих векторов адресами ваших собственных процедур, обязательно сохраняйте и восстанавливайте регистры сегмента es и ds, как это здесь показано.

Замечание

Всегда восстанавливайте все векторы прерываний, которые вы изменили в вашей программе. Когда ваша программа заканчивается, адреса подлежат перекрытию последующими программами. Поэтому оставлять работающую ISR после окончания программы без принятия мер по защите памяти, занимаемой ISR, нельзя, так как это почти наверняка приведет к краху системы. DOS не восстанавливает векторы, которые изменила ваша программа

В строках 52–56 программа показывают тестовую строку и ожидает нажатия любой клавиши, завершающей программу. В этом цикле ISR SlowMo выполняется как будто сама по себе, но в действительности осуществляется вызов процедуры прерывания таймера 01Ch из ROM BIOS со скоростью 18.2 раза в секунду. Хотя это может выглядеть так, как будто цикл в строках 52–56 и ISR исполняется взаимоконкурентно, помните, что прерывание вынуждает программу останавливаться, пока ISR отработает. Таким образом, конкуренция — это только иллюзия, созданная магией прерывания от часов PC.

После того как вы нажали клавишу, программа заканчивает работу. Перед этим в строках 58–63 еще раз вызывается функция DOS 025h, но на этот раз с ранее сохраненным вектором. Тем самым устанавливается изначальный адрес ISR прерывания 01Ch (вероятно, но не обязательно, она содержит одинокую команду iret), которая действовала до начала SLOWMO.

Прерывания и переменные

ISR процедура SlowMo из программы 10.1 (строки 70–123) выполняется, когда прерывание таймера ROM BIOS вызывает программное прерывание 01Ch. Из-за того, что это может случиться в любой момент времени — между командами основной программы, во время вызова DOS и даже во время вызова другой процедуры ROM BIOS — нельзя полагаться на то, что значения регистров сегментов будут указывать на местонахождение сегмента данных программы. По этой причине ISR должна быть осторожной при инициализации ds (и es, если необходимо) перед загрузкой или изменением переменных из сегмента данных. Один из вариантов, как это сделать — сохранить ds и затем приписать ему значение сегмента данных, как это обычно выполняется в начале программы:

```
push ds                 ; Сохранить текущий ds
mov  ax, @data          ; Приписать адрес сегмента данных
mov  ds, ax             ; ds через ax
;
;----- Здесь идет код прерывания
;
pop  ds                 ; Восстановить ds
iret                    ; Вернуться из прерывания
```

ISR должна выполнять это в начале подпрограммы каждый раз, когда она запускается, сохраняя текущее значение ds, которое код прерывания может использовать в качестве адреса своих собственных переменных. Другой метод, который представлен в строках 84–85, состоит в том, чтобы определять переменные ISR внутри сегмента кода. Этот метод требует использования cs: сегмент переопределяется для того, чтобы дать указание ассемблеру (и CPU) использовать cs как базовый адрес для поиска переменных в памяти. Например, для того чтобы загрузить байт inProgress в al, можно выполнить:

```
mov al, [byte cs:inProgress]
```

Если вы не используете cs:, после переопределения ассемблер будет предполагать, что ds адресует текущий сегмент данных — распространенная ошибка, которая обычно ведет к гибели системы.

Прерывание ISR

Как объяснялось ранее, прерывание таймера — сердце PC. Поскольку это жизненно важно, чтобы часы не выводились из работы надолго, необходимо обеспечить возможность прерываний в SlowMo (строка 97). При этом возникает задача, решаемая трюком. Если прерывание обрабатывается, очень возможно, что ISR на самом деле сможет прервать себя. В этом случае код ISR приостановит свое выполнение, флаги, регистры cs и ip запишутся в стек и прерывание часов обработается. Если это повторится многократно и у вызываемых ISR не будет возможности завершиться, стек в конце концов переполнится, что, по всей видимости, приведет к зависанию системы.

Когда процедура позволяет прервать саму себя, она считается *реентерабельной* — другими словами, если новая порция последовательности кода может начать выполняться с начала, перед тем как предыдущая порция закончила работу. Такой код должен выделить новое пространство для переменных, но не для глобальных. Чтобы понять по-

чему это так, рассмотрим ISR SlowMo. Из-за того, что существует только один inProgress в каждом случае, а переменные в строках 84, 85 различны, новый вызов кода будет использовать те же самые переменные, возможно, меняя их значения, если ISR позволено прерывать себя. Поэтому, когда второе выполнение SlowMo закончилось, вызвав тем самым изначальную порцию на выполнение, переменные, по-видимому, изменились — побочный эффект, которого можно избежать, если бы процедура была действительно реентерабельной.

Замечание

Вы могли слышать, что DOS и ROM BIOS нереентерабельны Это означает, что процедуры имеют доступ к глобальным переменным, похожим на SlowMo Такие процедуры не могут повторно вызвать себя, так как для них существует только один набор переменных На самом деле некоторые процедуры DOS и BIOS реентерабельны, несмотря на использование ими глобальных переменных Прерывание часов — наилучший тому пример, оно может и прерывает себя без конфликтов Фактически так и должно быть, чтобы системное время было правильным

Очевидно, поскольку процедура SlowMo использует только один набор переменных, она определенно нереентерабельна. Но для того чтобы обеспечить работу системных часов во время длительной паузы SlowMo, прерывания *должны быть разрешены* — даже если это приведет к тому, что последовательные прерывания часов перезапустят SlowMo в виде "остановка паузы" и намертво остановят таким образом систему. Мы должны решить сложную проблему: жизненно важные прерывания от таймера должны быть разрешены во время продолжительной паузы, и в то же время наша собственная SlowMo должна быть защищена от прерывания самой себя, хотя она это будет делать, так как ISR таймера выполняет следующее прерывание 01Ch. Ух!

К счастью, существует простой ответ для разрешения этого типичного конфликта, который показан в строках 93–95. В первую очередь, проверяется байт inProgress. Если этот байт нулевой, ISR разрешается работать нормально. Если байт inProgress не нулевой, программа предполагает, что предыдущая ISR была прервана. Так должно случиться, поскольку только в строке 95 байт inProgress устанавливается в 1 и только в строке 117 inProgress обращается в 0. Если inProgress не равен 0 во время запуска SlowMo, значит, скорее всего, команды между строками 98–116 прерваны тиком часов, который привел к повторному входу в SlowMo Простой флаг inProgress определяет это состояние, позволяя выполняться только одной порции SlowMo. В результате ISR таймера ROM BIOS может продолжать выполняться во время работы SlowMo, ведя системные часы по времени

Замечание

Функция Печать экрана (Print Screen) использует аналогичный фокус для того, чтобы игнорировать ваши нажатия на клавишу <PrtSc> во время печати копии экрана Когда вы нажимаете клавишу <PrtSc> и начинается печать, повторное нажатие клавиши <PrtSc> на самом деле перезапускает функцию „Печать экрана" Но флаг, аналогичный inProgress, показывает, что предыдущая операция печати выполняется, таким образом предотвращая многократный вывод на печать, когда требуется только одна копия экрана

Команда окончания прерывания

В строке 97 SLOWMO.ASM командой sti включаются прерывания, позволяя часам PC продолжать работу во время паузы SlowMo. Из-за того, что прерывания приходят через PIC 8259, как это описано раньше, одной sti недостаточно для того, чтобы разрешить распознавание будущих прерываний Вдобавок к sti вы должны также "сказать" PIC 8259, что вы хотите запускать только новые прерывания. Сделайте это подачей команды *конец-прерывания* (end-of-interrupt EOI) в порт 8259:

```
EOI   EQU     020h            , Значение EOI
PIC8259       EQU     020h    , Адрес порта 8259

sti                           , Разрешение на возникновение прерывания
mov   al, EOI                 , al = значение конец-прерывания
out   PIC8259, al             , Посылка команды EOI
```

Как EOI (константа EOI), так и PIC8259 (константа адреса порта) имеют одно и то же значение 020h, но это ничего не значащее совпадение. Команда sti устанавливает флаг if, который автоматически переопределяется процессором во время распознавания сигнала прерывания, заставляющего ISR начать работу. Установка if позволяет процессору снова распознать сигналы внешних прерываний. Из-за того, что эти сигналы приходят от микропроцессора 8259, команда end-of-interrupt также должна указать 8259 игнорировать прерывания, которые он получает для процессора. Одного только выполнения sti недостаточно. Когда обслуживаемые прерывания генерируются 8259 и любые прерывания вызываются изнутри связанных ISR, как в случае с SlowMo, — вы должны выполнить такую же последовательность из трех команд для того, чтобы разрешить возникновение будущих внешних прерываний

К этому времени вам, наверное, пришла в голову мысль, что обработка прерываний, в частности тех, которые присоединяются к часам PC, требует от вас "хождения на цыпочках". Большая часть работы над ISR — это нечто незем-

ное: избегать конфликтов с глобальными переменными, иметь дело с повторными вхождениями, быть уверенными, что будущие прерывания смогут возникнуть, сохранять и восстанавливать значения регистров и т.д. Сама внутренность ISR может быть относительно простой, какой она и есть в этом примере в строках 105–112. Эти команды проверяют младшее слово величины текущего времени, которую ISR часов ROM BIOS увеличивает, как это раньше описывалось Когда эта величина превосходит значение переменной difference, ISR SlowMo завершается.

Заметьте, что ни одна из команд цикла в строках 108–112 не изменяет значение LowTimer напрямую. Если прочесть этот код в отрыве от контекста, цикл будет выглядеть бесконечным, и вы можете предположить, что нашли ошибку. Если никакая команда не изменяет LowTimer, то вычитание в строке 110 будет всегда давать 0, приводя к тому, что jb в линии 112 будет повторяться бесконечно. Тот факт, что этого не происходит, доказывает, что ISR часов ROM BIOS выполняется независимо от этого цикла.

Прерывания и стеки

Поскольку внешние прерывания могут возникать в любое время, нет никакого способа предсказать значения регистров, когда ISR внешнего прерывания начинает работу. Единственный регистр, на который можно положиться, — cs. Очевидно, этот регистр всегда содержит значение текущей величины сегмента кода выполняемых в данный момент времени команд. Но es, ds и ss могут указывать на что угодно. Как объяснялось раньше, для ссылки на внутренние данные вы должны инициализировать ds и es, сохранив их текущие значения для восстановления перед окончанием ISR. К сожалению, корректная обработка регистра сегмента стека не так проста.

В программе 10 1 ISR-процедуры SlowMo три слова проталкиваются в стек в строках 98–100. Но в какой стек? DOS имеет свое собственное пространство для стека, как это и положено главной программе. Вдобавок, в памяти могут быть другие ISR, у которых свои собственные стеки. Если какая-либо из этих программ прерывается, величина ss будет равна тому значению, которое приписано этой программе. Другими словами, ISR обычно использует любой сегмент стека, который был текущим в момент возникновения прерывания. SlowMo просто предполагает, что по меньшей мере три слова стекового пространства свободны — в дополнение к трем словам, требуемым процессором, в стек проталкиваются флаги и регистры cs:ip перед выполнением ISR.

Во многих случаях, по-видимому, можно предположить, что небольшое стековое пространство всегда свободно Но для большинства программистов такое допущение — пилюля болезненной неопределенности, которую глотают в педантичном мире компьютерного программирования, требующем аккуратности от своих жителей. Если ваша ISR использует больше, чем несколько байтов стековой памяти, — вы должны перейти на локальный стек.

Замечание

В ваших собственных программах всегда добавляйте несколько дополнительных байтов в вашей директиве STACK к тем, которые точно нужны. В противном случае у вас могут быть проблемы с ISR, процедурами ROM BIOS, DOS и другим резидентным кодом, „считающим", что несколько баитов стека свободно Некоторые руководства по DOS рекомендуют 2,048-байт как минимальныи размер стека, хотя простые примеры, такие как программы в этои книге, могут обойтись и меньшеи величиной

Поменять стек в ISR не сложно, но вы должны выполнять команды в правильном порядке. Причина этого состоит в том, что 8086 временно запрещает прерывания на период в одну команду всякий раз, когда вы приписываете некоторое значение регистру сегмента. Другими словами, когда вы пишите знакомый код инициализации

```
mov  ax, @data
mov  ds, ax
mov  dx, offset string
```

прерывания не обрабатываются до mov в dx — факт, не очевидный из текста программы. В этом примере влияние на прерывания не существенно. Однако рассмотрим, что случится, когда меняется значение регистра сегмента стека:

```
mov  ax, offset stackSpace
mov  ss, ax
mov  sp, offset endOfStack
```

Регистр sp — это указатель на стек, локализующий текущую вершину стека относительно адреса сегмента в ss. Из-за того, что требуется две команды для изменения как ss, так и sp, если прерывание возникнет между присвоением ss и присвоением sp, старый указатель на стек будет использоваться вместе с новым сегментом стека — опасная ситуация, которая может привести к краху системы. По этой причине прерывания запрещены в течение одной команды после присвоения ss — как раз достаточный промежуток времени для того, чтобы приписать значение endOfStack регистру sp. Прерывания также запрещены на время выполнения команды pop, влияющей на регистр сегмента. Помните, что этот эффект длится только одну команду и mov в sp должен немедленно следовать за mov в ss.

Замечание

Всякий раз, когда вы приписываете значение ss, немедленно затем присваивайте соответствующую величину sp. Никогда не переставляйте эти команды и не вставляйте другие команды между этими присвоениями. Эта последовательность не произвольная!

Для того чтобы перейти на локальный стек в процедуре ISR, во-первых, выделите некоторое пространство в сегменте программного кода. Это можно осуществить многими способами, например:

```
ALIGN
myStack DB 512 DUP (0)                ; Внутренний 512-байтовый стек
endOfStack = $
```

Директива ALIGN гарантирует, что стек начнется с границы слова, другими словами, с четного адреса. Стек начинается в myStack и, в этом примере, равен 512 байт. Числовая константа endOfStack отмечает конец стекового пространства. Вслед за этим сохраните текущие значения ss и sp в глобальных переменных, которые вы будете позже использовать для восстановления состояния регистров в начале процедуры:

```
oldSS       DW 0              ; Содержит сегмент стека
oldSP       DW 0              ; Содержит смещение стека

PROC ISR
     mov    [cs:oldSS], ss    ; Сохраняется сегмент стека
     mov    [cs:oldSP], sp    ; Сохраняется указатель на стек
```

Вследствие того что переменные объявлены в сегменте кода, необходимо переопределить сегмент через cs — для сохранения ss и sp в нужном месте. Затем вы можете перейти к внутреннему стеку, присвоив для этого текущее значение сегмента кода ss и смещение endOfStack sp. Заметьте, что требуется одно слово стекового пространства для сохранения cs:

```
push cs                       ; Запомнить текущий сегмент кода
pop  ss                       ; Поместить величину cs в ss
mov  sp, offset endOfStack    ; Временно запретить прерывания
```

Для того чтобы устранить даже этот, слишком большой расход стека, используется третья переменная для сохранения ax (или другого регистра). Из-за того что вы не можете приписать значение одного сегментного регистра другому, текущее значение cs сначала присваивается ax, а затем переприсваивается ss:

```
oldAX       DW 0              ; Переменная в сегменте кода
mov  [cs:oldAX], ax           ; Сохранить ax в переменной
mov  [cs:oldSS], ss           ; Сохранить сегмент стека
mov  [cs.oldSP], sp           ; Сохранить указатель на стек
mov  ax, cs                   ; Приписать cs ax
mov  ss, ax                   ; Приписать ax ss (ss = cs)
mov  sp, offset endOfStack    ; Прерывания временно запрещены
```

Позже вы сможете восстановить ax по сохраненной в cs:oldAX величине. Обычно вам не надо добиваться такой длины — по меньшей мере три слова стекового пространства должны быть свободны для выполнения ISR, и разумно предположить, что хотя бы еще одно слово будет свободно.

Поскольку стек увеличивается от верхних адресов памяти к нижним, sp нужно инициализировать так, чтобы он указывал на конец стека, а не на начало. Из-за того что команда push уменьшает указатель на стек на 2, перед перемещением запомненного слова в место, на которое указывает ss:sp, будет безопасно адресовать значением sp некоторое место памяти сразу после последнего байта, выделенного стеку. Но некоторые программисты предпочитают использовать другую команду для загрузки sp:

```
mov  sp, offset endOfSteck-2
```

Таким образом, ss:sp указывает на последнее слово в стеке, а не на байт за концом стека. Так тратится впустую одно слово стекового пространства, но гарантируется, что sp никогда не выйдет за пределы стека.

После перехода на внутренний стек можно сохранить регистры, ссылаться на переменные относительно bp и т.д. Помните, что ваш новый стек должен делиться между любыми другими прерываниями, которые возникают во время выполнения ISR. После окончания ISR восстановите изначальный стек следующими командами:

```
mov  ss, [cs:oldSS]   ; Восстановить регистр сегмента стека
mov  sp, [cs:oldSP]   ; Восстановить регистр указателя на стек
```

Выполняйте эти команды только в таком порядке, без вставки любых других команд, поскольку прерывания будут временно запрещены во время перезаписи sp.

Замечание

Сохранение и восстановление ss и sp с использованием глобальных переменных вновь поднимает вопрос о реентерабельности. В предыдущих примерах, из-за того что новый стек являлся глобальной переменной, ISR были защищены от прерывания самих себя. При попытке написать полностью реентерабельную ISR, которая переходит на внутренний стек, понадобится новое пространство для стека и переменных при каждом вызове ISR.

Использование команд int и into

Как вы знаете, функции DOS вызываются командой программного прерывания int 21h. Настоящие прерывания генерируются внешним образом и могут возникать в любое время. Программные прерывания, инициируемые через int, могут возникать только тогда, когда программа выполнит эту команду. Поэтому программные прерывания еще больше похожи на обычные подпрограммы, чем ISR. За исключением этого различия, внутренние программные и внешние прерывания от оборудования идентичны и используют значения соответствующих векторов в нижней памяти для того, чтобы запустить ISR, сохранив флаги и регистры cs:ip в стеке. Программные прерывания также заканчиваются командой iret.

Интересно то, что вызовы int не запрещаются очисткой флага if командой cli. Вы всегда можете вызвать программное прерывание, даже когда внешние прерывания запрещены. Вы можете вызвать также внешнюю ISR, используя команду int. Например, совершенно законно "генерируется" ваш собственный тик часов как

```
int 08h       ; Заставить часы "тикнуть"
```

Наверное, не существует достаточно веских причин заставлять ISR ROM BIOS таймера выполняться по команде программного прерывания, но ничего не мешает вам сделать это — хотя если это будет происходить часто, по-видимому, часы придут в негодность. Знайте также, что некоторые ISR (программа BIOS для прерывания от клавиатуры 09h, например) "считают", что в определенных регистрах различных коммуникационных линий имеются данные, которые необходимо обработать. Этого может и не быть, если вы хотите, чтобы прерывание от оборудования возникло по команде int. Но вызов прерываний от устройств программной командой int полезно использовать для отладки внешних ISR, что позволяет эмулировать работу оборудования.

В дополнение к int вы также можете использовать команду into (прерывание по переполнению) для того, чтобы вызвать прерывание 4, если флаг переполнения установлен (of=1) в результате выполнения предыдущей арифметической операции. На практике команда into используется редко и вектор прерывания для прерывания номер 4 обычно указывает на простую команду iret. Таким образом, ничего не происходит, если программа выполняет into. Можно приписать этот вектор (используя функцию DOS 025h, как это ранее описано) вашей собственной ISR, если вы хотите обрабатывать ситуацию переполнения по своему усмотрению.

Отлавливание прерывания ошибки деления

Ошибочное название "деление на нуль" служит источником многих заблуждений. Команды div или idiv автоматически вызывают прерывание 0 всякий раз, когда результат деления больше, чем максимальная величина, которая может находиться в месте результата (ax или al), и, конечно же, когда произошло деление на 0. Например, этот код вызывает прерывание 0:

```
mov     ax, 100 ; Присвоить 100h ax (младшее слово)
xor     dx, dx  ; Обнулить dx (старшее слово)
xor     bx, bx  ; Обнулить bx (делитель)
div     bx      ; Поделить dx:ax на bx
```

Поскольку делитель (bx) равен 0, div потерпит неудачу и выполнит ISR, вектор которой сохраняется в 0000:0000. Многие оказываются не в состоянии понять, что следующий код также генерирует прерывание деления на нуль:

```
mov     ax, 100 ; Присвоить 100h ax (младшее слово)
mov     bl, 1   ; Установить делитель (bl) в 1
div     bl      ; Генерируется прерывание 0
```

Результат деления 100h на 1, конечно, 100h. Но поскольку эта величина слишком велика для того, чтобы поместиться в регистр результата al, генерируется прерывание деления на 0, хотя делитель определенно не 0. По этой причине прерывание деления на нуль лучше назвать прерыванием "ошибка деления", которое вы не можете обойти таким кодом, как следующий:

```
    or      bl, bl   , Равен ли нулю делитель
    jne     @@10     , Если да (bl = 0), осуществить переход
    call    Error    , Вызвать обработчик ошибки
@@10
    div     bl       , ??
```

Несмотря на внешний вид, это не предотвратит возникновения прерывания 0 Проверка того, равняется ли делитель 0 перед выполнением div, — потеря времени, так как прерывание 0 возникает всякий раз, когда результат деления превосходит вместимость регистра результата. Когда это случается, выполняется ISR внутри DOS, останавливая программу — событие, которого коммерческие программы должны избегать. Решение вопроса состоит в том, чтобы установить пользовательский обработчик прерывания ошибки деления, для того чтобы заменить ISR DOS прерывания 0. Как вы увидите, это гораздо труднее, чем можно предположить.

Исправление ошибки деления

Что должно произойти, когда возникнет ошибка деления? Ответ зависит от того, каково приложение. Программа калькулятора, вероятно, должна выдать символ ошибки Табличный процессор может вставить сообщение об ошибке в "ячейку" на экране. Другая, менее критичная программа может просто проигнорировать это событие — что полезно в некоторых случаях, пока программа, выполняющая деление, знает об этой возможности. Обычный подход к написанию простой ISR:

```
PROC DivFault
 xor ax,ax           , Необязательная установка в 0
 iret                , Выход из прерывания
ENDP DivFault
```

Переприсваивание вектора прерывания 0 DivFault приводит к тому, что при условии возникновения ошибки деления выполняется команда iret, что, по-видимому, является самым простым способом для того, чтобы игнорировать такую ошибку. Установка частного в 0 необязательна — это разумный (если не правильный) ответ в случае ошибки деления. К сожалению, этот подход работоспособен только на системах с процессором 8086/88. На системах с 80286 и более поздними моделями процессоров iret в этом примере приведет на самом деле к тем же div и idiv, которые вызывают прерывание, и машина зависает. Причина, по которой это происходит, состоит в том, что прерывание 0 запоминает адрес следующей команды для процессоров 8086/88, но оно запоминает адрес текущей команды для процессоров 80286 и более поздних моделей. Это чрезвычайно неприятная проблема для программистов, которые должны писать код, работающий на достаточно широком множестве PC, XT и AT

Корректная обработка этого необычного условия требует некоторого забавного "пританцовывания". Решение состоит в том, чтобы выровнять адрес смещения возврата в стеке, для того чтобы перескочить команды div и idiv, которые запускают ISR на выполнение. В некоторых руководствах рекомендуется просто добавить 2 к смещению возвращаемого адреса в стеке и затем закончить ISR командой iret. Но этот обычный подход неудачен, если принять во внимание, что команды div и idiv могут быть длиной 2 или 4 байта, в зависимости от того, является ли делитель регистром (2 байта) или объектом из памяти (4 байта). В такой ситуации требуется просмотреть имеющийся код команд div и idiv. Если первые два бита второго байта равны 1, тогда операнд — регистр; в противном случае операнд — ссылка на память. "Зная" это, программа может выровнять возвращаемый адрес на 2 или на 4, пропуская div и idiv во время выполнения iret.

Замечание

Расшифровка битов, из которых состоит конкретный машинный код, — это кропотливая работа, к счастью, редко необходимая (См документацию, где содержатся точные битовые форматы для других команд в машинном коде)

Установка обработчика ошибки деления

Хороший путь для обработки прерывания ошибки деления состоит в установке в памяти резидентной программы для того, чтобы отлавливать прерывания 0, если они возникнут. После того как это сделано, все ошибки делений пойдут через новую ISR, предотвращая неожиданную остановку программы средствами DOS. Программа 10.2, DIV286.ASM, осуществляет это, в то же время демонстрируя, как писать резидентные программы на языке ассемблера

Замечание

Несмотря на свое название, DIV286 ASM не ограничивается работой только на процессорах 80286 Можно запускать эту программу на любом PC с 80286, 80386, 80486, Pentium или совместимыми процессорами

Выполните ассемблирование и компоновку DIV286, используя команды:

```
tasm div286
tlink /t div286,,, mta
```

Не запускайте пока DIV286 — вам надо сначала выполнить вторую программу (которую мы сейчас опишем), для того чтобы проверить эффективность нового обработчика прерывания. Обратите внимание на параметр /t в команде tlink; он необходим для создания файла .COM, а не обычного формата .EXE. Резидентные файлы .EXE кода писать намного сложнее, хотя они могут использовать больше памяти, чем резидентные файлы .COM, размер которых ограничен 64K. Для наших целей формат .COM более чем достаточен.

Замечание

Вы должны иметь 80286 или более поздний процессор для использования DIV286. Для создания аналогичной программы для системы 8086 и 8088 замените строки 42–61 описанной раньше более простой процедурой DivFault. Возможно, вы назовете эту программу DIV86.ASM.

Программа 10.2. DIV286.ASM

```
1:    %TITLE "80286 and later-model Divide-Fault ISR -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL tiny
6:
7:    cr    EQU   13
8:    lf    EQU   10
9:
10:
11:         DATASEG
12:
13:   welcome     DB              cr, lf, '80286/386 Divide-Fault Handler Installed'
14:               DB              cr, lf, 'Address = ', 0
15:   string      DB              40 dup (?)
16:
17:
18:         CODESEG
19:
20:         ORG   100h            ; Стандартный адрес .COM (первоначальный)
21:
22:         EXTRN StrWrite:proc, BinToAscHex:proc, NewLine:proc
23:
24:   Start:
25:         jmp Begin             ; Перепрыгнуть через резидентную ISR
26:
27:   %NEWPAGE
28:   ;-------------------------------------------------------------------------------
29:   ; DivFault Divide-Fault handler ISR (ISR обработки ошибки деления)
30:   ;-------------------------------------------------------------------------------
31:   ; Вход:
32:   ; нет (вызывается внутренним образом при ошибке в DIV или IDIV)
33:   ; Выход:
34:   ; ax = 0 (al = 8-битовое частное, ax = 16-битовое частное)
35:   ;
36:   ; Замечание: программа нормально продолжается с команды,
37:   ; следующей за DIV или IDIV, которая вызвала ошибку
38:   ;
39:   ; Регистры:
40:   ; ax меняется
41:   ;-------------------------------------------------------------
42:   PROC DivFault
43:         sti                   ; Разрешить прерывания CPU
44:         push  bp              ; Сохранить текущий регистр bp
45:         mov   bp, sp          ; Адресация значений в стеке через ss:bp
46:         push  si              ; Сохранить другие изменяемые регистры
47:         push  ds
48:         lds   si, [bp + 2]    ; Адресовать DIV или IDIV через ds:si
49:         lodsw                 ; Получить DIV плюс второй байт (в ah)
50:         and   ah, 0C0h        ; Выделить первые два бита (поле MOD)
51:         cmp   ah, 0C0h        ; Биты = 1? (основанная на регистрах instr)
52:         je    @@10            ; Переход, если да - DIV длиной 2 байта
```

```
53:           add    [word bp + 2], 2     ; DIV равна 4 байт, добавить 2 к смещению
54:     @@10: add    [word bp + 2], 2     ; Добавить 2 (или еще 2) к смещению
55:           xor    ax, ax               ; Установить частное в 0 (remainder также 0
56:                                       ; в случае 8-битового деления)
57:           pop    ds                   ; Восстановить сохраненные регистры
58:           pop    si
59:           pop    bp
60:           iret                        ; Вернуться из прерывания
61:     ENDP DivFault
62:
63:     Begin:
64:           mov    ax, 2500h            ; Установить новый вектор для деления
65:           mov    dx, offset DivFault
66:           int    21h
67:           mov    di, offset welcome   ; Высветить сообщение приветствия
68:           call   StrWrite
69:           mov    ax, cs               ; Высветить значение сегмента
70:           call   ShowAX
71:           mov    dl, ':'              ; Высветить двоеточие (:)
72:           mov    ah, 2
73:           int    21h
74:           mov    ax, offset DivFault        ; Высветить значение смещения
75:           call   ShowAX
76:           call   NewLine
77:
78:     ;----- Закончить и остаться резидентной, оставив код вплоть до
79:     ; конца новой ISR обработки ошибки деления
80:
81:     Exit:
82:           mov    dx, offset Begin     ; Новый адрес свободной памяти
83:           int    27h                  ; Закончить, оставшись резидентной
84:
85:     ;----- Подпрограмма для выдачи AX в шестнадцатеричном представлении
86:
87:     PROC ShowAX  ; Показать величину AX
88:           mov    cx, 4                ; Минимальное количество символов
89:           mov    di, offset string    ; Адрес строковой переменной
90:           call   BinToAscHex          ; Преобразовать AX в шестнадцатеричный формат
91:           call   StrWrite             ; Высветить шестнадцатеричную строку
92:           ret                         ; Вернуться в вызвавшую программу
93:     ENDP ShowAX
94:
95:           END    Start                ; Конец программы / точка входа
```

Тестирование DIV286

Для того чтобы протестировать то, что происходит до и после выполнения DIV286, ассемблируйте программу 10.3, DIVFAULT.ASM, которая и приводит к возникновению ошибки деления. Как обычно, выполните ассемблирование и компоновку с MTA.LIB:

```
tasm divfault
tlink divfault,,, mta
```

Запустите тестовую программу, набрав divfault и нажав на <Enter>. Это должно привести к появлению сообщения DOS "Divide Overflow" и преждевременной остановке программы. В зависимости от вашей версии DOS (и, возможно, от того, какие резидентные программы загружены в память) вы, по-видимому, должны будете перезагрузиться, нажав <Ctrl+Alt+Delete>. Известно, что некоторые версии DOS работают непредсказуемо при возникновении ошибки деления.

Затем выполните DIV286 для установки резидентной TSR. (На системах 8086 и 8088 запустите модифицированную программу DIV86. Но не запускайте DIV86, если ваша система имеет процессор 80286 или более поздний.) Затем снова запустите DIVFAULT. На этот раз вы увидите сообщение — "Program continued normally" (Программа продолжает работу нормально). Тем самым доказано, что DOS не останавливает больше программу, получая прерывание ошибки деления.

Замечание

Запускайте DIV286 или DIV86 единственный раз, либо вы установите в памяти несколько бесполезных копий обработчика прерываний ошибки деления.

Программа 10.3. DIVFAULT.ASM

```
1:    %TITLE "Divide Fault Demonstration -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:    cr    EQU   13                ; ASCII возврат каретки
9:    lf    EQU   10                ; ASCII символ новой строки
10:
11:
12:         DATASEG
13:
14:   exCode      DB                0
15:   message1    DB                cr,lf,'Forcing a divide by zero fault...',0
16:   message2    DB                cr,lf,'Program continued normally',cr,lf,0
17:
18:
19:   CODESEG
20:
21:   ;----- Из STRIO.OBJ
22:         EXTRN StrWrite:proc
23:
24:   Start:
25:         mov   ax, @data         ; Инициализировать DS по адресу
26:         mov   ds, ax            ; сегмента данных
27:         mov   es, ax            ; Сделать es = ds
28:
29:         mov   dx, offset message1       ; Адрес приветственного сообщения
30:         call  StrWrite
31:
32:         mov   ax, 100h          ; Приписать значение ax
33:         xor   bx, bx            ; Нулевой делитель
34:         div   bx                ; Форсировать ошибку деления
35:
36:   Exit:
37:         mov   dx, offset message2       ; Адрес сообщения о продолжении
38:         call  StrWrite          ; Выдать строку
39.
40:         mov   ah,04Ch           ; Функция DOS : Выход из программы
41:         mov   al,[exCode]       ; Вернуть значение кода возврата
42:         int   21h               ; Вызвать DOS. Закончить программу
43:
44:         END   Start             ; Конец программы / точка входа
```

Как работает DIV286

DIV286 вызывает программное прерывание завершить-и-остаться-в-памяти (TSR) 27h в строке 83, устанавливая ISR ошибки деления. Выполнение int 27h возвращает управление COMMAND.COM, но говорит DOS о сохранении всей занимаемой памяти до адреса cs:dx. В строке 82 dx устанавливается по адресу смещения сразу после последней команды, остающейся в памяти — в примере это iret в строке 60. Есть и другие способы установить TSR, например с помощью функции DOS 031h. Но когда размер программы относительно невелик (меньше, чем 64 Кбайт), намного проще использовать прерывание 27h.

Заметьте, что директива DATASEG используется для того, чтобы объявить переменные в строках 11–15. Поскольку это .COM-программа, сегменты кода и данных — одно и то же. Сегмент стека в .COM-программе занимает тот же сегмент из 64 Кбайт; следовательно, в программе не определяется отдельный стек командой STACK.

Замечание

Кстати, переменные, определенные после директивы DATASEG в .COM-программе, хранятся вне выполняемого кода (по большим адресам). Переменные, которые должны остаться в памяти после окончания программы, определяются в сегменте кода по смещению меньше (по меньшим адресам) адресов, указанных в int 27h в cs:dx.

Установка кода TSR в памяти

Первая команда в резидентной программе обычно перескакивает код, остающийся в памяти после окончания программы (см. строку 25). Первая настоящая команда в программе находится в том месте, куда осуществляется этот переход — в DIV286 на метку Begin: (строка 63). Здесь вектор прерывания ошибки деления заменяется адресом новой ISR — резидентной части этой программы в строках 28–61.

Осознайте, что DIV286 — это на самом деле две программы, удобно объединенные. Код, который выполняется, когда вы запускаете DIV286, начинается в строке 25, перескакивает на строку 63 и заканчивается в строке 93. В это время резидентная процедура DivFault не выполняется. Вместо этого ISR остается в памяти после окончания DIV286, обрабатывая ошибку деления, когда та возникает. Единственная цель программы DIV286 состоит в том, чтобы установить ISR DivFault и выдать сообщение, об этом напоминающее. Для того чтобы помочь вам определить местонахождение кода в памяти (если это вам надо), DIV286 отображает также адрес, по которому находится DivFault.

После того как DIV286 закончилась, оставив ISR DivFault, последующие прерывания ошибки деления выполняют ISR, начиная со строки 43, где сразу же выполняется sti, разрешая обработку других прерываний во время работы DivFault. Здесь стек содержит системные флаги плюс адреса команд div и idiv, которые приводят к возникновению прерывания. Заимствуя популярный подход из языков высокого уровня, DivFault локализует адреса возврата в стеке, выполняя в начале следующие команды:

```
push bp       ; Сохранить текущее значение bp
mov  bp, sp   ; Адресовать стек через bp
```

Важен порядок этих команд. Сначала сохраняется текущее значение регистра bp, чем бы не было заполнено пространство стека. Затем значение указателя на стек sp переписывается в bp, таким образом адресуя стек через ss:bp. (Адресация памяти регистром bp не влияет на адресацию сегмента регистром ss. Вы могли бы использовать другие регистры для адресации данных в стеке, но bp наиболее удобен для этого.)

Рис. 10.1 иллюстрирует, как выглядит стек во время выполнения ISR DivFault. (Адрес возврата, флаги и другие величины стека составляют то, что называется фреймом стека.) Когда адресуются переменные стека, полезно нарисовать диаграмму фрейма стека. Обращение к неправильным данным в стеке может иметь разрушительные последствия, так что для ошибки не много надо. На рис. 10.1 показан указатель стека на различных стадиях выполнения DivFault:

```
sp0:   Указатель стека до возникновения прерывания ошибки деления
sp1:   Указатель стека после обработки сигнала прерывания ошибки деления
       Процессор протолкнул в стек флаг и регистры cs и ip
sp2:   Указатель стека после проталкивания текущей величины bp
sp3.   Указатель стека после проталкивания регистров si и ds
```

План состоит в том, чтобы прочитать значения cs:ip из стека, просмотреть команды div и idiv и увеличить адрес возврата либо на 2, либо на 4 байта. Для того чтобы это осуществить, регистру bp приписывается значение sp2, — тем самым устанавливается адресация на байт номер 4. (Эти номера на рис. 10.1 проставлены только для ссылок — они не являются реальными адресами памяти.) Так как каждый прямоугольник на рисунке представляет собой 2-байтовое слово, значение 16- битового регистра ip расположено в [bp+2]. Значение регистра cs находится в [bp+4]. Если вы хотите обратиться к флагам, хранимым в стеке, можно использовать [bp+6].

В строке 48 DIV286.ASM выполняется lds для того, чтобы загрузить регистры ds и si адресом команды div или idiv, вызвавшей прерывание ошибки деления. Вы могли бы точно так же использовать две команды mov для загрузки слов в [bp+2] и [bp+4], но lds выполняет ту же работу — она короче и немного быстрее. (Можно использовать в качестве получателя части смещения адреса любой 16-битовый регистр, а не только si.)

После строки 48 ds:si адресует вызвавшие ошибку div или idiv. В строке 49 первое слово этой команды загружается для проверки в ax. Если первые два бита равны 1, то это 2-байтовая команда; в противном случае это 4-байтовый вариант. В строках 53–54 часть смещения адреса возврата, находящегося в стеке, увеличивается на 2 или 4 байта соответственно.

Весь смысл этих действий состоит в том, чтобы проигнорировать div или idiv, которая вызвала прерывание 0. Регистр ax очищается (строка 55), устанавливая 8-битовое (al) или 16-битовое (ax) в 0. (Замечание: для 8-битового деления этим также устанавливается в 0 остаток.) Из-за того что адрес возврата был увеличен, когда прерывание заканчивается в строке 60, выполнение программы продолжается командой, которая следует за ошибочным делением.

Рис. 10.1. Область стека во время выполнения ISR DivFault в DIV286.ASM

Прерывания, управляющие последовательным обменом

У DOS есть свои критики, но даже ее приверженцы сходятся в одном: асинхронный последовательный порт ввода-вывода, применяемый через DOS, настолько же полезен, как туфли для русалки. Хотя в DOS есть две функции, позволяющие читать (функция 3) и писать (функция 4) символы в последовательный I/O порт, специалисты едины во мнении, что на программы, использующие эти функции, полагаться нельзя, за исключением, возможно, работы с самой низкой скоростью обмена. Есть, по меньшей мере, три способа решения этой проблемы.

1. Написать отдельный драйвер для чтения и записи в последовательный порт как в поименованный файл.
2. Вызывать асинхронное прерывание 14h BIOS непосредственно для всех последовательных обменов.
3. Установить код работы с прерыванием для независимого от DOS и BIOS чтения и записи символов.

Первый способ — это хорошая идея, в особенности если необходимо обеспечить доступ к специальному коммуникационному оборудованию — к внешней карте мультипорта, например. Однако написание пользовательских драйверов — предмет обсуждения, который потребовал бы целой главы, и, значит, нереально рассматривать это решение здесь. (Большинство хороших руководств по программированию DOS рассматривают эту задачу в деталях.) Второй способ также хорош. ROM BIOS на всех PC обрабатывает асинхронный последовательный I/O с отличным результатом. Но, даже несмотря на то что третий способ требует прямого обращения к регистрам системы — тем самым создавая трудности для перевода программы на несовместимые PC — средства управления асинхронным последовательным I/O через прерывание делают написание коммуникационных программ настолько простым делом, по сравнению с другими двумя методами, что большинство программистов предпочитают этот подход.

Программа 10.4, ASYNCH.ASM, может служить базисом для любых коммуникационных программ. Программа предоставляет буферизированный, управляемый прерываниями входной канал для приходящих данных и использует не управляемый прерываниями метод для выхода. После текста программы приведен пример программы, который демонстрирует, как использовать ASYNCH модуль. Выполните ассемблирование, компоновку и установите ASYNCH в MTA.LIB командами:

```
tasm /zi asynch
tlib /E mta -+ asynch
```

Как обычно, не обращайте внимания на предупреждения о том, что ASYNCH не находится в библиотеке. Если вы что-нибудь поменяете в ASYNCH.ASM, повторите эти два шага.

Замечание

Поменяйте присваиваемое значение в строке 9 на 0 для COM1: или на 1 для COM2:

```
1:    %TITLE "Asynch Serial Comm Module -- by Tom Swan"
2:
3:        IDEAL
4:
5:        MODEL small
6:
7:        PUBLIC ComPort
8:
9:    ComPort = 0                         ; 0 = COM1:, 1 = COM2:
10:
11:   IF ComPort EQ 0
12:       Port   EQU 03F8h                ; Базовый адрес 8250
13:       VectorNum    EQU 0Ch            ; Номер вектора прерывания
14:       EnableIRQ    EQU 0EFh           ; Маска для разрешения 8259 IRQ
15:       DisableIRQ   EQU 10h            ; Маска для запрета 8259 IRQ
16:
17:   ELSE IF ComPort EQ 1
18:       Port         EQU 02F8h          ; Те же комментарии, что и выше
19:       VectorNum    EQU 0Bh
20:       EnableIRQ    EQU 0F7h
21:       DisableIRQ   EQU 08h
22:   ELSE
23:       DISPLAY "ComPort must be 0 or 1"
24:       ERR
25:   ENDIF
26:
27:   ;----- Адреса регистров адаптера
28:
29:   TxRegister      = Port + 0          ; Регистр передачи
30:   RxRegister      = Port + 0          ; Регистр получения
31:   IntEnable       = Port + 1          ; Регистр разрешения прерывания
32:   IntIdent = Port + 2                 ; Идентификация прерывания
33:   LineControl     = Port + 3          ; Регистр контроля линии
34:   ModemControl    = Port + 4          ; Регистр контроля модема
35:   LineStatus      = Port + 5          ; Регистр статуса линии
36:   ModemStatus     = Port + 6          ; Регистр статуса модема
37:
38:   ;----- Другие константы
39:
40:   Ctrl8259_0 EQU  020h                ; Порт 8259
41:   Ctrl8259_1 EQU  021h                ; Порт 8259 (маскирован)
42:   EOI        EQU  020h                ; 8259 end-of-interrupt
43:   BufSize    EQU  2048                ; Размер буфера ввода
44:
45:
46:       DATASEG
47:
48:   vectorSeg       DW                  ?    , Сегмент старого вектора
49:   vectorOfs       DW                  ?    , Смешение старого вектора
50:   bufHead   DW    ?                   ; Указатель на начало буфера
51:   bufTail   DW    ?                   ; Указатель на хвост буфера
52:   buffer    DB    BufSize DUP (?)     ; Буфер ввода
53:
54:
55:       CODESEG
56:
57:       PUBLIC AsynchInit, AsynchStop, AsynchStat
58:       PUBLIC AsynchOut, AsynchIn, AsynchInStat
59:
60:   %NEWPAGE
61:   ;---------------------------------------------------------------
62:   ; EmptyBuffer Очистка буфера ввода
63:   ;---------------------------------------------------------------
64:   ; Замечание:
65:   ; Собственный модуль (Private to module)
66:   ; Вход:
67:   ; нет
68:   ; Выход:
69:   ; нет
70:   ; Регистры:
71:   ; нет
```

```
72:    ;-------------------------------------------------------------
73:    PROC EmptyBuffer
74:       cli    ; Предотвращение прерываний
75:       push   ax     ; Сохранить ax
76:       mov    ax, offset buffer          ; Когда буфер пуст,
77:       mov    [bufHead], ax              ; указатели на хвост и голову
78:       mov    [bufTail], ax              ; совпадают
79:       pop    ax                         ; Восстановить ax
80:       sti                               ; Разрешить прерывания
81:       ret                               ; Вернуться в вызвавшую процедуру
82:    ENDP EmptyBuffer
83:    %NEWPAGE
84:    ;---------------------------------------------------------------
85:    ; AsynchInit Проинициализировать последовательный порт и установить ISR
86:    ;---------------------------------------------------------------
87:    ; Вход:
88:    ; нет
89:    ; Выход:
90:    ; нет
91:    ;
92:    ; Замечание: выполнять совместно с вызовом int 14h
93:    ; для установки скорости обмена
94:    ;
95:    ; Замечание: ввод, управляемый прерыванием, начинается сразу же
96:    ; после выхода из подпрограммы
97:    ;
98:    ; Предупреждение: вы должны вызвать AsynchStop перед тем, как ваша
99:    ; программа закончится, во избежание разрушения системы!
100:   ;
101:   ; Регистры:
102:   ; ax, bx, dx
103:   ;---------------------------------------------------------------
104: PROC AsynchInit
105:
106:      call   EmptyBuffer               ; Инициализировать буфер
107:
108:   ;----- Сохранить и переопределить вектор прерывания
109:
110:      push   ds                        ; Сохранить регистры сегмента
111:      push   es
112:      mov    ax, 3500h + VectorNum     ; Получить адрес вектора
113:      int    21h                       ; Вызов DOS
114:      mov    [vectorSeg], es           ; Сохранить адрес сегмента
115:      mov    [vectorOfs], bx           ; Сохранить адрес смещения
116:      push   cs                        ; Адресация AsynchISR
117:      pop    ds                        ; ds:dx и вызов
118:      mov    dx, offset AsynchISR      ; функции DOS 25h для
119:      mov    ax, 2500h + VectorNum     ; установки нового адреса вектора
120:      int    21h
121:      pop    es                        ; Восстановить сохраненные регистры
122:      pop    ds
123:
124:   ;----- Разрешить линию прерывания (IRQ) 8259 для этого асинхронного адаптера
125:
126:      in     al, Ctrl8259_1            ; Прочитать разрешающую маску 8259
127:      and    al, EnableIRQ             ; Очистить маскирующий бит
128:      out    Ctrl8259_1, al            ; Записать новую маску 8259
129:
130:   ;----- Разрешить прерывание 8250 по готовности данных
131:
132:      mov    dx, LineControl           ; Сначала считать регистр линии контроля
133:      in     al, dx                       ; и очистить бит 7 (бит DLAB)
134:      and    al, 07Fh
135:      out    dx, al
136:      mov    dx, IntEnable             ; При DLAB = 0 установить бит 0
137:      mov    al, 1                     ; регистра разрешения прерывания
138:      out    dx, al                       ; в 1, разрешая прерывания
139:
140:   ;----- Очистить статус 8250 и регистры данных
141:
142:   @@10:
143:      mov    dx, RxRegister            ; Очистить регистр данных
144:      in     al, dx                       ; чтением из порта
145:      mov    dx, LineStatus            ; Очистить статус линии
```

```
146:      in    al, dx                        ; чтением из порта
147:      mov   dx, ModemStatus               ; Очистить статус модема
148:      in    al, dx                        ; чтением из порта
149.      mov   dx, IntIdent                  ; Проверить регистр идентификации
150:      in    al, dx                        ; прерывания
151:      test  al, 1                         ; Бит 1 должен быть 1
152:      jz    @@10                          ; Переход, если ожидается прерывание
153:
154:   ;----- Установить бит 3 регистра контроля модема
155:
156:      mov   dx, ModemControl              ; Прерывания будут
157       in    al, dx                        ; распознаваться до тех пор, пока
158:      or    al, 08h                       ; этот бит установлен в 1
159:      out   dx, al                        ; Сделано!
160:
161:   ;----- Очистить буфер снова, в случае если случайный символ
162:   ; сумел проскользнуть в него
163:
164:      call  EmptyBuffer                   ; Снова очистить буфер
165:
166:      ret                                 ; Вернуться в вызвавшую процедуру
167:   ENDP AsynchInit
168:   %NEWPAGE
169:   ;---------------------------------------------------------------
170:   ; AsynchStop Деинсталяция ISR Asynch
171:   ;---------------------------------------------------------------
172:   ; Вход:
173:   ; нет
174:   ; Выход:
175:   ; нет
176:   ;
177:   ; Предупреждение: всегда вызывайте AsynchStop перед окончанием вашей
178:   ; программы или неизбежно наступит разрушение системы!
179:   ;
180:   ; Регистры:
181:   ; al, dx
182:   ;---------------------------------------------------------------
183:   PROC AsynchStop
184:
185:   ;----- Маскировать (запретить) 8259 IRQ прерывание
186:
187:      in    al, Ctrl8259_1                ; Считать маску 8259
188:      or    al, DisableIRQ                ; Маскировать IRQ бит
189:      out   Ctrl8259_1, al                ; Записать новою маску
190:
191:   ;----- Запретить прерывание 8250
192:
193:      mov   dx, LineControl               ; Сначала считать регистр линии контроля
194:      in    al, dx                        ; и очистить бит 7 (бит DLAB)
195:      and   al, 07Fh
196:      out   dx, al
197:      mov   dx, IntEnable                 ; При DLAB = 0 очистить все биты
198:      xor   al, al                        ; для запрещения прерываний
199:      out   dx, al                        ; Записать новое значение регистра
200:
201:   ;----- Установить бит 3 регистра контроля модема в 0
202:
203:      mov   dx, ModemControl              ; Приписать адрес порта
204:      in    al, dx                        ; Получить текущий регистр
205:      and   al, 0F7h                      ; Очистить бит 3
206:      out   dx, al                        ; Выдать новое значение регистра
207:
208:   ;----- Прерывания запрещены. Восстановить сохраненный вектор прерывания
209:
210:      push  ds                            ; Сохранить регистр сегмента
211:      mov   ax, 2500h + VectorNum         ; Установить вектор прерывания
212:      mov   dx, [vectorOfs]               ; Получить сохраненное смещение
213:      mov   ds, [vectorSeg]               ; Получить сохраненный сегмент
214:      int   21h                           ; Установить вектор прерывания
215:      pop   ds                            ; Восстановить сохраненный регистр
216:
217:      ret                                 ; Вернуться в вызвавшую процедуру
218:   ENDP AsynchStop
219:   %NEWPAGE
```

```
220:  ;---------------------------------------------------------------
221:  ; AsynchStat Получение статуса вывода
222:  ;---------------------------------------------------------------
223:  ; Вход:
224:  ; нет
225:  ; Выход:
226:  ; ah = статус линии
227:  ; al = статус модема
228:  ; Регистры:
229:  ; ax, dx
230:  ;---------------------------------------------------------------
231:  PROC AsynchStat
232:     mov    ah, 3                    ; Номер функции получения статуса
233:     mov    dx, ComPort              ; 0 = COM1:, 1 = COM2:
234:     int    14h                      ; Вызов обслуживания BIOS RS232_IO
235:     ret                             ; Вернуться в вызвавшую процедуру
236:  ENDP AsynchStat
237:  %NEWPAGE
238:  ;---------------------------------------------------------------
239:  ; AsynchOut Output a byte (to output port)
240:  ;---------------------------------------------------------------
241:  ; Вход:
242:  ; al = символ (или байт) для вывода
243:  ; Выход:
244:  ; нет
245:  ; Регистры:
246:  ; нет
247:  ;---------------------------------------------------------------
248:  PROC AsynchOut
249:     push   dx                       ; Сохранить модифицированный dx
250:     push   ax                       ; Сохранить символ в al
251:  @@10:
252:     mov    dx, LineStatus           ; Адресация регистра статуса линии
253:     in     al, dx                      ; Получить статус линии
254:     and    al, 020h                 ; Выделить Trasmit Holding Reg
255:     jz     @@10                     ; Переход, если THRE не пуст
256:     pop    ax                       ; Восстановить символ
257:     mov    dx, TxRegister           ; Адресация регистра передачи
258:     out    dx, al                      ; Выдать символ в al
259:     pop    dx                       ; Восстановить сохраненный dx
260:     ret                             ; Вернуться в вызвавшую процедуру
261:  ENDP AsynchOut
262:  %NEWPAGE
263:  ;---------------------------------------------------------------
264:  ; AsynchIn Принять байт (из буфера)
265:  ;---------------------------------------------------------------
266:  ; Вход:
267:  ; нет
268:  ; Выход:
269:  ; al = символ из буфера
270:  ;
271:  ; Замечание: если буфер пуст, al будет нуль
272:  ;
273:  ; Выполняется совместно с вызовом AsynchInStat во избежание чтения
274:  ; из пустого буфера
275:  ;
276:  ; Регистры:
277:  ; al, bx
278:  ;---------------------------------------------------------------
279:  PROC AsynchIn
280:     xor    al, al                      ; Изначальная установка результата в нуль
281:     mov    bx, [bufTail]            ; Получить указатель хвоста
282:     cmp    bx, [bufHead]            ; Тест на пустоту буфера
283:     je     @@99                     ; Выход, если он пуст (al = 0)
284:     mov    al, [byte ptr bx]        ; В противном случае считать символ из буфера
285:     inc    [bufTail]                ; Увеличить указатель хвоста
286:     cmp    [word ptr bufTail], offset buffer + BufSize    ; В конце?
287:     jb     @@99                     ; Переход, если это не так
288:     mov    [bufTail], offset buffer ; В противном случае переустановить указатель хвоста
289:  @@99:
290:     ret                             ; Вернуться в вызвавшую процедуру
291:  ENDP AsynchIn
292:  %NEWPAGE
293:  ;---------------------------------------------------------------------
```

```
294:  ; AsynchInStat Получение статуса буфера ввода
295:  ;-------------------------------------------------------------------------
296:  ; Вход:
297:  ; нет
298:  ; Выход:
299:  ; dx = число байтов (или символов) в буфере
300:  ; Регистры:
301:  ; dx
302:  ;-------------------------------------------------------------------------
303:  PROC AsynchInStat
304:     mov   dx, [bufHead]            ; Получить указатель на голову
305:     sub   dx, [bufTail]            ; Вычесть хвост из головы
306:     jge   @@99                     ; Переход, если результат >= 0
307:     add   dx, BufSize              ; Обработать отрицательный результат
308:  @@99:
309:     ret                            ; Вернуться в вызвавшую процедуру
310:  ENDP AsynchInStat
311:  %NE WPAGE
312:  ;-------------------------------------------------------------------------
313:  ; AsynchISR Процедура обслуживания прерывания асинхронного ввода
314:  ;-------------------------------------------------------------------------
315:  ; Вход:
316:  ; нет
317:  ; Выход:
318:  ; нет (символ читается и хранится в буфере)
319:  ;
320:  ; Замечание: в этой версии игнорируется переполнение буфера
321:  ;
322:  ; Регистры:
323:  ; нет
324:  ;-------------------------------------------------------------------------
325:  PROC AsynchISR
326:     push  ax                       ; Сохранить модифицируемые регистры
327:     push  bx
328:     push  ds
329:     push  dx
330:
331:     mov   ax, @data                ; Адресовать внутренние данные через ds
332:     mov   ds, ax
333:     mov   dx, RxRegister           ; dx = Регистру получения
334:     in    al, dx                      ; Считать байт из порта
335:     mov   bx, [bufHead]            ; Установить указатель на голову
336:     mov   [byte ptr bx], al        ; Запомнить байт в буфере
337:     inc   bx                       ; Увеличить указатель на голову
338:     cmp   bx, offset buffer + BufSize   ; Указатель в конце?
339:     jb    @@10                     ; Перейти, если нет
340:     mov   bx, offset buffer        ; В противном случае переустановить на начало
341:  @@10:
342:     cmp   bx, [bufTail]            ; Проверка переполнения
343:     jne   @@20                     ; Перейти, если нет переполнения
344:     mov   bx, [bufHead]            ; Отменить увеличение указателя
345:  @@20:
346:     mov   [bufHead], bx            ; Сохранить новый указатель на голову
347:     mov   al, EOI                  ; Выдать end-of-interrupt (конец прерывания) в
348:     out   Ctrl8259_0, al           ; порт 8259
349:
350:     pop   dx                       ; Восстановить сохраненные регистры
351:     pop   ds
352:     pop   bx
353:     pop   ax
354:     iret                           ; Вернуться из прерывания
355:  ENDP AsynchISR
356.
357:     END   ; Конец модуля
```

Запуск демонстрации ASYNCH

Программа 10.5, TRM.ASM, демонстрирует, как использовать набор программ ASYNCH. Хотя TRM и не является завершенным терминальным эмулятором, она полезна для отладки связи с удаленной системой. Часто бывает полезно иметь возможность видеть не только нормальный текст ASCII, но также каждый контрольный байт и все, что происходит в процессе коммуникационного обмена. TRM отображает обычный текст нормальным образом, а ASCII-значения

управляющих кодов берет в скобочки. Например, перевод строки отображается, как [13][10]. Показанная выше последовательность управляющих кодов, приходящая от удаленного источника, — часто это все, что надо для решения коммуникационных проблем. Выполните ассемблирование и компоновку TRM, используя команды:

```
tasm /zi trm
tlink /v trm,,, mta
```

Замечание

Если вы имеете доступ к двум PC, свяжите их последовательным кабелем и запустите TRM на обеих машинах. Затем наберите управляющие коды и нажмите <Esc>, <Enter> и т.д. для того, чтобы увидеть, как TRM отображает текст и управляющие символы. Если у вас нет двух PC, вы, наверное, сможете использовать TRM с модемом, но нужно будет либо вводить команды инициализации модема вручную, либо использовать полноценную терминальную программу для того, чтобы подключиться к удаленной системе до запуска TRM

Программа 10.5. TRM.ASM

```
%TITLE "Terminal Emulator -- Copyright (c) 1989,1995 -- by Tom Swan"

        IDEAL

        MODEL small
        STACK 1024

;----- Из ASYNCH.OBJ
        EXTRN ComPort:abs

cr              EQU 13                  ; ASCII-возврат каретки
lf              EQU 10                  ; ASCII-символ новой строки
bd9600          EQU 0e3h                ; 9600 бод, без проверки четности, 1 стоп, 8 бит
ExitKey         EQU 100                 ; GetCh величина для F10


        DATASEG

exCode          DB                  0

welcome         DB cr, lf, 'Terminal Emulator by Tom Swan', cr, lf
                DB cr, lf, 'Configured for 9600 baud. Displays'
                DB cr, lf, 'control codes in brackets for debugging'
                DB cr, lf, 'an RS232 serial line. Press F10'
                DB cr, lf, 'to exit.', cr, lf, lf, 0

string          DB 80 DUP (?)           ; Многоцелевая строка


        CODESEG

;----- Из ASYNCH.OBJ
        EXTRN AsynchInit:proc, AsynchStop:proc, AsynchStat:proc
        EXTRN AsynchOut:proc, AsynchIn:proc, AsynchInStat:proc

;----- Из KEYBOARD.OBJ
        EXTRN KeyWaiting:proc, GetCh:proc

;----- Из BINASC.OBJ
        EXTRN BinToAscDec:proc

;----- Из STRIO.OBJ
        EXTRN StrWrite:proc

Start:
        mov     ax, @data               ; Инициализировать DS адресом
        mov     ds, ax                  ; сегмента данных
        mov     es, ax                  ; Сделать es = ds

        mov     di, offset welcome      ; Выдать приветственное сообщение
        call    StrWrite

;----- Инициализировать скорость обмена и пакет Asynch

        mov     ah, 0                   ; Функция инициализации BIOS RS232
```

```
56:         mov   al, bd9600          ; конфигурации
57:         mov   dx, ComPort         ; Номер порта (0 or 1)
58:         int   14h                 ; Вызвать обслуживание RS232_IO
59:         call  AsynchInit          ; Инициализировать пакет асинхронного обмена
60:
61:     ;----- Выполнить эмуляцию терминала I/O
62:
63:     Emulate:
64:         call  AsynchInStat        ; Появился ли уже какой-либо символ?
65:         or    dx, dx              ; Проверить, если dx > 0
66:         jz    @@10                ; dx = 0, проверить нажатие клавиш
67:         call  AsynchIn            ; Считать символ из буфера
68:         call  DispChar            ; Локально отобразить символ
69:         jmp   Emulate             ; Продолжить эмуляцию
70:     @@10:
71:         call  KeyWaiting          ; Проверить, была ли нажата клавиша
72:         jz    Emulate             ; Цикл, если нет
73:         call  GetCh               ; В противном случае получить нажатую клавишу
74:         jnz   @@20                ; Переход, если не клавиша fn или ctrl
75:         cmp   al, ExitKey         ; Нажата ли клавиша окончания программы?
76.         je    Exit                ; Переход на Exit, если да
77:     @@20:
78:         call  AsynchOut           ; В противном случае послать символ дальше
79:         jmp   Emulate             ; Цикл, пока все не сделано
80:
81:     ;----- Конец эмуляции. Деинициализировать пакет Asynch и выйти
82:
83:     Exit:
84:         call  AsynchStop          ; Остановить пакет Asynch
85:         mov   ah,04Ch             ; Функция DOS: Конец программы
86:         mov   al,[exCode]         ; Вернуть код возврата
87:         int   21h                 ; Вызвать DOS  Закончить программу
88:
89:     %NEWPAGE
90:     ;-------------------------------------------------------------------------
91:     ; DispChar/OneChar Отобразить любой символ ASCII
92:     ;-------------------------------------------------------------------------
93:     ; Вход:
94:     ; al = значение ASCII (0..255)
95:     ; Выход:
96:     ; нет
97:     ;
98:     ; Замечание: управляющие коды отображаются как [13] [10] и т.д.
99:     ; для отладки линии последовательного I/O
100:    ; Регистры:
101.    ; ax, cx, dl, di
102:    ;-------------------------------------------------------------------------
103:    PROC DispChar
104:        cmp   al, 32              ; Символ управляющий?
105:        jae   OneChar             ; Переход, если нет
106:
107.    ;----- Отобразить управляющие символы в скобках
108:
109:        xor   ah, ah              ; Преобразовать al в 16-битовую величину
110:        mov   cx, 1               ; Установить по меньшей мере один символ
111:        mov   di, offset string   ; Адресация строковой величины
112:        call  BinToAscDec         ; Преобразовать в строку
113:        mov   al, '['             ; Выдать символ "["
114.        call  OneChar             ; Выдать символ в al
115:        call  StrWrite            ; Выдать строку ctrl-кода
116:        mov   al, ']'             ; "Пролететь насквозь" в OneChar
117:
118:    PROC OneChar
119:        mov   dl, al              ; Приписать символ dl
120:        mov   ah, 2               ; Функция DOS вывода символа
121:        int   21h                 ; Вызов DOS для отображения символа
122:        ret                       ; Вернуться в вызвавшую процедуру
123:    ENDP OneChar
124:
125:    ENDP DispChar
126:
127:        END   Start               ; Конец программы / точка входа
```

Как работает TRM

Программа 10.5, TRM.ASM, демонстрирует, как использовать процедуры из набора программ ASYNCH. В строках 55–58 вызывается функция BIOS 14h для того, чтобы инициализировать главный последовательный порт, посылая значение скорости обмена и другие параметры в регистр al. Используемая по умолчанию установка — скорость 9600 бод, без проверки четности, 1 стоп-бит и 8 бит данных (см. строку 13).

В табл. 10.3 перечислены назначения битов в 8-битовой величине, посылаемой в al, когда ah=0 и dx определен величиной ComPort прерывания BIOS 14h. Вверху таблицы перечислены порядковые номера битов и их назначения, внизу представлены битовые установки, которые можно использовать для выбора различных конфигурационных параметров.

Таблица 10 3. **Конфигурационные биты прерывания 14h**

7	6	5 (скорость обмена)	4	3 (четность)	2 (биты остановки)	1	0 (биты данных)
0	0	0 (110)	0	0 (нет)	0 (1)	0	0 (???)
0	0	1 (150)	0	1 (нечетный)	1 (2)	1	0 (7)
0	1	0 (300)	1	1 (четный)		1	1 (8)
0	1	1 (600)					
1	0	0 (1200)					
1	0	1 (2400)					
1	1	0 (4800)					
1	1	1 (9600)					

В строке 59 вызывается AsynchInit для установки обработчика прерываний AsynchISR. Имейте в виду, что приходящие данные будут храниться в буфере ввода до тех пор, пока AsynchInit не закончит работу — так что не медлите с проверкой приходящих данных после этого этапа. В цикле в строках 63–79 проверяется вход, читаются символы из входного буфера, отслеживается нажатие клавиш и осуществляется выход, когда вы нажмете <F10> (Нажатие <ESC> для завершения этой программы некорректно, при этом символ Esc будет послан на удаленную машину.) Подпрограмма DispChar в строках 90–125 отображает значения ASCII или управляющие коды.

DispChar демонстрирует ассемблерный трюк, стоящий изучения. Исследуем вложенную процедуру OneChar в строках 118–123, которая отображает отдельный символ, вызывая функцию DOS 2. Перед этим в строке 114 (во внешней процедуре) вызывается OneChar. Но посмотрите внимательнее на всю программу DispChar — там только *одна команда возврата в строке 122, несмотря на то что здесь две подпрограммы. Это не ошибка! После команды* mov в строке 116 программа "проваливается насквозь" к подпрограмме OneChar, выполняя этот код как расширение внешней программы DispChar. Раньше, однако, DispChar вызывает эту внутреннюю часть самой себя как подпрограмму. Когда выполняется call в строке 114, ret в строке 122 возвращает управление обратно в 115. Когда программа проваливается насквозь к OneChar после строки 116, та же самая команда ret возвращает управление обратно коду, изначально вызвавшему DispChar. Когда вы используете этот фокус, обязательно тщательно продокументируйте вашу программу, чтобы другие смогли разобраться, что к чему.

Как использовать набор программ ASYNCH

ASYNCH.ASM содержит семь программ для чтения и записи асинхронных последовательных данных при любой скорости обмена, поддерживаемой вашим оборудованием. Перечислим эти программы.

1. AsynchInit — Инициализация пакета ASYNCH
2. AsynchStop — Деинициализация пакета ASYNCH
3. AsynchStat — Возврат статуса последовательного порта
4. AsynchOut — Запись одного байта в последовательный порт
5. AsynchIn — Чтение одного байта из буфера ввода
6. AsynchInStat — Возврат статуса буфера ввода
7. AsynchISR — Процедура обработки прерывания ввода

Программы никогда не вызывают AsynchISR напрямую — программу обработки прерывания, которая автоматически обрабатывает ввод из последовательного порта.

В основном вы будете использовать другие шесть процедур в таком порядке.

1. Вызвать прерывание ROM BIOS 14h для установки скорости связи. Поскольку в PC этот код инициализации уже встроен, ASYNCH не повторяет его.
2. Вызвать AsynchInit для инициализации пакета ASYNCH и установки кода AsynchISR.

3. Использовать AsynchStat для определения статуса последовательного порта — например, чтобы увидеть, готово ли оборудование для выдачи символа.
4. Вызвать AsynchOut для того, чтобы послать символы удаленной системе.
5. Вызвать AsynchInStat для выяснения, есть ли в буфере ввода какие-либо символы.
6. Если AsynchInStat сообщит, что хоть один символ в буфере есть, необходимо вызвать AsynchIn для изъятия этого символа.
7. Вызвать AsynchStop для того, чтобы отключить AsynchISR код и остановить ввод, управляемый прерыванием.

Замечание

Удостоверьтесь, что AsynchStop вызвана до окончания программы, в противном случае разрушение системы практически гарантировано. Оставление AsynchISR (как и любой другой ISR) в памяти после передачи управления COMMAND.COM, без сомнении, вызовет серьезные проблемы

Переменные и константы ASYNCH

В ASYNCH.ASM осуществляется задание набора констант для адресации двух взаимосвязанных цепей: микросхемы 8250 асинхронного порта ввода-вывода и контроллера прерываний 8259, которым вы учились управлять в этой главе ранее. Строка 9 определяет, будет ли пакет использовать доступ через первый (ComPort=0) или второй (ComPort=1) последовательный порт, имеющийся в распоряжении на большинстве PC. В строке 7 эта константа определяется как общая. В ваших собственных программах заимствуйте величину ComPort добавлением такой строки к вашим константам:

```
EXTERN ComPort:abs
```

В строках 11–25 приписываются значения четырем константам, в зависимости от величины ComPort. Обратите внимание, как в строках 22–25 обрабатываются ошибки. Попробуйте ассемблировать программу с ComPort, равной 3, для того чтобы увидеть эти команды в работе. В строке 23 директивой DISPLAY выводится сообщение об ошибке. Затем строка 24 выполняет ERR, отображая пользовательское сообщение об ошибке Turbo Assembler и предотвращая создание .OBJ-файла.

В строках 29–36 приписываются значения константам, необходимые для чтения и записи регистров, расположенных по различным смещениям относительно базового значения Port, которое инициализируется либо в 12-й, либо в 18-й строке. Программа использует эти величины для управления микросхемой 8250 напрямую, без вызова процедур DOS или BIOS. Некоторые другие константы в строках 40–43 описывают работу с контроллером прерываний 8259.

Вы можете изменить BufSize (строка 43) для того, чтобы увеличить или уменьшить размер буфера ввода. Оптимальный размер зависит от того, какого типа коммуникационную программу вы пишете. Программа, которая читает и пишет строки текста, может обойтись маленьким буфером, возможно, не больше 256 байт. Терминальный эмулятор должен, наверное, иметь возможность сохранить эквивалент нескольких экранов текста в памяти. Значение 2048, задаваемое по умолчанию, — это разумный компромисс.

Круг вокруг буфера Asynch

Переменные в строках 50–52 резервируют пространство для буфера ввода. Два указателя — bufHead и bufTail адресуют байты в этом буфере. Когда эти переменные указывают на один и тот же адрес, буфер пуст. Новые байты сохраняются в буфере по адресу bufHead. Байты извлекаются из буфера по адресу bufTail. Эти два указателя увеличиваются до тех пор, пока не достигнут конца буфера, и тогда они переустанавливаются в начальные значения. По мере того как данные приходят и уходят, bufHead и bufTail "гоняются" один за другим вокруг буферного пространства, создавая структуру, называемую *очередью*, в которой данные, пришедшие в буфер последними, первыми его покидают. Изучите строки 280–346 для того, чтобы разобраться, как эти структуры применяются в ASYNCH.

Замечание

Обработка ошибок в AsynchISR минимальная. Если буфер переполняется, последующий символ просто игнорируется. Это означает, что ваша программа должна вызывать AsynchIn достаточно часто для того, чтобы предотвратить переполнение. Если это невозможно, вы должны модифицировать AsynchISR так, чтобы а) установить флаг, указывающий на то, что переполнение произошло; б) послать сигнал остановки на удаленную систему, для того чтобы предотвратить новый ввод. *Обычно стоп-сигнал посылается за несколько символов до того, как произойдет переполнение, чтобы дать удаленной системе* возможность определить состояние переполнения. Конечно, затем вы должны послать удаленной системе сигнал о готовности получать входные данные снова. Смотрите в документации ссылки на подробные руководства по последовательному обмену.

Отладка с прерываниями

Прерывание точки остановки, номер 3, зарезервировано для отладки, (см. табл. 10.2). Несмотря на то что Turbo Debugger позволяет вам нажать <F2> для установки точки остановки, прерывающей программу непосредственно перед выполнением данной команды, вы также можете вызвать временную остановку, вставив такую строку:

```
int 3 ; Установить точку остановки
```

Когда вы запустите программу с этой командой под управлением Turbo Debugger (и большинства других отладчиков), программа остановится по выполнении int 3. Когда же программа будет запущена из-под DOS, точка остановки не даст никакого эффекта, так как вектор прерывания 3 в DOS обычно указывает на команду iret. Можно вставить в программу столько команд int 3, сколько захотите. Вы, возможно, выясните, что установка большого числа точек остановки в большой программе осуществляется проще, чем другие методы, предоставляемые Turbo Debugger.

Пошаговое выполнение

Установка флага ловушки, или трассировки (tf=1), заставляет процессор работать в пошаговом режиме. В этом состоянии почти за всеми командами автоматически следует сигнал прерывания 1, позволяющий ISR проверять регистры и память, отображать переменные и отслеживать другие программные действия. Установка вашей собственной ISR для прерывания этого номера дает вам возможность контролировать выполняемую программу почти после любой команды.

Замечание

Turbo Debugger устанавливает tf для своих собственных команд пошагового режима, так что не используйте эту методику в программах, которые вы хотите запускать под управлением отладчика. Это также верно для других отладчиков.

Некоторые команды не вызывают прерывание 1. В эти команды входят все префиксы, такие как rep, присваивания через mov и pop регистрам сегментов (что, как вы помните, временно отключает все прерывания, в том числе и 1) и команда wait. Но после других команд, выполняемых при tf=1, осуществляются следующих три шага:

1. Флаги, регистры cs и ip проталкиваются в стек.
2. Флаги tf и if очищаются.
3. Выполняется ISR по вектору прерывания 1.

Из-за того что на втором шаге флаг ловушки и флаг прерывания очищаются, пошаговая ISR не работает в пошаговом режиме. Поэтому не беспокойтесь о том, что эта ISR будет пытаться самопроверяться, прерывая саму себя, даже если вы разрешите распознавание прерываний (что вы, вероятно, и должны сделать) выполнением sti в ISR. Когда ISR закончится, команда iret восстановит установки флагов, возвращая процессор обратно в пошаговый режим.

Установка и очистка tf

Поскольку встроенных команд по установке и очистке tf нет, надо найти для этого другой метод. Возможно, вы захотите использовать команды lahf и sahf, которые передают данные между флагами процессора и ah. Но это не сработает, потому что lahf и sahf влияют только на флаги af, cf, sf и zf, а tf не может быть изменен sahf.

Одно из решений этой проблемы состоит в том, чтобы протолкнуть флаги в стек по команде pushf, вытолкнуть значения флагов в ax, поменять бит tf, протолкнуть флаги обратно в стек и выполнить popf, передав значения измененных флагов обратно в регистр флагов:

```
pushf                   ; Протолкнуть флаги в стек
pop  ax                 ; Переправить флаги в ax
or   ax, 0100h          ; Установить бит tf = 1
push ax                 ; Протолкнуть модифицированные флаги в стек
popf                    ; Вытолкнуть стек в регистр флагов
```

Для восстановления tf, отменяющего пошаговое выполнение, замените команду or на and ax, 0FEFFh. Единственная проблема в этом методе заключается в том, что команды, отменяющие пошаговое исполнение, должны выполняться в пошаговом режиме. Хотя это, вероятно, никакого вреда не принесет, существует более элегантный метод — устанавливать флаг ловушки в разрешающее или запрещающее состояние внутри ISR, что, как вы помните, выполняется на полной скорости.

Программа 10.6, SINGLE.ASM, демонстрирует этот метод, запуская процессор в пошаговый режим для простой подпрограммы, считающей до 100. На протяжении этого времени, если локальный счетчик достигает 50, пошаговая ISR делает паузу для выдачи сообщения. Нажатие любой клавиши продолжает программу. Этим эмулируется напи-

сание пошаговой ISR для проверки переменных в памяти, что и вам, наверное, стоит сделать для того, чтобы выяснять, в какой части ошибочной программы эти переменные меняются. (В Turbo Debugger, конечно, есть команды для выполнения аналогичных операций, но знание того, как установить свой собственный код отладчика, пока еще является хорошей практикой.) Ассемблируйте и скомпонуйте SINGLE.ASM командами:

```
tasm single
tlink single,,, mta
```

Замечание

Не запускайте SINGLE в Turbo Debugger (как и в любом другом отладчике). Если отладчик переведет процессор в пошаговый режим, то возможен конфликт.

Программа 10.6. SINGLE.ASM

```
1:    %TITLE "Single-Step (Trap) Demo -- Copyright (c) 1989,1995 - by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:    cr    EQU   13                        ; ASCII-возврат каретки
9:    lf    EQU   10                        ; ASCII-символ новой строки
10:   Trapping    EQU               0       ; "Пошаговое выполнение разрешено"
11:   TurnOnTrap  EQU               1       ; Код для разрешения пошагового режима
12:   TurnOffTrap EQU               2       ; Код для запрещения пошагового режима
13:
14:
15:         DATASEG
16:
17:   exitC DB    0
18:
19:   spaces      DB                ' ', 0  ; Строка из 4 пустых символов
20:
21:   offMsg      DB                cr, lf, 'Single-step trap is off', cr, lf, 0
22:   onMsg       DB                cr, lf, 'Single-step trap is on', cr, lf, 0
23:   pauseMsg    DB                'Press any key to continue...', 0
24:   countMsg    DB                cr, lf, lf, 'Count = 50!', cr, lf, 0
25:
26:   trapSwitch  DB                0       ; Переключатель разрешения/запрещения ловушки
27:   string      DB                40 DUP (?)      ; Многоцелевая строка
28:   count       DW                ?       ; Для подпрограммы Counter
29:   trapSeg     DW                ?       ; Адрес вектора старого прерывания 1
30:   trapOfs     DW                ?       ,
31:
32:
33:         CODESEG
34:
35:   ;----- Из STRIO.OBJ, BINASC.OBJ, KEYBOARD.OBJ
36:         EXTRN StrWrite:proc, NewLine:proc, BinToAscDec:proc
37:         EXTRN GetCh:proc
38:
39:   Start:
40:         mov   ax, @data           ; Инициализировать DS адресом
41:         mov   ds, ax              ; сегмента данных
42:         mov   es, ax              ; Сделать es = ds
43:
44:   ;----- Сохранить вектор прерывания 1 и приписать его Stepper ISR
45:
46:         mov   ax, 3501h           ; Получить вектор прерывания 1
47:         int   21h                 ; Вызов DOS
48:         mov   [trapSeg], es       ; Сохранить значение сегмента
49:         mov   [trapOfs], bx       , Сохранить значение смещения
50:         push  ds                  ; Сохранить текущий регистр ds
51:         mov   ax, 2501h           ; Установить вектор прерывания 1
52:         push  cs                  ; по адресу
53:         pop   ds                  ; Stepper ISR
54:         mov   dx, offset Stepper
55:         int   21h
56:         pop   ds                  ; Восстановить ds
57:         push  ds                  ; Установить es равным ds
```

```
58:         pop   es
59:
60:     ;----- Выполнить код примера на полной скорости
61:
62:         mov   di, offset offMsg   ; Высветить "Trapping is off"
63:         call  Counter             , Вызвать подпрограмму примера
64:
65:     ;----- Выполнить код примера в пошаговом режиме
66:
67:         mov   di, offset onMsg    , Высветить "Trapping is on"
68:         mov   [trapSwitch], TurnOnTrap  ; Заставить ISR выполняться
69:         int   1                   , при установленной ловушке
70:         call  Counter             , Вызвать подпрограмму примера
71:         mov   [trapSwitch], TurnOffTrap ; Заставить ISR выполняться
72:     ; при выключенной ловушке
73:     ;----- Снова выполнить код примера на полной скорости
74:
75:         mov   di, offset offMsg   ; Высветить 'Trapping is off"
76:         call  Counter             ; Вызвать подпрограмму примера
77:
78:     Exit:
79:         push  ds                  ; Сохранить текущий регистр ds
80:         mov   ax, 2501h           ; Переустановить вектор прерывания 1
81:         mov   ds, [trapSeg]       ; по сохраненному адресу
82:         mov   dx, [trapOfs]       ; в trapSeg и trapOfs
83:         int   21h
84:         pop   ds                  , Восстановить ds
85:         mov   ah, 04Ch            , Функция DOS  Выход из программы
86:         mov   al, [exitC]         , Вернуть код возврата
87:         int   21h                 ; Вызов DOS. Закончить программу
88:
89:
90:     ;----- Подпрограмма: выдает строку, делает паузу и считает до 100
91:
92:     PROC Counter
93:         call  StrWrite            ; Высветить id сообщение
94:         call  Pause               ; Ожидать нажатия клавиши
95:         mov   [count], 0          ; Обнулить count
96:     @@10:
97:         inc   [count]             ; count <- count + 1
98:         mov   ax, [count]         ; Преобразовать count в строку
99:         mov   cx, 4               ; Минимальный размер строки
100:        mov   di, offset string
101:        call  BinToAscDec
102:        call  StrWrite            ; Выдать строку
103:        mov   di, offset spaces   ; Выдать 4 пробела
104:        call  StrWrite
105:        cmp   [count], 100        , Продолжать до тех пор, пока count = 100
106:        jb    @@10
107:        ret                       ; Вернуться в вызвавшую процедуру
108:    ENDP Counter
109:
110:
111:    ;----- Подпрограмма: Выдать сообщение и ожидать нажатия клавиши
112:
113:    PROC Pause
114:        mov   di, offset pauseMsg       ; Высветить сообщение о паузе
115:        call  StrWrite
116:        call  GetCh               ; Ждать нажатия клавиши
117:        call  NewLine             ; Начать новую строку для выдачи сообщений
118:        ret                       ; Вернуться в вызвавшую процедуру
119:    ENDP Pause
120:
121:
122:    %NEWPAGE
123:    ;---------------------------------------------------------------------
124:    ; Stepper ISR отлавливания в пошаговом режиме
125:    ;---------------------------------------------------------------------
126:    ; Вход:
127:    ; [trapSwitch] = TurnOnTrap
128:    ; Пошаговый режим разрешен
129:    ; [trapSwitch] = TurnOffTrap
130:    ; Пошаговый режим запрещен
131:    ; [trapSwitch] = ???
```

```
132:  ; no action
133:  ; Выходт:
134:  ; нет
135:  ; Регистры:
136:  ; нету
137:  ;-------------------------------------------------------------------------
138:  PROC Stepper
139:        sti   ; Разрешить прерывания
140:        push  bp                ; Сохранить текущий регистр bp
141:        mov   bp, sp            ; Адресовать стек через bp
142:        push  ax                ; Сохранить все регистры
143:        push  bx
144:        push  cx
145:        push  dx
146:        push  di
147:        push  si
148:        push  ds
149:        push  es
150:
151:  ;----- Адресовать внутренние данные через ds, es
152:
153:        mov   ax, @data         ; Инициализировать DS адресом
154:        mov   ds, ax            ; сегмента данных
155:        mov   es, ax            ; Сделать es = ds
156:
157:  ;----- Тестирование trapSwitch по включению/выключению пошагового режима
158:
159:        cmp   [trapSwitch], TurnOnTrap
160:        jne   @@10
161:        or    [word bp+6], 0100h ; Установить tf (разрешить ловушку)
162:        mov   [trapSwitch], Trapping    ; "Ловушка разрешена"
163:        jmp   @@99              ; Выход
164:  @@10:
165:        cmp   [trapSwitch], TurnOffTrap
166:        jne   @@20
167:        and   [word bp+6], 0FEFFh       ; Переустановить tf (запретить ловушку)
168:        jmp   @@99              ; Выход
169:
170:  @@20:
171:
172:  ;----- Вставить код пошагового выполнения здесь
173:
174:        cmp   [count], 50       ; count = 50?
175:        jne   @@99              ; Если нет, выход
176:        mov   di, offset countMsg       ; В противном случае высветить сообщение отсчета
177:        call  StrWrite
178:        call  Pause             ; и ожидать нажатия клавиши
179:        inc   [count]           ; для разрешения программе считать
180:        call  NewLine
181:
182:  @@99:
183:        pop   es                ; Восстановить все регистры
184:        pop   ds
185:        pop   si
186:        pop   di
187:        pop   dx
188:        pop   cx
189:        pop   bx
190:        pop   ax
191:        pop   bp
192:        iret                    ; Выйти из прерывания
193:  ENDP Stepper
194:
195:        END   Start             ; Конец программы / точка входа
```

Как работает SINGLE

Когда вы запускаете SINGLE, то получаете первое сообщение о том, что пошаговый режим выключен. Нажмите <Enter> и программа вызовет подпрограмму для счета от 1 до 100 на полной скорости. После этого установкой tf подключается пошаговый режим. Нажатие <Enter> еще раз вызывает подпрограмму счета, которая, как вы сможете увидеть, работает значительно медленнее из-за прерывания команд, передающего управление пользовательской

ISR. Когда эта ISR определит, что счетчик равен 50, она останавливает отсчет и просит вас нажать любую клавишу. Нажмите <Enter> для возобновления работы. Для того чтобы показать, что можно вернуться от пошагового выполнения к работе на полной скорости в любой момент времени, программа восстанавливает значение флага ловушки. Нажмите <Enter> в последний раз для нового отсчета на максимальной скорости исполнения.

Три константы в SINGLE — Trapping, TurnOnTrap и TurnOffTrap в строках 10–12 — определяют три состояния, распознаваемых Stapper ISR (строки 138–193). Байтовая переменная trapSwitch в строке 26 содержит одно из этих трех значений, определяющих путь, по которому выполняется Stepper. Если trapSwitch равен TurnOnTrap, Stepper разрешает пошаговое выполнение установкой флага tf. Если trapSwitch равен TurnOffTrap, то Stepper запрещает пошаговое выполнение переустановкой tf. Если trapSwitch равен Trapping, то Stepper запускает маленькую часть кода для изучения глобальной переменной count (см. строку 28). Когда count равен 50, программа отображает сообщение и просит вас нажать клавишу.

SINGLE начинается с сохранения текущего вектора для прерывания 1 и затем меняет этот вектор для того, чтобы адресовать пользовательскую ISR Stepper (строки 46–58). Затем программа вызывает подпрограмму Counter (строки 92–108), которая считает до 100, выводя колонки цифр на экране. После этого первого вызова Counter, который осуществляется на полной скорости, trapSwitch устанавливается в TurnOnTrap (строки 67–68). Затем строка 69 сразу же попадает в ловушку прерывания 1 командой программного прерывания:

```
int 1
```

Это вызывает выполнение ISR Stepper в первый раз. Когда ISR чувствует, что trapSwitch установлен в TurnOnTrap (строки 159—163), команда or меняет флаг tf, сохраненный в стеке командой int, используя регистр bp для адресации стековых переменных. После установки бита флага в стеке последующая команда iret, которая обновляет текущие флаги значением из стека, переводит процессор в пошаговый режим.

Если необходимо выключить пошаговый режим, команда and в строке 167 модифицирует бит флага в стеке (аналогично тому, как этот бит был установлен ранее) и переходит на выход ISR. На этот раз, до выполнения iret, tf остается выключенным (он отключен во время работы ISR, как вы помните), приводя к тому, что программа продолжает работать на полной скорости.

Изменив строки 174–180, можно использовать ISR Stepper в ваших собственных программах для просмотра, почти после любой команды, всего, что пожелаете. Для того чтобы это сделать, скопируйте строки 10–12, 26 и ISR Stepper, удалите строки 174–180 и вставьте ваши собственные команды. Затем, для разрешения пошагового выполнения, используйте команды:

```
mov [trapSwitch], TurnOnTrap
int 1
```

Для того чтобы запретить пошаговое выполнение, вернув процессор к полной скорости, выполните:

```
mov [trapSwitch], TurnOffTrap
```

Замечание

Перед возвратом в DOS вы должны запретить пошаговое выполнение в любой программе, которая устанавливает флаг tf. *Неподчинение этому правилу может привести к зависанию компьютера, что заставит вас его перезапустить.* Если программа закончилась непредвиденным образом, перезапускайтесь как можно быстрее — если сможете.

Резюме

Прерывание это сигнал, который заставляет выполняющуюся программу сделать паузу, запускает специальную подпрограмму, называемую процедурой обработки прерывания (ISR), после чего возобновляется нормальное выполнение. Для семейства процессора 8086 существует два вида сигналов прерываний: внешние и внутренние. Внешние прерывания могут возникать в любое время. Внутренние, или программные прерывания, возникают, только когда программа выполняет команду int либо into или когда выполняются некоторые условия, такие как ситуация деления на нуль.

Вследствие того что сигнал внешнего прерывания может возникнуть в любое время, внешние ISR должны сохранять все регистры. Флаги сохраняются процессором автоматически, когда он распознает сигнал прерывания. Внутренние ISR могут передавать значения регистров в программы, так же как это делают обычные подпрограммы. В любом случае прерывание не обрабатывается процессором до тех пор, пока не закончится текущая команда.

Маскируемые прерывания могут временно запрещаться командой cli и разрешены sti. Немаскируемые прерывания не могут быть запрещены. На PC прерывания, генерируемые внешним образом, направляются в микросхему контроллера прерываний (PIC) 8259, который разрешает конфликты между несколькими прерываниями и передает сигналы прерываний по одной входной линии процессора INTR.

Векторы прерываний хранятся в нижних адресах памяти в сегменте 0000 со смещением от 0000h до 03ffh. Можно установить свой собственный код ISR, вставив адрес вашей процедуры в позицию правильного вектора для соответствующего номера прерывания В DOS есть функции, возвращающие значения вектора прерывания и вставляющие новые значения в таблицу векторов прерываний Если вы меняете какой-либо вектор, то вы ответственны за то, чтобы восстановить изначальные значения перед окончанием вашей программы.

Ошибки деления возникают, когда делитель в div либо idiv равен 0 или когда результат от деления слишком велик для того, чтобы вместиться в 8- или 16-битовое место результата. Ошибка деления автоматически вызывает генерацию прерывания 0, выполняя ISR по вектору, хранящемуся в позиции 0000.0000, и обычно останавливает программу. Это состояние можно предотвратить установкой пользовательской ISR для того, чтобы отловить прерывание. Но задача усложнена тонким различием между 8086/88, 80286/386 и более поздними процессорами. Решение этой проблемы требует трюкачества, но это можно сделать так, как в примере, демонстрируемом в этой главе.

Хороший способ для написания программ коммуникации с удаленным компьютером через последовательную линию или через модем состоит в использовании управляемых прерываниями процедур для захвата данных по мере их поступления.

Несмотря на то что Turbo Debugger может запускать программы в пошаговом режиме, полезно знать, как установить свою собственную программу пошаговой работы. Программа SINGLE в этой главе иллюстрирует, как это сделать, и может служить оболочкой для ваших собственных сеансов пошаговой отладки.

Упражнения

10.1. Почему так важно сохранять значения регистров во внешней ISR?

10.2. Что делает iret?

10.3. Какие команды запрещают прерывания? Какие команды разрешают прерывания? Что делают эти команды? Для ISR — где логично расположить эти команды?

10 4. Напишите код для установки новой ISR с названием NewISR для прерывания номер 01Ch. Напишите код для восстановления изначального вектора прерывания перед окончанием программы.

10.5. *Может ли программа обработки прерывания прерываться другим прерыванием?*

10.6. После обработки сгенерированного внешним образом прерывания, какие команды вы должны выполнить, чтобы будущие прерывания распознавались?

10 7. Внешнее прерывание "Печать экрана" на PC — это прерывание 5. Напишите подпрограмму, которая печатает экран. При этом нажатие клавиши <PrtSc> не должно быть обязательным!

10.8. Какая разница между прерыванием ошибки деления на 8086/88 и 80286/386 и более поздних процессорах?

10.9. Какую команду можно использовать для того, чтобы вставить точку прерывания в вашу программу?

10.10. Напишите команды для установки флага трассировки, используя подход, отличный от тех двух, которые описываются в этой главе.

Задания

10.1. Перепишите программу TRM, добавив подпрограммы для полной эмуляции CRT терминала.

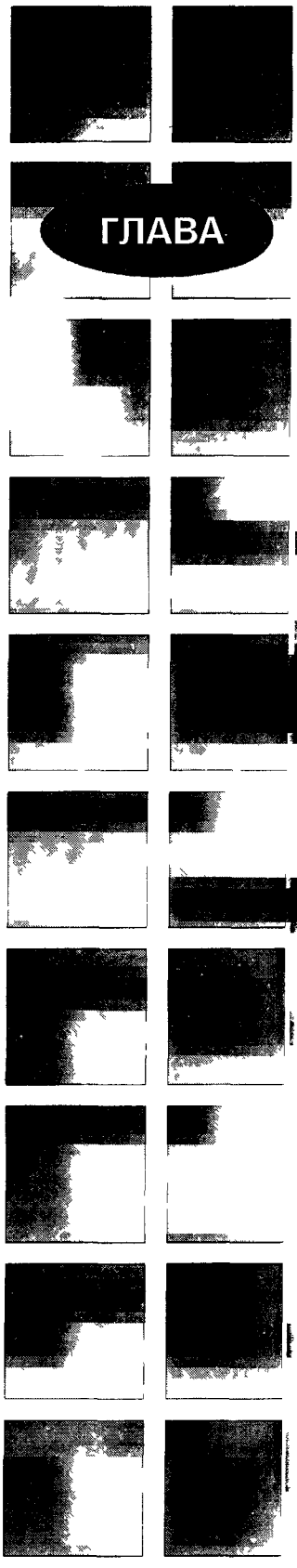
10.2. Улучшите модуль ASYNCH, добавив код, посылающий стоп-сигнал (обычно это ASCII 013h или <Ctrl+S>) перед переполнением буфера ввода. Добавьте также код для посылки сигнала начала (обычно это ASCII 011h или <Ctrl+Q>), разрешающий прием ввода вновь.

10.3 [Сложное] Добавить подпрограммы вывода, управляемые прерыванием в ASYNCH.ASM. (Замечание: вам потребуются дополнительные описания микросхем 8250 и 8259 для выполнения этого задания.)

10.4. Напишите версию программы обработки ошибки деления (DIV286.ASM), которая использует условную компиляцию для создания программы для всех моделей процессоров

10.5. Преобразуйте программу SINGLE в библиотечный модуль для того, чтобы добавлять код пошаговой отладки в любую программу. (Подсказка: используйте метод call bx из программы 9.5, DR.ASM, строка 91, для вызова пользовательского кода из ISR пошагового выполнения)

10.6. Напишите программу для печати отчета о всех номерах прерываний и адресов их векторов.



ГЛАВА 11

Дополнительные темы

- Углубление ваших знаний ассемблера
- Двоично-кодированные десятичные числа (BCD)
- Математическая библиотека для работы с BCD
- Отдельные дополнительные директивы
- Обработка таблиц
- Жесткий способ определения сегментов
- Директива SEGMENT
- Директива ASSUME
- Директива GROUP
- Использование сегментов в программах
- Дальние сегменты данных
- Программирование 80286 и старших моделей процессоров
- Программирование процессора 80386
- Директива VERSION
- Перечисляемые типы данных
- Становимся SMART (ушлыми)
- Некоторые дополнительные команды
- Резюме
- Упражнения
- Задания

Углубление ваших знаний ассемблера

В предыдущих главах вы изучили, как использовать большинство команд процессора 8086, ввели и исполнили множество примеров, иллюстрирующих различную технику работы на языке ассемблера В данный момент вы, вероятно, готовы писать ваши собственные программы, если вы этого еще не сделали Но все еще остались неизученными некоторые богатые возможности, специальные команды 80286, 80386 и более поздних моделей процессоров, а также директивы, которые упрощают распределение данных в мультипрограммных модулях

Многие из вас когда-нибудь возьмутся за большие проекты на ассемблере, требующие специального обращения с сегментами данных, не обеспечиваемого упро-

щенными моделями памяти, которые использовались в большинстве приведенных программ. Для этого вы, вероятно, захотите установить сегмент жестким образом, указав Turbo Assembler и Turbo Linker точное положение и размер сегментов данных и кода. Вы также можете захотеть добавить *дальний* сегмент данных — быстрый способ удвоить емкость данных вашей программы. Настоящая глава охватывает эти и другие предметы изучения, собранный здесь материал представляет собой "портфель" с намеками, советами и программами для серьезного программирования на ассемблере.

Двоично-кодированные десятичные числа (BCD)

Числа, используемые в деловых приложениях, должны быть большими и точными — эти два требования создают особые проблемы для программистов на ассемблере, приученных иметь дело с относительно малыми двоичными величинами. Так, например, представление долларовой величины одинарным словом в диапазоне от –32768 до 32767 не позволит после добавления воображаемой десятичной точки превысить величину, ограниченную мизерным диапазоном от –$327.68 до $327.67. Диапазон 32-битовых величин от –$21,474,836.48 до $21,474,836.47 подходит больше, но может по-прежнему быть слишком ограниченными для приложений, нуждающихся в обработке результатов в задачах складского учета и других бухгалтерских задачах. Кроме того, преобразование таких величин двойной точности в ASCII-кодировку и обратно требует времени. Представление с плавающей точкой еще хуже; появление возможности ошибок округления, которые могут быть приемлемы для научных измерений, допускающих такие ошибки, невозможно в бизнесе.

Одним из решений этой проблемы может быть сохранение значений в форме двоично-кодированных десятичных чисел (binari coded decimals — BCD), которые легко преобразуются в ASCII-кодировку и обратно, а также могут сохранять очень большие числа, содержащие до 20 цифр, что соответствует максимальной долларовой величине $999,999,999,999,999,999.99 (почти триллион триллионов). Существуют две основные разновидности BCD-чисел:

- *Упакованные BCD-числа* сохраняют две цифры на байт, в обычном для отдельных цифр порядке — от старших разрядов к младшим, но с байтами — в порядке от младших к старшим.
- *Неупакованные BCD-числа* сохраняют одну цифру на байт, упорядочивая байты как от младшего к старшему, так и от старшего к младшему.

Упакованные BCD-числа, вероятно, больше распространены, они сохраняют две десятичные цифры в каждом байте — одну цифру в старших четырех битах и другую — непосредственно в двоичном виде. Поскольку 4 бит могут представлять двоичную величину с диапазоном от 0 до 15, использование 4 бит для представления величин с диапазоном от 0 до 9 оставляет немного места в каждом байте. (При другом рассмотрении можно представить, что BCD-байт может содержать величины в диапазоне от 0 до 99, в то время как двоичный байт может нормально представлять значения от 0 до 255.)

Неупакованные BCD-числа обычно используются как промежуточная форма для преобразования упакованных BCD-чисел в ASCII-символы и обратно. Как вы сейчас увидите, существует непосредственная связь между ASCII и неупакованными BCD. К сожалению, этот формат не эффективен, поскольку способен представлять величины в диапазоне только от 0 до 9 на один байт.

Представление BCD в памяти

Вы можете создавать упакованные и неупакованные BCD-переменные в памяти, используя директивы dt и db. Директива dt создает 10-байтовую 20-цифровую упакованную BCD-величину.

Например:

```
packed dt 81659247      ,Упакованное BCD-число
```

Эта команда выделяет 10 байт, и в этом случае запоминает величину 00000000000081659247 в метке packed. Игнорируя ведущие нули, на рис. 11.1 показано, как эта величина располагается в памяти. Младшие две цифры (4 и 7) занимают первый байт, следующие две цифры по возрастанию (9 и 2) — второй байт и т.д. Как вы увидите, этот полуобратный порядок упрощает выполнение математических операций на двух упакованных BCD-числах.

У Turbo Assembler нет директив для создания неупакованных BCD-чисел, однако вы осторожно можете использовать директиву db. Например, то же число 81659247 выделяется как неупакованное 20-байтовое BCD-число:

```
unpacked    db 7,4,2,9,5,6,1,8,0,0          , неупакованное BCD-число
            db 0,0,0,0,0,0,0,0,0,0
```

Рис. 11.2 иллюстрирует, как эта величина представляется в памяти, также игнорируя ведущие нули. Как и в упакованной форме (рис. 11.1), цифры располагаются в обратном порядке, произвольный выбор порядка зависит только от того, как другое программное обеспечение использует неупакованные величины. Вы даже можете сохранять упакованные BCD любым возможным способом, как только будете готовы написать необходимый код для поддержки этого формата.

Замечание

Turbo Debugger распознает упакованные BCD-числа и может отображать их значения в окнах Watch и Variable. Отладчик не распознает неупакованные BCD-числа. Для просмотра байтов неупакованной величины используйте команду View.Dump.

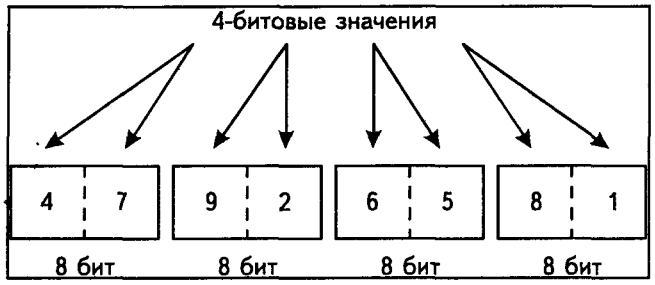


Рис. 11.1. Упакованная величина BCD 81,659,247 в памяти

Рис. 11.2. Неупакованная величина BCD 81,659,247 в памяти

Команды работы с неупакованными BCD

Четыре команды процессора 8086 — aaa, aad, aam и aas преобразуют неупакованные BCD-цифры в двоичные величины и обратно, делая операции с BCD-числами легкими для написания. Рассмотрим это последовательно.

Используйте aaa (ASCII Adjust After Addition — выравнивание после сложения) после сложения двух одноцифровых BCD-величин с использованием команд add или adc. Сумма должна располагаться в al. Если сумма превышает 09, то ah увеличивается и al приводится к значению в диапазоне 0–9. Например, для того чтобы сложить две цифры — 04 и 08, вы можете написать:

```
mov  bl,04h          ; Первую цифру в bl
mov  al,08h          ; Вторую цифру в al
add  al,bl           ; Сумма должна быть в al
xor  ah,ah           ; Обнулить ah
aaa                  ; Привести к неупакованному BCD
```

Этот фрагмент складывает неупакованные величины 04 и 08 и помещает сумму в al. Поскольку сложение выполняется в двоичном виде, al в этом примере принимает значение 0Ch. Для преобразования этой величины обратно в неупакованный BCD-формат выполняются команды xor обнуления ah и aaa. Поскольку в этом примере сумма в al больше 9, ah увеличивается на единицу и al выравнивается. В результате ax=0102 — ответ (12) в неупакованном BCD-формате.

Аналогичная инструкция aas (ASCII Adjust After Subtraction — приведение после вычитания) приводит разность двух неупакованных BCD-чисел после команд sub и sbb. Если требуется заем, то 1 вычитается из ah и al приводится к диапазону 0–9. Например, для вычитания 08 из 0406 можно написать:

```
mov  ax,0406h                ; Присвоить первую величину ax
mov  bl,08h            ; Присвоить вторую величину bl
sub  al,bl             ; Вычесть 0406h - 08h
aas                     ; Привести к неупакованной BCD-форме
```

Двоичное вычитание оставляет регистр ax=04FEh, который aas затем преобразует в неупакованную BCD-величину 0308h, или 38 десятичное — результат вычитания 46–8.

Две другие инструкции – aad (ASCII Adjust Before Division — приведение перед делением) и aam (ASCII Adjust After Multiplication — приведение после умножения) преобразуют неупакованные BCD-величины в двоичную форму и обратно. Вы можете использовать их до, а также после умножения и деления BCD-величин. Однако не стоит заблуждаться из-за мнемонического значения команд — их можно использовать и в других случаях. Необязательно выполнять после команды aad деление или выполнять умножение перед командой aam.

Для того чтобы преобразовать два неупакованных BCD-числа в регистре ax в двоичную форму, используйте aad. Поскольку наибольшее такое число, которое может храниться в ax, 0909h, aad всегда обнуляет регистр ah и устанавливает в al двоичный эквивалент BCD-цифр. Например:

```
mov  ax,0406h                , Присвоить неупакованное BCD-значение ax
aad                    ; Преобразовать, ax = 002Eh (46 десятичное)
```

Неупакованная BCD-величина 0406h в регистре ax преобразуется в эквивалентную двоичную величину 002Eh (46 десятичное) командой aad. Для обратного процесса преобразования двоичной величины в неупакованную BCD используйте aam, как в следующем примере:

```
mov  ax,005Fh                ; Присвоить двоичную величину ax
aam                  ; Преобразовать, ax = 0905h (05F шестнадцатеричное)
```

Двоичная величина 005Fh (95 десятичное) в ax преобразуется в неупакованный BCD-эквивалент 0905 командой aam. Наибольшая величина, которую команда aam позволяет записать в ax, 0063h (99 десятичное).

Преобразование неупакованных BCD в ASCII

Поскольку старшие 4 бит неупакованного BCD-байта всегда равны 0 (см. рис. 11.2), преобразование неупакованных BCD-чисел в ASCII-форму и обратно представляется достаточно простым. Вспомним, что ASCII-цифры 0–9 кодируются как шестнадцатеричные величины 30h–39h. Поэтому для преобразования неупакованных BCD-цифр в ASCII необходимо просто установить значение старших 4 бит равным 3:

```
mov  ax,0307h                ; Установка неупакованной BCD-величины в ax
or   ax,3030h        ; Преобразование в ASCII-формат (ax = 03337h)
```

Выполнение операции "побитового или" регистра ax с 3030h устанавливает старшие 4 бит, как в ah, так и в al, в значение 3, и тем самым преобразует 0307h в 3337h — две ASCII-кодированные цифры 33h (3) и 37h (7). Преобразование ASCII-цифр в неупакованный BCD-формат представляется столь же простым — необходимо только использовать операцию "побитового и" для удаления ASCII-информации из каждой цифры:

```
mov  ax,'81'          ;Устанавливаем 03831h в ax
and  ax,0F0Fh                ;Преобразуем в неупакованное BCD-значение (ax=0801h)
```

После установки строки '81' (равной 03831h) в ax логическое И с маской 0F0Fh устанавливает старшие 4 бит регистров ah, al в 0 и, таким образом, преобразует цифры в неупакованный BCD-формат.

Замечание

Порядок цифр в предыдущих двух примерах не является обратным, как показано на рис 11 2. При преобразовании неупакованных BCD-величин в ASCII необходимо обращать на это внимание

Команды работы с упакованными BCD

Две "десятичные" команды daa и das оперируют упакованными BCD-величинами аналогично командам "ASCII" aaa и aas. Команда daa используется после сложения двух упакованных BCD-байтов, каждый из которых содержит по две цифры:

```
xor  ah,ah           ;Обнулить ah
mov  al,087h         ;Установить в al упакованное BCD 87
mov  bl,035h         ;Установить в bl упакованное BCD 35
add  al,bl           ;Сложить al<-al+bl
daa                  ;Преобразовать, al=22h, cf, ah=1
```

Команда xor обнуляет регистр ah по причинам, которые будут объяснены позже. Две упакованные BCD-величины 87h и 35h устанавливаются в al и bl. Команда add складывает величины и помещает двоичную сумму в регистр al, который становится равным 0BCh. Выполнение команды daa преобразует эту двоичную величину в упакованную BCD, записывая в al значение 22h. Но правильный ответ составляет 122 (87+35), а не 22, поэтому код необходимо дополнить проверкой флага переполнения:

```
      jnc     @@10           ;Перескочить увеличение если cf=0
      inc     ah             ;Увеличить ah на 1
@@10:
```

Технически, если daa обнаруживает переполнение, когда упакованный BCD-результат оказывается больше 99 (максимальная BCD-величина, которая может содержаться в 1 байте), то флаги cf и af устанавливаются в 1; в противном случае оба флага очищаются. На практике, для обнаружения этого условия, можно проверять только cf. В данном примере значение ah увеличивается, устанавливая в ax правильный ответ 0122h. Именно по этой причине регистр ah обнулялся.

Замечание

После выполнения команды daa, если af=1 и cf=0, значит, результат в al находится в пределах диапазона 10h–99h, т.е., другими словами, перенос осуществлялся из младших 4 бит.

Дополнением к daa является команда das, которая преобразует упакованные BCD-величины после вычитания командами sub или sbb. Поскольку вычитание может создавать отрицательные числа, использование das требует дополнительной осторожности. Вначале рассмотрим пример, в котором получается положительный результат:

```
mov  al,062h          ;Устанавливаем в al упакованное BCD 62
mov  bl,036h          ;Устанавливаем в bl упакованное BCD 36
sub  al,bl            ;Вычитаем al<-al-bl
.das                  ;Преобразуем, al = 026h
```

Упакованные BCD-величины 62h и 36h устанавливаются в регистры al и bl. Команда sub производит вычитание и устанавливает двоичную разность (02Ch) в al. Выполнение команды das преобразует эту двоичную величину в упакованную BCD, изменяя значение в al на 026h —правильный ответ в десятичной форме для вычитания 62–36. После этого, если cf равен 0, то заем не требуется и, следовательно, ответ в регистре al можно непосредственно использовать.

Замечание

Технически, в случае отсутствия заема и cf, и af должны равняться 0, однако если cf=0, а af=1, то заем требуется для младшей цифры. Если вы выполните предыдущий пример в Turbo Debugger, вы увидите, что именно это и происходит. Вычитание 62–36 требует заема для двух младших цифр (2 и 6). Обычно можно игнорировать это специальное условие и только проверять cf для обнаружения заема при выполнении полного вычитания.

В случае, когда при вычитании получается отрицательный результат, процедура усложняется. При этом необходимо контролировать флаг переноса для обнаружения заема при вычитании. Возникновение заема означает, что результат в al представляет собой отрицательное десятичное дополнение, которое необходимо использовать для получения окончательного ответа. Данный пример поможет прояснить, как это сделать:

```
      mov     al,036h       ;Установить в al упакованное BCD 36
      mov     bl,062h       ;Установить в bl упакованное BCD 62
      sub     al,bl         ;Вычесть, al<-al-bl
      das                   ;Преобразовать, al = 074h
      jnc     @@10          ;Переход в случае отсутствия заема
      neg     al            ;Преобразовать в отрицательное (двоичное)
      das                   ;Преобразовать в упакованное BCD
@@10:
```

Как и раньше, в al и bl устанавливаются упакованные BCD-значения для проведения вычитания. Команда sub вычитает bl из al, при этом, в данном примере, в регистре al получается отрицательный двоичный результат, равный 0D4h. Эта величина преобразуется в упакованный BCD-формат командой das, устанавливающей в al значение 74h. Но это неправильный ответ (36–62)=–26, а не 74. Проверка флага переноса командой jnc обнаруживает условие, говорящее о том, что al представляет собой десятичное дополнение, преобразованное в абсолютную величину вычитанием 100. (74–100=–26 — правильный ответ.) Более простым (хотя, возможно, и менее очевидным) способом найти десятичное дополнение является выполнение команды neg, которая производит вычитание своего операнда (в данном случае al) из 0. Поскольку эта операция оставляет ответ в al в двоичном виде, требуется еще одна команда das для преобразования результата обратно в упакованный BCD-формат, которая наконец устанавливает правильную абсолютную величину ответа в al.

Математическая библиотека для работы с BCD

Выполнение математических операций над *высокоточными величинами*, т.е. такими, которые содержат больше байтов или слов, чем может вместиться в регистрах, и, следовательно, требуют выполнения множества операций для осуществления сложения, вычитания, умножения и деления, создает дополнительные трудности при программирования BCD-процедур. Для демонстрации некоторых приемов, используемых при написании таких процедур, и для того чтобы привести несколько полезных процедур, которые вы можете использовать в своих программах, в программу 11.1, BCD.ASM, были включены шесть подпрограмм для сложения и вычитания упакованных BCD-величин и для преобразования BCD-чисел между упакованным, неупакованным и ASCII- форматами. Приводится также процедура, которая копирует упакованную десятибайтовую величину в другую BCD-переменную. Ассемблируйте и сохраните полученный модуль в файле MTA.LIB, используя следующие команды:

```
tasm /zi bcd
tlib /E mta -+bcd
```

Как обычно, игнорируйте предупреждение о том, что модуля BCD нет в библиотеке — он не появится там до тех пор, пока вы не проинсталируете его первый раз. Если вы сделаете какие-либо изменения в программе, используйте те же команды для реассемблирования и установки нового модуля. Инструкции по использованию BCD-модуля находятся в тексте программы.

Программа 11.1. BCD.ASM

```
1: %TITLE "Binary Coded Decimals (BCD) -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:
7:
8:     ;----- Константы
9:
10:    ASCIINull    EQU 0              ; Нулевой ASCII-символ конца строки
11:    PackedSize   EQU 10             ; Байты в упакованной величине BCD
12:    UnpackedSize                    EQU 20  ; Байты в неупакованной величине BCD
13:
14:    ,----- замечание. PackedSize должен быть четным!
15:
16:
17:         UDATASEG
18:
19:    TempUPBCD DT ?, ? ; Пространство неупакованного слова BCD (20 байт)
20:
21:
22:         CODESEG
23:
24:         PUBLIC BCDAdd, BCDSubtract, PackedToUnpacked
25:         PUBLIC UnpackedToPacked, BCDToASCII, BCDCopy
26:
27:    %NEWPAGE
28:    ;---------------------------------------------------------------
29:    ; BCDAdd Сложение двух чисел BCD
30:    ;---------------------------------------------------------------
31:    ; Вход:
32:    ; si = адрес исходного значения BCD (10 байт)
33:    ; di = адрес результата величины BCD (10 байт)
34:    ; Выход:
35:    ; destinationBCD <- destinationBCD + sourceBCD
36:    ; cf = 0 : Ошибки нет
37:    ; cf = 1 : Произошла ошибка переполнения
38:    ; Регистры:
39:    ; нет
40:    ;---------------------------------------------------------------
41:    PROC BCDAdd
42:         push   ax                  ; Сохранить модифицируемые регистры
43:         push   cx
44:         push   di
45:         push   si
46:
47:         cld                        ; Автоувеличение si & di
48:         clc                        ; Очистить носитель первого adc
49:         mov    cx, PackedSize      ; Присвоить счетчик цикла cx
50:    @@10:
51:         lodsb                      ; Взять две цифры исходной величины
52:         adc    al, [byte di]       ; Добавить две цифры результата + cf
53:         daa                        ; Выровнять по формату упакованных BCD
54:         stosb                      ; Сохранить результат в месте назначения
55:         loop   @@10                ; Цикл до тех пор, пока все не сделано (cx = 0)
56:
57:         pop    si                  ; Восстановить сохраненные регистры
58:         pop    di
59:         pop    cx
60:         pop    ax
61:         ret                        ; Вернуться в вызвавшую процедуру
62:    ENDP BCDAdd
```

```
%NEWPAGE
;-------------------------------------------------------------------------
; BCDSubtract Вычитание двух упакованных чисел BCD
;-------------------------------------------------------------------------
; Вход:
; si = адрес исходного значения BCD (10 байт)
; di = адрес результата величины BCD (10 байт)
; Выход
; destinationBCD <- destinationBCD - sourceBCD
; cf = 0 : Ошибки нет
; cf = 1 : Произошла ошибка переполнения (underflow)
; Регистры:
; нет
;-------------------------------------------------------------------------
PROC BCDSubtract
        push   ax                   ; Сохранить модифицируемые регистры
        push   cx
        push   di
        push   si

        cld                         ; Автоувеличение si & di
        clc                         , Очистить носитель первого sbb
        mov    cx, PackedSize       , Присвоить счетчик цикла cx
@@10:
        lodsb                       ; Взять две цифры исходной величины
        sbb    [byte di], al        ; dest <- dest - source byte
        mov    al, [byte di]        ; Загрузить двоичный результат в al
        das                         ; Выровнять по формату упакованных BCD
        stosb                       , Сохранить результат в месте назначения
        loop   @@10                 ; Цикл до тех пор, пока все не сделано (cx = 0)

        pop    si                   ; Восстановить сохраненные регистры
        pop    di
        pop    cx
        pop    ax
        ret                         ; Вернуться в вызвавшую процедуру
ENDP BCDSubtract
%NEWPAGE
;-------------------------------------------------------------------------
; PackedToUnpacked Преобразование упакованного BCD в неупакованный BCD
;-------------------------------------------------------------------------
; Вход:
; si = адрес исходного значения упакованного BCD (10 байт)
; di = адрес результата неупакованной величины BCD (20 байт)
; Выход:
; destinationBCD <- unpacked(sourceBCD)
; Регистры:
; нету
;-------------------------------------------------------------------------
PROC PackedToUnpacked
        push   ax                   ; Сохранить модифицируемые регистры
        push   cx
        push   di
        push   si

        cld                         ; Автоувеличение si & di
        mov cx, PackedSize          ; Присвоить счетчик цикла cx
@@10:
        lodsb                       ; Взять две цифры исходной величины
        mov    ah, al               ; Скопировать цифры из al в ah
        shr    ah, 1                , Сдвинуть верхние цифры в
        shr    ah, 1                ; нижние 4 бита ah
        shr    ah, 1
        shr    ah, 1
        and    al, 0Fh              ; Маскировать верхние цифры al
        stosw                       ; Сохранить ax в месте назначения
        loop   @@10                 ; Цикл до тех пор, пока все не сделано (cx = 0)

        pop    si                   ; Восстановить сохраненные регистры
        pop    di
        pop    cx
        pop    ax
        ret                         ; Вернуться в вызвавшую процедуру
ENDP PackedToUnpacked
```

```
%NEWPAGE
;-------------------------------------------------------------------------
; UnpackedToPacked Преобразование неупакованного BCD в упакованный BCD
;-------------------------------------------------------------------------
; Вход:
; si = адрес исходного значения неупакованного BCD (20 байт)
; di = адрес результата упакованной величины BCD (10 байт)
; Выход:
; destinationBCD <- packed(sourceBCD)
; Регистры:
;-------------------------------------------------------------------------
PROC UnpackedToPacked
        push   ax                   ; Сохранить модифицируемые регистры
        push   cx
        push   di
        push   si

        cld                         ; Автоувеличение si & di
        mov    cx, PackedSize       ; Присвоить счетчик цикла cx
@@10:
        lodsw                       ; Взять две цифры исходной величины
        shl    ah, 1                ; Сдвинуть цифры в
        shl    ah, 1                ; верхние 4 бита ah
        shl    ah, 1
        shl    ah, 1
        or     al, ah               ; Упаковать две цифры в al
        stosb                       ; Сохранить al в месте назначения
        loop   @@10                 ; Цикл до тех пор, пока все не сделано (cx = 0)

        pop    si                   ; Восстановить сохраненные регистры
        pop    di
        pop    cx
        pop    ax
        ret                         ; Вернуться в вызвавшую процедуру
ENDP UnpackedToPacked
%NEWPAGE
;-------------------------------------------------------------------------
; BCDToASCII Преобразование упакованного BCD в ASCII
;-------------------------------------------------------------------------
; Вход:
; si = адрес исходного значения упакованного BCD (10 байт)
; di = адрес результирующей строки ASCIIZ (21 байт)
; Выход:
; ASCIIZ <- ASCII( sourceBCD) + нулевой символ
; Регистры:
; нету
;-------------------------------------------------------------------------
PROC BCDToASCII
        push   ax                   ; Сохранить модифицируемые регистры
        push   cx
        push   di
        push   si

        push   di                   ; Сохранить адрес места назначения
        mov    di, offset TempUPBCD       ; Использовать временную рабочую область
        call   PackedToUnpacked     ; Распаковать исходное значение в temp
        pop    di                   ; Восстановить адрес места назначения

;----- Адресовать последнее слово временной рабочей области
        mov    si, offset TempUPBCD + UnpackedSize - 2

        mov    cx, PackedSize       ; Присвоить счетчик цикла cx
@@10:
        std                         ; Автоуменьшение si
        lodsw                       ; Взять две цифры в ax
        or     ax, 03030h           ; Преобразовать в ASCII
        xchg   ah, al               ; Поменять местами символы
        cld                         ; Автоувеличение di
        stosw                       ; Сохранить символы в месте назначения
        loop   @@10                 ; Цикл до тех пор, пока все не сделано (cx = 0)
        mov    [byte di], ASCIINull       ; Сохранить маркер конца строки

        pop    si                   ; Восстановить сохраненные регистры
        pop    di
```

```
211:        pop    cx
212:        pop    ax
213:        ret                          ; Вернуться в вызвавшую процедуру
214:  ENDP BCDToASCII
215:  %NEWPAGE
216:  ;---------------------------------------------------------------------
217:  ; BCDCopy Копирование упакованных величин BCD
218:  ;---------------------------------------------------------------------
219:  ; Вход:
220:  ; si = адрес исходного значения BCD (10 байт)
221:  ; di = адрес результата величины BCD (10 байт)
222:  ; Выход:
223:  ; destinationBCD <- sourceBCD
224:  ; Регистры:
225:  ; нет
226:  ;---------------------------------------------------------------------
227:  PROC BCDCopy
228:        push   cx                  ; Сохранить модифицируемые регистры
229:        push   di
230:        push   si
231:
232:        cld                        ; Автоувеличение si & di
233:        mov    cx, PackedSize/2    ; Присвоить счетчик цикла cx
234:        rep    movsw               ; Скопировать, используя перемещение слов
235:
236:        pop    si                  ; Восстановить сохраненные регистры
237:        pop    di
238:        pop    cx
239:        ret                        ; Вернуться в вызвавшую процедуру
240:  ENDP BCDCopy
241:
242:        END                        ; Конец модуля BCD
```

Использование BCD-модуля

Шесть процедур в BCD-модуле работают с упакованным и неупакованным BCD-форматами чисел, представленными в начале этой главы (см. рис. 11.1 и 11.2). Упакованные BCD-числа имеют длину 10 байт и могут содержать до 20 цифр. Неупакованные BCD-числа должны иметь длину 20 байт и также могут содержать до 20 цифр. Вы сами можете убедиться, что эти переменные достаточно велики, для того чтобы хранить результаты различных операций. Поскольку инструкции работы со строками используются во всех процедурах, сегментные регистры es и ds должны указывать на один и тот же сегмент данных. Для того чтобы применить данный набор процедур в программе, опишите процедуры, которые вы собираетесь использовать с помощью директивы EXTERN.

Обычно это делается сразу после директивы CODESEG. Например:

```
CODESEG
EXTRN BCDadd:proc, BCDSubtract:proc, PackedToUnpacked:proc
EXTRN UnpackedToPacked:proc, BCDToASCII:proc, BCDCopy:proc
```

Вы можете исполнять любую из шести процедур с помощью команды call.

Замечание

Все BCD-величины должны быть положительными. Для того чтобы использовать эти процедуры с отрицательными числами, нужно сохранять знак отдельно. Вам также необходимо знать, что Turbo Assembler 1.0 содержит ошибку, из-за которой невозможно правильно описать отрицательное BCD-число с помощью директивы dt. В следующих версиях эта ошибка исправлена.

Отдельные дополнительные директивы

Turbo Assembler имеет три директивы, которые могут сгладить некоторые проблемы, связанные с большими многомодульными ассемблерными программами. Данный раздел описывает использование этих директив:

- COMM — Общий (Communal)
- GLOBAL — Глобальные Переменные (Global Variables)
- INCLUDELIB — Включить Библиотечный Модуль (Include Library Module)

Замечание

Turbo Linker 2.0 и более ранние версии не поддерживают директиву COMM.

Использование общих переменных

Директива COMM определяет *общие переменные*, которые аналогичны неинициализированным переменным и могут быть описаны в множественных модулях. Например, предположим, что несколько модулей используют 100-байтовый массив байтов и индексную переменную. Для их описания можно использовать директиву COMM следующим образом:

```
DATASEG
COMM near index:Word
COMM near array Byte 100
```

Одно утверждение COMM может содержать множественное описание переменных, разделяемых запятыми, но отдельные строки, как показано выше, удобнее для чтения. Первый элемент после директивы COMM является опциональным, может быть либо near, либо far, и указывает, относится ли данная переменная к текущему сегменту данных или к другому сегменту. При использовании упрощенной модели памяти нет необходимости указывать параметры near или far — Turbo Assembler сам проверит все ссылки на общие переменные и выдаст ошибку при попытке обращения к переменной в неверном сегменте Следующий элемент — это наименование переменной, за которым следует двоеточие и размер, причем размер может быть byte, word, dword, fword, pword, qword или tbyte. Помимо этого можно определить наименование структуры. Затем может идти двоеточие и числовая величина (:100 во второй строке примера), сообщающая ассемблеру количество байтов, которые необходимо выделить для данного элемента. Если вы не укажите число, Turbo Linker выделит пространство только для одного элемента указанного размера

В действительности пространство под общие переменные не выделяется до тех пор, пока вы не скомпонуете модули. Переменные с одним и тем же именем, определенные в различных модулях, перекрываются. Таким образом, вместо декларирования PUBLIC переменных в модулях определения и EXTERN в используемых модулях, можно просто объявить все переменные общими (COMM) и оставить Turbo Linker работу по сведению этих множественных ссылок к одним переменным.

Недостатком такого соглашения является невозможность инициализации общих переменных. Как и все неинициализированные переменные, общие переменные не имеют определенных значений на момент запуска программы. Нет также гарантий того, когда и в каком порядке переменные появятся в памяти — таким образом, необязательно две общие переменные будут расположены последовательно во время исполнения программы. Для того чтобы избежать этих ограничений, но не потерять преимущества отказа от использования директив PUBLIC и EXTRN, Turbo Assembler имеет аналогичную, но более гибкую директиву GLOBAL, описанную ниже.

Использование глобальных переменных

Директива GLOBAL аналогична директиве COMM, но при этом позволяет устанавливать начальные значения переменным, которые используются несколькими модулями. Применяя те же две переменные, которые описывались в предыдущем разделе, один из модулей мог бы описать и инициализировать переменные array, index с помощью команд:

```
DATASEG
GLOBAL index Word
GLOBAL array Byte 100
;
index       dw 0
array       db 100 dup (1)
```

В текущем сегменте данных две директивы GLOBAL декларируют машинное слово index и байт array. Типы данных после двоеточия могут быть такими же, как и у директивы COMM. Число (100) после описания переменной array сообщает ассемблеру, какое количество байтов занимает данная переменная. Это число необходимо указывать только в том случае, если директивы размещения (db, dw и т.п.) описывают множественные переменные или используют оператор dup; в противном случае ассемблер не может определить, что переменная array из примера не является одним байтом. Две действительные переменные описываются и инициализируются, как обычно, создаваемая переменная index инициализируется 0 с помощью команды dw, а массив из 100 байт, каждый из которых равен 1, — командой db.

Для ссылки на эти же переменные в других модулях необходимо только повторить в них директивы GLOBAL. Действительное размещение переменных (например, с использованием dw и db) должно проводиться только в одном модуле. Как показывает этот пример, теперь переменные доступны из всех программных модулей без использования одной из директив — PUBLIC или EXTERN.

Включение глобальных переменных

Хорошим способом организации больших многомодульных программ является вынесение глобальных переменных в отдельный файл с последующим включением его во все модули. Это позволяет держать переменные в одном подручном месте и избежать неприятных сюрпризов и конфликтов, которые могут возникнуть при использовании сотен директив PUBLIC и EXTERN. В подобных ситуациях вы начнете ощущать настоящую силу директивы GLOBAL. Хорошим подходом может быть описание глобальных переменных в текстовом файле, возможно GLOBAL.ASM:

```
;GLOBAL.ASM файл
;
GLOBAL index:Word
GLOBAL array:Byte:100
;
;другие глобальные переменные
```

Затем в каждом модуле, который ссылается на одну (или более) глобальную переменную, необходимо добавить директиву INCLUDE, обычно где-то после директивы DATASEG:

```
;AMODULE.ASM
;
DATASEG
INCLUDE 'GLOBAL.ASM'
;
; другие локальные переменные
```

Вы можете по-прежнему описывать остальные локальные переменные в данном модуле, только глобальные переменные разделяются между модулями. Директива INCLUDE загружает глобальные описания из файла GLOBAL.ASM и делает эти описания доступными в данном модуле. В дополнение необходимо создать инициализационный модуль, который в действительности и создаст переменные:

```
;INIT.ASM
DATASEG
INCLUDE 'GLOBAL.ASM'
index       dw 0
array       db 100 dup (1)
```

INIT.ASM описывает и инициализирует переменные. И снова файл GLOBAL.ASM включается, как и в другие модули. (Вы можете либо ассемблировать файл INIT.ASM так же, как и другие модули, либо включить его текст в основную программу директивой INCLUDE.) Применяя директиву GLOBAL, можно избежать использования PUBLIC, EXTRN и получить возможность сохранения и инициализации всех глобальных переменных в одном или двух подручных файлах. Устраняется также ограничение директивы COMM, которая не позволяет инициализировать переменные.

Использование директивы INCLUDELIB

В большинстве предыдущих глав давались инструкции по добавлению объектных модулей в файл библиотеки MTA.LIB. Затем команды Turbo Linker ссылались на этот файл, для того чтобы извлечь модули, содержащие процедуры, которые описаны директивой EXTRN в кодовом сегменте программы (или одного из модулей). Для упрощения команд построения можно включить директиву INCLUDELIB, сообщающую компоновщику имя библиотеки, в которой нужно искать необходимые модули. Например, можно добавить следующую команду вблизи начала основной программы:

```
INCLUDELIB 'MTA'
```

Если не указывать расширение имени файла, то компоновщик предполагает, что это LIB. Помимо этого имя файла может содержать информацию о пути поиска, как "c:\library\MTA.LIB". Теперь можно ассемблировать и компоновать программу с помощью таких команд, как:

```
tasm myprog
tlink myprog
```

Поскольку в тексте программы содержится директива INCLUDELIB, необходимые модули извлекаются из MTA.LIB автоматически, без ссылки на файл библиотеки, непосредственно из командной строки tlink. Включайте директиву INCLUDELIB только в основной модуль — не используйте эту директиву для ссылки на один файл библиотеки из более чем одного модуля одновременно.

Замечание

Даже при использовании директивы INCLUDELIB по-прежнему необходимо применять директивы EXTRN для импорта процедур, объявленных PUBLIC в библиотечном модуле.

Обработка таблиц

Обычно, если можно найти переменную в таблице, а не вычислять то же значение с помощью численного выражения, программа работает быстрее. Требуется только пара команд для поиска величины, в то время как вычисление требует большего числа команд. Еще больше сэкономить время можно, если пользоваться специальными командами табличной обработки процессора 8086 xlat (преобразовать из таблицы — Translate From Table).

Команда xlat требует установки в ds:bx адреса таблицы байтов. Индексная величина из регистра al добавляется к этому адресу и указывает на один из байтов таблицы. Выполнение команды xlat загружает значение этого байта в al, замещая начальное значение регистра. Другими словами, индексная величина в al преобразуется в связанный байт из таблицы. Небольшой пример поможет объяснить, как это происходит. Ассемблируйте и скомпонуйте программу 11.2, TABLE.ASM, с использованием команд

```
tasm /zi table
tlink /v table
```

Программа 11.2. TABLE.ASM

```
1: %TITLE "Table translation -- Copyright (c) 1989,1995 -- by Tom Swan"
2:
3:              IDEAL
4:
5:              MODEL small
6:              STACK 256
7:
8:              DATASEG
9:
10:     ;-----------------------------------------------
11:     ;indexes     0, 1, 2, 3, 4, 5, 6, 7, 8, 9
12:     ;-----------------------------------------------
13:     btable db    0, 1, 4, 9, 16, 25, 36, 49, 64, 81
14:
15:             CODESEG
16:
17:     Start:
18:             mov     ax,@data            ; Инициализировать DS по адресу
19:             mov     ds,ax               ; сегмента данных
20:
21:             mov     bx, offset btable   ; Адресовать btable через ds:bx
22:             mov     cx, 9               ; Присвоить счетчик цикла cx
23:     @@10:
24:             mov     al, cl              ; Скопировать значение индекса в al
25:             xlat                        ; Преобразовать из btable
26:             loop    @@10                ; Цикл с cx
27:
28:     Exit:
29:             mov     ax,04C00h           ; Функция DOS: Конец программы
30:             int     21h                 ; Вызов DOS. Закончить программу
31:
32:             END     Start               ; Конец программы / точка входа
```

Как работает программа TABLE.ASM

Загрузите ассемблерную программу TABLE в Turbo Debugger, используя команду td table, и нажмите <Alt+V> для переключения в окно CPU. Нажмите <F5> для того, чтобы распахнуть окно на весь экран, и выполните следующие шаги.

1. Нажмите <F7> дважды, затем еще один раз для того, чтобы загрузить в bx смещение переменной btable в строке 13. Нажмите <F7> опять для загрузки в cx счетчика циклов (9).
2. Курсор должен находиться на команде mov. Нажмите <F7> для копирования cl в al. Вы должны увидеть, что регистр ax (в верхнем правом углу экрана) получил значение 09.

3. Нажмите <F7> для выполнения команды xlat, преобразующей значение в al в величину из таблицы btable с адресом ds:bx. При первом проходе цикла в al установится значение 51h (81 десятичное), при втором — начальное значение.
4. Нажимайте <F7> для выполнения всех проходов по циклу, устанавливая в al меньшие и меньшие индексные значения, которые преобразуются в другие байты из таблицы btable.

Этот эксперимент демонстрирует работу команды xlat, преобразуя индексные величины в al в байты из таблицы, хотя можно выполнить те же действия проще. Более полезные примеры будут представлены дальше.

Практическое использование xlat

Одной из основных областей применения xlat является преобразование ASCII-символов в другие, возможно в программе терминального эмулятора, которая должна передать определенные значения в удаленную систему при нажатии управляющей клавиши. Простейший способ запрограммировать — создать таблицу величин, индексируемых ASCII-символами. В качестве демонстрационного примера программа 11.3, BOXCHAR.ASM, преобразует <Alt+1>, <Alt+2>, ..., <Alt+0> в десять расширенных ASCII-символов, обычно используемых PC для рисования рамок. Ассемблируйте, скомпонуйте и исполните программу при наличии библиотеки MTA.LIB на диске следующими командами:

```
tasm boxchar
tlink boxchar,,,mta
boxchar
```

Для отображения символа рамки нажмите <Alt> и цифровую клавишу. Этот пример демонстрирует использование команды xlat для преобразования кодов клавиш в ASCII-символы. Для окончания демонстрации нажмите <F10>.

Программа 11.3. BOXCHAR.ASM

```
1:    %TITLE "Box char demonstration -- by Tom Swan"
2:
3:          IDEAL
4:
5:          MODEL small
6:          STACK 256
7:
8:    cr            EQU 13              ; ASCII-символ возврата каретки
9:    lf            EQU 10              ; ASCII-символ новой строки
10:   Fn10          EQU 100             ; Значение GetCh для <F10>
11:   LowIndex      EQU 152             ; Значение GetCh для <Alt+1>
12:   HighIndex     EQU 161             ; Значение GetCh для <Alt+0>
13:
14:         DATASEG
15:
16:   message       db cr, lf, 'Sample Character Table Translation'
17:               db cr, lf, 'Press <Alt+1> to <Alt+0> to display characters'
18:               db cr, lf, 'Press <F10> to end', cr, lf, lf, 0
19:
20:   ctable      db 179, 180, 191, 192, 193, 194, 195, 196, 217, 218
21:
22:         CODESEG
23:
24:         EXTRN StrWrite:proc, GetCh:proc
25:
26:   Start:
27:         mov   ax, @data           ; Инициализировать DS по адресу
28:         mov   ds, ax              ; сегмента данных
29:         mov   es, ax              ; Установить es = ds
30:
31:         mov   di, offset message  ; Выдать инструкцию по использованию программы
32:         call  StrWrite
33:   @@10:
34:         call  GetCh               ; Получить нажатую клавишу
35:         jnz   @@10                ; Повторять, пока клавиша не будет функциональной
36:         cmp   al, Fn10            ; Проверить, не <F10> ли?
37:         je    Exit                ; Выход, если нажата <F10>
38:         cmp   al, LowIndex        , Проверить, лежит ли al в
39:         jb    @@10                ; интервале от LowIndex до
40:         cmp   al, HighIndex       , HighIndex
41:         ja    @@10
```

```
42:         sub   al, LowIndex       ; Преобразовать al в 0..n
43:         mov   bx, offset ctable  ; Адресовать ctable через ds:bx
44:         xlat                     ; Преобразовать al из ctable
45:         mov   dl, al             ; Переместить новый символ из al в dl
46:         mov   ah, 2              ; Функция DOS "высветить символ"
47:         int   21h                ; Вызвать DOS для высвечивания символа
48:         jmp   @@10               ; Повторять, пока все не сделано
49:   Exit:
50:         mov   ax,04C00h          ; Функция DOS: Конец программы
51:         int   21h                ; Вызвать DOS. Закончить программу
52.
53:         END   Start              ; Конец программы / точка входа
```

Как работает программа BOXCHAR.ASM

Переменная ctable в строке 20 определяет расширенные ASCII-символы для клавиш <Alt+1>, <Alt+2>, ..., <Alt+0>. Код в строках 31–39 вызывает процедуру GetCh из модуля KEYBOARD (см. главу 7) для определения нажатой клавиши, результат возвращается в al. Остальные команды в этой части проверяют нажатие <F10>, которая заканчивает работу программы, и контролируют, чтобы значение в al попадало в диапазон LowIndex — HighIndex. Затем в строке 42 производится вычитание величины LowIndex из al, таким образом уменьшая значение диапазона с 151–161 до 0–10. Затем строки 43 и 44 преобразуют этот дополнительный индекс в одну из табличных величин и отображают символ посредством вызова функции 2 DOS (строки 45–47).

Использование xlat в многомерных таблицах

Иногда xlat оказывается полезной для преобразования величины из al, представляющей номер колонки в двумерной матрице. Совместно с командой lea (Load Effective Address — загрузить эффективный адрес) работа с такими сложными массивами оказывается не столь трудной на языке ассемблера, как можно было себе представить. Предположим, например, что имеется следующая матрица размером 4×8:

```
DATASEG
matrix      db 00fh, 04bh, 087h, 0c3h, 00fh, 04bh, 01eh, 05ah
            db 096h, 0d2h, 01eh, 05ah, 02dh, 069h, 05ah, 01eh
            db 02dh, 069h, 03ch, 078h, 0b4h, 0f0h, 03ch, 078h
            db 09dh, 0d2h, 04fh, 067h, 003h, 079h, 099h, 000h
```

Затем предположим, что программа устанавливает номер колонки (0–7) в al и номер строки (0–3) в si. Для того чтобы загрузить байт matrix[строка, колонка], требуется всего несколько команд:

```
CODESEG
mov al,4              ;Установить номер колонки в al
mov si,2              ;Установить номер строки в si
mov cl,3              ;Установить смещение в cl
shl si,cl             ;si<-si*8
lea bx,[matrix+si]    ;ds:bx адресует строку таблицы
xlat                  ;al<-таблица[строка,колонка]
```

В данном примере al равняется 4, а si равняется 2 — соответственно номер строки и колонки. Третья команда mov и команда shl умножают номер строки в si на количество байтов в одной строке — в данном примере 8. Затем команда lea загружает в регистр bx смещение заданной строки. После установления регистра bx команда xlat преобразует индекс колонки в регистре al в байт, указываемый номером колонки заданной строки таблицы. Команда lea приводит к тому же результату, что и две команды:

```
mov  bx, offset matrix
add  bx, si
```

Вместо этого всегда используйте lea — это быстрее, чем вычисление сложного адреса путем сложения. В качестве параметра команды lea можно применять любые типы адресов, обсуждавшиеся в главе 5. Для получения результата можно использовать любой из регистров общего назначения, хотя обычно для этих целей применяется bx.

Другие формы команды xlat

Команда xlat имеет несколько вариантов использования. Вы можете передавать переменную, указывающую на таблицу, в качестве параметра команде xlat. Тогда Turbo Assembler сам проверит возможность адресации переменной посредством ds:bx, которые необходимо предварительно инициализировать. Например:

```
mov  bx,offset atable        ;Адресовать atable регистром bx
mov  al,[index]              ;Установить индексное значение в al
xlat [atable]                ;Преобразовать al в значение таблицы (ds bx)
```

При использовании параметра с командой xlat Turbo Assembler проверяет, находится ли переменная atable в сегменте, адресуемом регистром ds. Можно использовать аналогичную конструкцию с измененным сегментом для ссылки на таблицу, расположенную в сегменте, адресуемом es:

```
mov  bx,offset atable        ;Адресовать atable регистром bx
mov  al,[index]              ;Установить индексное значение в al
xlat [es:atable]             ;Преобразовать al в значение таблицы (es bx)
```

Переустановка сегмента меняет обычный базовый регистр команды xlat ds на es. Таким способом можно устанавливать параметр, но если вы не хотите ссылаться на переменную по имени, то для этой цели можно воспользоваться регистром bx:

```
mov  bx,offset atable        ;Адресовать atable регистром bx
mov  al,[index]              ;Установить индексное значение в al
xlat [es:bx]                 ;Преобразовать al в значение таблицы (es bx)
```

В дополнение вы можете использовать сокращенную мнемонику xlatb таким же способом, как и xlat без параметра:

```
mov  bx,offset atable        ;Адресовать atable регистром bx
mov  al,[index]              ;Установить индексное значение в al
xlatb                        ;Преобразовать al в значение таблицы (ds bx)
```

Честно говоря, мне не совсем понятно почему мнемоника xlatb вообще существует — ведь для выполнения той же задачи можно просто использовать xlat без параметра. Единственной существенной разницей между именами является то, что мнемоника xlatb вообще не может иметь параметра, а xlat может использоваться как с параметром, так и без него.

Жесткий способ определения сегментов

Большинство программ в этой книге используют преимущества, предоставляемые Turbo Assembler в упрощенной модели памяти, применяя такие директивы, как CODESEG и DATASEG, для обозначения начала кодового сегмента и сегмента данных программы. Для большинства целей это дает все необходимые возможности разделения кода и данных и разумной организации программы. Однако в более редких случаях, когда необходимо больше возможностей по управлению названиями и размерами сегментов, упрощенные модели памяти могут оказаться неподходящими. В этих случаях необходимо пользоваться "жестким способом" определения сегментов, используя директивы SEGMENT, ASSUME и GROUP.

Директива SEGMENT

Директива SEGMENT указывает Turbo Assembler на необходимость размещения всех последующих данных в одном сегменте памяти, который может содержать данные, код или стек программы. Программа может определять множество сегментов, приписывая им различные имена и атрибуты, которые приводят к объединению данных или кода в соответствии со всевозможными правилами и условиями. Ниже приводится полный синтаксис директивы SEGMENT:

SEGMENT *имя [выравнивание] [объединить] [использовать] ["класс"] [доступ]*

1. *имя* — идентификатор, такой как MYDATA или SEGA45X. Можно повторять одно имя при описании нескольких сегментов, даже в разных программных модулях. Turbo Assembler объединяет одноименные сегменты в один большой сегмент. Можно размещать этот сегмент в памяти, указывая смещение *имени* в сегментном регистре. Последующие за именем атрибуты необходимо указывать только при первом описании сегмента.
2. *выравнивание* (align) — указывает граничные ограничения на начало сегмента. Табл. 11.1 приводит различные символы, которые используются при *выравнивании*. В процессе ассемблирования, если текущая позиция в начале сегмента не удовлетворяет установленным правилам для заданного типа выравнивания, ассемблерный счетчик положения увеличивается на соответствующую величину, смещая начало сегмента в область старших адресов и, возможно, теряя несколько байтов. Если не указывать тип выравнивания, сегмент выравнивается на границу следующего 16-байтового параграфа (опция PARA).

3. *объединить* (combine) — устанавливает правила организации сегментов и объединения множественных сегментов в памяти. Табл. 11.2 приводит список символов, используемых при *объединении*. По умолчанию применяется Private.
4. *использовать* (use) — применяется только для процессоров 80386 и старших моделей в программах, использующих директивы типа P386, P386N, P486, P586 и аналогичные, которые позволяют применять специальные команды процессора и расширенные регистры не доступные на процессорах 8088 и 8086. Табл. 11.3 приводит список используемых символов.
5. *"класс"* — выполняет роль спецификатора категории. Все сегменты с одинаковым именем класса, даже с различными собственными именами, во время работы программы физически располагаются совместно в памяти.
6. *доступ* (access) — используется только в программах защищенного режима, которые компонуются под расширение DOS с применением компоновщика Phar Lap. (Turbo Linker не поддерживает эти возможности.) Ассемблируйте программу с использованием ключа /ор. Типы ограничения доступа для сегментов защищенного режима приведены в табл. 11.4.

Таблица 11.1. **Символы выравнивания директивы SEGMENT**

Символ	Выравнивать сегмент на...
Byte	Байтовый адрес (текущая позиция)
Word	Адрес слова (LSD адреса = 0)
Dword	Адрес двойного слова (2 LSD адреса = 0)
Para	16-байтовый параграф (4 LSD адреса = 0)
Page	256-байтовая страница (8 LSD адреса = 0)
MemPage	Начало сегмента на следующей странице памяти (граница 4 Кбайта)

Таблица 11.2. **Символы объединения директивы SEGMENT**

Символ	Значение
At-выражение	Размещает сегмент по адресу, указанному в выражении, которое должно представлять абсолютный адрес параграфа, такой как 0F00h или 0040h. Используйте эту опцию для доступа к данным, которые уже находятся в памяти, таким как переменные ROM BIOS
Common	Сегменты с одинаковым именем перезагружаются. Размер сегмента равен размеру наибольшего из всех сегментов. Используйте эту опцию для ссылок на общие переменные из разных модулей
Memory	Идентична опции Public. Приводит к последовательному объединению сегментов с одинаковыми именами
Private	Устанавливается по умолчанию. Приводит к тому, что сегменты с одинаковыми именами рассматриваются как различные сегменты. При этом перед обращением к переменным данного сегмента необходимо инициализировать сегментный регистр адресом этого сегмента
Public	Приводит к последовательному объединению сегментов с одинаковыми именами в памяти, в порядке, установленном в программе. В результате получается один большой сегмент, содержащий все данные или код из всех сегментов. При этом для доступа к переменным во всех сегментах необходимо инициализировать сегментный регистр только адресом первого из объединяемых сегментов
Stack	Используйте эту опцию только для выделения пространства под стек, обычно в основном модуле программы. Все .EXE-программы должны определять сегмент стека. Не следует загружать этот регистр в программе Несколько сегментов с одинаковым именем и установленным типом объединения STACK объединяются один после другого и формируют один большой сегмент стека. Это позволяет раздельным программным модулям выделять столько стекового пространства, сколько им нужно. (Помните о необходимости выделения дополнительного пространства для DOS, BIOS и векторов прерываний.)
UnInit	Дает указание о том, что данные записываются в неинициализированный сегмент. Можно использовать это свойство для предупреждения пользователей о том, что сегмент в модуле требует инициализации перед использованием, поскольку память под сегмент выделяется только во время выполнения программы
Virtual	Определяет общее пространство, которое должно находиться внутри другого сегмента. Обычно используется для объединения статических данных (или инициализированных переменных) из различных модулей в общее пространство внутри другого сегмента. Виртуальный сегмент имеет те же атрибуты, что и сегмент, в котором он определяется

Таблица 11.3. **Символы использования директивы SEGMENT**

Символ	Значение
Use16	Устанавливается по умолчанию. Позволяет 16-битовое внутрисегментное адресное смещение и ограничивает размер сегмента до 64K
Use32	Позволяет 32-битовое адресное смещение и допускает максимальный размер сегмента в 4GB

Таблица 11.4. **Символы доступа директивы SEGMENT**

Символ	Значение
ExecOnly	Сегмент может содержать только код (без данных)
ExecRead	Сегмент может содержать исполняемый код и данные только для чтения
ReadOnly	Сегмент может содержать только данные для чтения (без исполняемого кода)
ReadWrite	Сегмент может содержать изменяемые данные (без кода)

Использование директивы SEGMENT

Несколько примеров помогут объяснить, как устанавливать сегменты в собственной программе. Предположим, вам необходимо поместить три переменные типа "слово" в сегмент данных. Для этого можно описать его следующим образом:

```
SEGMENT Dseg Para Public 'DATA'
v1   dw 0
v2   dw 1
v3   dw 2
ENDS Dseg
```

Директива ENDS указывает на окончание сегмента и является обязательной. Здесь, после ENDS, можно указать имя сегмента Dseg или оставить эту позицию незаполненной. Сегмент выравнивается на ближайшую границу 16-байтового параграфа в памяти (Para) и, поскольку тип *объединения* установлен Public, добавляется к другим сегментам с именем Dseg или сегментам с тем же именем класса 'DATA'. Для того чтобы найти переменные в этом сегменте, необходимо инициализировать соответствующий сегментный регистр, обычно ds. Например, для установки в dx значения переменной v2 необходимо выполнить следующие шаги:

```
mov  ax,Dseg
mov  ds,ax
ASSUME ds:Dseg
mov  dx,[v2]
```

Так; скоро доберемся и до ASSUME, а сейчас необходимо осознать, что перед тем как ссылаться на переменные из сегмента, вы должны инициализировать сегментный регистр. В большинстве случаев это можно сделать, устанавливая значение имени сегмента (в данном примере — Dseg). Однако на данный момент эти команды подвешены в воздухе, а они также должны входить в некоторый сегмент. Типичный сегмент кода для главного программного модуля может выглядеть следующим образом:

```
SEGMENT Cseg Para Public 'CODE'
Start:
mov  ax,Dseg          ;Устанавливаем адрес сегмента
mov  ds,ax            ;в ds
mov  es,ax            ;и в es
ASSUME ds:Dseg, es:Dseg
;
; другие команды
;
ENDS Cseg             ;Конец сегмента кода
END  Start            ;Конец текста
```

Сегмент кода называется Cseg; он выравнивается к ближайшей границе параграфа, объединяется с другими сегментами Cseg или с другими сегментами из класса 'CODE'. Отметим, что директива END устанавливает адрес входа в программу, которую затем использует компоновщик, включая информацию в .EXE-файл, для того чтобы DOS загрузил кодовый сегмент (или сегменты) непосредственно в память, инициализировал cs и передал управление первой исполняемой команде программы.

В дополнение к сегментам кода и данных требуется стековый сегмент, в противном случае Turbo Linker предупредит, что в программе отсутствует сегмент стека — это серьезная ошибка, если вы не создаете .COM-программу. Типичный сегмент стека описывается так:

```
SEGMENT Sseg Word Stack 'STACK'
theStack      db 128 dup ('**Stack*')
ENDS
```

Поскольку установлен тип *объединения* Stack, регистры ss:sp автоматически устанавливаются на пространство, выделенное под стек, которое выравнивается на границу ближайшего слова. Имя класса 'STACK' приводит к тому, что различные сегменты одного класса объединяются так же, как и для других сегментов. Не путайте эти два элемента, только тип *объединения* сообщает компоновщику, что данный сегмент является стековым. Пространство под стек в данном примере выделяется директивой db, сохраняющей 128 копий строки '**Stack*' в 1024 байт. Во время отладки это упростит поиск стека в памяти — необходимо только найти строку '**Stack*'. После окончания программы вы можете изучить установленный стек и увидеть, сколько стекового пространства использовалось, найдя место нарушения строки. (Не забывайте добавлять дополнительное пространство для векторов прерываний — не устанавливайте размер стека ниже чистого минимума.)

Замечание

Одним из недостатков этого метода является то, что данные стека сохраняются в .EXE-файле на диске. В окончательной версии вы можете преобразовать ваш стек, применив директиву STACK из упрощенной модели памяти или описав неинициализированное стековое пространство с использованием оператора (?) вместо строки символов. Это позволит уменьшить размер исполняемого файла

Эти три элемента — сегменты данных, кода и стека — являются минимальными требованиями в программе, которая определяет сегменты "жестким образом". Но перед использованием этих идей для написания полных программ необходимо понять, что делает директива ASSUME.

Директива ASSUME

Для того чтобы понять действие директивы ASSUME, необходимо представить, что ваша программа существует в двух временных измерениях. Первое измерение — *время ассемблирования* — действия, происходящие во время ассемблирования текста программы. Второе измерение — *время исполнения* — действия, происходящие при загрузке вашей программы в память и выполнении первой команды.

Директива ASSUME относится исключительно ко времени ассемблирования — она не действует во время исполнения программы. Используйте ASSUME для того, чтобы сообщить Turbo Assembler, что определенный сегментный регистр адресует указанный сегмент. Например, предположив, что предыдущее описание сегмента данных — Dseg, можно установить регистр es для адресации сегмента в памяти:

```
mov  ax,Dseg                ;Устанавливаем адрес Dseg
mov  es,ax                  ;в es посредством ax
ASSUME es:Dseg              ;Сообщаем Turbo Assembler, куда указывает es
```

Во время исполнения программы две команды mov устанавливают в es адрес сегмента данных Dseg. Во время ассемблирования директива ASSUME сообщает Turbo Assembler, куда в данный момент указывает es. Причина необходимости обоих шагов состоит в том, что Turbo Assembler ассемблирует, но не "понимает" языка ассемблера; следовательно, необходимо сообщить ассемблеру, куда указывает es, хотя предыдущая команда загружает в es именно этот регистр. Общая форма директивы ASSUME:

```
ASSUME сегРег:сегИмя|NOTHING, ..., сегРег:сегИмя|NOTHING
```

Параметр *сегРег* может быть cs, ds, es или ss. Программы для 80386 и старших моделей процессоров могут также устанавливать регистры fs и gs, которые недоступны на 8086 и 80286. Параметр *сегИмя* должен ссылаться на имя сегмента, описанного директивой SEGMENT. (Как вы скоро увидите, *сегИмя* может также ссылаться на группу сегментов.) Вместо *сегИмя* можно использовать ключевое слово NOTHING, которое сообщает Turbo Assembler, что заданный регистр не адресует в данный момент никакого определенного регистра.

Используя ASSUME, вы даете Turbo Assembler возможность выполнить два действия.

- Проверить возможность адресации переменных в сегменте данных.
- Проводить автоматическую переустановку сегментов при необходимости.

Второе преимущество наиболее важно. При использовании ASSUME Turbo Assembler может вставлять команду переустановки сегмента es:. Например, предположим, что сегмент Dseg адресуется только регистром es. Тогда строки

```
ASSUME es:Dseg
mov  dx,[v1]
```

в действительности ассемблируются как

```
mov  dx,[es:v1]
```

Вы по-прежнему можете специфицировать переустановку сегментов, но не обязаны это делать. Директива ASSUME предоставляет Turbo Assembler решать, является ли переустановка необходимой. Это особенно удобно при использовании строковых команд и при ссылке на разные сегменты посредством как ds, так и es. Используя ASSUME после каждого назначения сегментного регистра, вы будете уверены, что Turbo Assembler сделает все возможное для проверки корректности ссылок и что переменные действительно находятся в сегменте, адресуемом сегментными регистрами.

Можно также осуществлять множественные описания, разделяемые запятыми. Например, при использовании описания сегмента из предыдущего обсуждения директивы SEGMENT типичная директива ASSUME может иметь вид

```
ASSUME cs:Cseg, ds:Dseg, es:NOTHING, ss Sseg
```

Директива GROUP

Сейчас, когда вы получили средства, необходимые для жесткого описания сегментов, вы, вероятно, захотите использовать директиву GROUP для упрощения ссылок к множественным сегментам. Директива GROUP имеет вид

```
GROUP имя сегИмя[, ..., сегИмя]
```

Имя и другие элементы директивы GROUP меняются местами при ассемблировании в режиме MASM. В качестве *имени* может выступать любой неиспользованный идентификатор, такой как dgroup или stacksegs. После параметра *имя* следует один или более параметров *сегИмя*, которые должны быть именами сегментов, использованными в директивах SEGMENT.

Замечание

Параметр *сегИмя* в директиве GROUP может быть и выражением, начинающимся с SEG, как в GROUP newgroup SEG myLabel, хотя это используется довольно редко Обычно лучше определять именованные сегменты директивой SEGMENT и затем использовать имена в директиве GROUP

Используйте директиву GROUP в случае, когда имеется множество сегментов с различными именами, которые вы хотите адресовать единственным сегментным регистром. При этом сегменты могут иметь различные имена классов. В действительности, если имена сегментов и имена классов сегментов отличаются, применение директивы GROUP является единственным способом объединения нескольких сегментов в памяти. Например, если три модуля определяют сегменты данных с именами Dseg, LocalSeg и OtherSeg, можно использовать директиву GROUP:

```
GROUP DataGroup Dseg, LocalSeg, OtherSeg
```

Несмотря на то, относятся эти сегменты к одному классу или нет, они будут объединены в один большой сегмент в памяти. После этого можно ссылаться на все переменные из трех сегментов, инициализировав ds (или es) и сообщив Turbo Assembler, куда указывает ds:

```
mov  ax,DataGroup          ,Установить адрес DataGroup
mov  ds,ax                 ,в ds посредством ax
ASSUME ds:DataGroup        ,Сообщить Turbo Assembler, куда указывает ds
```

Вместо того чтобы загружать смещения отдельных сегментов, вы можете сейчас загружать смещение имени группы, в данном случае DataGroup. То же имя группы используется и в директиве ASSUME, сообщая Turbo Assembler, куда указывает ds.

После такого объединения нескольких сегментов смещения всех переменных во всех объединяемых сегментах вычисляются автоматически. И как только регистр ds или es указывает на заданную группу, все переменные могут быть непосредственно адресованы. Единственным ограничением является то, что все объединенные сегменты не могут занимать более 64K.

Использование сегментов в программах

При отказе от упрощенной модели памяти описание сегментов программы требует тщательного планирования В большинстве случаев упрощенная модель выполняет все необходимое, но существует одно малоизвестное ограничение на эти модели памяти В Руководстве по Turbo Assembler, при обсуждении директивы MODEL, приводится несколько таблиц со списками имен сегментов, используемых различными упрощенными моделями Для демонстрации в табл 11 4 приведены имена для малой модели памяти и показаны директивы, использованные для описания каждого типа сегмента

Замечание

Большинство остальных моделеи использует имена, аналогичные именам приведенным в табл 11 4 Если вам необходимо точно знать эти имена, обратитесь к руководству по Turbo Assembler или ассемблируите программу с помощью команды tasm /l имя фаила Имена сегментов можно наити в конце фаила листинга (LST)

Табл 11 5 показывает недостатки упрощенных моделей памяти Данные, неинициализированные данные, константы и сегмент стека объединены в группу с именем DGROUP Это означает, что *общий* размер сегментов ограничен размером 64K! Другими словами, чем больше стекового пространства вы выделяете, тем меньше места для данных Это нехорошо даже для малой модели памяти Все упрощенные модели памяти объединяют сегменты данных и стека в DGROUP

Таблица 11 5 **Сегменты упрощенной малой модели памяти (Small)**

Директива*	Имя	Выравнивание	Объединение	Класс	Группа
CODESEG	_TEXT	Word	Public	'CODE'	
FARDATA	FAR_DATA	Para	Private	'FAR_DATA'	
UFARDATA	FAR_BSS	Para	Private	'FAR_BSS	
DATASEG	_DATA	Word	Public	'DATA'	DGROUP
CONST	CONST	Word	Public	'CONST'	DGROUP
UDATASEG	_BSS	Word	Public	'BSS'	DGROUP
STACK	STACK	Para	Stack	'STACK'	DGROUP

***Замечание** только для Ideal mode

Устанавливая ваши собственные сегменты, вы можете избежать этого ограничения, как показано в программе 11 4, HARDSHEL ASM — версия с "жестким" определением сегментов программы EXESHELL ASM из главы 2. Вы можете использовать программу HARDSHEL ASM как шаблон для написания собственных программ с полным управлением сегментами Программа позволяет иметь два сегмента данных по 64K, один стековый сегмент 64K и сегмент кода 64K с общим потенциальным размером около 256K

Для ассемблирования оболочки (которая ничего не делает, хотя и запускается на исполнение) и распечатки листинга и карты памяти введите команды

```
tasm /l hardshel
tlink hardshel
type hardshel lst >prn
type herdshel map >prn
```

Программа 11 4 HARDSHEL ASM

```
1     %TITLE ' EXE shell  nonsimple segments -- Copyright (c) 1989 1995 by Tom Swan"
2
3           IDEAL
4
5     ;----- Вставляйте присваивания EQU и = здесь
6
7
8     SEGMENT SSeg Para Stack 'STACK
9
10    ,     db    1024 dup ('**Stack* )      , 8K стека отладки
11          db    8192 dup (?)               , 8K неинициализированного стека
12
```

```
13:    ENDS SSeg
14:
15:
16:    SEGMENT DSeg Word Public 'DATA'
17:
18:    exCode DB 0
19:
20:    ;----- Объявляйте остальные переменные через DB, DW и т д. здесь
21:
22:    ;----- Специфицируйте любые переменные EXTRN здесь
23:
24:    ENDS DSeg
25:
26:
27:    SEGMENT ESeg Word Public 'EDATA'
28:
29:    ;----- Другой (far) сегмент данных
30:
31:    ENDS ESeg
32:
33:
34:    SEGMENT CSeg Word Public 'CODE'
35:
36:    ,----- Специфицируйте любые процедуры EXTRN здесь
37:
38:    Start:
39:          ASSUME ds:DSeg
40:          mov    ax, DSeg            ; Инициализировать DS адресом
41:          mov    ds, ax              ; сегмента данных
42:          ASSUME es ESeg
43:          mov    ax, ESeg            ; Инициализировать ES адресом
44:          mov    es, ax              ; дополнительного сегмента данных
45:
46:    ;----- Вставляйте программы, вызовы подпрограмм и т.д. здесь
47:
48:    Exit:
49:          mov    ah, 04Ch            ; Функция DOS: Конец программы
50:          mov    al, [exCode]        ; Вернуть код возврата
51:          int    21h                 , Вызов DOS  Завершить программу
52:
53:    ENDS CSeg                        ; Конец сегмента кода
54:
55:          END    Start               ; Конец программы / точка входа
```

Использование программы HARDSHELL.ASM

Несколько замечаний помогут вам использовать HARDSHEL.ASM в качестве шаблона. Строка 10 закомментирована. Удалите точку с запятой и закомментируйте строку 11, добавив тем самым 8 Кбайт строки '**Stack*' в исполняемый файл. При отладке вы сможете изучать память, выделенную под стек, и увидеть какое количество стековой памяти в действительности используется программой.

Два сегмента — Dseg и Eseg определяются в строках 16–31. Эти сегменты не объединяются вместе, хотя при желании вы можете это сделать. (Конечно, объединение сегментов данных также ограничивает общий размер объединенного сегмента размером 64 Кбайта.) Изучите, как код в строках 39–44 устанавливает в регистры es и ds адреса двух раздельных сегментов.

Замечание

В большинстве модулей в этой книге предполагается, что es и ds адресуют один и тот же сегмент данных. При использовании HARDSHEL ASM вы должны будете модифицировать эти модули или временно переустанавливать значение es в ds перед вызовом подпрограмм из них

Сегмент кода в стоках 34–53 может содержать до 64K. Если вам требуется больше пространства, можно определить дополнительный сегмент кода и осуществлять дальние вызовы процедур из этого модуля. Если вы сделаете это, то проверьте, чтобы процедуры заканчивались командой retf, а не ret.

Тип объединения At

В табл. 11.2 приводится список типов *объединения*, которые можно использовать в директиве SEGMENT. Один из них — At, который устанавливает сегмент по заданному адресу в памяти. Такой сегмент является *фантомом* — средством установить переменные, описанные в программе и уже существующие в памяти в результате каких-либо других процессов. Эта техника особенно полезна для ссылки на переменные, которые относятся к DOS или ROM BIOS. Очевидно, что такие переменные не создаются в вашей программе, а устанавливаются при включении компьютера. Существует два способа обращения к данным BIOS. Можно просто приравнять символ адресу в памяти и читать или записывать значения по этому адресу. (Проконсультируйтесь в руководстве по аппаратному обеспечению относительно этого адреса.) Чтобы избежать ошибок, вызванных записью значений по неверным адресам, используйте определение сегмента с опцией At, как это показано в программе 11.5, COLDBOOT.ASM. Ассемблируйте и скомпонуйте программу командами:

```
tasm coldboot
tlink coldboot
```

Замечание

Запуск программы COLDBOOT перезагрузит ваш компьютер и удалит все данные из памяти. Не исполняйте программу, пока перезагрузка вам не понадобится

Программа 11.5. COLDBOOT.ASM

```
1:    %TITLE "Perform Warm or Cold Reboot -- by Tom Swan"
2:
3:          IDEAL
4:          MODEL small
5:          STACK 256
6:
7:    WarmBoot     EQU 1234h          ; Проскочить системные тесты на наличие питания (POST)
8:    ColdBoot     EQU 1234d          ; Могут работать и другие значения
9:
10:   BIOSDataLoc  EQU 0040h          ; Адрес сегмента данных BIOS
11:   ResetFlagLoc EQU 0072h          ; Смещение ResetFlag среди данных BIOS
12:
13:
14:   ;----- Сказать ассемблеру, где слово ResetFlag расположено
15:
16:   SEGMENT BIOSData at BIOSDataLoc
17:         ORG ResetFlagLoc
18:   LABEL  ResetFlag Word
19:   ENDS
20:
21:         CODESEG
22:
23:   Start:
24:         mov   ax, BIOSDataLoc    ; Адресовать сегмент BIOSData
25:         mov   ds,ax              ; через ds
26:
27:   ASSUME DS:BIOSData
28:
29:         mov   [ResetFlag],ColdBoot      ; Установить ResetFlag
30:
31:         END   Start              ; Конец программы / точка входа
```

Как работает программа COLDBOOT.ASM

Программа COLDBOOT определяет один "жесткий" сегмент, хотя помимо этого она использует упрощенную модель памяти. В этом нет ошибки — вы можете по желанию объединять модели памяти с собственными сегментами. Эта программа определяет один сегмент по абсолютному адресу 0040h, который является началом сегмента данных ROM BIOS:

```
SEGMENT BIOSData at BIOSDataLoc
```

Во время работы программы этот сегмент не загружается в память, следовательно, нельзя устанавливать инициализированные переменные в BIOSData. Это в любом случае плохая идея —вы изменили бы переменные, при-

надлежащие ROM BIOS. Обычно вы будете ссылаться на уже существующие переменные, как это показано в строках 16–19. Директива ORG устанавливает начало на 0072h (символьное имя ResetFlagLoc), которое представляет адрес флага переустановки системы. Директива LABEL устанавливает метку ResetFlag на этот адрес, с тем чтобы дальнейшие команды могли ссылаться на данные в этом месте. Причина использования ORG состоит в том, чтобы избежать описания переменных по младшим адресам, в которых программа не нуждается. Нет смысла включать ссылку на весь сегмент данных BIOS, для того чтобы сослаться на единственную переменную.

Строки 24–27 выполняют шаги по загрузке в ds адреса BIOSData и, используя директиву ASSUME, сообщают Turbo Assembler, куда указывает ds. После этого команда mov устанавливает в ResetFlag величину ColdBoot, описанную в строке 8.

На некоторых компьютерах простое установление этой величины в флаг переустановки системы приводит к перезагрузке компьютера. На других компьютерах вам необходимо будет выполнить переход на адрес F000:0000, по которому команда jmp осуществляет переход на процедуру перезагрузки ROM BIOS. На некоторых системах для перезапуска можно выполнить int 19h. К сожалению, тяжело определить, какой из этих различных методов будет работать на определенной машине. Программа COLDBOOT.ASM работает на моей системе, но может не работать на вашей.

Замечание

Выйдите из Microsoft Windows и закройте все программы перед запуском COLDBOOT.

В строке 7 показана величина, устанавливаемая в ResetFlag для выполнения теплой загрузки — эффект, который соответствует нажатию <Ctrl+Alt+Delete>. Использование этой величины в ColdBoot в строке 29 по-прежнему перезапускает систему, но запускает тест памяти и другие аппаратные тесты, таким образом, можно сэкономить некоторое время.

Дальние сегменты данных

Если вам необходимо дополнительное пространство для данных, но вы хотите воспользоваться упрощенными моделями памяти, то можно использовать директиву FARDATA для создания нужного количества сегментов данных. Следует помнить только одно правило — вам необходимо устанавливать сегментные регистры для доступа к данным в дальних сегментах. За исключением этого усложнения, использование дальних сегментов данных совсем просто. Например, предположим вы хотите поместить все строки из программы в отдельный сегмент и тем самым освободить место в основном сегменте данных для других переменных. Во-первых, определим сегмент директивой FARDATA:

```
FARDATA
s1   db 'Добро пожаловать в TurboCalc',0
s2   db 'Copyright 1999 by PC Universe',0
s3   db 'Линия поддержки· 800-555-1212',0
```

Это все, что необходимо сделать для создания дальнего сегмента данных. Поскольку этот сегмент не включается в DGROUP (см. табл. 11.4), он не объединяется с остальными сегментами. Следовательно, для доступа к переменным в дальнем сегменте данных необходимо установить один или более сегментных регистров в программе Например, если вы хотите обрабатывать строки из этого примера с использованием подпрограмм из модуля STRIO, необходимо будет установить как es, так и ds:

```
CODESEG
mov  ax,@farData                ;Установить адрес сегмента
mov  ds,ax                      ,в ds
mov  es,ax                      ,и в es
ASSUME ds:@farData, es:@farData ;Сообщить Turbo Assembler
```

Вначале в es и ds устанавливается адрес дальнего сегмента данных, с использованием предопределенного символа @farData. Директива ASSUME сообщает Turbo Assembler об изменении ds. Далее можно импортировать подпрограммы из других модулей, таких как STRIO, и отображать строки:

```
EXTRN StrWrite:proc, NewLine.proc
mov  di,offset s1
call StrWrite
call NewLine
```

Для восстановления начальных значений es и ds выполним обычные действия:

```
mov  ax,@data              ;Установить адрес сегмента
mov  ds,ax                 ;в ds
mov  es,ax                 ,и в es
ASSUME ds:@data, es:@data  ;Сообщить Turbo Assembler
```

Не забывайте директиву ASSUME. Помните, что лучше (и в этом случае желательно) всегда сообщать Turbo Assembler о переназначениях сегментных регистров. Другая возможность состоит в занесении в стек и извлечении сегментных регистров для временного переключения на дальний сегмент данных. Предположим, вы хотите загрузить в dx переменную v1, расположенную в сегменте FARDATA:

```
FARDATA                ;Переменная в дальнем сегменте
v1   dw 99
CODESEG
push ds                ;Сохранить текущий сегмент в стеке
mov  ax,@farData       ;Встановить адрес дальнего сегмента
mov  ds,ax             ;в ds
ASSUME ds:@farData     ;Сообщить Turbo Assembler
mov  dx,[v1]           ;Загрузить переменную в dx
pop  ds                ;Восстановить оригинальный сегмент
ASSUME ds:@data        ;Сообщить Turbo Assembler
```

И снова директива ASSUME сообщает Turbo Assembler об изменении ds. Не забывайте ставить директиву ASSUME *после команды pop. Хотя pop и восстанавливает ds в исходное значение, но это происходит во время* исполнения программы. А вам по-прежнему необходимо сообщать Turbo Assembler, что происходит во время ассемблирования.

Множественные дальние сегменты данных

Обычно, если вы включаете несколько директив FARDATA в различные модули, все дальние сегменты данных *объединяются в один сегмент размером до 64K. Добавив имя, вы можете создать столько дальних сегментов* данных, сколько вам необходимо. Допустим, вам нужно два таких сегмента. Тогда вы можете начать так:

```
FARDATA FarOut
v1   dw 1
v2   dw 2

FARDATA FartherOut
v3   dw 3
v4   dw 4
```

Сейчас программа имеет два различных дальних сегмента данных — FarOut и FartherOut. Каждый из этих сегментов может достигать 64K, увеличивая общую емкость данных программы до 192K (включая начальный сегмент данных без стекового пространства и других элементов DGROUP). Уникальные имена в FARDATA исключают объединение сегментов.

Замечание

Если вы повторите одинаковые имена в различных директивах FARDATA, сегменты будут объединены, как будто эти имена не указывались

Для обращения к данным в различных дальних сегментах данных установите в сегментный регистр имя, используемое в директиве FARDATA. Применяйте директиву ASSUME для сообщения Turbo Assembler, куда в данный момент указывает сегментный регистр.

Предположим, вы хотите установить в cx значение из v1 (в сегменте FarOut) и в dx v3 (в FartherOut).

```
mov  ax,FarOut          ;Установить в ds
mov  ds,ax              ;адрес сегмента FarOut
ASSUME ds:FarOut        ;Сообщить Turbo Assembler
mov  ax,FartherOut      ;Установить в es
mov  es,ax              ;адрес сегмента FartherOut
ASSUME es FartherOut    ;Сообщить Turbo Assembler
mov  cx,[v1]            ;Установить значение v1 в cx
mov  dx,[v3]            ;v3 в dx
```

Поскольку директивы ASSUME сообщают Turbo Assembler информацию о том, куда указывают регистры ds и es, последние две команды mov могут просто загружать переменные по имени. Ассемблер проверяет возможность обращения к переменным v1 и v3 этих команд, и в случае загрузки переменной v3 в dx включает переустановку

сегмента es:, которая необходима, поскольку es адресует сегмент, в котором находится переменная v3. Вы можете это увидеть, если изучите машинный код программного фрагмента под Turbo Debugger. Ищите шестнадцатеричное число 26h, значение машинного кода для префикса переустановки сегмента es:.

Неинициализированные дальние сегменты данных

Другая директива — UFARDATA начинает описание неинициализированного дальнего сегмента данных, аналогичного обычному неинициализированному сегменту данных, описываемому директивой UDATASEG. Поскольку дальний сегмент не является частью DGROUP, он занимает отдельный сегмент, как и FARDATA сегмент, но с переменными, которые не содержат предопределенных значений. При описании переменных в UFARDATA всегда пользуйтесь директивой знак вопроса (?). Например:

```
UFARDATA
index       dw ?
array       db 1024 up (?)
```

Поскольку вы не определили никаких начальных значений, переменные будут существовать только во время исполнения программы. Для обращения к переменным в неинициализированных сегментах данных используйте символ @FarData:

```
mov  ax,@FarData?
mov  ds,ax
ASSUME ds @FarData?
```

Этот код установит адрес дальнего сегмента данных в ds. При описании нескольких дальних сегментов данных с помощью UFARDATA добавьте имя, как это ранее объяснялось для FARDATA, установите это имя в сегментный регистр и в директиву ASSUME. Например, представим два различных неинициализированных дальних сегмента данных, каждый емкостью до 64K:

```
UFARDATA BlackHole
space       dw ?
moreSpace   dw ?

UFARDATA BlakerHole
deepSpase   dw ?
deeperSpace dw ?
```

Для установки в ds адреса BlackHole и в es адреса BlakerHole необходим следующий код:

```
CODESEG
mov  ax,BlackHole
mov  ds,ax
mov  ax,BlakerHole
mov  es,ax
ASSUME ds BlackHole,es BlackerHole
```

Программирование 80286 и старших моделей процессоров

Если вы уверены, что ваши программы будут работать только на системах с процессором 80286 (или старших совместимых моделях процессоров), можно пользоваться специальными командами, введенными фирмой Intel в систему команд процессора 80286.

Если вы это сделаете, то будьте уверены, что ваша программа не будет работать на системах с процессорами 8086 и 8088. Для разрешения использования специальных команд используйте одну из двух директив:

- P286 — Позволяет использовать все команды 80286
- P286N — Позволяет использовать только команды незащищенного режима

В основном вы будете использовать P286N — команды защищенного режима P286 применяются для написания многозадачного программного обеспечения и редко бывают полезными в прикладном программировании, на вопросах которого сконцентрирована данная книга. Дополнительную информацию о написании операционных систем можно почерпнуть из руководства фирмы Intel и других, приведенных в библиографии.

Замечание

Использование P286 или P286N не ограничивает области применения программного обеспечения машинами с 80286 процессорами Поскольку старшие модели процессоров совместимы с 80286, эти директивы также позволяют использовать специальные инструкции для 80386, 80486 и Pentium В этом разделе эти процессоры рассматриваются совместно как 80286

Поскольку флаги и регистры процессора 80286 идентичны используемым в процессоре 8086, можно сразу приступать к программированию процессора 80286. (В действительности существует несколько новых флагов, но они используются только в защищенном режиме, на котором мы не будем подробно останавливаться.) В дополнение, процессор 80286 понимает все команды процессора 8086, как это было описано в данной и предыдущих главах. Табл. 11.6 приводит список новых команд, доступных при использовании 80286 и старших моделей.

Можно также ссылаться на главу 16 для получения более детальной информации по командам из табл. 11.6. Две строковые команды, которые могут читать и записывать строки в аппаратные порты и каждая из которых имеет сокращенные формы, приведены здесь отдельно, хотя мнемонические имена представляют те же команды. Мнемоники ins, insb, insw представляют одну команду, так же как и outs, outsb, outsw.

Три команды — bound, enter и leave были добавлены в 80286 специально для использования компиляторами языков высокого уровня, хотя, безусловно, вы можете применять их в чисто ассемблерном коде, как это будет показано позже.

Использование команды bound

Команда bound проверяет, находится ли индекс в заданном диапазоне — и иногда называется проверкой диапазона в языках высокого уровня. Поскольку большинство таких языков выполняет вызов подпрограмм для проверки индексов массивов, использование bound может увеличить скорость программы, сохраняя безопасность проверки диапазонов, которую многие программисты запрещают для увеличения скорости.

Команда bound требует два операнда. Первый из операндов должен быть 16-битовым регистром, таким как dx или bx, и должен содержать проверяемое индексное значение. Второй операнд – это адрес 32-битового двойного слова в памяти, содержащего младшее и старшее значение диапазона индекса. Если величина первого операнда не входит в установленный диапазон, процессор осуществляет прерывание 5. Очевидно, вам необходимо установить соответствующую процедуру перехвата и обслуживания этого прерывания.

Замечание

Прерывание 5 используется на PC и совместимых для процедуры „Печать экрана" и приводит к классическому конфликту, возникающему при отказе от микросхем 8086 и 8088 При этом Intel резервирует прерывание 5 для собственных нужд — ограничение, которое игнорируется фирмой IBM при проектировании PC Позднее, при выпуске 80286, Intel использовала прерывания для команды bound (Несомненно, компания должна была знать, что это приводит к конфликту с клавишей <PrtSc> PC) Таким образом, если вы используете команду bound для проверки индекса массива и индекс оказывается за пределами диапазона, а вы не запретили использование клавиши <PrtSc>, возникновение ошибки приведет также к печати содержимого экрана К сожалению, это будет происходить снова и снова, до тех пор пока вы не перезагрузите систему Это смешная история, но всем не до смеха

В качестве примера того, как инсталировать вектор прерываний при использовании команды bound, программа 11.6 моделирует возникновение ошибки проверки диапазона. Ассемблируйте, скомпонуйте и запустите на выполнение программу командами:

```
tasm bound286
tlink bound286,,, mta
bound286
```

Таблица 11 6 **Команды 80286 (незащищенный режим)**

Мнемоника/операнды	Описание
bound destinatin,source	Проверка границ массива
enter immediate,immediate	Установка границы стека процедуры
ins destination,dx	Ввод строки из порта
insb	Ввод строки байтов из порта
insw	Ввод строки слов из порта
leave	Выйти из процедуры (после enter)
outs dx,source	Вывод строки в порт
outsb	Вывод строки байтов в порт
outsw	Вывод строки слов в порт
popa	Восстановить все основные регистры
pusha	Сохранить все основные регистры

Замечание

Исполняйте следующую программу только на системах с процессором 80286 или старших.

Программа 11.6. BOUND286.ASM

```
%TITLE "Bound Test--80286/386 only! -- by Tom Swan"

    P286N
    IDEAL

    MODEL small
    STACK 256

DATASEG

exCode DB 0

errorMsg        db '**Error: array index out of bounds', 0
normalMsg       db 'Program ending with no errors', 0

lowRange        DW 100              ; Нижний предел индекса
highRange                           DW 199  ; Верхний предел индекса
oldSeg          DW ?                ; Сохраняет сегмент прерывания 5
oldOfs          DW ?                ; Сохраняет смещение прерывания 5

CODESEG

;----- Из STRIO.OBJ
EXTRN StrWrite:proc, NewLine:proc

Start:
    mov   ax, @data                 ; Инициализировать DS адресом
    mov   ds, ax                            ; сегмента данных
    mov   es, ax                            ; Установить es = ds

    push  es                        ; Сохранить es
    mov   ax, 03505h                ; Получить вектор прерывания 5
    int   21h                       ; Вызов DOS
    mov   [oldSeg], es              ; Сохранить адрес сегмента
    mov   [oldOfs], bx              ; Сохранить адрес смещения
    pop   es                        ; Восстановить es

    push  ds                        ; Сохранить ds
    mov   ax, 02505h                ; Установить новый вектор прерывания 5
    mov   dx, offset Int5ISR        ; по этому адресу смещения
    push  cs                        ; и по этому адресу
    pop   ds                        ; сегмента кода
    int   21h                       ; Вызвать DOS
    pop   ds                        ; Восстановить ds

    mov   bx, 2                     ; Приписать значение индекса bx
    bound bx, [lowRange]            ; Проверить границы индекса

    mov   dx, offset normalMsg      ; Выдать сообщение
    call  StrWrite                  ; "no errors"
    call  NewLine

Exit:
    push  ds                        ; Сохранить ds в стеке
    mov   ax, 02505h                ; Установить вектор прерывания 5
    mov   dx, [oldOfs]              ; по этому смещению и
    mov   ds, [oldSeg]              ; этому сегменту
    int   21h                       ; Вызвать DOS
    pop   ds                        ; Восстановить ds

    mov   ah, 04Ch                  ; Функция DOS: конец программы
    mov   al, [exCode]              ; Вернуть значение кода возврата
    int   21h                       ; Вызвать DOS. Закончить программу

;----- Процедура обработки прерывания 5: завершение программы

PROC    Int5ISR
```

```
68.    mov   ax, @data              ; Снова установить ds и es просто
69:    mov   ds, ax                       ; для надежности
70:    mov   es, ax
71:    mov   dx, offset errorMsg    , Сообщение об ошибке адресации
72:    call  StrWrite               , Высветить сообщение
73:    call  NewLine
74:    jmp   Exit                   , Конец программы
75: ENDP     Int5ISR
76:
77:    END   Start                  ; Конец программы / точка входа
```

Как работает программа BOUND286.ASM

Большая часть программы BOUND286.ASM занимается изменением и восстановлением вектора прерывания 5; эти действия описываются в главе 10. Программа обслуживания прерывания приведена в строках 67–75 и немного отличается от обычной. Вместо сохранения и восстановления регистров код просто устанавливает es и ds (возможно, это не является необходимым, но это неплохая идея на все случаи) и после вывода сообщения об ошибке переходит на метку выхода из программы, прерывая выполнение программы при обнаружении ошибки командой bound.

Строки 46, 47 демонстрируют команду bound. В регистр bx устанавливается проверяемое значение индекса. Измените значение 2 на 150 (или любое другое допустимое значение индекса 100–199). При запуске программы вы получите другое сообщение, доказывающее, что программа обслуживания 5-го прерывания не активизируется.

Строки 16, 17 сохраняют младшее и старшее значения индексного диапазона, используемого bound. Обе эти величины должны одновременно находиться в памяти в указанном здесь порядке. Хотя строка 47 использует простую прямую адресацию для обращения к этим переменным, вы можете применять другой способ адресации (см. главу 4).

Использование команд enter и leave

Команды enter и leave полезны при подготовке границ стека процедуры. Они выделяют и освобождают пространство стека для локальных переменных в подпрограммах. Такие переменные динамические — они существуют только во время работы подпрограммы. Эти методы обычно применяются в языках высокого уровня как составная часть их методов использования функций и процедур, но при желании вы можете применять их и в чисто ассемблерном коде (см. главы 12 и 13 для получения более подробной информации об адресации локальных переменных в стеке).

Используйте команду enter в первой строке процедуры. Enter требует двух операндов, каждый из которых должен быть числом. (Операнды могут быть выражениями или уравнениями, если их результатом является непосредственно числовое значение.)

Первый операнд представляет собой число байтов, которые необходимо зарезервировать в стеке. Второй операнд представляет собой уровень вложения процедуры. Если три процедуры включаются одна в другую, самая внутренняя процедура находится на уровне 2, средняя — на уровне 1, а внешняя — на уровне 0. Уровни вложения обеспечиваются в основном для поддержки таких языков, как Паскаль, который позволяет вложенным (дочерним) процедурам получать доступ к локальным переменным, описанным во внешних (родительских) процедурах.

Выполнение команды enter аналогично выполнению трех команд процессора 8086:

```
push bp              ;Сохранить текущее значение bp
mov  bp,sp           ,Записать указатель стека в bp
sub  sp,n            ;Выделить стековое пространство для переменных
```

Вначале bp засылается в стек для сохранения его исходной величины. Затем в bp устанавливается значение указателя стека sp, что позволит командам использовать этот регистр для адресации локальных переменных. Затем выделяется место под переменные путем вычитания первого параметра команды enter из указателя стека.

В любой процедуре, использующей enter, перед выходом из процедуры командой ret необходимо выполнить команду leave для освобождения стекового пространства, выделенного enter, и для восстановления sp и bp. Команда leave выполняет те же действия, что и две команды 8086:

```
mov  sp,bp           ;Восстановить указатель стека из bp
pop  bp              ;Восстановить значение bp
```

Копирование bp в sp освобождает пространство, занятое в стеке перед восстановлением величины bp, которая могла использоваться другими процедурами для адресации их собственных локальных переменных. В качестве примера полной процедуры, использующий enter и leave, приведем подпрограмму, которая выделяет пространство для четырех переменных типа слово в стеке:

```
P286N
PROC AnyProc
enter 8,0                    ;Резервируем 8 байт стека
mov  [word bp - 0],4         ;Записываем 4 в v1
mov  [word bp - 2],3         ;Записываем 3 в v2
mov  [word bp - 4],2         ;Записываем 2 в v3
mov  [word bp - 6],1         ;Записываем 1 в v4
leave                        ;Освобождаем пространство
ret                          ;Возврат в вызывающую процедуру
ENDP AnyProc
```

Команда enter резервирует 8 байт стекового пространства — место для четырех переменных типа слово. Помимо этого команда подготавливает bp для адресации переменных, как иллюстрировалось соответствующей командой mov. Первое слово находится по адресу [bp-0], второе — [bp-2] и т.д. Вместо слова (word) можно указывать byte, dword и другие описатели для адресации данных различного размера. Команда leave освобождает стековое пространство, используемое локальными переменными (при этом их значения разрушаются), и восстанавливает sp и bp, подготавливая почву для команды ret.

Использование команд pusha и popa

Эти две команды засылают в стек (push) и восстанавливают (pop) все регистры основного назначения. Обычно это необходимо в начале и конце процедур обслуживания прерываний, хотя вы можете использовать эти команды и в простых процедурах.

Выполнение команды pusha сохраняет регистры ax, cx, dx, bx, sp, bp, si и di в стеке в перечисленном порядке. Обратите внимание, что указатель стека также сохраняется. Но величина sp, копируемая в стек, соответствует значению sp перед выполнением pusha.

Соответствующая команда popa удаляет все регистры основного назначения из стека. Выполнение popa (обычно после предшествующей pusha) извлекает регистры di, si, bp, sp, bx, dx, cx и ax в соответствующем порядке. На самом деле величина sp отбрасывается, поскольку, если popa в действительности восстановит sp перед восстановлением оставшихся di, si и bp, эти три регистра получат неверные значения и стек будет сдвинут на три лишних слова. Действие popa соответствует тому, что вы и могли ожидать — все регистры общего назначения восстанавливают значения, которые они имели перед выполнением предыдущей команды pusha. Сегментные регистры не сохраняются и не восстанавливаются командами pusha и popa.

Чтение и запись строк в порты

Две строковые команды процессора 80286 (и старших моделей) — ins, outs читают и записывают строки в аппаратные порты, указываемые регистром dx. Эти команды и их сокращенные формы (см. табл. 11.6) действуют аналогично другим строковым командам. В случае команды ins регистры es:di адресуют пространство, куда должны записываться строковые данные. Выполнение команды ins читает один байт или слово из указанного порта и сохраняет данные в es:di. Если df=0, то di увеличивается на 1 для байта или на 2 для слова. Если df=1, то di уменьшается на соответствующее значение. Обычно команде ins предшествует префикс rep и установка счетчика в cx для выполнения загрузки ряда байтов или слов, как это показано ниже:

```
DATASEG
string db 80 dup (?)
strlen = $ - string
CODESEG
P286N
mov  dx,port number          ;Устанавливаем номер порта в dx
mov  ax,SEG string           ;Устанавливаем адрес сегмента,
mov  es,ax                   ,содержащего string в es
ASSUME es:SEG string         ,Сообщает Turbo Assembler
mov  di,offset string                ,Адресуем строку парой es di
mov  cx,strlen               ;Счетчик повторений в cx
cld                          ;Автоматическое увеличение di
rep  insb                    ,Загрузить строку байтов из порта
```

Для того чтобы завершить этот пример, необходимо установить действительный адрес порта в dx. И даже после этого, возможно, вы не сможете запустить на исполнение этот пример, пока в вашей системе не будет порта, из которого можно читать строки (в большинстве PC таких нет). Однако это показывает, как применять insb для периферийных или пользовательских систем с соответствующим аппаратным обеспечением.

Можно использовать аналогичный код для вывода строк на порты вывода, применяя команду outs, номер порта в dx и адрес строки в паре ds:si. Можно также воспользоваться переустановкой для адресации строки с помощью es:

```
cld
rep outs dx,[byte es:si]     ,Вывод строки в порт
```

Обычно outs используется, как и в данном примере, с префиксом повторения и счетчиком в регистре cx для передачи группы байтов или слов в аппаратные порты. Если df=0, то si увеличивается на 1 для байтов и 2 для слов. Если df=1, то si уменьшается на соответствующее значение.

Непосредственный сдвиг и циклическая ротация величин

Малозаметным улучшением в командах 80286 является возможность устанавливать величины непосредственного сдвига и ротации, большие, чем на 1. Это означает, что команды 8086

```
mov  cl,4              ,Установить значение смещения в cl
shl  ax,cl             ;Сдвинуть ax влево 4 раза
```

могут быть упрощены:

```
shl  ax,4
```

Такое же изменение применимо ко всем командам сдвига и вращения 8086. При необходимости вы можете по-прежнему указывать значение сдвига в cl.

Программирование процессора 80386

Если в вашей системе установлен процессор 80386 или старше, то вы имеете в распоряжении все команды 8086 и 80286 — плюс к этому преимущество сверхбыстрой обработки, которую вы и не могли предполагать. Как и 80286, процессор 80386 и старшие имеют команды защищенного и незащищенного режимов. Если не считать некоторых исключений, команды защищенного режима идентичны командам 80286. В дополнение к работе в защищенном и незащищенном режимах, новые процессоры имеют третий режим для запуска программ на *виртуальной 8086 машине*. Эта улучшенная техника программирования является царством многозадачного программного обеспечения, такого как Xenix, OS/2, Windows. Как уже упоминалось раньше, Turbo Debugger может исполнять программы в этом режиме для лучшего контроля над разрушением системы, доступами к закрытым областям памяти и т.д. Здесь нет места описывать системное программное обеспечение, но хорошей новостью является то, что если вы базируетесь на командах 8086, то ваши программы будут работать независимо от режима 80386.

Если вы уверены, что ваши программы будут исполняться только на процессорах 80386 или старших, вы можете воспользоваться преимуществами, предоставляемыми дополнительными командами, приведенными в табл. 11.6.

Начало программирования 80386

Рис. 11.3 иллюстрирует семейство 32-разрядных регистров и флагов процессора 80386. Обратите внимание, что все регистры 8086 доступны, но расширены до 32-битовой величины. Сегментные регистры идентичны, хотя и появилось два новых (fs и gs). Расширенными регистрами можно пользоваться для большинства команд 8086. Например, для очистки 32-разрядного аккумулятора следует применять команды:

```
P386N
xor  eax,eax
```

Для разрешения использования команд 80386 воспользуйтесь директивой P386N (незащищенный режим) или P386 (все режимы). Вы можете сделать это на любой системе — нет необходимости иметь процессор 80386 для ассемблирования и компоновки программ. Однако для исполнения полученных программ вам придется воспользоваться 80386 или старшим процессором.

Большинство команд в табл. 11.7 является 32-битовыми вариациями аналогичных команд 8086, которые вы уже хорошо знаете. Например, cmpsd действует идентично cmps (сравнить строки), но добавляет возможность сравнения двойных слов, наряду со сравнением обычных байтов и слов. Аналогично, insd, lodsd, movsd, outsd, scasd и stosd добавляют возможность работы с двойными словами командам 8086 lods, movs, scas и командам 80286 ins, outs. Другие команды используют 32-битовые расширенные регистры для выполнения операций, аналогичных операциям 8086 и 80286.

Замечание

Более подробное описание команд из табл. 11.7 вы найдете в главе 16

Битовое сканирование и установка

Используйте команды bsf (Bit Scan Forward — битовое сканирование вперед) и bsr (Bit Scan Reverse — обратное битовое сканирование) для установки в регистр номера позиции первого ненулевого бита в байте, слове или двойном слове. Сканирование вперед проходит от LSD (бит 0) до MSD; обратное сканирование проходит

от MSD до LSD. Если не найдено ни одного единичного бита, zf устанавливается в 0. Одним из способов использования этих команд может быть установка в cl числа битов, необходимых для сдвига единичного бита в LSD-позицию. Например:

```
P386N
     mov     bx,00100000b     ,Установить 5-й бит в 1
     xor     cl,cl            ;Обнулить cl
     bsf     cx,bx            ;Сканировать с 0-го по 15-й биты
     shr     bx,cl            ;Установить бит в LSD-позицию
@@10:
```

В этом примере тестируемая величина находится в bx, представленном для ясности в двоичном виде. Бит номер 5 в этом значении установлен в 1; следовательно, команда bsf установит в cx 5. После этого команда shr сдвигает bx для установки единичного бита в LSD позицию. В этом случае обе команды — bsf и bsr дадут одинаковый результат. А теперь рассмотрим случай, когда больше, чем один бит, установлен в 1:

```
P386N
mov  bx,00010110b          ; Устанавливаем биты 1,2 и 4
bsf  cx,bx                 ; cx=1
bsr  cx,bx                 ; cx=4
```

Команда bsf ищет первый единичный бит, начиная с 0-й позиции, и, следовательно, устанавливает cx в 1. Команда bsr сканирует в противоположном направлении и устанавливает cx в 4 — номер позиции первого от MSD единичного бита в bx.

Таблица 11.7. **Команды 80386 (незащищенный режим)**

Мнемоника/Операнды	Описание
bsf назначение,источник	Битовое сканирование вперед
bsr назначение,источник	Обратное битовое сканирование
bt назначение,источник	Проверка бита
btc назначение,источник	Проверка бита и дополнение
btr назначение,источник	Проверка бита и переустановка
bts назначение,источник	Проверка бита и установка
cdq	Преобразование двойного слова в четверное
cmpsd	Сравнение строк двойных слов
cwde	Преобразование слова в расширенное двойное слово
insd	Ввод строк двойных слов
lfs назначение,источник	Установить указатель и fs
lgs назначение,источник	Установить указатель и gs
lss назначение,источник	Установить указатель и ss
lodsd	Установить строку двойных слов
movsd	Копировать строку двойных слов
movsx назначение,источник	Копировать и расширить знак
movzx назначение,источник	Копировать и расширить 0-й знак
outsd	Вывод строки двойных слов
popad	Восстановить все 32-битовые регистры
popfd	Восстановить все 32-битовые флаги
pushad	Сохранить все 32-битовые регистры
pushfd	Сохранить все 32-битовые флаги
scasd	Сканировать строку двойных слов
set условие	Установить байт по условию
shld назначение,источник,счетчик	Сдвиг с двойной точностью влево
shrd назначение,источник,счетчик	Сдвиг с двойной точностью вправо
stosd	Сохранить строку двойных слов

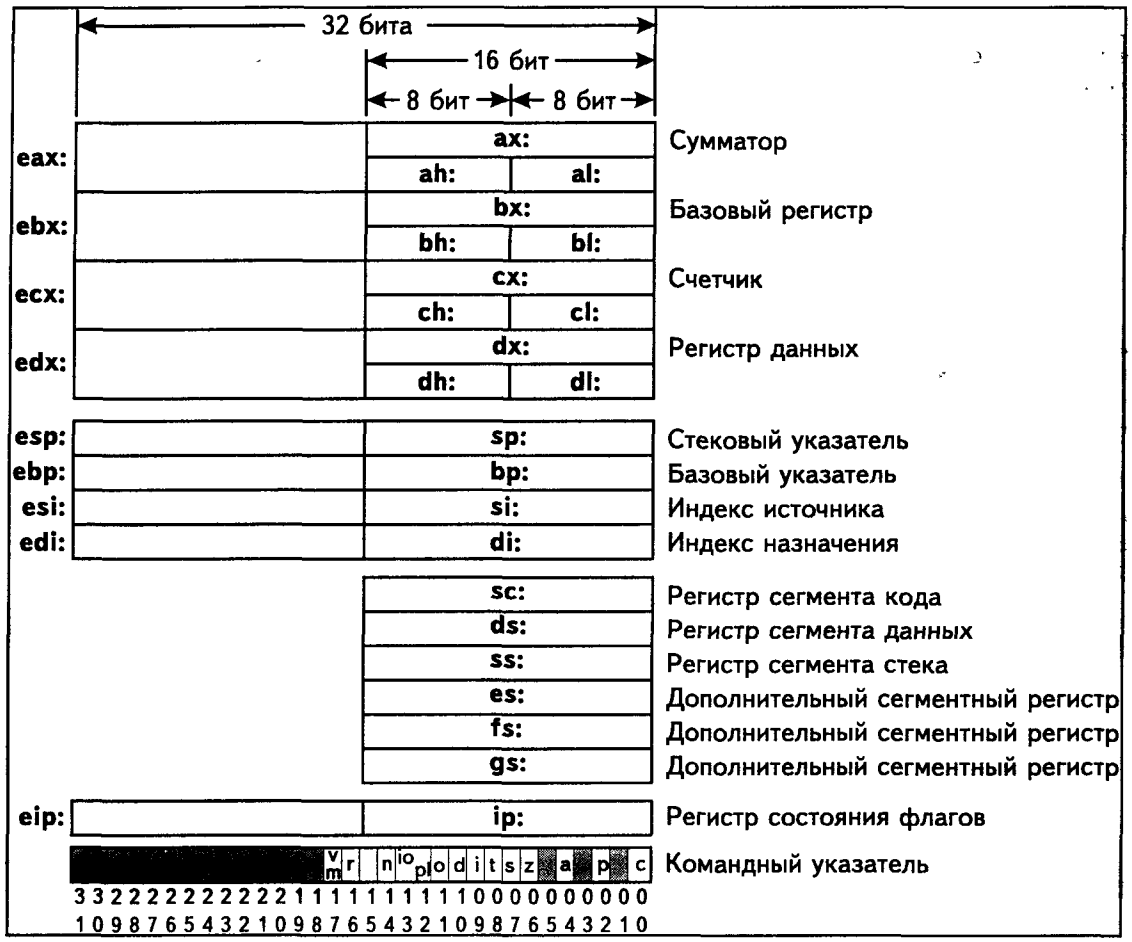


Рис. 11.3. Регистры и флаги 80386

Проверка битов

Команды bt, btc, btr и bts все выполняют одни и те же, хотя и слегка различающиеся задачи. Каждая из команд использует два операнда. Каждый из операндов может быть 16- или 32-разрядным регистром; кроме того, второй операнд может задаваться непосредственным значением. Независимо от формы представления, второй операнд представляет собой номер бита, который будет копироваться в флаг переноса. Вот пример, в котором cf устанавливается в 1:

```
mov  dx,0010000b          ;Устанавливаем бит 5 в 1
bt   dx,5                 ;Копируем 5-й бит в cf
```

Другие три команды действуют таким же образом, но по-разному влияют на исходное значение после копирования бита в cf. Команда btc дополняет (переключает) исходное значение бита; btr переустанавливает значение в 0, а bts в 1. Несколько примеров помогут понять их действия:

```
mov  dx,01010011b         ;Установить начальное значение в dx
btc  dx,7                 ;cf=0; dx=11010011 (бит 7 <- 0 )
btr  dx,0                 ;cf=1; dx=11010010 (бит 1 <- 0 )
bts  dx,3                 ;cf=0; dx=11011010 (бит 3 <- 0 )
```

Команда btc из примера копирует 7-й бит из dx в cf и дополняет исходный бит в dx. Команда btr копирует бит 0 в cf и затем устанавливает этот бит в 0. Команда bts копирует бит 3 в cf и затем устанавливает этот бит в 1.

Еще одни преобразования

В дополнение к cbw, которая преобразует байты в слова, и cwd, преобразующей слова в двойные слова, можно использовать cdq, для того чтобы конвертировать 32-битовые двойные слова в 64-битовые четверные слова и cwde для конвертации слов в двойные слова в расширенный аккумулятор eax. Эти команды полезны, когда выполняется работа со знаковыми целыми различных размеров. Простой пример показывает, как использовать новые добавления к 80386:

```
mov  ax, -3          ; Установить ax в -3 (ax = 0FFFDh)
cwde                 ; Установить eax в -3 (eax = 0FFFF FFFDh)
cdq                  ; Установить edx.eax в -3 (edx = 0FFFF FFFFh;
                     ;        eax = 0FFFF FFFDh)
```

16-битовая величина в ax (–3) переводится к полной 32-битовой длине расширенного аккумулятора eax командой cdw. Эта величина затем дальше расширяется на два регистра — edx и eax. Во всех случаях приписывания регистру фиксированы, как это здесь показано — можно только расширять величины в ax до eax и edx. Вы не можете расширять значения в других регистрах общего назначения.

Другие команды процессора 80386

Можно загружать указатели в регистры общего назначения и плюс в дополнительные регистры сегмента fs и gs с помощью lfs и lgs. Третья команда — lss позволяет вам инициализировать ss и sp. Они работают идентично les и lds, но загружают значения величин сегмента в специальные регистры сегмента. Например:

```
DATASEG
ptr48       dw      1,2,3
CODESEG
P386N
lfs  ebx, [pword ptr48]          ; Загружает ptr48 в fs:ebx
lgs  edi, [pword ptr48]          ; Загружает ptr48 в gs:edi
;lss esp, [pword ptr48]          ; Загружает ptr48 в ss:esp
```

Заметьте, что pword специфицирует ссылку в памяти во втором операнде каждой команды. Этим указывается Turbo Assembler на то, что переменная, определенная здесь как состоящая из нескольких частей директивы dw, в действительности — 48-битовый указатель (16 бит сегмента и 32 бита смещения). Команда lfs устанавливает ebx в 000200001h и fs в 0003h, собирая эти величины под меткой ptr48 в сегменте данных. Команда lgs аналогичным образом устанавливает edi в 000200001h и gs в 0003h. Команда lss устанавливает ss и esp аналогичными значениями, но, вероятно, разрушает компьютерную систему. По этой причине команда lss показана здесь как комментарий. Вы должны проявлять большую осторожность, когда используете lss для изменения сегмента стека и указателя.

К другим полезным командам можно отнести еще две команды — movsx и movzx. Используйте их для того, чтобы приписывать знаковые и беззнаковые значения из маленьких регистров или переменных памяти большим регистрам. Для обеих команд первый операнд должен быть 16- или 32-битовым расширенным регистром. Второй операнд может быть 8- либо 16-битовым регистром или указателем на память. Например, если у вас есть 8-битовая величина со знаком в bl, можно перевести эту величину в 16-битовый регистр dx с помощью команд:

```
mov   bl, -7         ; Инициализировать bl как -7 (8 бит)
movsx dx, bl         ; Установить dx в -7 (16 бит)
```

Или вы можете скопировать 16-битовую величину в 32-битовый регистр с помощью команд:

```
mov   dl, -8         ; Инициализировать dx как -8 (16 бит)
movsx eax, dx                ; Установить eax в -8 (32 бита)
```

Используйте movzx для того, чтобы сделать то же самое, но для беззнаковых величин. Например:

```
mov   bl, 255                ; Инициализировать bl как 255 (8 бит)
movzx ax, bl         ; Установить ax в 255 (16 бит)
mov   bx, 25890      ; Инициализировать bx как 25890 (16 бит)
movzx eax, bx                ; Установить eax в 25890 (32 бита)
```

Аналогично командам pusha и popa для процессора 80286 используйте pushad и popad для сохранения и восстановления всех 32-битовых расширенных (двойного слова) регистров общего назначения. Выполняйте pushad для проталкивания регистров eax, ecx, edx, ebx, esp, ebp, esi и edi в таком порядке. Величина сохраненного esp равна величине указателя на стек до выполнения pushad. Выполняйте popad для изъятия тех же самых регистров из стека в указанном порядке: edi, esi, ebp, esp, ebx, edx, ecx и eax. Величиной esp можно пренебречь, хотя esp по-прежнему восстанавливается тем же значением, которое он имел до pushad.

Другая команда — set-condition аналогична условному переходу. Смысл, однако, состоит в том, чтобы установить байтовый регистр либо величину из памяти в 1 или 0, в зависимости от того, удовлетворилось ли определенное условие. Например,

```
cmp  ax, 1
sete bh
```

устанавливает bh в 1 только, если ax равен 1. Окончания у set такие же, как и у команд условного перехода: setb, seta, setz, setnle и т.д. (Полный список мнемонических сокращений и установок флагов, проверяемых этой командой, см. в главе 16 на set-condition.)

Сдвиги двойной точности

Последними двумя командами являются shld и shrd, требующие, не как обычно, три операнда. В большинстве случаев, когда вам надо сдвинуть 32-битовые регистры, вы можете использовать только сдвиг для процессора 8086 и команды вращения, такие как shr и rcr, определяя расширенный регистр, как в этих строках:

```
mov  eax, 4          , Инициализировать eax как 4 (32 бита)
shl  eax, 3          ; Умножить eax на 8
```

Команды сдвига двойного слова работают несколько иначе. Первый операнд в shld и shrd определяет приемник результата и может быть регистром слова либо двойного слова или ссылкой на память. Второй операнд, который должен быть регистром слова или двойного слова, содержит биты, сдвигаемые в первый операнд. Третий операнд представляет собой число битов, которые надо сдвинуть в указанном направлении (направо для shrd и налево для shld). Этот операнд может быть конкретной величиной от 0 до 31 или же регистром cl. Например,

```
shld eax, ebx, 4      , Сдвинуть первые четыре бита ebx -> eax
```

сдвигает 4 бита из ebx в eax. Величина в ebx не меняется. Циклы с командами shld и shrd особенно полезны для выполнения сдвигов разной-точности (multiple-precision) над очень большими величинами. Более полным примером того, как это работает, является код в главе 16 для shld.

Директива VERSION

В Turbo Assembler 4.0 добавлена новая директива VERSION, которая заменяет некоторые другие директивы в более ранних ассемблерах. Например, в некоторых вариантах TASM используется символ QUIRKS, теперь устаревший, для того, чтобы эмулировать некоторые синтаксические особенности Ассемблера Microsoft (MASM).

Можно использовать VERSION для того, чтобы ассемблировать программы, написанные для большинства версий MASM и TASM. В табл. 11.8 перечислены аргументы, которые вы можете задавать. Например, для ассемблирования программы TASM 2.5 с использованием Turbo Assembler 4.0 вставьте где-нибудь в начале исходного текста программы директиву

```
VERSION T250
```

Таблица 11 8 **Аргументы VERSION**

Аргумент	Ассемблер	Версия
M400	MASM	4.0
M500	MASM	5.0
M510	MASM	5.1
M520	MASM	5.2
T100	TASM	1.0
T101	TASM	1 01
T200	TASM	2.0
T250	TASM	2.5
T300	TASM	3.0
T310	TASM	3.1
T320	TASM	3.2
T400	TASM	4.0

Директива VERSION заменяет следующие символы, которые можно найти в предыдущих версиях языка ассемблера:

```
MASM51, NOMASM51, QUIRKS, SMART, NOSMART
```

Перечисляемые типы данных

Присваивание имен и чисел — это стиль программирования, экономящий время, для написания понятных компьютерных программ — в любом языке, не только в ассемблере. Например, написанная без всяких пояснений, эта команда лишена смысла:

```
mov  al, 8
```

Конечно же, таким образом величина 8 перемещается в регистр al. Но что представляет собой 8? В программе календаря это число может представлять месяц Октябрь, в игре — уровень игры. Ничто не говорит о том, что конкретно делает эта программа.

Вы можете добавить комментарий, для того чтобы сделать программу понятнее:

```
mov  al, 8 , Приписать al Октябрь
```

Но почему бы не пройти еще милю и не создать *символ*, представляющий число мнемонически? Например, можно определить символ OCTOBER, который равен величине 8:

```
OCTOBER EQU 8
```

Вы сможете затем использовать этот символ в программе, определяя назначение утверждения совершенно ясным образом, без необходимости проясняющих комментариев:

```
mov  al, OCTOBER
```

Перечисление типов данных — это техника программирования, которая автоматизирует присваивание символов и чисел (в основном, последовательных). Вместо того чтобы набирать директивы EQU и приписывать им буквальные значения, можно использовать директиву ENUM для создания набора символов.

Например, в программе, которая использует дни недели, вы могли бы создать перечисляемый тип данных, подобный следующему:

```
ENUM ETDays SUN, MON, WED, THU, FRI, SAT
```

Тип данных ETDays представляет собой символы от SUN до SAT, которые внутренним образом представляют числовые величины от 0 до 6. По общему соглашению, я записываю перед именем типа данных ET, означающее "перечисляемый тип" (enumerated type), но вы можете использовать и другое имя.

Перечисляемый тип данных — это просто объявление, он не занимает никакой памяти в конечной программе. Для того чтобы использовать перечисляемый тип данных, вы должны, вдобавок к его описанию, выделить пространство для объекта этого типа, обычно в сегменте данных программы. Например, таким образом создается переменная по имени aDay типа данных ETDays:

```
aDay ETDay ?
```

Это грубый эквивалент директивы DB. Знак вопроса означает, что переменная не определена и память под нее будет выделена во время работы. Для определения точного значения перечисляемой величины начальное значение задавайте таким способом, как этот:

```
aDay ETDay WED
```

Тем самым создается переменная по имени aDay, инициализированная символом WED. В aDay запоминается 3, но этот факт для символьного представления не важен.

Перечисляемые типы данных используются так же, как и символы присваивания. Предыдущие названия дней, например, аналогичны индивидуальному присваиванию:

```
SUN EQU   0
MON       EQU   1
...
SAT EQU   6
```

Но есть важная разница между символами присваивания и перечисляемыми типами данных Ассемблер не только приписывает за вас символьные выражения с помощью NUM, но он может также предотвратить некоторые недопустимые операции. Например, можно попытаться приписать символ TUE как 16-битовое слово переменной памяти

```
mov  ax, TUE
mov  [aDay], ax
```

Это приведет к сообщению об ошибке *"Несоответствие типов операндов"* (Operand types do not match) из-за того, что во второй строке делается попытка сохранить слово в 8-битовой переменной. Так как перечисляемый тип данных — это байт, сохранение в нем 16-битовых величин не допускается. Правильным кодом будет:

```
mov  al, TUE
mov  [aDay], al
```

Для символов, которым значения присвоены отдельно, ассемблер не может определить ошибку такого вида. Поэтому перечисляемые типы данных могут помочь в предотвращении ошибок.

Иногда полезна другая форма директивы ENUM, записанная в несколько строк. Вот как можно использовать ее для задания набора имен месяцев:

```
ENUM ETMonths {
     JANUARY
     FEBRUARY
     MARCH
     APRIL
     MAY
     JUNE
     JULY
     AUGUST
     SEPTEMBER
     OCTOBER
     NOVEMBER
     DECEMBER
}
```

Конечный результат — это набор перечисляемых символов от JANUARY до DECEMBER, которые равны последовательным величинам от 0 до 11. Обратите внимание, что используемые в такой форме символы записаны в отдельных строках между скобками и не разделены запятыми.

Иногда вы можете захотеть изменить значения, связанные с перечисляемыми символами. Например, JANUARY связан с 1, FEBRUARY — с 2 и т.д. Для того чтобы изменить значение, связанное с символом, припишите новое значение, как ниже:

```
ENUM ETMonths {
     JANUARY = 1
     FEBRUARY
     MARCH

     DECEMBER
}
```

Вы можете сделать аналогичные присваивания для всякого другого или любых других перечисляемых символов Следующий символ на один больше. Другими словами, FEBRUARY сейчас равен 2, MARCH — 3 и т.д.

Создайте переменную типа данных ETMonths таким образом:

```
aMonth       ETMonths ?
```

Затем припишите ей значение, используя следующее выражение:

```
mov  al, SEPTEMBER
mov  [aMonth], al
```

Становимся SMART (ушлыми)

Если директива SMART разрешена, Turbo Assembler может помочь написать вам более эффективную программу на языке ассемблера. С такой директивой ассемблер заменяет некоторые виды команд на более короткие и быстрые. Включите генерирование smart-кода, добавив в начале текста вашей программы такую директиву:

```
SMART
```

Выключение smart-кодировки происходит после вставки NOSMART:

```
NOSMART
```

Совет

Вы, может быть, захотите использовать NOSMART, когда программа отлаживается, для того чтобы увидеть те самые команды которые вы написали Когда SMART установлена, в процессе отладки в окне CPU Turbo Debugger может показывать команды, которые вы не писали

Эффективная адресация SMART

Используя SMART, Turbo Assembler может заменить некоторые виды вычислений с адресами более эффективными, со смещениями. Например, предположим, вы хотите адресоваться к переменной, определенной в сегменте данных:

```
DATASEG
data DW ?
```

Можно использовать команду lea (загрузить действительный адрес — load effective adress) для того, чтобы загрузить адрес data в регистр:

```
CODSEG
lea  ax, [data]
```

Однако эта команда расточительна по времени и памяти. Более короткая и быстрая команда, которая делает ту же операцию, просто помещает *смещение* адреса data (относительно ее сегмента данных) в ax:

```
mov  ax, offset data
```

Генерируя код со SMART, Turbo Assembler автоматически замещает команду lea на эквивалентную ей mov, более быструю, имеющую меньше байтов. Ассемблер делает замену только тогда, когда целевой адрес может быть приравнен относительному смещению.

Расширенные булевы операции со знаком

Некоторые команды, такие как and, имеют расширенную форму со знаком (sign-extended), которая требует на байт или два меньше памяти Директива SMART Turbo Assembler может выбрать эти, более эффективные, команды автоматически. Например, в этом фрагменте кода определяется слово данных и затем над ним выполняется логическая команда and с маской –2.

```
DATASEG
data DW 1234h
CODESEG
and  [data], -2
```

При нормальных условиях команда and ассемблируется с использованием 16-битовой символьной формы команды, закодированной в машинном языке как поток следующих байтов кода (ассемблированная команда показана справа):

```
81260100FEFF          and word ptr [0001], FFFE
```

Шестнадцатеричная величина FFFE (байты в команде переставлены) представляет –2 как 16-битовую строковую величину. Эта величина, однако, может быть эффективнее представлена как шестнадцатеричный байт FE с использованием расширенной формы команды and со знаком, которая растягивает байт внутренним образом до слова. Когда действует SMART-генерация кода, Turbo Assembler выбирает эту альтернативную команду and, записывая эти байты в поток кода

```
83260100FE            and word ptr [0001], FFFE
```

Зови меня Smart

Когда вызываются дальние подпрограммы внутри одного и того же сегмента кода, следующая команда генерирует неэффективный код:

```
call far Subroutine
```

В этом случае более эффективно протолкнуть регистр текущего кода сегмента (cs) в стек и выполнить близкий call:

```
push cs
call near Subroutine
```

Это действует так же, но быстрее. Когда разрешена генерация кода SMART, Turbo Assembler автоматически заменяет дальние вызовы сохранением и близким вызовом в случае, когда исходный и целевой адреса сегмента кода одни и те же.

Замечание

Мое тестирование показывает, что, несмотря на документацию Borland по генерации кода со SMART, директива NOSMART в действительности не выключает оптимизацию нефрагментированных команд вызова Хотя это, по-видимому, является ошибкой Turbo Assembler 4.0, трудно себе представить какую-либо достойную причину для того, чтобы отказаться от этого, так что проблема не очень велика.

Проталкивание, проталкивание...

80386 и более поздние модели процессоров позволяют сохранять константные значения в стеке. Это может оказаться полезным для передачи аргументов в функции. Например, используя только команды 8086 процессора, вы должны загрузить регистр и протолкнуть его в стек таким образом:

```
mov  ax, 10          ; Загрузить значение в ax
push ax              ; Протолкнуть значение в стек
call Subroutine      ; Вызвать подпрограмму
pop  ax              ; Вытолкнуть значение из стека
```

Для 80386 и более поздних процессоров вы можете запомнить значение символьной константы непосредственно, заменив предшествующий код следующим:

```
push 10              ; Протолкнуть значение в стек
call Subroutine      ; Вызвать подпрограмму
pop  ax              ; Вытолкнуть значение из стека
```

Для лучшей переносимости программ генерация кода в Turbo Assembler со SMART позволяет использовать ту же технологию даже для процессоров 8086, в которых команда push не может сохранять константные величины. Если вы разрешите только команды процессора 8086, вставив директиву P8086 в программу, Turbo Assembler заменит предшествующий код следующими командами:

```
push ax                          ; Пробить дырку в стеке
push bp                          ; Сохранить текущий регистр bp
mov  bp, sp                      ; Адресовать стек через bp
mov  word ptr [bp+02], 000Ah     ; Опустить значение в дырку
pop  bp                          ; Восстановить сохраненный bp
```

Эта последовательность представляет милый фокус для вставки константных значений в стек 8086. Первая команда push "пробивает дырку" в памяти стека, создавая пространство, в которое константное значение будет вставлено. Второй push сохраняет bp для адресации стека. После установки bp равным sp команда mov опускает константу 10 (шестнадцатеричные 000A) в пробитую дырку. В конце pop восстанавливает сохраненное значение bp.

Замечание

Вы все еще должны поставить pop после предыдущего кода для того, чтобы вернуть сохраненное слово из стека. Turbo Assembler не сделает это за вас

Некоторые дополнительные команды

В Turbo Assembler 4.0 добавлено несколько новых сокращений для команд, определенных для 80386 и более поздних процессоров. Это не новые команды. Они определяют другие, и иногда более эффективные, формы команд, которые могут оказаться удобными время от времени. В следующих разделах обсуждается, как использовать эти альтернативные команды.

Замечание

Все примеры программ в следующих частях требуют 80386 или более поздних моделей процессоров. Используйте директиву P386 в вашей программе для разрешения выполнения этих команд.

Цикл цикла

Команда loop — одна из наиболее полезных в наборе команд 80386. Для нее вы можете установить счетчик цикла в cx и автоматически создать цикл, который будет повторяться заданное число раз. Например, в этом фрагменте кода loop используется для вызова подпрограммы (не показано):

```
    mov     cx, 10          ; Установить счетчик цикла в cx
@@99:
    call    Subroutine      ; Вызвать подпрограмму
    loop    @@99            ; Цикл по cx
```

Команда loop уменьшает cx, и если регистр не равен 0, переходит на определенную метку (@@99).

Все это работает прекрасно до тех пор, пока вы не начнете программирование с 32-битовым сегментом кода, используя 80386 и более поздние процессоры. При нормальных условиях Turbo Assembler ассемблирует команду loop, которая использует регистр cx, если сегмент кода является 16-битовым, но применяет 32-битовый регистр ecx для 32-битового сегмента кода.

Если вы захотите использовать 16-битовый регистр cx для команды loop 32-битового сегмента кода, вам не повезет — пока вы не примените одну из альтернативных команд loop, обеспечиваемых Turbo Assembler. Например, вы можете использовать loopw (w символизирует *слово (word)*):

```
loopw @@99          ; Цикл с 16-битовым cx
```

Это не новая команда. Она просто определяет, что cx должен быть использован как счетчик цикла даже для 32-битового сегмента кода. Аналогичным образом вы можете использовать расширенный 32-битовый регистр ecx как счетчик для 16-битового сегмента кода применением альтернативной команды loopd (d символизирует двойное слово (doubleword)):

```
loopd @@99          ; Цикл с 32-битовым ecx
```

Представленная выше форма особенно полезна для написания циклов, которые должны повторяться больше, чем 65536 раз.

Как вы, наверное, помните, существует пять стандартных команд цикла — loop, loope, loopz, loopne и loopnz (Просмотрите их в главе 16, если вам надо освежить в памяти то, что делают эти команды.)

Добавьте w после loop в этих командах для того, чтобы избрать их альтернативные формы со словом (16-битовый cx) — loopw, loopwe, loopwz, loopwne и loopwnz. Добавьте d после loop для того, чтобы избрать формы с двойным словом (32-битовый ecx) — loopd, loopde, loopdz, loopdne и loopdnz.

Enter или Leave (войти или остаться)

В этой главе я уже объяснял, как использовать enter и leave. Когда применяется процессор 80386 или более поздние модели процессоров и 32-битовый сегмент кода, ассемблер обычно вставляет команды так, чтобы использовать 32-битовые расширенные регистры ebp и esp.

Как и с командой loop, вы можете применять альтернативные формы enter и leave для того, чтобы заставить использовать 16- или 32-битовые регистры независимо от размера сегмента. Замените enter на enterw и leave на leavew, выбирая 16-битовые команды с регистрами bp и sp. Замените enter на enterd и leave на leaved, выбирая 32-битовые команды с регистрами ebp и esp.

Вернуть отправителю

Программирование процессоров семейства 80×86 требует постоянного внимания к формату адресов. Например, когда вызываются подпрограммы, вам надо использовать 16-битовый call, если эта подпрограмма заканчивается локальной (near) командой ret. С использованием режима Ideal, директив PROC и упрощенных моделей памяти вы обычно можете игнорировать эти вопросы и доверить Turbo Assembler за вас выбрать правильные команды call и ret.

В тех случаях, когда вам необходим больший контроль над командами подпрограмм, можно задать retn, для того чтобы возврат всегда был локальным. Или используйте retf, для того чтобы всегда получался 32-битовый возврат. Когда вы так поступаете, то на вас лежит ответственность за правильное использование команды call. Предварите адрес подпрограммы обозначением near или far по необходимости:

```
call near Subroutine          ; Должна заканчиваться retn
call far Subroutine           , Должна заканчиваться retf
```

Чтобы сделать это по-иному, вы можете использовать команду retcode в Turbo Assembler 2.0 или более поздних версиях. Эта команда автоматически выбирает локальный или дальний возврат, основываясь на текущей модели памяти.

Совет

Ассемблируите несколько тестовых программ и изучите их в Turbo Debugger, для того чтобы проверить, вставляет ли retcode ожидаемые команды возврата

Прерывания с 32-битовым сегментом кода

Когда 16-битовый сегмент кода используется вместе с процедурами обработки прерывания, Turbo Assembler обычно выбирает команду возврата из прерывания, основанную на текущем размере сегмента кода. Это оказывает влияние на размер значений регистров, восстанавливаемых из стека. Для 32-битовых сегментов регистры двойного слова выталкиваются; для 16-битовых сегментов регистры слова выталкиваются.

Обычно команды, используемые по умолчанию, — именно те, которые вам надо. Если же тем не менее вы хотите заставить ассемблер восстанавливать 16-битовые регистры слова в 32-битовом сегменте кода, используйте команду iretw вместо iret. Если вы хотите восстанавливать 32-битовые расширенные регистры в 16-битовом сегменте кода, применяйте команду iretd.

Еще одни команды проталкивания

Еще один набор команд позволяет делать выбор между 16- и 32-битовыми командами pusha, popa, pushf и popf (см. ссылки в главе 16). Обычно эти команды проталкивают и выталкивают 16-битовые регистры и флаги в 16-битовом сегменте кода и 32-битовые расширенные регистры и флаги в 32-битовом сегменте кода.

Другие формы этих команд всегда проталкивают определенные регистры, независимо от размера сегмента кода. Используйте pushaw, popaw, pushfw и popfw для сохранения и восстановления 16-битовых регистров и флагов, а pushad, popad, pushfd и popfd — 32-битовых регистров и флагов.

Замечание

В Руководстве пользователя по Turbo Assembler эти альтернативные команды документированы некорректно (там даже не указаны формы команд двоиного слова) Предыдущая информация основана на тестировании программ — вы должны использовать Turbo Debugger для проверки того, правильные ли команды вставляются в ваши программы

Команды сдвига

80386 и более поздние процессоры обеспечивают альтернативную форму команд ротации и сдвига rcl, rcr, rol, ror, shl, shr, sal и sar. Например, для сдвига содержимого аккумулятора ax на три позиции влево вы можете использовать команду

```
shl  ax, 3
```

Процессор 8086, однако, может сдвигать только одну битовую позицию величины за раз, когда используется константа для задания значения сдвига. Применяя код 8086 (вставьте директиву P8086 в вашу программу), вы должны написать три отдельные команды для выполнения предыдущей операции:

```
shl  ax, 1
shl  ax, 1
shl  ax, 1
```

Вы можете использовать новую форму в программах 8086, Turbo Assembler заменит сдвиг на константную величину, большую, чем 1, соответствующим количеством команд сдвига в случае, когда генерация кода 8086 включена.

Быстрое умножение

Программисты на языке ассемблера очень гордятся своими находками наиболее эффективных методов программирования различных операций. Часто можно умножить две величины быстрее, если использовать комбинации сдвигов и других логических команд вместо команды imul (целочисленное умножение). (Найдите ее в справочнике по функциям, если вы не знакомы с ней.)

Погрузившись в сложности написания кода для различных процессоров, от 8086 до 80386, становится вдвойне тяжело найти наилучшую последовательность команд для умножения. Вот почему в Turbo Assembler 3.0 введена новая псевдокоманда FASTIMUL, которая генерирует наиболее эффективные команды для умножения для всех процессоров.

Несколько примеров показывают, как воспользоваться преимуществами этой новой команды. Синтаксис FASTIMUL такой:

```
FASTIMUL destination_reg, source_r/m, value
```

Первый аргумент должен быть регистром назначения — тем местом, где вы будете сохранять результат умножения. Второй аргумент может быть регистром или ссылкой на переменную памяти. Третий аргумент должен быть символьной величиной. На месте FASTIMUL Turbo Assembler генерирует одну или несколько команд, которые умножают source на число value и сохраняют результат в destination. Вы можете использовать 32-битовые регистры с соответствующими процессорами, такими как 80386 и 80486.

FASTIMUL обманчиво проста в использовании, но результаты вас могут поразить. В следующей строке, например, bx умножается на 4 и результат сохраняется в ax:

```
FASTIMUL ax, bx, 4
```

Из-за того, что это умножение эффективнее выполняется с использованием команды левого сдвига, Turbo Assembler на месте FASTIMUL пишет следующие команды:

```
shl  bx, 02
mov  ax, bx
```

Аналогично, с соответствующим процессором, можно умножать 32-битовые регистры:

```
P386
FASTIMUL eax, ebx, 4
```

Вместо команды FASTIMUL Turbo Assembler генерирует следующий 32-битовый код:

```
shl  ebx, 02
mov  eax, ebx
```

Спецификирование 16-битовой модели процессора, такого как 8086, генерирует другую последовательность Рассмотрим то же самое умножение с использованием директивы P8086:

```
P8086
FASTIMUL ax, bx, 4
```

В этом случае FASTIMUL генерирует три команды, так как сдвиги в 8086 могут смещать только одну битовую позицию за раз:

```
shl  bx, 1
shl  bx, 1
mov  ax, bx
```

Предыдущие примеры описали толику того, что может делать FASTIMUL. Менее очевидная оптимизация возникает, когда умножение производится на величину, не являющуюся степенью двух. Рассмотрим такую команду, при условии генерации кода 8086:

```
P8086
FASTIMUL ax, bx, 3
```

Вместо команды FASTIMUL Turbo Assembler генерирует следующие три команды:

```
mov  ax, bx
shl  bx, 1
add  ax, bx
```

Требуется некоторое умственное усилие для того, чтобы проверить, что эти команды действительно умножают bx на 3, и более чем легкий просмотр — для осознания того, что код, полученный в результате, является более эффективным решением. Многие программисты на языке ассемблера, вероятно, написали бы следующий код:

```
mov  al, 16
mov  bl, 3
imul bl
```

Как общее правило, можно отметить, что *любая* замена команды imul, которая использует непосредственную величину (3 в этом случае), вероятно, лучше из-за многочисленных циклов CPU, которые эта "времязатратная" команда потребляет. В некоторых случаях, особенно для таких процессоров, как 80386 и 80486, imul может все еще быть лучшим выбором, как показывает следующий пример:

```
P386
FASTIMUL eax, ebx, 123456
```

Умножение ebx на постоянную величину 123456 лучше всего сделать командой imul, которую Turbo Assembler и генерирует для предыдущей команды:

```
imul eax, ebx, 0001E240
```

Аналогично, с установленной генерацией кода 80386 или более поздних моделей, ассемблер использует команду imul для непростых символьных операндов (таких, как 1234 в этом 16-битовом умножении):

```
P386
FASTIMUL ax, bx, 1234
```

На месте этой FASTIMUL ассемблер генерирует команду imul:

```
imul ax, bx, 04D2
```

Эта команда, однако, не разрешена для процессора 8086, который имеет только ограниченную форму imul. Когда код генерируется для 8086 процессора, вы можете использовать команды FASTIMUL не только ради повышения эффективности, но и для улучшения переносимости. Например, если вы специфицируете директиву P8086 для того же умножения, что и раньше,

```
P8086
FASTIMUL ax, bx, 1234
```

ассемблер сгенерирует следующую последовательность команд сдвигов и добавлений:

```
shl  bx, 1
mov  ax, bx
shl  bx, 1
shl  bx, 1
shl  bx, 1
add  ax, bx
shl  bx, 1
shl  bx, 1
add  ax, bx
shl  bx, 1
add  ax, bx
shl  bx, 1
shl  bx, 1
shl  bx, 1
add  ax, bx
```

Несмотря ни на что, это и работает, и делает возможным написание переносимых команд умножения для процессоров 80×86. Вы должны быть в курсе, что FASTIMUL может в некоторых случаях увеличивать ваш файл кода в размерах.

Резюме

Двоично-кодированные десятичные величины позволяют хранить 20-цифровые числа в формате BCD, из которого и в который легко преобразовывать ASCII-символы. Упакованный BCD хранит 2 цифры на байт, неупакованный — 1 цифру на каждый байт. Директива dt создает 20-цифровые переменные BCD. Хотя аналогичной директивы для создания неупакованных BCD-переменных нет, db является адекватной заменой.

Команды aaa и aas выравнивают двоичные результаты после сложения и вычитания неупакованных BCD-величин обратно к неупакованному формату BCD. Команды aad и aam осуществляют преобразования между двоичными и неупакованными BCD-величинами. Несмотря на сокращения в именах этих двух команд, не нужно связывать их с делением и умножением. Преобразование неупакованных BCD в ASCII и из них использует только простые команды and или or, так как схема кодировки ASCII использует цифры 0–9. Команды daa и das выравнивают двоичные результаты после сложения и вычитания упакованных BCD-величин обратно к упакованному формату BCD.

Общие переменные, которым нельзя приписать начальные значения, декларируются директивой COMM. Аналогично общим переменным, глобальные переменные определенные директивой GLOBAL, могут иметь начальные значения и быть поделены между различными модулями. GLOBAL устраняет необходимость объявлять переменные как PUBLIC в одном модуле и как EXTRN — в других. Просто поместите все ваши глобальные объявления в один или два файла и ассемблируйте и скомпонуйте выше приложение, используя директивы INCLUDE для того, чтобы загрузить глобальные определения в конкретные модули. В больших проектах вы, может быть, также захотите задать определенный библиотечный файл командой INCLUDE, что упростит компоновку.

Используйте xlat для перевода байтовых индексных величин в байты, хранящиеся в табличной форме в памяти. Таким образом, можно сэкономить время, так как поиск значений в памяти обычно быстрее, чем выполнение сложных вычислений. Типичное применение xlat — это перевод кодов ASCII в другие символы. Эта команда также может быть использована (часто вместе с lea) для выбора значений из двумерных матриц.

Упрощенные модели памяти требуют внимания ко многим деталям, которые вы должны определять сами, когда сегменты определяются "жестким образом" директивой SEGMENT. Типичная .EXE-программа требует по меньшей мере три таких сегмента — один для данных, один для кода и один для стека. Различные правила и соглашения об именах меняют способ, которым Turbo Assembler и компоновщик организуют вашу программу и сегменты в памяти, объединяя некоторые сегменты в целое и оставляя другие отдельными.

Когда вы объявляете свои собственные сегменты, вы должны инициализировать регистры сегментов, все время помня о том, что такие присваивания возникают во время работы программы. Используйте директиву ASSUME, которая выполняется на этапе ассемблирования, "говоря" Turbo Assembler о том, какие присваивания регистров сегментов делает ваша программа. Другая, связанная с ней, команда GROUP собирает несколько сегментов разных названий и, возможно, разных видов в один большой сегмент до 64K длиной.

Объявляя сегмент *комбинированного* типа через At, вы создаете призрачный сегмент, который перекрывает переменные или код, уже существующий в памяти, когда ваша программа работает. Это дает вам возможность читать и писать переменные, вызывать или переходить на процедуры, которые принадлежат другим процессам — таким, как ROM BIOS.

Когда вам надо дополнительное пространство для переменных, вы можете присоединить один или более дальних сегментов данных к упрощенной модели памяти. Дальние сегменты данных могут быть инициализированы или деинициализированы с произвольными именами после директив FARDATA и UFARDATA, позволяя резервировать много фрагментов памяти по 64K для использования даже в программах "малой" модели памяти.

В 80386 и более поздних процессорах к базовому набору 8086 сокращений добавлены некоторые новые команды. 80386 добавляет еще больше команд плюс расширенные 32-битовые регистры, флаги и еще два сегментных регистра. Хотя Turbo Assembler может ассемблировать код для этих процессоров на любой машине, полученные в результате программы могут выполняться только на соответствующих компьютерах.

Новая директива VERSION делает возможным ассемблирование программ, написанных для всех версий Turbo Assembler. VERSION заменяет старые опции, такие как QUIRKS.

Используйте ENUM для создания перечисляемых типов данных для серий символов, которые могут быть представлены символьно. Символы похожи на индивидуальные присваивания, но ассемблер может предостеречь от некоторых ошибок (например, от сохранения слова в байтовой переменной).

Директива SMART разрешает ассемблеру заменять команды на более эффективные во многих случаях. Используйте NOSMART для выключения генерации smart-кода.

Другие новые команды и способы оптимизации в Turbo Assembler 4.0, как объясняется в этой главе, помогают вам написать более эффективный код.

Упражнения

11.1. Сколько чисел было бы в гипотетической упакованной 4-байтовой величине BCD? Сколько чисел было бы в гипотетической неупакованной 6-байтовой величине BCD? Сколько чисел BCD-директива dt позволяет вам определять в некоторой величине?

11.2. Напишите код для конвертации упакованного байта BCD в регистре al в двоичное представление в регистр ax.

11.3. Какие директивы GLOBAL вам нужны для разделения следующих переменных среди нескольких модулей?

```
string  db      'This is an ASCII string,' 0
count   dw      0
BCD     dt      123456789
```

11.4. Используя xlat, напишите код для перевода значения в cl в следующие величины (равные кубу 0–6):

```
cl      cl*cl*cl
------------------
0       0
1       1
2       8
3       27
4       64
5       125
6       216
```

11.5. Что делает ASSUME?

11.6. Объявите сегмент данных по имени MoreData, выровненный по следующей наибольшей 256-байтовой странице, и скомбинируйте другими сегментами в класс 'DATA'. Сохраните переменную в слово по имени MyWord в вашем сегменте и покажите код, необходимый для загрузки ax значением MyWord.

11.7. Что делает GROUP? Как бы вы использовали GROUP для ссылки на четыре сегмента —SomeData, MoreData, TableSeg и StringSeg?

11.8. Байт PC kbFlag (флаг клавиатуры) хранится по смещению 017h в сегменте данных BIOS 040h. Бит 6 этой величины указывает на то, что клавиша CapsLock включена (1) (или нет (0)). Напишите программу для отображения текущей установки этой клавиши. Используйте в вашем ответе абсолютный сегмент данных At.

11.9. Напишите оболочку процедуры обработки прерывания 80286, которая сохраняет и восстанавливает все регистры общего назначения.

11.10. Напишите эквивалентный код 8086, который дублирует следующие команды 80386:

```
bt      dx, 3
btc     dx, 12
btr     dx, 8
bts     dx, 1
```

Задания

11.1. Добавьте процедуры умножения и деления в BCD.ASM. Подсказка. распакуйте упакованные переменные BCD и используйте aad и aam для перевода значений в двоичную форму и из нее.

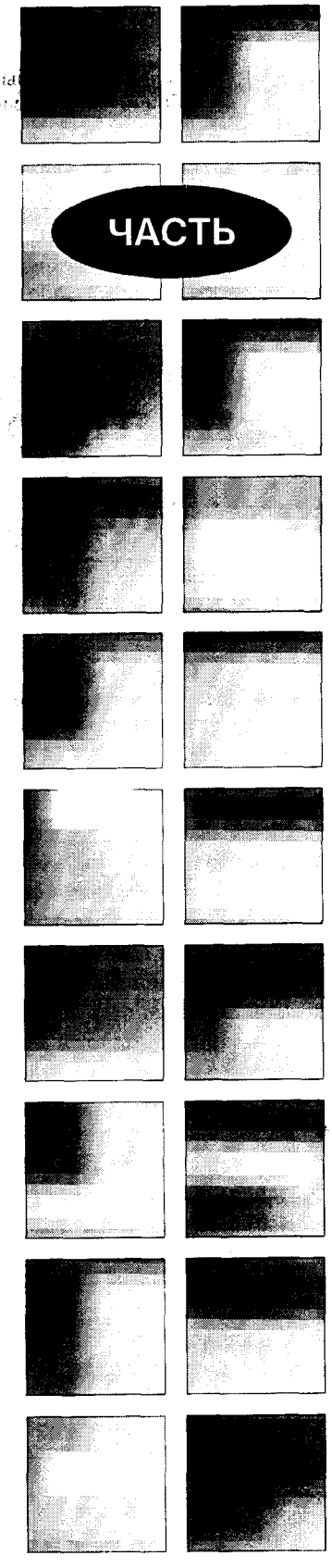
11.2. Напишите команды форматирования строк ASCIIZ, для того чтобы добавить десятичные точки и знаки доллара и (по желанию) удалить ведущие нули из упакованных величин BCD. Подсказка: используйте процедуру BCDToASCII из BCD.ASM, для того чтобы выполнить грубое преобразование BCD в цифры ASCII, затем используйте процедуру STRING, для того чтобы вставить и удалить нужные символы

11.3. Применяя технический справочник по PC, напишите include-файл, в котором определялись бы абсолютные (At) сегменты данных для всех или почти всех переменных ROM BIOS.

11.4. Разработайте набор макросов для ассемблирования программ с командами 8086, 80286, 800386 (и более поздними), основанный на условном присваивании символов в начале модуля. Продублируйте, насколько сможете, специальные команды 80286 и 80386, используя только команды 8086

11.5. Найдите примеры программ в этой книге, которые можно улучшить ассемблированием со специальными командами 80286 и 80386. Используйте свои макросы из задания 11.4, для того чтобы реассемблировать код и запустить его в качестве теста времени исполнения для проверки ваших предположений.

11.6. Напишите, с использованием команды xlat, модуль, который позволит перепрограммировать нажатия различных функциональных клавиш в нажатия других клавиш. Сконструируйте модуль так, чтобы вы могли перепрограммировать управляющие клавиши в программе.



ЧАСТЬ II

Прикладное программирование

ГЛАВА 12

Использование языка ассемблера при программировании на Pascal

- Возможность усовершенствования
- Выявление критичных участков программы (критичного кода)
- Преобразование программы на Pascal к ассемблерному коду
- Внешние процедуры и функции
- Модель памяти Pascal
- Вызов процедур Pascal
- Демонстрационный пример
- Адресация данных кодового сегмента
- Адресация констант с определенным типом
- Вызов функций Pascal
- Адресация переменных Pascal
- Вызов внешних функций
- Передача параметров
- Использование модели памяти TPASCAL
- Использование директивы ARG
- Написание внешних строчных функций
- Проблема быстродействия
- Резюме
- Упражнения
- Задания

Возможность усовершенствования

В идеальном мире программирования компиляторы с языков высокого уровня должны генерировать самый быстрый, самый компактный и наилучший машинный код для любой программы. Если бы это было возможно, то вполне вероятно, что в данной главе, а может быть, и в самой книге, не было бы никакой необходимости. Но это невозможно. Несмотря на множество усовершенствований при разработке компиляторов, ни один язык высокого уровня пока не в состоянии продублировать краткий, быстрый и изящный код, написанный опытным специалистом по ассемблеру.

В чем же дело? А в том, что компиляторы обобщают решаемые ими задачи. В Pascal существует только один способ написания цикла FOR. В то же время с помощью ассемблера этот цикл можно организовать десятками способов. Чтобы компилятор выбирал идеально подходящие в каждой конкретной ситуации методы реализации алгоритма и при этом учитывал их влияние на остальные части программы — он должен обладать интеллектом гения, вкусом художника и интуицией предсказателя. Сегодняшние компиляторы с языков высокого уровня достаточно сообразительны, но не до такой степени.

Среди всех существующих компиляторов с Pascal наиболее приблизился к идеалу Turbo Pascal. Порождаемый им машинный код работает достаточно быстро, занимает на диске немного места и в большинстве случаев может использоваться без каких-либо модификаций. Однако несмотря на все достоинства Turbo Pascal, все равно остается место для улучшений, благодаря чему даже небольшие участки программы, переписанные на ассемблере, могут значительно повысить скорость работы программы и уменьшить размер выполняемого файла. Кроме того, добавив ассемблер к Pascal, можно облегчить доступ к регистрам аппаратных средств и решение других задач низкого уровня, таких как запись символов непосредственно в видеопамять.

Замечание

В этой главе подразумевается, что вы достаточно хорошо знаете Pascal, а также как установить и запустить используемый вами компилятор. Примеры программ, приведенные в данной главе, можно использовать с большинством версий Turbo Assembler, Turbo Pascal и Borland Pascal.

Очень важно не только знать, как подключать язык ассемблера к Pascal, но и точно определять, когда это делать, а когда нет. При написании участков программы на языке ассемблера всегда помните, что вам придется переписать эти же самые участки снова, если позже возникнет необходимость перенести их на компьютер, базирующийся на процессоре, отличном от 8086. Чтобы избавить себя в будущем от головной боли, руководствуйтесь следующими простыми принципами:

- Пишите на ассемблере только критичный код
- Отлаживайте процедуры и функции Pascal и только затем переписывайте их на ассемблере
- Сохраняйте резервную копию преобразованных процедур и функций на Pascal, чтобы облегчить при необходимости возврат к исходному коду на Pascal.

Критичный код (critical code) — термин, применяемый для описания тех фрагментов программы, на которые приходится наиболее значительная часть общего времени выполнения программы. В большинстве программ существует несколько процедур, функций и циклов, выполняющихся чаще, чем все остальное. Поскольку эти критичные процедуры вносят основной вклад в общее время исполнения программы, запись соответствующих операторов на языке ассемблера может существенно повысить производительность программы. Действительно, многие специалисты сходятся во мнении, что большинство программ тратит до 90% общего времени работы, выполняя около 10% операторов всей программы; следовательно, небольшое улучшение в участке с критичным кодом может оказать решающее влияние на общее быстродействие.

Вместе с тем написание оставшихся 90% операторов на языке ассемблера может дать менее ощутимые результаты. По существу, общий эффект может даже оказаться равным нулю. Например, вероятно, не стоит переписывать простое приглашение, позволяющее кому-либо набрать имя файла. Люди могут печатать только с ограниченной скоростью, и, даже если программа работает эффективнее, полученный выигрыш будет нулевым. Не тратьте напрасно время, переписывая части программы, которые уже работают с достаточной скоростью.

Выявление критичных участков программы (критичного кода)

Выявление критичных участков программы не всегда является простой задачей. В некоторых случаях ваш опыт программиста подскажет, какие разделы должны быть переделаны. В частности, вы можете знать, что определенные операции вывода изображения на экран не должны сопровождаться потрескиванием, писком и другими шумами, на которые только способен компьютер. В других случаях ваш опыт программирования на Pascal подскажет, что определенные действия (например, прямой доступ к аппаратным портам), вероятно, будут исполняться быстрее в ассемблерном коде.

В других случаях выбор не всегда столь очевиден и вам может потребоваться *программа-профайлер* (profiler), такая как Turbo Profiler, которая поставляется с некоторыми версиями Turbo и Borland Pascal, чтобы она помогла вам определить критичные участки. Профайлер отслеживает прохождение программы и создает статистические таблицы для выявления операторов, исполняющихся чаще других. После обработки программы с помощью профайлера вы можете перепрограммировать эти части на языке ассемблера, оставляя остальную менее критичную программу на Pascal. Такой подход к оптимизации программы помогает уменьшить затраты времени на программирование и сулит существенное увеличение производительности программы.

Однако даже с помощью профайлера легко утратить из виду конечную цель и в результате закончить пересмотром слишком большой части программы. Помните, что вашей целью является обнаружение критичных участков программы и их преобразование к ассемблерному коду. При этом вы должны непрерывно проверять и перепроверять программу, оценивая результаты своей работы. Чтобы облегчить свой труд:

- Не применяйте профайлер к программам, использующим оверлейную структуру
- Используйте разные значения частоты опроса
- Оптимизируйте большие программы по частям.

В больших программах, которые используют оверлейную структуру для резервирования памяти путем загрузки независимых частей в одни и те же области оперативной памяти, вероятно, лучше оптимизировать оверлейные модули так, как если бы они были отдельными программами. Большинство программистов, разрабатывающих большие системы программного обеспечения, предпочитают сначала писать оверлейные модули как отдельные программы, вместо того, чтобы тратить время, компилируя и объединяя уже законченные куски. При таком подходе завершающий программный код создается на одном из последних этапов, непосредственно перед окончанием работы. Следование такому подходу упрощает оптимизацию. Применяя профайлер, вы имеете возможность облегчить работу, исследуя отдельные оверлейные модули перед их объединением в конечную программу. Возможно, в своих будущих программах вы захотите применить этот метод.

Частота опроса (sample rate) — величина, показывающая, насколько часто профайлер прерывает работу выполняющейся программы для проведения замеров. Внутренний таймер IBM PC, генерирующий 18,2 "тика" в секунду, слишком медленный, чтобы обеспечить приемлемое профилирование, так как за 1/18 секунды выполняется огромное количество инструкций — это практически целая вечность для компьютера. По этой причине некоторые профайлеры перепрограммируют внутренние часы, чтобы повысить частоту опроса от 40 до 30000 выборок в секунду. Подобрать правильное значение частоты опроса нелегко, поэтому целесообразно профилировать одну и ту же программу, используя по меньшей мере три частоты опроса — например 500, 1000 и 2000.

Никогда не пытайтесь профилировать и оптимизировать большую программу всю целиком. Если ваша программа на Pascal превышает 10000 строк, вам потребуется составить план ее оптимизации по частям. Одной из возможностей является профилирование оверлейных модулей в отдельности. Или же можно воспользоваться такими средствами профайлера, как помещение в исходный код специальных команд, которые ограничат работу профайлера отдельными областями.

Преобразование программы на Pascal к ассемблерному коду

После выявления критичного участка программы на Pascal можно приступать к преобразованию операторов Pascal на язык ассемблера. При этом вы можете пользоваться тремя способами:

- Операторы *Inline*
- Процедуры и функции *Inline*
- Внешние процедуры и функции

Операторы *Inline* по существу являются командами компилятору Pascal для подключения машинного кода непосредственно к коду, обычно генерируемому компилятором. Предположим, что вам необходимо отключить прерывания. Поскольку в Pascal отсутствуют операторы, способные выполнить это действие непосредственно, оператор *Inline* вставляет в откомпилированную программу код, вызывающий выполнение процессором 8086 инструкции cli:

```
Inline ($FA); {cli -- отключение прерываний}
{Операторы, которые должны выполняться с отключенными прерываниями}
Inline ($FB); {sti - снова подключает прерывания}
```

Обычно операторы Inline удобно использовать при вставке ограниченного числа инструкций на машинном коде. Поскольку приходится использовать цифровые выражения машинных кодов, операторы Inline не применяются для преобразования больших участков программ с Pascal на ассемблер.

Замечание

Рациональным способом получения двоичных машинных кодов для различных инструкций представляется написание небольших ассемблерных программ с последующим выполнением ассемблерного кода с помощью Turbo Debugger. Затем необходимо воспользоваться командой View/CPU и скопировать нужные байты в операторы Inline.

Другим способом является использование процедур или функций Inline. Эти средства по своему действию в значительной степени сходны с макросами ассемблера, помещая машинный код в том месте программы, где появляется имя процедуры или функции. Ранее в программе на Pascal вам уже приходилось объявлять процедуры, подобные следующим:

```
PROCEDURE ClrInt; Inline($FA);
PROCEDURE SetInt; Inline($FB);
```

Функции объявляются аналогично. Этим достигается то, что байты операторов Inline, содержащие машинные коды, связываются с идентификаторами процедур ClrInt и SetInt. Позже, при использовании этих идентификаторов, компилятор Pascal вставит машинные коды непосредственно в скомпилированную программу.

Например, можно использовать следующие операторы:

```
ClrInt;
Writeln('Прерывания отключены');
SetInt;
Writeln('Прерывания включены')
```

Преимуществом данного метода является то, что он маскирует язык машинных кодов. Хотя он и появляется тогда, когда происходит обращение к процедурам ClrInt и SetInt, в действительности компилятор помещает машинные коды непосредственно в кодовую последовательность. Это увеличивает переносимость программы путем помещения машинных кодов только в одном месте исходного текста программы. Для другой системы легко можно преобразовать код замещением процедур Inline соответствующими процедурами на Pascal. Подобный подход гораздо предпочтительнее, чем отыскание операторов Inline, разбросанных по всей программе.

Замечание

Предшествующие примеры аналогичны тем, что приведены в моей книге „Mastering Turbo Pascal", где применение ассемблера в программах на Pascal рассматривается подробнее.

Внешние процедуры и функции

Несмотря на то что на это требуется больше усилий, написание внешних процедур и функций, исполняющихся отдельно от исходной программы на Pascal, зачастую является наиболее рациональным подходом. Тому есть несколько причин:

- Программы на Pascal сохраняют высокую степень переносимости
- Внешние подпрограммы можно отлаживать по отдельности
- Внешние подпрограммы можно использовать совместно с программами на других языках программирования.

Написание программ сначала на Pascal, с последующим избирательным преобразованием отдельных процедур и функций, повышает переносимость этих программ. После оптимизации, если требуется перенести программу на другой компьютер, например на Macintosh с процессором 68000, относительно несложно заменить оптимизированный модуль с ассемблерным кодом исходной программой на Pascal, предусмотрительно сохраненной вами на диске. Затем, после того как вы убедитесь, что программа работает надлежащим образом, можно заняться оптимизацией, применяя родной язык данного компьютера.

Еще одним преимуществом использования внешних подпрограмм, написанных на языке ассемблера, является то, что при этом упрощается отладка. В большинстве случаев можно написать простые тестовые программы (либо на Pascal, либо на ассемблере) для помещения требуемого машинного кода между шагами отлаживаемой основной программы. Причем один и тот же код может использоваться в программах, написанных на других языках программирования, таких как C или BASIC. Многие программисты создают библиотеки подобных подпрограмм, готовых к использованию со своими программами, написанными на языках высокого уровня.

Замечание

Соглашения о вызове подпрограмм и модели памяти отличаются в различных языках программирования, поэтому не всегда можно использовать одни и те же внешние подпрограммы без каких-либо изменений. Но даже в этом случае внешние процедуры, содержащие ассемблерный код, легче преобразовать, чем непосредственные вставки Inline.

Вызов внешних подпрограмм из Pascal

Для использования в программе на Pascal ассемблерных процедур необходимо выполнить следующее:

- Написать ассемблерную процедуру PROC типа NEAR или FAR

- Объявить процедуру PROC PUBLIC, чтобы сделать метку внешней процедуры экспортируемой для компилятора Pascal
- Воспользоваться командой компилятора Pascal {$L <file>} для загрузки с диска соответствующего модуля .OBJ в процессе компиляции
- Объявить данную процедуру в Pascal как EXTERNAL.

Процедуры на ассемблере обладают таким же форматом, как и другие отдельные модули объектного кода, описанные в этой книге. Будьте внимательны при объявлении процедур NEAR или FAR, чтобы компилятор "знал", какой вызов генерировать при обращении к этой процедуре — длинный (в другой сегмент) или короткий (в этот же сегмент). Кроме того, чтобы Pascal мог определить код начала процедуры в модуле .OBJ, необходимо поместить в оператор PUBLIC имя процедуры. Общий формат имеет следующий вид:

```
PUBLIC    ProcName
PROC ProcName NEAR
     ;----- Код процедуры
     ret          ; Возврат управления в точку вызова
ENDP ProcName
```

Применительно к процедурам FAR (в другой сегмент) необходимо в приведенном примере заменить NEAR на FAR. Для загрузки соответствующего объектного кода во время компиляции в тексте программы на Pascal следует использовать команду компилятора {$L <file>}. Кроме того, необходимо объявить процедуру в Pascal как EXTERNAL, чтобы сообщить компилятору имя процедуры, а также имена, количество и типы всех необходимых параметров. Так, для процедуры, имеющей имя MYCODE.OBJ, нужно использовать следующие строки:

```
{$L MYCODE OBJ}
PROCEDURE ProcName; EXTERNAL;
```

В этом примере ProcName не имеет ни одного параметра. Если бы они имелись, пришлось бы их здесь объявить. (Вопрос передачи параметров будет освещен в этой главе несколько позже). По завершении указанных шагов можно вызывать внешнюю процедуру. Чтобы осуществить это, достаточно использовать имя процедуры (в данном примере — ProcName) в качестве оператора — точно так же, как при вызове остальных процедур Pascal. Можно также объявить внешние функции, как демонстрируется в дальнейших примерах. Прочтя директиву {$L}, Turbo Pascal автоматически вставляет коды внешней процедуры из файла .OBJ в выходной файл EXE на диске (или же в память, если вы работаете с интегрированной средой). Все, что вам остается сделать, — это откомпилировать программу, так как нет необходимости предпринимать какие-либо дополнительные промежуточные шаги.

Модель памяти Pascal

Несмотря на то что выше были описаны все элементы, необходимые для написания внешних процедур на ассемблере для включения их в программы на Pascal, один важный момент был упущен: формат исходного текста ассемблерной программы. К сожалению, формат, использованный в большинстве программ в этой книге, не подойдет, так как Pascal обладает своим собственным способом организации памяти. Поэтому при написании внешней процедуры на ассемблере необходимо воспользоваться одной из двух, отличных одна от другой, моделей, чтобы компилятор Pascal имел возможность соединить соответствующий файл объектного кода с компилируемыми операторами Pascal.

Программа 12.1, PASSHELL.ASM, является шаблоном, не выполняющим никаких действий, в который вы можете вставить реальные коды и данные для использования в своих внешних модулях. Как легко заметить, в этом шаблоне кодовые сегменты и сегменты данных объявляются более сложным образом, а не с использованием упрощенных моделей памяти, как это делалось в большинстве других примеров этой книги. Обратитесь к тексту программы, и я объясню, чем это вызвано.

Программа 12.1. PASSHELL.ASM

```
1:   %TITLE "Turbo Pascal .OBJ shell -- Copyright (c) 1989,1995 -- by Tom Swan"
2:
3:   IDEAL
4:
5:   SEGMENT DATA word public
6:
7:   ;----- Поместите здесь объявление данных EXTRN
8:
9:   ;----- Поместите здесь статические (неинициализированные) переменные
10:
11:  ENDS DATA
```

```
12:
13:
14:  SEGMENT CODE byte public
15:
16:  ASSUME cs:CODE, ds:DATA
17:
18:  ;----- Поместите здесь объявления кода PUBLIC
19:
20:  ;----- Поместите здесь объявления кода EXTRN
21:
22:
23:  %NEWPAGE
24:  ;---------------------------------------------------------
25:  ; ПРОЦЕДУРА ProcName( <параметры> );
26:  ;---------------------------------------------------------
27:  PROC ProcName NEAR
28:       ret                              ; Возврат в точку вызова
29:  ENDP ProcName
30:
31:  %NEWPAGE
32:  ;---------------------------------------------------------
33:  ; ФУНКЦИЯN FuncName( <параметры> ) . <тип>;
34:  ;---------------------------------------------------------
35:  PROC FuncName NEAR
36:       ret                              , Возврат в точку вызова
37:  ENDP FuncName
38:
39:  ENDS CODE
40:
41:  END                                   ; Конец модуля
```

Сегмент DATA PASSHELL

В программе PASSHELL сегменты данных и кодов объявляются "жестким образом" с применением директив SEGMENT вместо использования упрощенной модели памяти в директиве MODEL. В строках 5–11 объявляется сегмент общедоступных данных, выровненный по адресу четного слова, и таким образом Pascal может идентифицировать начало и конец сегмента.

Замечание

Выравнивание сегмента данных по четным адресам может увеличить скорость доступа к 16-битовым данным. Указание выравнивания на границу слова в директиве SEGMENT заставляет первую переменную в сегменте размещаться по четному адресу с пропуском, при необходимости, одного байта. Однако если вы объявляете в сегменте данных какие-то переменные с размерами, кратными байту, из-за выравнивания на границу слова вы можете „испортить" все последующие переменные. Во избежание этого за каждой директивой объявления одного байта db добавляют фиктивную однобайтовую переменную, чтобы обеспечить выравнивание на границу слова для всех переменных. Однако все это имеет значение только для чрезвычайно критичного по времени кода; для большинства программ можно не обращать внимания на эти тонкости.

Внутри сегмента данных можно объявлять переменные совершенно так же, как в любом другом ассемблерном модуле. Однако существует одно важное отличие: все переменные должны быть *неинициализированными* (*uninitialized*). Иначе говоря, следующие объявления не будут работать:

```
astring          db 15, 'Пример строки'
counter          dw 100h
asciiEsc         db 27
```

Turbo Assembler воспринимает эти объявления, но Turbo Pascal не распознает инициализированные данные. Это происходит потому, что глобальный сегмент данных в откомпилированной программе на Pascal является фантомом, существующим только во время выполнения программы; поэтому вы не можете объявить во внешнем модуле заранее инициализированные переменные. Вместо этого нужно использовать объявления, подобные следующим:

```
astring          db 16 DUP (?)
counter          dw ?
asciiEsc         db ?
```

Эти команды распределяют место для 16-символьной строки, для слова и байта. В ходе выполнения программы переменные не будут иметь никаких конкретных значений и вам самим придется позаботиться о том, как их ини-

циализировать. Кроме того, такие переменные ограничены использованием только в ассемблерных модулях — нельзя переносить в Pascal метки переменных. Причиной такого ограничения является то, что в Pascal отсутствует директива EXTERNAL, применимая к переменным. Ключевое слово EXTERNAL в Pascal действует только применительно к процедурам и функциям. (Имеется способ обойти эту проблему, используя методику, описанную в данной главе несколько позже.)

Использование статических переменных

Вы можете в ассемблерном модуле воспользоваться и инициализированными переменными, однако этот метод требует небольшой помощи со стороны компилятора Pascal. Для этого в исходном тексте ассемблерного модуля не надо использовать директивы db и dw, а на Pascal объявите переменные *типизированными константами (typed constants)*. Например, в программе на Pascal могут использоваться следующие строки:

```
CONST astring  string[ 13 ] = 'Пример строки',
 counter  integer = $100,
 asciiEsc  byte = 27,
```

Теперь в сегменте данных ассемблерного модуля можно импортировать эти константы с помощью директивы EXTRN, сообщающей Turbo Assembler, что действительные адреса реальных данных будут предоставлены позже во время компиляции:

```
SEGMENT DATA word public
EXTRN astring  BYTE, counter  WORD  asciiEsc  BYTE
ENDS DATA
```

После этого можно пользоваться переменными astring, counter и asciiEsc, как будто они были определены непосредственно в ассемблерном модуле. Обратите внимание, что строка в Pascal соответствует на ассемблере указателю байта.

Сегмент CODE в PASSHELL

В строках 14–39 в программе PASSHELL объявляется сегмент модуля CODE, выровненный на любой адрес (байт), и задается как Public для компилятора Pascal. В строке 16 используется директива ASSUME, чтобы сообщить Turbo Assembler о взаимосвязанности сегментных регистров cs и ds с сегментами модуля. Pascal не накладывает никаких ограничений на регистр es, поэтому для него не требуется никаких объявлений. Если вы планируете адресовать сегмент данных es, то можете заменить строку 16 следующей:

```
ASSUME cs CODE, ds DATA, es DATA
```

Помните, что директива ASSUME просто сообщает ассемблеру об организации модуля — она не генерирует никакого кода и не обеспечивает действительной адресации сегментных регистров конкретным сегментам. Требуется особая внимательность при инициализации es, который не резервируется между обращениями к внутренним подпрограммам Pascal. Pascal инициализирует ds таким образом, чтобы адресовать глобальный сегмент данных, который может быть только один, длиной до 64K. Следовательно, вы не должны инициализировать ds в тексте своего модуля.

Замечание

Pascal заботится о том, чтобы распределить место для стека. Поэтому никогда не объявляйте область стека и не переназначайте ss в своих внешних модулях.

Вызов процедур Pascal

В строке 18 показано, где необходимо вставлять объявления PUBLIC. После ключевого слова PUBLIC поместите имена всех процедур модуля, которые вы хотите экспортировать в Pascal; вам не нужно просматривать каждую процедуру. Например, модуль может содержать локальную подпрограмму для индивидуального использования другими процедурами, содержащимися в модуле. Но каждое имя в объявлении PUBLIC должно соответствовать объявлению EXTERNAL процедуры или функции в тексте программы на Pascal. Кроме того, помните, что только код, но не данные, может быть объявлен для общего пользования.

Строка 20 показывает, где нужно помещать объявления EXTRN. Это относится к процедурам и функциям Pascal, которые вы хотите вызвать из своей программы на ассемблере. Предположим, у вас есть подпрограмма на Pascal с именем Pause, которая выводит на дисплей сообщение и ждет нажатия клавиши <Enter>:

```
PROCEDURE Pause;
BEGIN
 Writeln;
 Write('Для продолжения нажмите клавишу <Enter>...');
 Readln
END; { Pause }
```

Прежде чем экспортировать Pause из Pascal в ассемблерный модуль, вы должны быть уверены, что компилятор узнает имя процедуры до того, как загрузит соответствующий объектный код. Одним из путей достижения этого является объявление FORWARD процедуры Pause *перед* директивой {$L <file>}, которая загружает файл с диска. Если ассемблерный модуль назван ANYCODE.OBJ, в первых строках программы можно использовать следующие операторы Pascal:

```
PROCEDURE Pause; FORWARD;
{$L ANYCODE.OBJ}
```

Чтобы вызвать Pause из внешнего ассемблерного модуля, создайте сегмент CODE наподобие следующего:

```
SEGMENT CODE byte public
ASSUME cs:CODE, ds:DATA
EXTRN Pause:NEAR
PROC MyProc NEAR
 call Pause ; Вызов процедуры Pascal
 ret
ENDP MyProc
ENDS
```

Директива EXTRN сообщает Turbo Assembler, что Pause является ближней процедурой (в том же самом кодовом сегменте). Если это не так — например, если в тексте программы на Pascal вы используете директиву {$F+} для включения генерации дальнего кода или если процедура упоминается в разделе интерфейса блока — тогда вы должны объявить Pause как FAR. Само обращение к Pause ничем не отличается от обращений к другим подпрограммам на языке ассемблера. В этом примере, однако, отсутствуют параметры. Если бы они имелись, вам нужно было бы также передать параметры в соответствии с требованиями кода Pascal. Этим вопросом мы сейчас и займемся.

Тело кодового сегмента

В строках 24–37 содержатся пустые заготовки для внешних процедур и функций. Единственным отличием между процедурами и функциями является то, что функции возвращают в вызывающую программу значения, а процедуры — нет. (В Pascal функции используются в выражениях, в то время как процедуры вызываются по имени в операторах.)

Заключительная часть текста программы PASSHELL представлена строками 39–41. Поскольку в данном случае используется не упрощенная модель памяти, сегмент CODE должен заканчиваться директивой ENDS (строка 39). END в строке 41 сообщает ассемблеру, что эта строка является последней в исходном тексте. Можно не указывать метку точки входа, как это делается для самостоятельных ассемблерных .EXE-программ.

Демонстрационный пример

Программы 12.2 PASDEMO.ASM и 12.3 PASDEMO.PAS помогут прояснить многие вопросы относительно передачи кода и данных между модулями на Pascal и ассемблере. Пример немного "сумасшедший" — он не служит никакой другой цели, кроме демонстрации различных вопросов (обсуждаемых после текста программ). Помимо передачи параметров, программа иллюстрирует почти все возможные комбинации разделения кода и данных и послужит вам хорошим подспорьем при разработке собственных проектов. Чтобы загрузить и откомпилировать контрольные примеры, воспользуйтесь следующими командами:

```
tasm /zi pasdemo
tpc /v pasdemo
```

Если у вас Borland Pascal, замените tpc на bpc. Проделывайте это со всеми инструкциями данной главы, использующими tpc. Чтобы эти команды работали, необходимо инсталировать компилятор командной строки.

Опции /zi и /v подключают к PASDEMO.EXE информацию об отладке, чтобы Turbo Debugger мог отображать исходные кодовые строки как Pascal, так и ассемблера, вместе с загруженным и откомпилированным машинным кодом. Другой возможностью является создание файла MAKEFILE, содержащего следующие строки:

```
pasdemo.exe: pasdemo.obj pasdemo.pas
 tpc /v pasdemo

pasdemo.obj: pasdemo.asm
 tasm /zi pasdemo
```

После сохранения этого текста на диске в файле MAKEFILE наберите на клавиатуре make, чтобы создать PASDEMO.EXE. (Если вы назвали MAKEFILE как-то иначе, например MAKEPAS.MAK, наберите make -fmakepas.mak, чтобы создать PASDEMO.EXE.) Операторы MAKEFILE указывают, что PASDEMO.EXE создается из PASDEMO.OBJ и PASDEMO.PAS. Если любой из этих двух файлов изменяется, то команда tpc компилирует программу на Pascal, объединяя его коды с соответствующим объектным кодом. Вторая часть MAKEFILE устанавливает зависимость PASDEMO.OBJ от PASDEMO.ASM. Если ассемблерный файл изменяется, то Turbo Assembler формирует из PASDEMO.ASM новый PASDEMO.OBJ (что также вызывает рекомпиляцию PASDEMO.PAS).

Программа 12.2. PASDEMO.ASM

```
1:  %TITLE "Test Pascal .OBJ module -- Copyright (c) 1989,1995 by Tom Swan"
2:
3:  IDEAL
4:
5:  ;----- Сегмент данных объединяется с глобальным сегментом данных Pascal
6:
7:  SEGMENT DATA word public
8:
9:  ;----- Импортирование констант и переменных из Pascal
10:      EXTRN value : WORD, cr : BYTE, lf : BYTE
11:
12: asmCount    dw ?                    ; Статическая переменная
13:
14: ENDS DATA
15:
16:
17: ;----- Кодовый сегмент объединяется с основной программой Pascal
18:
19: SEGMENT CODE byte public
20:
21: ASSUME cs:CODE, ds:DATA             ; Поясняет модель памяти ассемблеру
22:
23: ;----- Экспортирует общие процедуры в Pascal
24:      PUBLIC AsmProc, CountPtr
25:
26: ;----- Импортирует процедуры и функции из Pascal
27:      EXTRN PasProc : NEAR, PasFunc : NEAR
28:
29:
30: ;------------------------------------------------------------
31: ; ПРОЦЕДУРА AsmProc;
32: ;------------------------------------------------------------
33:
34: ;----- Ранее инициализированные переменные должны быть помещены в кодовый сегмент
35: testString db 'AsmProc: Should be a "hatch mark" --> ', '$'
36:
37: PROC AsmProc NEAR
38:
39: ;----- Вызов процедуры Pascal
40:
41:      call   PasProc                 ; PasProc помещена в PASDEMO.PAS
42:
43: ;----- Использует локальные данные, сохраненные в кодовом сегменте
44:
45:      push   ds                      ; Сохраняет регистр ds Pascal
46:      push   cs                      ; Адресует кодовый сегмент
47:      pop    ds                      ; регистром ds
48: ASSUME      ds : CODE                  ; Информирует ассемблер
49:      mov    dx, offset testString   ; Адресует контрольную строку
50:      mov    ah, 09h                    ; Отображает контрольную строку посредством
51:      int    21h                     ; вызова функции ДОС "9"
52:      pop    ds                      ; Восстанавливает регистр ds Pascal
53: ASSUME      ds : DATA                  ; Информирует ассемблер
54:
55:
```

```
56:  ;----- Получить от Pascal константы с определенным типом и использовать статические переменные
57:
58:       mov    ax, [value]              ; Получить от Pascal значения
59:       mov    [asmCount], ax           ; Инициализировать статические переменные
60:
61:
62:  ;----- Вызвать функцию Pascal для значения символа
63:
64:       call   PasFunc                     ; Получить контрольный символ от Pascal
65:       mov    dl, al                   ; Присвоить значение символа dl
66:       mov    ah, 2                    ; Отобразить символ посредством
67:       int    21h                      ; функции ДОС "2"
68:
69:
70:  ;----- Получить переменные от Pascal
71:
72:       mov    ah, 2                    ; Функция ДОС отображения символа
73:       mov    dl, [cr]                 ; Получить cr от Pascal
74:       int    21h                      ; Выполнить возврат каретки
75:       mov    dl, [lf]                 ; Получить lf от Pascal
76:       int    21h                      ; Выполнить переход на новую строку
77:       ret                             ; Возврат в точку вызова
78:
79.  ENDP AsmProc
80:
81:
82:  %NEWPAGE
83:  ;-------------------------------------------------------
84.  , ФУНКЦИЯ CountPtr : IntPtr;
85.  ;-------------------------------------------------------
86:  PROC CountPtr NEAR
87:       mov    dx, SEG asmCount         ; Переслать сегмент адреса в dx
88:       mov    ax, OFFSET asmCount      ; Переслать смещение адреса в ax
89.       ret                             ; Возврат в точку вызова
90   ENDP CountPtr
91.
92   ENDS CODE                           ; Конец кодового сегмента
93:
94:  END                                  ; Конец модуля
```

Программа 12.3. PASDEMO.PAS

```
1: { Тестовая программа, которая должна быть в качестве внешней подключена к PASDEMO.OBJ}
2:
3: {$D+} { Включает отладочную информацию }
4:
5: PROGRAM PasDemo;
6:
7: CONST
8: value: Integer = 1234; { Объявление констант с определенным типом }
9:
10: TYPE
11: IntPtr = ^Integer; { Указатель на целый тип }
12:
13: VAR
14: cr, lf : Char; { Глобальные переменные }
15:
16: PROCEDURE PasProc, FORWARD, {Указывается перед директивой $L}
17: FUNCTION PasFunc: Char; FORWARD;
18:
19. {$L PASDEMO.OBJ} { Загрузка подключаемого объектного кода }
20:
21: { Внешние объявления, сообщающие Pascal формат внешних
22: подпрограмм в PASDEMO.ASM. }
23:
24: PROCEDURE AsmProc; EXTERNAL;
25: FUNCTION CountPtr. IntPtr; EXTERNAL;
26:
27: PROCEDURE PasProc;
28: VAR I: Integer; { Не может быть экспортировано в модуль ASM }
29: BEGIN
30: Writeln('PasProc: Inside the Pascal procedure')
```

```
31: END; { PasProc }
32:
33: FUNCTION PasFunc: Char;
34: BEGIN
35: PasFunc := '#' { Пересылка символа в модуль ASM }
36: END; { PasFunc }
37:
38: BEGIN
39: cr := chr(13);
40: lf := chr(10);
41: AsmProc;
42: Writeln('Main: asmCount = ', countPtr^)
43: END.
```

Замечание

В последующих разделах номера строк, помеченные „p", относятся к PASDEMO.PAS, а помеченные „a"— к PASDEMO.ASM.

Как понимать PASDEMO

Строки a7–14 объявляют сегмент данных ассемблерного модуля. Директива EXTRN импортирует из программы Pascal значение одной изменяющейся константы и две переменные — cr и lf (см. строки p8, p14). Обратите внимание, что не программа Pascal должна пересылать переменные и изменяющиеся константы, а ассемблерный модуль должен импортировать их, чтобы сделать имена соответствующими инструкциям языка ассемблера.

Строка a12 объявляет статическую неинициализированную переменную. Здесь должен использоваться знак вопроса (?), поскольку во внешней программе не допускаются инициализированные переменные.

Директива PUBLIC в строке a24 экспортирует в программу Pascal ассемблерные модули AsmProc и countPtr (см. строки a30–90). Соответственно строки p24–25 объявляют эти две подпрограммы EXTERNAL, давая возможность вызвать их из программы Pascal. Обратите внимание, как строка p25 определяет тип результата функции, который раньше в строке p11 объявляется типом данных Pascal (определенный как целый (integer)).

Еще одна директива EXTRN, на этот раз в сегменте кода (строка a27), импортирует процедуру PasProc и функцию PasFunc из Pascal в ассемблерный модуль. Вызов их осуществляется в строках a41 и a64, где иллюстрируется, как вызывать подпрограммы Pascal из внешних ассемблерных модулей. Указатели NEAR в директиве EXTRN (строка a27) указывают ассемблеру, что этот код находится в этом же сегменте. Указатели FAR были бы необходимы в том случае, если бы подпрограммы Pascal компилировались директивой {$F+} или появлялись в разделе интерфейса всего блока. В тексте программы Pascal PasProc и PasFunc объявляются FORWARD (см. строки p16–17), сообщая эти идентификаторы компилятору перед командой {$L} в строке p19, которая загружает с диска ассемблерный объектный код. Текст программы Pascal, относящийся к этим подпрограммам, приведен в строках p27–36.

Адресация данных кодового сегмента

Несмотря на то что нельзя объявить инициализированные переменные в сегменте данных ассемблерного модуля, который должен быть подключен к Pascal, можно поместить данные в кодовый сегмент, как показано в строке a35 в тексте программы PASDEMO.ASM. Позаботьтесь о том, чтобы разделить код и данные; предпочтительнее помещать переменные вне директив PROC.

Замечание

Основной кодовый сегмент в откомпилированной программе Pascal ограничивается 64 Кбайтами и включает в себя тело основной программы плюс глобальные процедуры и функции, поэтому лучше сводить число и размеры инициализированных переменных к минимуму.

Адресация переменных, расположенных в кодовом сегменте, требует использования переадресации на кодовый сегмент (cs:) при обращении к этим переменным в памяти. Еще большую трудность вызывает адресация таких переменных в другие подпрограммы, особенно при обращении к функции DOS, как это демонстрируется в строках a45–53. Во-первых, текущий ds регистр сохраняется в стеке — это совершенно необходимо. Pascal требует, чтобы ds постоянно указывал на глобальный сегмент данных. Если вы измените ds в ассемблерном модуле и забудете восстановить первоначальное значение регистра перед тем, как вернуться в Pascal, программа, наверно, разрушит свою область памяти, что приведет к "зависанию" компьютера.

Замечание

Несмотря на столь грозное предупреждение относительно изменения ds, es можно изменять в любой необходимый момент. Pascal не накладывает никаких ограничений на сегментный регистр es. Тем не менее не следует полагать, что es сохранит свое значение между обращениями к внешним подпрограммам.

В строках a46–47 ds устанавливается равным cs, адресуя кодовый сегмент регистром сегмента данных. Поэтому рационально использовать директиву ASSUME, чтобы сообщить Turbo Assembler об изменении регистра ds. Вслед за этим в строках a49–51 осуществляется вызов функции ДОС "9", отображающей последовательность ASCII$ на экран. Затем строка a52 восстанавливает значение сегментного регистра ds Pascal, что влечет за собой необходимость использования еще одной директивы ASSUME (строка a53), чтобы информировать Turbo Assembler о том, что ds снова адресует сегмент DATA (сегмент данных).

Адресация констант с определенным типом

Строки a58–59 в программе PASDEMO.ASM инициализируют глобальную переменную asmCount, объявленную в строке a12. Сначала значение константы с определенным типом (см. строку p8) в тексте программы Pascal перемещается в регистр ax (строка a58). Благодаря объявлению EXTRN в строке a10 Turbo Assembler "знает", что значение адресует 16-битовое слово. Как легко понять из иллюстрации, только от вас зависит, чтобы директивы EXTRN специфицировали правильные типы данных для переменных, объявленных в Pascal. Если вы объявили, что значением должен быть байт, Turbo Assembler никак не может определить, что это не так.

В строке a59 значение, хранящееся в ax, присваивается неинициализированной статической переменной, сохраненной в сегменте данных. Как следует из этого примера, в программе нет никаких указаний (строки a58–59) о том, где объявляются переменные. Вы можете считывать и записывать переменные (и изменяющиеся константы) независимо от того, были ли они объявлены в ассемблерном модуле либо в тексте программы Pascal.

Замечание

В отличие от переменных и констант с определенным типом, нельзя экспортировать из Pascal в ассемблер объявления CONST и TYPE. Простые константы и идентификаторы типа данных могут использоваться только в программе Pascal.

Вызов функций Pascal

Вызов функций Pascal из ассемблерного модуля аналогичен вызову процедур Pascal. После вызова PasFunc (строка a64) значение, возвращенное функцией в ax, направляется в регистр dl. В связи с тем, что PasFunc возвращает символ, требуется только младшая половина ax. Этот символ затем отображается посредством функции ДОС "2".

Замечание

Вам вполне по силам надлежащим образом использовать значения функций в ассемблерном модуле и разобраться в том, какие регистры изменяются при вызове функций Pascal. Табл. 12.1 приводит размеры результатов функций и регистры, использованные для возврата значений соответствующих типов.

Таблица 12.1 **Типы и размеры функций Pascal**

Тип функции	Размер в байтах	Регистр(ы)
Boolean (булев)	1	al
Char (символьный)	1	al
Enumerated (8-bit) (числовой 8-битовый)	1	al
Enumerated (16-bit) (числовой 16-битовый)	2	ax
ShortInt (целый короткий)	1	al
Byte (байт)	1	al
Integer (целый)	2	ax
Word (слово)	2	ax
LongInt (целый длинный)	4	dx=старший, ax=младший
Single (одинарный)	см. примечание 1	
Double (двойной)	см. примечание 1	

Окончание таблицы 12.1

Тип функции	Размер в байтах	Регистр(ы)
Real words (действительные слова)	6	dx=старший, bx=средний, ax=младший
Extended (расширенный)	см. примечание 1	
Comp (сравнение)	см. примечание 1	
Pointer (указатель)	4	dx=сегмент, ax=смещение
String (строка)	см. примечание 2	

Примечание 1. Функции этих типов возвращают значение в вершину стека математического сопроцессора.

Примечание 2 Строковые функции получают указатель на пространство памяти для временного использования, а также сохраняют результаты по этим адресам, оставляя указатель в стеке

Адресация переменных Pascal

Строки a72–76 осуществляют возврат каретки и перевод строки, передавая функции ДОС "2" значения двух переменных Pascal — cr и lf, которые объявляются в строке p14 и инициализируются в Pascal в строках p39–40. (Если вы считаете такой способ установки новой строки на дисплее странным, то вы совершенно правы. Но, несмотря на это, программа служит иллюстрацией того, как можно передавать данные из Pascal во внешний ассемблерный модуль.) Обратите внимание, что эти переменные импортируются в ассемблерный модуль побайтно в объявлении EXTRN строки a10. Переменные (и константы с назначенным типом) хранятся в глобальном сегменте данных Pascal и, как показано в данных примерах, легко доступны.

Замечание

К переменным, которые являются локальными для процедур и функций Pascal (например, к целой переменной I в строке p28), нельзя получить доступ из ассемблерного модуля. Локальные переменные существуют в Pascal только до тех пор, пока активны объявляющие их процедуры или функции; поэтому, до того момента как программа не будет запущена, вы никак не можете сообщить Turbo Assembler, где будут размещаться в памяти эти переменные. Чтобы обойти это ограничение, вы вынуждены передавать локальные переменные либо посредством их значений, либо адресацией их во внешние процедуры и функции в качестве параметров.

Вызов внешних функций

В строках a83–90 приведена небольшая внешняя функция, которая демонстрирует несколько дополнительных возможностей. Функция названа функцией общего пользования (строка a24) и объявлена в программе Pascal функцией EXTERNAL (строка p25). Типом данных для этой функции является указатель, указывающий на целый тип, определенный как IntPrt в строке p11 программы Pascal. Ассемблерный модуль не может использовать этот тип данных, поэтому программа должна возвращать значения в соответствующие регистры, которые ожидает Pascal в отношении определенного типа функции. Turbo Assembler не в состоянии проверить корректность результатов выполнения внешних функций.

В данном случае, поскольку типом является указатель, Pascal ожидает, что dx будет содержать сегмент, а ax — смещение адреса (см. табл. 12.1 и строки a87–88). В программе Pascal строка p42 использует этот адрес путем ссылки на идентификатор функции и отображения целого значения оператором Writeln. Но каково это значение? В очередной раз обратившись к тексту ассемблерной программы, легко увидеть, что строки a87–88 присваивают адрес неинициализированной переменной asmCount, объявленной в сегменте данных в строке a12. Оператор SEG возвращает значение сегмента помеченного адреса. Оператор OFFSET, в свою очередь, возвращает значение смещения. Таким образом, совместно эти два значения точно определяют размещение asmCount в памяти, выводя величину переменной в операторе Writeln. Все изложенное выше демонстрирует, каким образом можно передавать внешние переменные в Pascal. Следует помнить, что объявление PUBLIC в отношении меток данных воспринимается Turbo Assembler, но является неприемлемым для Turbo Pascal, поскольку за исключением переменных-констант сегмент данных Pascal не существует до тех пор, пока программа не начинает выполняться. Для передачи переменных из внешних модулей программе Pascal требуется передача их адресов.

Замечание

Можно также передавать указатели в виде параметров процедур или функций. Однако использование функций-указателей для определения размещения переменных, объявленных в ассемблерных модулях, обычно наиболее рациональным, так как для манипулирования параметрами процедур и функций требуются дополнительные усилия при программировании.

Передача параметров

Внешние ассемблерные подпрограммы усложняются, когда в них передаются переменные и числовые параметры. Это вызывает целый ряд дополнительных вопросов: передаются параметры своими значениями или ссылками; как поступать в особых случаях, в частности со строками и массивами; как убедиться, что стек организован правильно для возвращения в Pascal; как проделать все это в обратном направлении — вот краткий перечень проблем, вызываемых передачей параметров из ассемблерного модуля в процедуры и функции Pascal.

Лучшим способом разобраться в этом переплетении нюансов является доскональное овладение программированием на Pascal и ясное представление о том, как компилятор Pascal выполняет процедуры и функции, написанные в машинных кодах. Не следует забывать, что в вашем распоряжении находится один из лучших учителей — Turbo Debugger. Проверка тестируемой программы Pascal на уровне машинного кода с помощью команды View:CPU является прекрасным способом узнать, как Pascal выполняет операторы, написанные в машинных кодах.

Числовые параметры

Числовые параметры передаются как простые переменные в стеке. Например, чтобы передать целочисленный параметр, Pascal выталкивает его на верхушку стека, прежде чем вызвать процедуру, которая в нем нуждается. Процедура Pascal, подобная следующей

```
PROCEDURE Count(I Integer);
```

на машинном языке вызывалась бы таким образом:

```
mov ax, [I]             ; получение значения I
push ax          ; выталкивание значения I на верхушку стека
call Count              , вызов процедуры
```

Откомпилированная программа процедуры Count должна получить значение I из стека. В Pascal процедуры и функции осуществляют это посредством ссылки на стек через регистр bp. Соответственно откомпилированные процедуры и функции обычно начинаются со следующих операторов:

```
push bp          ; сохранение текущего значения bp
mov bp, sp              , адресация стека регистром bp
```

Рис. 12.1 иллюстрирует состояние стека при запуске Count после выполнения этих двух инструкций. Значение I находится под 2-байтовым адресом возврата, который, в свою очередь, находится под сохраненным значением bp. (Каждый пронумерованный прямоугольник на этой диаграмме представляет один байт. Числа не совпадают с реальными адресами в памяти.)

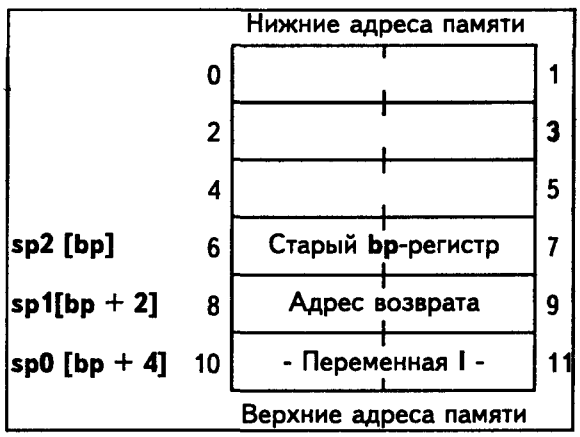


Рис. 12.1. Стек, показывающий один числовой параметр Pascal

Подсчитав байты от bp до начала I, легко убедиться, что, прибавив к bp 4 байта, можно получить адрес начала I. Следовательно, чтобы загрузить ax значением, сохраненным в этой ячейке стека, следует написать:

```
mov ax, [word bp+4]      , Присвоение значения I ax
```

Можно также обратиться к значению непосредственно, воспользовавшись инструкцией

```
inc [word bp+4]          , I := I+1
```

Одно осложнение, связанное с этим, возникает в процедурах FAR. В этом случае адрес точки возврата имеет длину 4 байта:2 байта — сегмент и 2 байта — смещение (рис. 12.2). В связи с этим, чтобы загрузить значение I, следует использовать приращение, равное 6, вместо 4:

```
mov ax, [word bp+6]      , Загрузка I в ax (подпрограмма FAR)
```

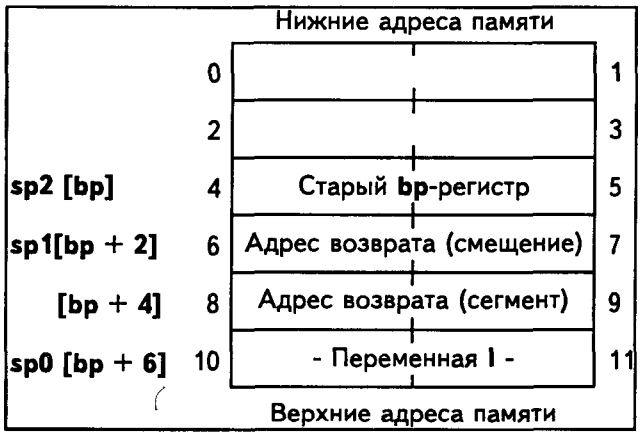


Рис. 12.2. Стек после вызова процедуры FAR с одним численным параметром

Возврат из внешнего кода

Когда внешняя ассемблерная подпрограмма заканчивается, она должна воспользоваться специальной инструкцией ret, которая, кроме определения адреса точки возврата и продолжения работы программы, должна также удалить байты параметров из стека. В данном случае в стеке находятся два байта параметра, значит, подпрограмма должна заканчиваться следующей инструкцией:

```
ret 2 , Возврат и удаление (выталкивание) двух байтов из стека
```

Указанное после ret значение прибавляется к указателю стека *после* выталкивания адреса точки возврата в ip (в cs — в случае межсегментного дальнего (FAR) вызова). Следует помнить, что значение представляет число *байтов* всех параметров, продвигаемых по стеку. Поскольку Pascal никогда не продвигает менее 2 байт — даже однобайтовые символы продвигаются как двухбайтовые слова — конкретное значение в инструкции ret внешней подпрограммы Pascal всегда должно быть четным числом.

Переменные параметры

Переменные параметры (которые предваряются описателем VAR в списке параметров процедур или функций Pascal) передаются посредством ссылок, т.е. адресов. Четырехбайтовый адрес каждой такой переменной передается в стек, и ссылка на него осуществляется так же, как и на любое другое значение. Ассемблерная программа может использовать такой адрес в качестве указателя на действительный адрес какой-либо другой ячейки памяти. Сказанное выше понять из нескольких примеров. Предположим, что предыдущая процедура объявила переменный параметр:

```
PROCEDURE Count(VAR I. Integer),
```

В откомпилированной программе обращение к процедуре Count выталкивает адрес I в стек. Если предположить, что I хранится в сегменте данных программы, то компилирующие инструкции могли быть подобны следующим:

```
mov di, offset I          ; Получение смещения переменной I
push ds                   ; Проталкивание адреса сегмента, хранящего I
push di                   , Проталкивание смещения адреса I
call Count                                  , Вызов процедуры Count
```

При запуске процедуры Count, после сохранения и присвоения bp значения регистра указателя стека, стек выглядит так, как показано на рис. 12.3. После того как стек сконфигурирован согласно рисунку, ассемблерный модуль может получить значение I несколькими способами. Одним из них является загрузка из стека es и еще одного регистра (di подходит для этого так же, как и bx):

```
mov es, [word bp+6]                    ; Получение значения сегмента
mov di, [word bp+4]                    ; Получение значения смещения
```

Затем es:di адресует значение I в память. Можно не сомневаться, что эта ячейка может находиться где угодно — в сегменте данных или, возможно, в стековом сегменте, если I является локальной переменной, объявленной в процедуре или функции Pascal. Другим способом получить этот же результат является использование инструкции les:

```
les di, [bp+4]          ; Загрузка es:di значением адреса I
mov ax, [word es:di]                   ; Загрузка значения I в ax
```

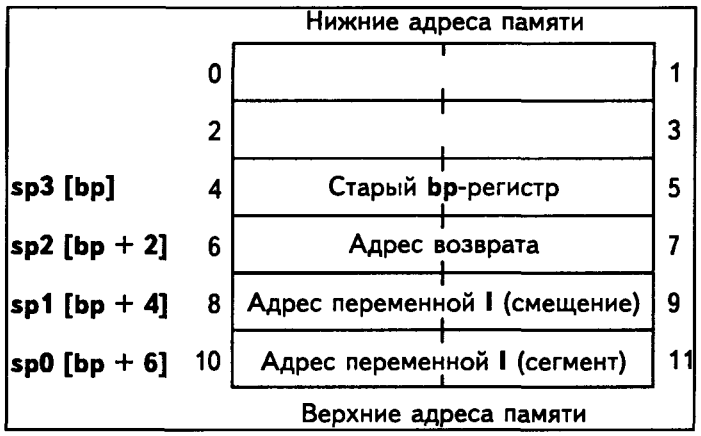


Рис. 12.3. Стек с одним переменным параметром

Инструкция les загружает и в es, и в di (либо в любой другой из основных регистров) адрес, сохраненный в специфицированной ячейке, в данном случае — 4 байта в соответствии с указанием ss:bp. Затем вторая инструкция адресует эту ячейку для загрузки значения переменного параметра в ax. Аналогичную инструкцию lds можно применять для загрузки сегментной части адреса в ds. Поскольку Pascal требуется ds для адресации глобального сегмента данных, при использовании lds убедитесь в том, что вы сохранили и восстановили значение ds до окончания работы внешней подпрограммы.

Использование модели памяти TPASCAL

Одним из способов упрощения адресации переменных в стеке является использование специализированной для Turbo Assembler модели памяти TPASCAL, созданной для применения с ранними версиями Turbo Pascal. Однако не следует использовать TPASCAL с Borland Pascal. Этот метод имеет следующие преимущества.

- Возможность использовать упрощенные директивы CODESEG и DATASEG вместо ручного объявления именованных сегментов.
- Turbo Assembler автоматически подготавливает и восстанавливает регистр bp.
- Параметр, адресующийся в стек, вычисляется, что дает возможность адресовать его по имени, а не вычислять смещение в стеке, как, например, в [bp+8].
- Верное конкретное значение в инструкцию ret для удаления байтов параметра из стека вводится автоматически.

Однако недостатком модели памяти TPASCAL является невозможность воспрепятствовать Turbo Assembler сгенерировать инструкции, подготавливающие bp для адресации переменных в стек. Даже в тех процедурах, в которых отсутствуют параметры, эти инструкции тем не менее слепо вставляются в тело программы. А ведь одной из основных причин использования ассемблерных модулей совместно с Pascal является как раз стремление избавиться от всех ненужных инструкций! Вместе с тем применение TPASCAL вполне приемлемо в некоторых случаях — как в следующих ниже примерах. Но для оперативного управления на низком уровне вам придется объявлять директивы SEGMENT вручную, как в PASHELL.ASM.

Использование директивы ARG

Если модель памяти TPASCAL задействована, для упрощения адресации параметров можно использовать директиву ARG. Эта директива Turbo Assembler имена и размеры параметров, переданных в стек для внешних процедур. Ассемблер использует эту информацию, чтобы вычислить значения смещений относительно ss:bp, где сохранены значения параметров.

Замечание

ARG работает и с другими моделями памяти, как с неупрощенными сегментами. Однако имеется определенное отличие. С моделью памяти Pascal параметры должны появляться в директивах ARG в том же порядке, что и в объявлениях процедур и функций Pascal. Если вы не используете TPASCAL, то необходимо перечислять параметры в обратном порядке.

ARG требует, чтобы последовательности элементов разделялись запятыми и при этом каждый элемент описывал один параметр. Рассмотрим следующее объявление процедуры Pascal:

```
PROCEDURE StoreNum( MyNumber   Integer ),
```

На ассемблере эта команда имеет соответствующее объявление PROC:

```
PROC StoreNum NEAR
 ARG MyNumber   WORD
```

После их исполнения необходимо с помощью приведенной ниже ассемблерной инструкции переместить значение MyNumber в регистр:

```
mov ax, [MyNumber] , Загрузка ax значением MyNumber
```

Сравните это с обычным методом адресации переменных в стеке по отношению к bp:

```
mov ax  [word bp+4]
```

Если в дальнейшем вы изменяете число параметров, передаваемых процедуре, или меняете тип процедуры с NEAR на FAR — просто переассемблируйте внешний объектный код, чтобы привести в соответствие размещение MyNumber в стеке. В случае использования директивы ARG вам придется пересчитать и изменить цифру 4 вручную, что значительно повышает вероятность сделать программу дефектной, если вы ошибетесь.

Удаление параметров из стека

Если вы приводите список параметров ARG со знаком равенства "=" и назначенным временным именем, Turbo Assembler подсчитывает число байтов, занятых всеми параметрами, и присваивает это значение имени, которое вы назначили. Например, следующие директивы присваивают ArgSize число байтов, занимаемых двумя параметрами — his и hers:

```
PROC Share NEAR
 ARG his WORD  hers WORD = ArgSize
```

Если не использовать модель памяти TPASCAL, то можно применять ArgSize совместно с ret, чтобы удалить байты параметров из стека:

```
ret ArgSize , Возврат и удаление параметров из стека
```

При использовании модели памяти TPASCAL этого не следует делать, так как в этом случае Turbo Assembler автоматически прибавляет верное значение к ret (естественно, если вы правильно определили в директиве ARG число и размеры параметров). Используя модель памяти TPASCAL, всегда заканчивайте свои внешние процедуры PROC простой инструкцией ret. (Вы можете также дополнить директиву ARG командой =ArgSize и использовать значение, поставленное в соответствие ArgSize, для других целей.)

Написание внешних строчных функций

Третья опция директивы ARG позволяет специфицировать параметры, которые не нужно удалять из стека по окончании работы внешней подпрограммы. Для этого вслед за элементами списка (возможно, с добавленной командой =ArgSize) необходимо добавить RETURNS, за которой следует список параметров, не подлежащих удалению из стека по завершении PROC.

В Pascal, вероятно, единственным случаем, когда вам потребуется RETURNS, является написание внешних строчных функций. Когда Pascal обращается к строчной функции, сначала он выталкивает в стек результат функции — 4-байтовый указатель, перед тем как вытолкнуть остальные параметры (если они имеются), передаваемые функции. Указатель результата функции адресует временную область, где ваша внешняя программа может хранить символы строки, возвращаемой функцией. Когда внешняя подпрограмма заканчивается, Pascal ожидает, что указатель строчной функции останется в стеке. (Инструкции, следующие за вызовом подпрограммы, позже либо удаляют эти байты, либо просто передают адрес другой процедуре или функции, которая использует результат этой строчной функции.) Из-за такого специфичного действия программы, если вы объявите результат функции в основном списке параметров ARG, процедура не будет работать, поскольку Turbo Assembler удалит байты параметров и указатель результата функции из стека инструкцией ret.

Программа 12.4, FILLSTR.ASM, демонстрирует правильный способ написания внешних строчных функций для Pascal. Программа 12.5, FILLSTR.PAS, показывает, как подключать внешний модуль к программе Pascal. Ассемблирование, компиляция и запуск программы Pascal осуществляется следующими командами:

```
tasm fillstr
tpc fillstr
fillstr
```

Программа 12.4. FILLSTR.ASM

```
1:   %TITLE "Pascal String-Filler Function -- Copyright (c) 1989,1995 -- by Tom Swan"
2:
3:        IDEAL
4:
5:        MODEL TPASCAL
6:
7:        CODESEG
8:
9:        PUBLIC FillString
10:
11:  %NEWPAGE
12:  ;--------------------------------------------------------
13:  ; ФУНКЦИЯ FillString( n : Byte; ch : Char ) : String;
14:  ;--------------------------------------------------------
15:  PROC FillString NEAR
16:       ARG    n:BYTE:2, c:BYTE:2 RETURNS string:dword
17:
18:       les    di, [string]              ; es:di адресует результат функции
19:       mov    al, [n]                       ; Загрузка n в al
20:       cld                              ; Автоувеличение di
21:       stosb                            ; Инициализирует длину строки
22:       xor    ch, ch                    ; Обнуляет старшую половину cx
23:       mov    cl, al                    ; cx = требуемой длине строки
24:       mov    al, [c]                       ; al = заполняющему символу
25:       jcxz   @@99                      ; Выход, если length = 0
26:       repnz  stosb                     ; Накапливает символы в строку
27:  @@99:
28:       ret                              ; Возврат в точку вызова
29:  ENDP FillString
30:
31:       END    ; Конец модуля
```

Программа 12.5. FILLSTR.PAS

```
1: PROGRAM FillStr;
2:
3: { Проверка с использованием внешней функции FillString }
4:
5. VAR s : String;
6:
```

```
7: FUNCTION FillString( n : Byte, ch : Char ) : String; EXTERNAL;
8: {$L FILLSTR.OBJ}
9:
10: BEGIN
11: s := FillString( 45, '@' );
12: Writeln( 'After filling: ', s )
13: END.
```

Как работает FILLSTR

Строка 16 в программе FILLSTR.ASM создает объявление ARG для возврата параметра в стек. Для удобства эта строка приведена ниже:

```
ARG n:BYTE:2, c:BYTE:2 RETURNS string:dword
```

Сначала появляются два параметра — n и c, каждый из которых является единственным байтом. Обратите внимание, что параметр ch:char из текста программы Pascal здесь переименовывается на c, поскольку ch в языке ассемблера относится к старшему байту регистра cx и не может использоваться в качестве идентификатора. (Я умышленно создал подобную конфликтную ситуацию, чтобы показать, что в объявлении ARG имена могут быть какими угодно — они вовсе не должны полностью соответствовать своим аналогам в Pascal.)

Поскольку Pascal всегда выталкивает значения в стек по 2 байта, к обоим параметрам добавлены дополнительные определители :2, показывающие Turbo Assembler, что даже если он адресует n и c в виде байта, то он должен считать эти переменные занимающими по 2 байта стекового пространства. Если не включить определитель :2, Turbo Assembler неверно вычислит число байтов, занимаемых параметрами, и не зафиксирует состояние стека корректно по завершении внешней подпрограммы.

Замечание

Признаком неверного закрытия стека является ошибка „Stack Overflow". Если вы получили сообщение об этой ошибке, проверьте, все ли однобайтовые параметры имеют спецификацию :2 в вашем списке ARG.

Следующие два параметра содержатся в предложении RETURNS string:dword. Имя string может быть любым идентификатором, который просто дает возможность обратиться к результату функции внутри внешней программы. Часть :dword является директивой, указывающие Turbo Assembler на то, что строка адресует в стек 4-байтовое значение. (В действительности строчная функция возвращает в Turbo Pascal указатель, а он всегда занимает 4 байта.)

Результат строчной функции можно адресовать несколькими способами. Простейшим методом является загрузка es:di или ds:si адресом области, зарезервированной для результата:

```
les di, [string] , es:di адресует результат функции
```

После этого байт длины цепочки помещается в es:di. Первый символ цепочки располагается в es:di+1 и т.д. Сохранение символов в es:di передает эти символы обратно в качестве результата строчной функции. Вам не нужно предпринимать никаких других шагов, чтобы возвратить строку процедуре, вызывающей внешнюю функцию. В программе FILLSTR.ASM повторяющаяся инструкция stosb использует этот метод для возврата строки, заполненной любыми ASCII-символами.

Объявление параметров без ARG

Существует и другой метод объявления параметров, не использующий ARG, — достаточно просто поместить список параметров после PROC и NEAR или FAR директивы. Например, можно заменить строки 15–16 в тексте программы FILLSTR.ASM строкой

```
PROC FillString NEAR n:BYTE:2, c:BYTE:2 RETURNS string.dword
```

Другими словами, если вы записываете все одной строкой, вам не требуется директива ARG. Однако, поскольку длинные объявления PROC, подобные приведенной выше, трудно читать, я предпочитаю перечислять аргументы в отдельной директиве ARG. Результат при этом достигается одинаковый, и вы можете использовать тот из двух методов, который вам больше нравится.

Проблема быстродействия

Существует только одна причина тратить время, оптимизируя программу на Pascal или на любом другом языке с помощью аппаратно-зависимого языка ассемблера: добиться того ошеломительного быстродействия, которое может обеспечить, если использовать его правильно, только ассемблер. В этом разделе мы напишем программу на Pascal, рассмотрим отдельно машинный код, сгенерированный Turbo Pascal, и напишем в значительной степени оптимизированную замещающую внешнюю процедуру на ассемблере. Как вы убедитесь, результаты стоят затраченных усилий.

Программа на Pascal

Прежде всего, нам требуется программа на Pascal. Программа 12.6, STR.PAS, содержит две полезные процедуры — ASCIIZtoStr и StrToASCIIZ, которые преобразуют строки Pascal в формат ASCIIZ и обратно и используются многими составными программами в этой книге. Чтобы сэкономить место, здесь приведен оптимизированный текст программы на Pascal. Поэтому для проверки, после того как вы введете эту программу, скопируйте STR.PAS в другой файл, названный STRSLOW.PAS.

Следующим шагом загрузите STRSLOW.PAS в свой редактор и удалите строки 12, 18, 19, 30, 36, 37 и 45. Это действие преобразует текст в программу на чистом Pascal, убирая ссылки на внешние подпрограммы, которые вы снова добавите позже. После этих изменений откомпилируйте STRSLOW с помощью команды

```
tpc /v strslow
```

Программа 12.6. STR.PAS

```
1: PROGRAM StringConversion;
2:
3: { Преобразование строк ASCIIZ и строк Pascal }
4:
5: TYPE    ASCIIZString = ARRAY[ 0 .. 255 ] OF Char;
6:         ASCIIZptr = ^ASCIIZString;
7:
8: VAR     a : ASCIIZString;
9:         s : String;
10:
11:
12: {$L STR.OBJ}
13:
14:
15: { Преобразование строки ASCIIZ (a) в строку Pascal (s) }
16:
17: PROCEDURE ASCIIZtoStr( a : ASCIIZString; VAR s : String );
18: EXTERNAL;
19: (*
20:  VAR len : Integer;
21:   BEGIN
22:     len := 0;
23:     WHILE ( len < 255 ) AND ( a[ len ] <> Chr( 0 ) ) DO
24:       BEGIN
25:         len := len + 1;
26:         s[ len ] := a[ len - 1 ]
27:       END; { while }
28:     s[ 0 ] := Chr( len )
29:   END; { ASCIIZtoStr }
30: *)
31:
32:
33: { Преобразование строки Pascal (s) в строку ASCIIZ (a) }
34:
35: PROCEDURE StrToASCIIZ( s : String; VAR a : ASCIIZString );
36: EXTERNAL;
37: (*
38:  VAR len, i : Integer;
39:   BEGIN
40:     len := Length( s );
41:     FOR i := 1 TO len DO
42:       a[ i - 1 ] := s[ i ];
43:     a[ len ] := Chr( 0 )
44:   END; { StrToASCIIZ }
45: *)
```

```
46:
47:
48: { Отображение строки ASCIIZ }
49:
50: PROCEDURE ShowASCIIZ( a : ASCIIZString );
51:  VAR i : Integer;
52:   BEGIN
53:     i := 0;
54:    WHILE ( i < 255 ) AND ( a[ i ] <> Chr(0) ) DO
55:      BEGIN
56:       Write( a[ i ] );
57:       i := i + 1
58:      END { while }
59:   END; { ShowASCIIZ }
60:
61:
62: BEGIN
63:   s := 'This is a test';
64:   StrtoASCIIZ( s, a );
65:   ShowASCIIZ( a );
66:   Writeln;
67:   s := '';
68:   ASCIIZtoStr( a, s );
69:   Writeln( s )
70: END.
```

Разбор текста программы STRSLOW

После компиляции STRSLOW, чтобы увидеть машинные коды, созданные Turbo Pascal, запустите программу под управлением Turbo Debugger командой

```
td strslow
```

Затем нажимайте клавишу <F7> для пошагового прохождения программы. Когда вы войдете в процедуры StrToASCIIZ и ASCIIZtoStr, воспользуйтесь командой View:CPU, чтобы посмотреть машинный код, который Turbo Pascal генерирует для этих подпрограмм. Вы будете поражены длиной программы в машинных кодах, в которую скомпилировал Pascal внешне простые операторы высокого уровня. Во многих случаях Turbo Pascal генерирует очень компактные и быстро исполняемые коды. Но, как совершенно очевидно, это не тот случай. С небольшими дополнениями на языке ассемблера все это можно сделать гораздо лучше.

На рис. 12.4 приведен текст дизассемблированной составной программы, которая соответствует процедуре программы Pascal StrToASCIIZ (строки 35–44 в тексте программы STR.PAS). Комментарии помогут вам в основном понять, какие действия в ней выполняются. Обращения к недокументированным подпрограммам Turbo Pascal, выполняющимся в процессе работы основной программы, помечены "internal sub" (внутренняя подпрограмма) и не поясняются. При просмотре этой части программы посредством Turbo Debugger некоторые инструкции могут иметь несколько другие форматы.

Замечание

В зависимости от вашей конкретной версии Turbo или Borland Pascal, реальный машинный код может отличаться от того, который приведен на рис. 12.4.

Строки 7–18 на рис. 12.4 показывают одну из причин, из-за которой даже самые лучшие компиляторы языков высокого уровня, таких как Turbo Pascal, иногда генерируют медленно исполняемые коды. Так уж случилось, что данная процедура имеет строчный параметр s, передаваемый своим значением; вот почему компилятор совершенно правильно полагает, что эта строка может быть изменена внутри процедуры. В результате, а также потому, что строки и другие множества *всегда* внутри программы передаются по адресу, код в строках 7–18 слепо копирует всю строку во временную рабочую область стека — процесс, который повторяется каждый раз при вызове процедуры. Однако, как можно видеть из программы Pascal, эта строка не изменяется и весь этот код оказывается ненужным, просто компилятор не настолько сообразителен, чтобы им пренебречь.

Есть и другие места в этом тексте программы (как и во второй процедуре — ASCIIZtoStr), которые также можно улучшить. Например, совершенно очевидно, что строка 28 лишняя, так как ах должен уже иметь значение, сохраненное в [bp-0102h] в соответствии с более ранней инструкцией в строке 24. Эти наблюдения приводят к заключению, что подпрограммы на чистом языке ассемблера экономят память и выполняются быстрее.

```
1:      ; PROCEDURE StrTOASCIIZ( s : String; VAR a : ASCIIZString);
2:
3:      PROC StrTOASCIIZ NEAR
4:
5:           push   bp                          ; Сохранение bp в стеке
6:           mov    bp, sp                      ; Адресация параметров посредством bp
7:           mov    ax, 0106h                   ; Проверка доступности 106h байт
8:           call   far ptr (internal sub)      ; стека
9:           sub    sp, 0106                            ; Резервирование области стека для s
10:          les    di, [dword ptr bp + 8]      ; es:di = адресу стека
11:          push   es                          ; Проталкивание исходного адреса
12:          push   di                          ; (сегмента и смещения)
13:          lea    di, [bp - 0100h]            ;es:di = адресу копии
14:          push   ss                          ; Проталкивание адреса назначения
15:          push   di                          ; (сегмента и смещения)
16:          mov    ax, 00ffh                   ; Число байтов, подлежащих копированию
17:          push   ax                          ; Проталкивание счетчика
18:          call   far ptr (internal sub)      ; Копирование строки во временную переменную
19:
20:     ; Len := Length (s)
21:
22:          mov    al, [bp-0100h]              ; Получение длины s
23:          xor    ah, ah                      ; Обнуление старшей части ax
24:          mov    [bp - 0102h], ax            ; Инициализация переменной Len
25:
26:     ; For I := 1 To Len DO
27:
28:          mov    ax, [bp - 0102h]            ; Присвоение Len ax
29:          mov    [bp - 0106h], ax            ; Присвоение Len значению стопа
30:          mov    ax, 0001h                   ; Присвоение начального значения ax
31:          cmp    ax, [bp - 0106h]            ; Значение старта > значения стопа?
32:          jg     @@09                        ; Если да, закончить цикл For
33:          mov    [bp - 0104h], ax            ; В противном случае инициализировать I
34:          jmp    short @@08                  ; Переход в цикл
35:     @@07:
36:          inc    [word bp - 0104h]           ; Увеличение управляющей переменной (I)
37:
38:     ; a[ I - 1 ] := s[I]
39:
40:     @@08
41:          mov    di, [word bp - 0104h]       ; Присвоение I di
42:          mov    dl, [byte bp+di-0100h]      ; Получение символа из s[I]
43:          mov    ax, [word bp - 0104h]       ; Установка ax на I
44:          dec    ax                          ; Настройка ax на I-1
45:          les    di, [dword bp + 04]         ; es:di адресует a
46:          add    di, ax                      ; Добавление di к a[I-1]
47:          mov    [byte es:di], dl            ; Выборка символа из s[I]
48:          mov    ax, [word bp - 0104h]       ; Пересылка управляющей переменной (I) в ax
49:          cmp    ax, [word bp - 0106h]       ; Сравнение со значением стопа
50:          jne    @@07                        ; Переход, если ax <> значению стопа
51:
52:     , a[ Len ] .= Chr (0)
53:
54:     @@09
55:          mov    ax, [word bp - 0102h]       ; Пересылка Len в ax
56:          les    di, [word bp + 04]          , es:di адресует a
57:          add    di, ax                      ; es:di адресует a[Len]
58:          mov    [byte es:di], 0             ; Засылка в a[Len] 0
59:
60:     ; END; { StrTOASCII }
61:
62:          mov    sp,bp                       ; Восстановление указателя стека
63:          pop    bp                          ; Восстановление сохраненного регистра bp
64:          retn   8                           ; Возвращение сохраненного регистра ip
65:
66:     ENDP StrToASCIIZ
```

Рис. 12.4. Текст составной программы для процедуры StrToASCIIZ в STR.PAS (измененная, неоптимизированная версия, переименованная в STRSLOW.PAS) с комментариями

Оптимизация STR.PAS

Программа 12.7, STR.ASM, замещает процедуры ASCIIZtoStr и StrToASCIIZ в STR.PAS внешними ассемблерными подпрограммами. (Если вы следовали приведенным рекомендациям, то несколько раньше скопировали STR.PAS в STRSLOW.PAS. Прежде чем сделать следующий шаг, убедитесь, что у вас на диске хранится первоначальная копия STR.PAS). Ассемблируйте, скомпилируйте и запустите программу с помощью команд:

```
tasm str
tpc str
str
```

Программа 12.7. STR.ASM

```
1:  %TITLE "ASCIIZ and Pascal Strings -- Copyright (c) 1989,1995 -- by Tom Swan"
2:
3:      IDEAL
4:
5:      MODEL TPASCAL
6:
7:      CODESEG
8:
9:      PUBLIC ASCIIZtoStr, StrToASCIIZ
10:
11: %NEWPAGE
12: ;-----------------------------------------------------------
13: ; PROCEDURE ASCIIZtoStr( a : ASCIIZString; VAR s : String );
14: ;-----------------------------------------------------------
15: PROC ASCIIZtoStr NEAR
16:     ARG     a:dword, s:dword = ArgSize
17:     push    ds                          ; Сохранение регистра ds Pascal
18:     les     di, [s]                         ; Адресация s посредством es:di
19:     push    di                          ; Сохранение адреса на будущее
20:     inc     di                          ; Адресация s[1] посредством es:di
21:     lds     si, [a]                         ; Адресация a посредством ds:si
22:     cld                                 ; Автоувеличение si, di
23:     xor     cl, cl                      ; Установка len (cl) в нуль
24: @@10:
25:     cmp     cl, 255                         ; Проверка условия <len = 255 уже?>
26:     je      @@20                        ; Если да, выход
27:     lodsb           ; Получение символа (al <- a[i])
28:     or      al, al                      ; Проверка условия al = 0 (ASCII нуль)?
29:     jz      @@20                        ; Если символ = нулю, выход
30:     inc     cl                          ; Len := Len + 1
31:     stosb           ; s[ Len ] := a[ Len - 1 ]
32:     jmp     @@10                        ; Цикл до завершения
33: @@20:
34:     pop     di                          ; es:di снова адресует s[0]
35:     mov     [byte es:di], cl            ; s[ 0 ] := Chr( Len )
36:     pop     ds                          ; Восстановление регистра ds Pascal
37:     ret                                 ; Возврат в точку вызова
38: ENDP ASCIIZtoStr
39:
40: %NEWPAGE
41: ;-----------------------------------------------------------
42: ; PROCEDURE StrToASCIIZ( s : String; VAR a : ASCIIZString );
43: ;-----------------------------------------------------------
44: PROC StrToASCIIZ NEAR
45:     ARG     s:dword, a:dword = ArgSize
46:     push    ds                          ; Сохранение регистра ds Pascal
47:     les     di, [a]                         ; Адресация a посредством es:di
48:     lds     si, [s]                         ; Адресация s посредством ds:si
49:     cld                                 ; Автоувеличение si, di
50:     xor     ch, ch                      ; Обнуление старшей половины cx
51:     lodsb           ; al := Length( s )
52:     mov     cl, al                      ; cx = длине строки
53:     jcxz    @@10                        ; Выход, если длина строки = 0
54:     repnz   movsb                       ; Перенос s в a
55: @@10:
56:     mov     [byte es:di], cl            ; a[ Len ] := Chr( 0 )
57:     pop     ds                          ; Восстановление регистра ds Pascal
58:     ret                                 ; Возврат в точку вызова
59: ENDP StrToASCIIZ
60:
61:     END                                 ; Конец модуля
```

Как работает STR.ASM

Вы просмотрели все инструкции, команды и процедуры, нашедшие свое отражение в STR.ASM, следовательно, у вас не должно возникнуть особых проблем с пониманием того, как работает эта программа. Обратите внимание, как используется директива ARG, для того чтобы сделать адресацию параметров в стек более простой, не требующей сложных и чреватых ошибками спецификаций типа [word bp+4]. Кроме того, модель памяти TPASCAL позволяет Turbo Assembler инициализировать и сохранять bp автоматически, а также прибавлять соответствующее конкретное значение к инструкциям ret, удаляя параметры из стека при необходимости.

Я использовал модель памяти таким образом, чтобы этот пример правильно работал с большинством версий Turbo Assembler и Turbo Pascal. Однако следует помнить, что не нужно применять TPASCAL совместно с Borland Pascal.

Обратите особое внимание на строки 18, 21 и 47–48, которые загружают es:di и ds:si адресами параметров a и s. Поскольку имеется потенциальная возможность того, что при этом будет изменен ds, переменные могут и не находиться в сегменте данных программы Pascal — процедура предусмотрительно заранее сохраняет ds.

Еще одна возможность оптимизации, продемонстрированная здесь, использует тот факт, что даже несмотря на то, что a является численным параметром для ASCIIZtoStr, а s — для StrToASCIIZ, Turbo Pascal всегда передает строки и множества по адресам. Следовательно, поскольку эти переменные не изменяются, оптимизированная внешняя программа опускает шаги, связанные с копированием, которое осуществляется в коде, сгенерированном компилятором.

Замечание

В этом примере переменные a и s сохранены в сегменте данных, и если вы запустите тест в Turbo Debugger, то сможете убедиться, что ds действительно не изменяется в строках 21 и 48. Но другая программа могла бы передать этим процедурам параметры, которые не хранятся в сегменте данных. В этом случае ds изменился бы. Подобные нюансы — источник многочисленных сбоев, и лучшим способом предотвратить их является досконалное знание того, как Pascal работает с машинным кодом. Не следует полагать, что если в одном случае значение регистра не меняется, то оно не будет меняться и в другом.

Быстродействие, достигаемое в STR.ASM, почти полностью обязано применению быстродействующей строчной инструкции 8086 в строках 27, 31 и 54. Сравните эти инструкции с обычным методом, использованным в выходной программе на чистом Pascal (см. рис. 12.4). Просто нет никакой подходящей альтернативы хранению значений в быстродоступных регистрах общего пользования, применению строчных индексов si и di и использованию преимуществ мощных строчных инструкций, таких как lodsb и movsb. Как легко видеть, небольшие ассемблерные фрагменты, вставленные в программу Pascal, могут существенно улучшить ее характеристики.

Отдавая должное Turbo Pascal, я вынужден признать, что STR.PAS может быть написана таким образом, что она будет выполняться значительно быстрее с использованием инструкций, присущих только Turbo Pascal, подобных Move. Но, хотя меня можно обвинить в том, что я "пеку книги" ради удачного примера оптимизирующих возможностей языка ассемблера, бывают случаи, когда вы захотите избежать команд, уникальных для Turbo-систем — даже если это приведет к более медленному коду. Ограничивая свои программы стандартными командами Pascal, вы достигнете того, что вашу программу будет легче перенести в другую систему. Действительно, я иногда писал три версии программы: одну — на стандартном Pascal, вторую, оптимизированную, — с помощью Turbo Pascal и третью, оптимизированную, — с применением ассемблера, заменяющего процедуры и функции в обеих первых версиях. Это, конечно, требовало дополнительной работы, но и значительно увеличивало возможности по запуску программы на различном оборудовании с минимальными модификациями.

Резюме

Компиляторы сегодня сообразительней, чем когда-либо раньше, но они по-прежнему не могут сравниться по уму с умным ассемблерным программистом. Даже программы, скомпилированные в сверхбыстрый код Turbo Pascal, часто могут быть улучшены. Но знание того, как вставлять ассемблерные фрагменты в программу на Pascal, — это только полдела, так же важно знать — когда это делать. Обычно лучше всего преобразовать только критичный код, оставляя остальные части на Pascal. Для получения переносимости на другие системы представляется разумным писать операторы Pascal до того, как преобразовывать критичные участки на язык ассемблера.

Найти критичный код иногда оказывается непросто, но большинство специалистов сходятся во мнении, что программы до 90% времени затрачивают на выполнение около 10% своих инструкций. Оптимизация этих критичных 10% может значительно увеличить скорость выполнения программы, оптимизация остальных 90% может оказаться пустой тратой времени. Профайлер может оказать помощь в нахождении критичных участков, предоставляя статистические данные о выполняемой программе.

Turbo и Borland Pascal дают возможность использовать операторы Inline, процедуры и функции Inline, а также внешние процедуры и функции для подключения языка ассемблера к Pascal. Последнее, как правило, предпочтительнее, поскольку повышает возможность переноса программы в другую систему. Внешние подпрограммы могут ассемблироваться и отлаживаться отдельно, а также использоваться с другими языками программирования.

Необычная модель памяти Pascal обусловливает особый подход, требуя от вас объявления сегментов кода и данных достаточно сложным образом для получения наиболее гибких результатов. В качестве альтернативы с ранними версиями Turbo Pascal может использоваться модель памяти TPASCAL, хотя у нее имеется недостаток — необходимость включения в каждую процедуру запускающих инструкций, независимо от того, нужны они или нет.

Можно вызывать внешние процедуры и функции из Pascal, а процедуры и функции Pascal — из ассемблерной программы. Следует быть внимательным в отношении того, какие процедуры и функции являются NEAR, а какие — FAR. Можно также импортировать данные из Pascal во внешние модули, но, поскольку в Pascal отсутствует директива EXTERNAL для переменных, нельзя экспортировать данные из ассемблера в Pascal. (Можно передать адреса внешних данных в Pascal и, таким образом, получить доступ к внешним переменным.)

Написание функций Pascal требует особой внимательности, чтобы быть уверенным в том, что соответствующие значения передаются обратно в вызывающие процедуры в верные регистры. Параметры еще больше усложняют работу по написанию внешней программы, требуя, чтобы ассемблерные модули адресовали переменные в стек. Применение директивы ARG может помочь в этом случае (особенно при использовании модели памяти TPASCAL), давая возможность внешней программе адресовать параметры именами, а не чреватыми ошибками выражениями типа [bp+8]. С Borland Pascal TPASCAL не требуется.

Безусловно, конечной целью подключения языка ассемблера к Pascal является увеличение быстродействия программ. Как показывают примеры в этой главе, результаты оптимизации могут экономить память, уменьшать размеры файла, содержащего программу, и значительно улучшать ее характеристики.

Упражнения

12.1. Что такое "критичный код"?

12.2. Что делает профайлер?

12.3. Машинный код инструкции clc — 0F8h, инструкции stc — 0F9h. Напишите операторы и процедуры Inline, использующие эти инструкции для установки и сброса флага.

12.4. Назовите правильный способ написания внешнего ассемблерного модуля, соответствующего следующему объявлению процедуры Pascal:

```
{$F+}
PROCEDURE PlayBall
```

12.5. Предположим, у вас имеется внешний ассемблерный модуль, названный NEWSTUFF.ASM, подключенный к NEWSTUFF.OBJ. В нем имеется одна процедура OldStuff и целая функция OlderStuff. Какие операторы Pascal требуются для объединения файлов Pascal и ассемблера?

12.6. Почему в модели памяти TPASCAL имеется потенциальный недостаток?

12.7. Даны следующие объявления Pascal. Напишите директиву или директивы, требуемые для импортирования значений в ассемблерный модуль. Какое из следующих объявлений может быть импортировано во внешний модуль:

```
TYPE Month = (Jan, Feb, Mar, Apr, May,
      Jun, Jul, Aug, Sep, Oct, Nov, Dec);

CONST MaxLevel = 17;
      AreaCode : Integer = 555;
      Esc = #27;

VAR YourName : String;
       Score : Integer;
       SalesPerMonth : Array[ Month ] OF Integer;
```

12.8. Дана процедура Pascal. Напишите необходимые инструкции для вызова подпрограммы из ассемблерного модуля, названного ASCII.OBJ:

```
PROCEDURE WriteASCII( ch :Char );
BEGIN
      Writeln( 'ASCII value = ', Ord(ch))
END;
```

12.9. Предположим, что имеется глобальная переменная, объявленная посредством dd и названная LongValue. Какая требуется ассемблерная инструкция, чтобы передать это значение обратно в Pascal в качестве результата функции с объявленным типом?

12.10. (Усложненное). Применяя директиву ARG и исходя из предположения, что используется модель памяти TPASCAL, применительно к Turbo Pascal напишите ассемблерную программу, требуемую для замены процедуры Pascal LotsOfParams (показанной здесь) внешним модулем, загружающим параметры — a в cx, b в dx, прибавляющим 5 к числу и загружающим ch в al:

```
PROCEDURE LotsOfParams( a, b : Integer;
 VAR number : Integer; VAR ch : char );
```

Напишите программу Pascal для сравнения. (Подсказка: сначала напишите версию на Pascal, проверьте код с помощью Turbo Debugger и затем напишите ассемблерный модуль.)

Задания

12.1. Преобразуйте модуль STRINGS из главы 5 во внешние процедуры, которые можно подключить к программам Pascal, дополняя возможности Turbo Pascal возможностями ASCIIZ.

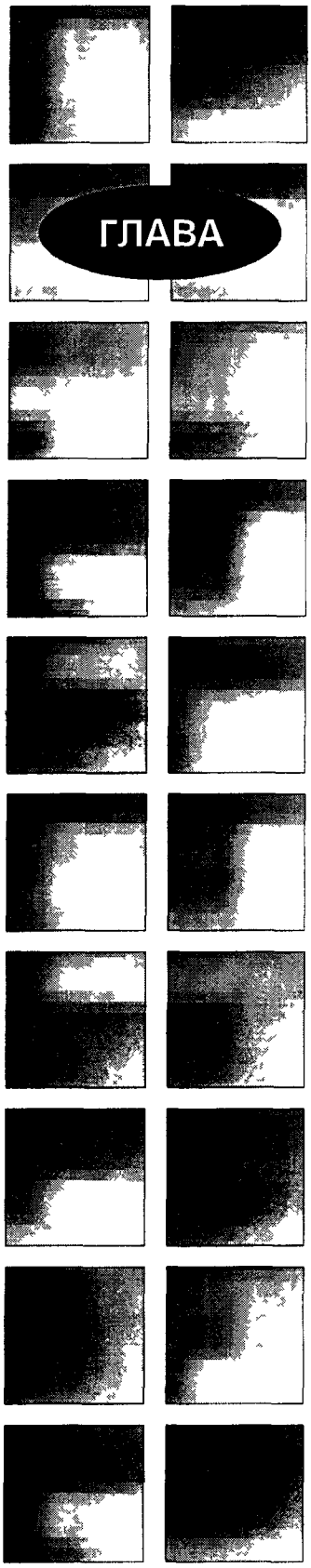
12.2. Используя внешние процедуры из модуля ASYNCH главы 10, напишите эмулятор терминала в Pascal для инициализации и управления последовательным портом I/O (ввода-вывода).

12.3. Как можно точнее определите критичный код в объемной программе Pascal, желательно содержащей до 1000 строк. Оптимизируйте ключевые процедуры и функции в программе и задокументируйте улучшения.

12.4. Write и Writeln Pascal вынуждены оперировать множеством параметров, целых, действительных чисел и строк. Они и оперируют ими, но при этом могут создавать неоправданно длинные последовательности инструкций в машинных кодах. Напишите упрощенные строчные процедуры для записи строчных переменных Pascal на стандартный выход.

12.5. Разработайте процедуру на языке ассемблера для отображения строк на ПК с высокой скоростью.

12.6. Используйте Turbo Debugger для проверки машинного кода стандартной утилиты CRT Turbo Pascal, поставляемой с большинством версий Turbo и Borland Pascal. Выявите и прокомментируйте как можно больше инструкций. (Это хороший способ изучить, каким образом Pascal компилирует программы — но не переживайте, если вы не сможете описать все инструкции.)



ГЛАВА 13

Использование языка ассемблера при программировании на C и C++

- Применение C и C++ совместно с языком ассемблера
- Определение критичного кода
- Использование регистров
- Ассемблерные операторы Inline
- Разделение данных
- Внешние программы на ассемблере
- Вызов ассемблерных функций из C
- Ассемблирование и компоновка внешних модулей
- Отладка многоязыковых программ
- Вызов функций C из языка ассемблера
- Результаты выполнения функции
- Локальные переменные
- Вызов функций C++ из ассемблера
- Передача аргументов функции
- Использование классов C++ с языком ассемблера
- Резюме
- Упражнения
- Задания

Применение C и C++ совместно с языком ассемблера

Причины подключения модулей на языке ассемблера к программам на Turbo C и Borland C++ те же самые, что и обсуждавшиеся в главе 12, — это скорость и получение доступа к самым нижним уровням оборудования. Однако и ловушки, подстерегающие на этом пути, тоже идентичны — снижение переносимости программ и повышение вероятности сбоев. Для большинства программ компиляторы Borland C и C++ генерируют компактный и быстрый машинный код, с которым трудно конкурировать. И все же ни один компилятор не может сравниться по "уму" с опытным специалистом по языку ассемблера. Во многих случаях единственный способ для прибавления программе скорости — немного дополнить совершенный "C", используя один из двух методов:

- Использование встроеных операторов (операторов Inline)
- Использование внешних функций

Операторы Inline вставляют язык ассемблера непосредственно в исходную программу на C и C++. Эта методика быстрая и простая, но имеет несколько недостатков, которые я поясню ниже. *Внешние функции*, хотя их и труднее использовать, чем операторы inline, имеют то преимущество, что позволяют наиболее полно использовать все возможности Turbo Assembler. В этой главе рассматриваются оба метода и приведено много примеров, которые можно применять в качестве заготовок для ваших собственных проектов.

Замечание

Готовя материал для этой главы, я исходил из предпосылки, что вы имеете некоторое представление о C и C++, а также знаете, как инсталировать и запустить свой компилятор. Примеры из этой главы можно использовать с большинством версий Turbo Assembler, Turbo C, Turbo C++ и Borland C++, хотя некоторые примеры в конце главы требуют наличия Borland C++ и Turbo Assembler версий 4.0 и более поздних. И еще, если, читая эту главу, вы почувствуете, что вам слегка не хватает опыта, не отчаивайтесь. Несколько абзацев из главы 12 умышленно здесь повторяются.

Определение критичного кода

Как пояснялось в главе 12, критичный код программы обычно составляет около 10% всех ее инструкций, которые зачастую выполняют до 90% всех действий. Переписывание этих критичных 10% на быстрый язык ассемблера может дать просто поразительное увеличение быстродействия, в то время как оптимизация остальных 90% будет пустой тратой времени. Поэтому первейшим правилом при использовании нескольких языков является следующее: не нужно переписывать операторы, которые и так работают достаточно быстро.

Нахождение критичного кода программы — не всегда простая задача. Иногда ваше знание программы подскажет вам, из каких участков можно еще кое-что выжать. В других случаях для прослеживания программы и создания статистического отчета с перечнем перегруженных подпрограмм вам может понадобиться помощь профайлера типа Turbo Profiler Borland.

Хорошим стратегическим подходом является написание программы полностью на C или C++, а затем, после отладки кода, преобразование отдельных участков на язык ассемблера. Во время этой работы отслеживайте результаты и старайтесь свести объем ассемблерных процедур к минимуму. Это может показаться странным советом для книги, посвященной ассемблеру, но одной из главных причин для написания программ на языке высокого уровня является стремление повысить возможность переноса кода на другой компьютер. Чтобы такая возможность существовала, вероятно, лучше использовать ассемблер как можно меньше. (Кроме того, к маленькой программе на машинных кодах ведет долгий путь разработки.)

Использование регистров

В своих ассемблерных подпрограммах вы можете использовать все регистры процессора. Но чтобы предотвратить путаницу с функциями C и C++, необходимо восстанавливать значения bp, cs, ds, sp и ss, которые они имели до запуска вашей подпрограммы. Тогда можно быть совершенно уверенными, что обращение к другим функциям не изменит эти регистры.

К регистрам ax, bx, cx, dx, di, si и es можно обращаться свободно и не нужно резервировать их значения до окончания вашей подпрограммы. Эта свобода касается и других функций, поэтому помните, что эти регистры могут изменяться, если вы вызываете функции C или C++ из ассемблерных подпрограмм.

Компилятор использует di и si для регистровых переменных, поэтому, если вы используете любой из этих регистров в процессе выполнения ассемблерных операторов, компилятор отключает оптимизацию регистровых переменных, избегая возможного конфликта с вашей программой. К несчастью, это может также привести к замедлению любой программы, которую вы пытаетесь ускорить. Поэтому обычно лучше избегать использования di и si, если только в этом не возникает крайней необходимости. При подключении внешних ассемблерных модулей только от вас зависит резервирование si и di для других функций, использующих регистровые переменные.

Ассемблерные операторы Inline

Ассемблерные операторы inline начинаются словом asm, за которым следует инструкция ассемблера и ее операнды. Например, чтобы синхронизировать программу с внешним сигналом прерывания, можно написать:

```
/* ожидание прерывания */
asm sti
asm hlt
printf("Прерывание получено\n");
```

Когда ранние версии Turbo C компилируют программу со встроенными командами asm, компилятор сперва создает ассемблерный текст для *всей* программы, вставляя в текст ваши ассемблерные инструкции вместе с откомпилированным кодом для остальных операторов C. Затем компилятор вызывает Turbo Assembler и Linker (компоновщик), чтобы провести ассемблирование и подключить программу к конечному файлу кода. Более поздние версии Turbo и Borland C++ могут компилировать операторы asm без вызова TASM. Полный синтаксис asm:

```
asm[метка] мнемоника/директива операнды [,] [/* C комментарий */]
```

Необязательная *метка* допускается только для директив, оперирующих данными. Нельзя помечать мнемонику инструкций. Например, чтобы создать переменную типа слова, названную ForWord, можно написать:

```
asm ForWord dw ?
```

Чтобы пометить функцию в инструкции, необходимо использовать *метку C* — идентификатор, за которым следует колонка:

```
ThisLocation
asm inc ax

asm or ax, ax
asm jz ThisLocation
```

Мнемоникой/директивой может быть любая допустимая инструкция языка ассемблера или директива Turbo Assembler. *Операнды* мнемоники те же, что и при использовании в "чистом" ассемблерном коде. Например, можно увеличить целую переменную с именем Level посредством следующей команды:

```
int Level
asm inc [word Level],
```

Обратите внимание на то, что определитель word необходим для сообщения ассемблеру размера Level. Следует прибавлять word, byte, tbyte и другие определители только тогда, когда размер переменной не ясен. В тех же случаях, когда он не вызывает сомнения, определитель можно опустить:

```
int Bevel,
asm mov ax, [Bevel],
```

Приведенная команда помещает значение Bevel в ax. Поскольку ax является регистром слова, Turbo Assembler заключает, что и Bevel имеет такой же размер Кроме того, как показано здесь, нет нужды беспокоиться о том, *где* размещаются переменные — одни и те же построения ассемблера применимы независимо от того, помещаются переменные в стеке или в сегменте данных, достаточно использовать имена переменных подобно приведенным примерам.

Точка с запятой в конце оператора ассемблера необязательна. Не следует путать точку с запятой с символом комментария в ассемблере — перед ассемблированием компилятор удаляет точку с запятой. Поэтому для комментариев ассемблерных операторов необходимо применять стиль, используемый в C, подобно следующему:

```
int Swivel,
int Drivel,
asm mov cx, [Swivel]  /* Загрузка cx значением Swivel */
asm shl ax, cl /* Сдвиг ax влево на значение, хранимое в cl */
asm mov [Drivel], ax, /* Сохранение сдвинутого ax в Drivel */
```

Точки с запятыми в конце строк asm и комментарии C, расположенные между /* и */, удаляются из текста перед ассемблированием. Я предпочитаю опускать точки с запятыми, как в среднем операторе asm в этом примере.

Компиляция и ассемблирование кода Inline

Существует несколько способов компиляции программ C со встроенными инструкциями asm. Чтобы продемонстрировать отличия между этими методами, обратимся к тексту программы 13.1, TALLY.C. Замечания, приведенные после текста программы, поясняют, как осуществить компиляцию программы.

Программа 13.1. TALLY.C

```
1: /* TALLY.C -- Короткий пример с применением ассемблерных операторов inline */
2:
3: #include <stdio.h>
4:
5: int main(void)
6: {
7: int votes;
8: int tally;
9:
10: votes = 100;
11: tally = 500;
12: printf("Tally : %d\n", tally);
13: asm mov ax, [votes];
14: asm add [tally], ax;
15: printf("Tally : %d\n", tally);
16: return 0;
17: }
```

Как откомпилировать TALLY.C

В TALLY.C используются два встроенных оператора asm, чтобы сложить значение целой переменной votes с другой переменной tally, производя то же самое действие, что и оператор C:

```
tally = tally + votes
```

В ранних версиях Turbo C для компиляции, ассемблирования и компоновки программы используется команда

```
tcc tally
```

Для этого необходимо воспользоваться компилятором TCC.EXE из командной строки ДОС. Вы не сможете использовать интегрированный редактор и программу компиляции TC.EXE для компиляции программ, содержащих операторы Inline. Во время компиляции, когда Turbo C доходит до первого оператора asm, он выводит "Warning 13" ("Предупреждение 13") и начинает компиляцию программы сначала. Обычно Turbo C сперва проводит компиляцию непосредственно в файлы с объектным кодом, а затем вызывает Turbo Linker для компоновки объектных и библиотечных модулей программы в окончательном файле с .EXE кодом. Однако из-за наличия встроенных операторов asm Turbo C компилирует исходный файл в текстовый типа .ASM, в данном случае создавая файл TALLY.ASM. Этот файл содержит полностью всю программу C, переведенную на язык ассемблера и объединенную с операторами asm. Затем Turbo C вызывает Turbo Assembler для ассемблирования TALLY.ASM в файл TALLY.OBJ. После удаления с диска TALLY.ASM компилятор вызывает Turbo Linker для компоновки TALLY.OBJ с соответствующими библиотечными и другими файлами, создавая конечный программный файл TALLY.EXE — другими словами, даже столь коротенькая команда порождает массу действий.

Проблема при использовании этого метода состоит в том, что тратится время на компиляцию программы вплоть до первого оператора asm, когда Turbo C наконец "понимает", что ему нужно вместо этого генерировать файл с программой на ассемблере. Можно избежать этого, указав в командной строке –B. (B должна быть заглавной.) Например:

```
tcc -B tally
```

компилирует TALLY.C в TALLY.ASM, ассемблирует TALLY.ASM в TALLY.OBJ и компонует TALLY.OBJ с библиотечным файлом, чтобы создать TALLY.EXE. После этого TALLY.ASM удаляется с диска. Для сохранения текста ассемблерной программы следует использовать опцию –S (которая также должна быть заглавной):

```
tcc -S tally
```

Эта команда компилирует TALLY.C в TALLY.ASM, но не проводит ассемблирования и компоновки в конечную программу. Этой командой следует пользоваться, если вы хотите проверить ассемблер, сгенерированный Turbo C, что даст вам возможность подробнее разобраться с инструкциями, использованными для выполнения таких команд, как цикл и вызов функций. После проверки ассемблерного текста программы повторите компиляцию с командой –B для создания конечной программы. (Можно также ассемблировать TALLY.ASM отдельно, но тогда вам придется запускать Turbo Linker, чтобы скомпоновать TALLY.OBJ с соответствующей библиотекой Turbo C.)

Более поздние версии Turbo C++ и Borland C++ могут компилировать операторы asm непосредственно, без создания промежуточного текстового .ASM файла. Поскольку компиляторы C++ обычно ожидают имена файлов, оканчивающиеся расширением .CPP, для программ C необходимо указывать .C. Например, чтобы скомпилировать TALLY.C с Borland C++4 или 4 5, наберите следующую инструкцию вслед за приглашением DOS (замените bcc на tcc, если вы используете Turbo C++):

```
bcc tally c
```

Вы по-прежнему можете создать промежуточный текст на ассемблере и предоставить компилятору вызвать Turbo Assembler для ассемблирования программы, определив опцию –B:

```
bcc -B tally c
```

Можно воспользоваться –S для сохранения ассемблерного файла (TALLY.ASM) для проверки его с помощью текстового редактора:

```
bcc -S tally c
```

Прагматичное ассемблирование

Другим методом компилирования программ, подобных TALLY.C, включающих в себя команды asm, является вставка строки

```
#pragma inline
```

в начале модуля. Чтобы проверить это, вставьте #pragma inline в TALLY.C между строками 1 и 2 (или в качестве первой строки) и откомпилируйте модуль следующей командой:

```
tcc tally
```

Как легко убедиться, директива #pragma — ANSI C стандартный метод для активирования пользовательских свойств компилятора — выполняет ту же работу, что и опция –B в командной строке, экономя время, которое в противном случае было бы потрачено попусту при перезапуске компилятора после достижения первого оператора asm.

Этот же метод можно применять при использовании самых последних версий Turbo C++ и Borland C++. Однако, поскольку эти компиляторы могут компилировать операторы asm непосредственно, обычно нет смысла использовать директиву #pragma inline. Ее применение приводит к тому, что компиляторы генерируют промежуточные .ASM файлы и обращаются к Turbo Assembler. Если у вас нет веских причин для компиляции программ этим способом, то это приведет только к лишней трате времени.

Размещение данных и операторов в тексте программы

Каждая строка текста программы C и C++ находится либо внутри, либо снаружи функции, и операторы asm могут вставляться как в одном, так и в другом месте. Конкретное положение оператора asm влияет на то, куда ассемблируется код или директива. Если оператор asm появляется снаружи функции, то он ассемблируется в сегмент данных программы, если внутри функции —в кодовый сегмент. Обычно, чтобы создать переменные, операторы asm вставляются снаружи функций; для создания кода их следует вставлять внутрь функций. Ниже приведены примеры обоих случаев:

```
asm count db ?
int main()
{
 asm shl [count], 1 /* умножение count на 4 */
 asm shl [count], 1
 return 0;
}
```

Переменная count объявляется в сегменте данных программы (относительно ds). Операторы внутри функции main умножают count на 4, используя вместо mul быстрые инструкции сдвига. Если вы объявляете переменные внутри функции, данные ассемблируются в кодовый сегмент, требуя особого обхождения:

```
int main()
{
 asm jmp OverThere
 asm count db ?
OverThere:
 asm shl [count], 1 /* умножение count на 4 */
 asm shl [count], 1
 return 0;
}
```

Поскольку теперь переменная count находится в кодовом сегменте, требуется инструкция jmp, чтобы избежать случайного восприятия значения count в качестве машинного кода и его исполнения. Обратите внимание, что обращения shl к count остались неизменными — компилятор автоматически помещает ссылку на сегмент (в данном случае cs:), как это требуется для адресации переменных в соответствующие сегменты.

Разрешение инструкций 80286/386

Инструкции 80286 и 80386 могут быть разрешены вставкой в программу соответствующих директив Turbo Assembler. Например, чтобы включить незащищенные инструкции 80286, применяйте команду

```
asm .286C
```

Не забудьте использовать формат MASM вместо его эквивалента P286N для Ideal-mode, иначе вы также включите и режим Ideal. Если вы все же делаете это, не забудьте выполнить обратное переключение на режим MASM, который используется для выходных файлов самого компилятора:

```
asm Ideal /* Включение режима Ideal */
asm P286N /* Разрешение незащищенных инструкций 80286*/
asm MASM  /* Переключение обратно в режим MASM */
```

Разделение данных

Inline операторы asm имеют свободный доступ к переменным и структурам C и C++ — одно из самых привлекательных преимуществ метода inline по сравнению с подходом посредством внешних модулей (этот метод описывается в данной главе дальше). В табл. 13.1 перечислены типы данных C и C++, приведены определители ассемблера, которые необходимо использовать в случаях неясных ссылок, число байтов, занимаемых переменными каждого типа, и эквивалентные директивы Turbo Assembler для создания переменных такого же размера. Обратите внимание, что dq может использоваться для создания инициализированных переменных двойной длины с плавающей точкой в языке ассемблера, но в Turbo Assembler отсутствуют директивы для непосредственного создания переменных с плавающей точкой.

В операторах asm можно ссылаться на переменные типов, приведенных в табл. 13.1.

Например:

```
unsigned char initial;

initial = 'T';
asm mov dl, [initial]   /* Загрузка символа в dl */
asm mov ah, 2           /* Пересылка символа в ДОС */
asm int 21h             /* Стандартная выходная функция */
```

Таблица 13.1. **Типы данных C и C++**

Типы данных	Определитель	Байты	Директива
unsigned char (беззнаковый символьный)	Byte ptr	1	db
char (символьный)	Byte ptr	1	db
enum (число)	Word ptr	2	dw
unsigned short (беззнаковый короткий)	Word ptr	2	dw
short (короткий)	Word ptr	2	dw

Окончание таблицы 13.1

Типы данных	Определитель	Байты	Директива
unsigned int (беззнаковый целый)	Word ptr	2	dw
int (целый)	Word ptr	2	dw
unsigned long (беззнаковый длинный)	Dword ptr	4	dd
long (длинный)	Dword ptr	4	dd
float (плавающий)	Dword ptr	4	-
double (двойной)	Dword ptr	8	dq
long double (длинный двойной)	Tbyte ptr	10	dt
near * (ближний)	Word ptr	2	dw
far * (дальний)	Dword ptr	4	dd

Беззнаковая символьная переменная initial загружается оператором asm в dl. Из табл. 13.1 видно, что, поскольку и dl, и беззнаковые символьные данные имеют длину один байт, нет необходимости в ссылке использовать определитель Byte, хотя его применение и не будет ошибкой:

```
asm mov dl, [Byte ptr initial]
```

Скобки обычно применяются для указания не на значение метки, а на ссылку в память (т.е. на адрес), в результате оператор ассемблера будет выглядеть следующим образом:

```
mov dl, [[Byte ptr initial]]
```

Двойные скобки не вызывают никаких проблем, поэтому не стоит обращать на них внимания. (Если только вы не осуществляете компиляцию с опцией –S, вы все равно не увидите этих скобок.) Можно избежать странного поведения двойных скобок, не используя их в операторе asm —

```
asm mov dl, initial
```

хотя теперь программа стала менее понятной. (К чему относится initial — к значению или к адресу этой переменной?)

Объявление ассемблерных данных

Можно объявить переменные для использования только ассемблерными операторами. Например, чтобы создать 16-битовое слово с именем TwoBytes и загрузить значение переменной в cx, можно написать:

```
asm TwoBytes db 1, 2
int main()
{
 asm mov cx, [Word ptr TwoBytes]
 return 0
}
```

Переменная TwoBytes объявляется в сегменте данных программы (снаружи функции), с использованием директивы db, чтобы хранить в памяти 2 байта (1 и 2). Оператор ассемблера затем загружает TwoBytes в cx. Определитель *Word ptr* необходим для ссылки на TwoBytes как на 16-битовое слово.

Поскольку TwoBytes объявляется в операторе asm, на эту переменную нельзя ссылаться в тексте программы C или C++. Поэтому, если только вам не требуются отдельные переменные для ассемблерных инструкций, следует объявлять их обычным образом и ссылаться на них из ассемблерного модуля.

Структуры C

Групповые имена (поля) в структурах во внутреннем представлении хранятся в виде значений смещений от начала структуры. Например, структура

```
struct PersonRec {
 char Name[50];
 char Address[60];
 char CityStZip[60],
 char AgeInYears;
} Person;
```

присваивает значения смещений Name, Address и CityStZip, представляя положение этих полей в структуре PersonRec. Помня об этом, необходимо при ссылке на поля структуры в ассемблерном операторе использовать и переменные, и групповые идентификаторы, разделенные точкой:

```
asm mov si, offset Person.Address
```

что ассемблируется в :

```
mov si, 0038h
```

Здесь 0038h (в вашей системе при просмотре в Turbo Debugger может быть другое значение) представляет собой смещение от начала сегмента данных для поля Address, т.е. Person+Address. Сравните это с инструкцией

```
asm mov al, Byte ptr Person AgeInYears
```

которая ассемблируется в

```
mov al, Byte ptr DGROUP:_Person + 170
```

В таком случае значение поля AgeInYear загружается в al. 170 представляет собой значение смещения этого поля от начала записи Person. (Компилятор прибавляет к именам переменных символы подчеркивания — но об этом позже.)

Часто вы хотите сослаться на структуру с помощью указателей, обычно загруженных в bx. Например, чтобы инициализировать bx адресом записи Person, следует воспользоваться оператором

```
asm mov bx, offset Person
```

Адресуя Person посредством ds:bx, можно теперь загрузить значения или адреса остальных полей относительно указателя:

```
asm mov dl, [bx.AgeInYears]
```

Никакой определитель размера типа Byte ptr не требуется, поскольку и поле, и регистр обладают одинаковым размером.

Если две или больше структур обладают идентичными именами полей, необходимо отличить сомнительные ссылки на указатели, добавив перед именами полей имя структуры в круглых скобках. Например, предположим, что существует еще одна запись, названная Customers, имеющая поле CityStZip — с тем же именем, что и в структуре PersonRec. Чтобы загрузить si смещением адреса поля CityStZip от переменной TheBakery из Customers, адресуемой bx, можно написать:

```
asm mov bx, offset TheBakery
asm lea si, [bx.(struct Customers) CityStZip]
```

Первая команда asm загружает bx смещением адреса TheBakery, вторая — загружает si действительным адресом поля CityStZip относительно bx. Имя структуры в круглых скобках дает возможность компилятору различить двусмысленную ссылку по имени поля на CityStZip.

Разделение кода

Ассемблерные операторы inline могут вызывать функции C и C++, а операторы C и C++ обращаться к функциям, написанным полностью на ассемблере. Я начну с более простой из двух методик, показав, как писать законченную функцию на ассемблере и вызывать эту функцию с помощью операторов C. Откомпилируйте, ассемблируйте и скомпонуйте программу 13.2, UPDOWN.C посредством команды

```
tcc -v updown c
```

Если вы применяете Borland C++, замените tcc на bcc.

Опцию -v необходимо использовать только в том случае, если вы хотите проверить исходный текст при запуске программы в Turbo Debugger. Если же вы хотите проверить результирующий ассемблерный модуль, введите следующую команду и воспользуйтесь своим испытанным редактором для просмотра файла UPDOWN.ASM:

```
tcc -S updown.c
```

Программа 13.2. UPDOWN.C

```
1: /* Демонстрация ассемблерной функции Inline */
2:
3: #pragma inline
4:
5: #include <stdio.h>
6: #include <string.h>
7:
8: extern void BumpStrUp(unsigned char far * TheString,
9: int StringLength);
10:
11: extern void BumpStrDown(unsigned char far * TheString,
12: int StringLength);
13:
14: char *MixedUp = "UppER aNd LOwEr CaSE";
15:
16: int main()
17: {
18: printf("Before BumpStrUp: %s\n", MixedUp);
19: BumpStrUp( MixedUp, strlen(MixedUp) );
20: printf("After BumpStrUp. %s\n", MixedUp);
21: BumpStrDown( MixedUp, strlen(MixedUp) );
22: printf("After BumpStrDown: %s\n", MixedUp);
23: return 0;
24: }
25:
26: void BumpStrUp(unsigned char far * TheString,
27: int StringLength)
28: {
29: asm les di, TheString                    /* Адресация строки посредством es:di */
30: asm mov cx, StringLength                  /* Загрузка длины строки в cx */
31: asm jcxz Exit                            /* Выход, если длина = 0 */
32: asm cld                                  /* Автоувеличение di */
33: NextChar:
34: asm mov al, es:[Byte ptr di]             /* Загрузка следующего символа */
35: asm cmp al, 'a'                          /* Прерывание преобразования, если */
36: asm jb NotLower                          /* символ не */
37: asm cmp al, 'z'                          /* строчный */
38: asm ja NotLower
39: asm sub al, 32                           /* Преобразование в прописной */
40: NotLower:
41: asm stosb                                /* Накапливание символа в строку */
42: asm loop NextChar                        /* Цикл до завершения */
43: Exit:;
44: }
45:
46: void BumpStrDown( unsigned char far * TheString,
47: int StringLength )
48: {
49: asm les di, TheString                    /* Адресация строки посредством es:di */
50: asm mov cx, StringLength                  /* Загрузка длины строки в cx */
51: asm jcxz Exit                            /* Выход, если длина = 0 */
52: asm cld                                  /* Автоувеличение di */
53: NextChar:
54: asm mov al, es:[Byte ptr di]             /* Загрузка следующего символа */
55: asm cmp al, 'A'                          /* Прерывание преобразования, если */
56: asm jb NotUpper                          /* символ не */
57: asm cmp al, 'Z'                          /* прописной */
58: asm ja NotUpper
59: asm add al, 32                           /* Преобразование в строчный */
60: NotUpper:
61: asm stosb                                /* Накапливание символа в строку */
62: asm loop NextChar                        /* Цикл до завершения */
63: Exit:;
64: }
```

Как работает UPDOWN.C

Строки 8–12 объявляют две внешние функции — BumpStrUp и BumpStrDown, которые преобразуют строки полностью в прописные либо строчные символы. Для удобства функции приведены вместе с основной программой, но они могут находиться также и в отдельных модулях, если вы готовы справиться со всеми нюансами компоновки модулей для создания конечного исполняемого программного файла.

Основная функция (строки 16–24) вызывает внешние функции, оказывая влияние на строчную переменную (строка 14), адресуемую дальним указателем. Функция BumpStrUp (26–44) приводит два параметра, дальний символьный указатель и целое число, представляющее собой длину строки. Первая ассемблерная инструкция (строка 29) использует les для загрузки в регистры es.di полного 32-битового адреса строки Назначение остальных инструкций вы сможете понять из комментариев, приведенных в правой части большинства строк.

Строка 43 иллюстрирует неприятие меток в ANSI C, где определено, что вслед за меткой должен следовать оператор. Поскольку ваш ассемблерный код нуждается в методе перехода на конец функции, это создает проблему, которая здесь решается с помощью лишней точки с запятой после метки Exit.

Функция BumpStrdown (строки 46–64) почти полностью идентична BumpStrUp, за исключением строк 55–59, которые преобразуют прописные буквы в строчные.

За кулисами

UPDOWN C таит в себе несколько сюрпризов, которые не очевидны из текста программы. Как легко убедиться, если проанализировать выходной текст на ассемблере, BumpStrUp и BumpStrDown начинаются инструкциями:

```
push bp          ; Сохранение в стеке
mov bp, sp             ; Адресация стека посредством bp
push si          ; Сохранение si
push di          ; Сохранение di
```

Первые две инструкции сохраняют bp перед приравниванием этого же регистра sp, подготавливая адресацию параметров в стек. Вторая и третья инструкции сохраняют значения si и di. Это происходит потому, что функции используют di, таким образом, компилятор выбирает безопасный путь и сохраняет и si, и di, чтобы избежать всякой возможности конфликта между любыми регистровыми переменными, используемыми другими подпрограммами, которые могут вызывать данную. Позднее обе функции заканчиваются следующими инструкциями:

```
pop di           ; Восстановление сохраненного di
pop si           ; Восстановление сохраненного si
pop bp           ; Восстановление сохраненного bp
ret              ; Возврат в точку вызова
```

Тем самым восстанавливаются первоначальные значения di, si и bp перед тем, как вернуться к инструкции, следующей за вызовом данной функции.

Как легко понять из сказанного выше, при использовании встроенных операторов asm компилятор берет на себя заботу о нюансах, связанных с адресацией параметров, сохраняя и восстанавливая регистровые переменные, держа стек "в порядке" и манипулируя bp. Хотя это действительно очень удобно, но имеются и недостатки в столь навязчивой помощи. Во-первых, ни одна обычная функция не использует si; следовательно, сохранение и восстановление этого регистра является пустой тратой времени. Во-вторых, в данном случае нет никакой необходимости в сохранении и восстановлении di, поскольку основная программа, вызывающая обычные функции, не содержит регистровых переменных и никакой конфликт не может произойти при изменении di.

Для лучшего управления подобными нюансами и во избежание необходимости предварять каждый ассемблерный оператор ключевым словом asm можно написать внешние ассемблерные модули для подключения к программам Turbo C. Это требует больших затрат труда, однако результат часто стоит хлопот.

Внешние программы на ассемблере

Поскольку компиляторы — и Turbo Assembler, и Borland C, и Borland C++ могут создавать одинаковый формат программных .OBJ файлов, можно писать одни части программы на C или C++, а другие — на языке ассемблера, а затем использовать Turbo Linker для компоновки файлов с объектным кодом в конечную .EXE программу. Компиляторы также способны запускать ассемблер и компоновщик непосредственно, упрощая компиляцию, по крайней мере для относительно небольших программ. Несмотря на дополнительную сложность программирования, методы с использованием внешних ассемблерных модулей имеют несколько преимуществ по сравнению с inline операторами asm:

- Уменьшают время компиляции
- Ассемблерные модули могут использовать режим Ideal
- Не прибавляются никакие "замаскированные" инструкции

- Программы C и C++ сохраняют высокую степень переносимости
- Внешние подпрограммы могут отлаживаться отдельно
- Внешние подпрограммы могут использоваться с другими языками программирования

Время компиляции уменьшается, поскольку компилятор больше не должен генерировать файл с ассемблерным текстом, что требуется при встроенных inline операторах asm. Во внешних ассемблерных модулях можно использовать предпочтительный режим Ideal, что также помогает Turbo Assembler запускаться быстрее. Не прибавляются никакие дополнительные инструкции, манипуляции со стеками или инструкции pop и push — то, что компилятор вставляет в inline функции asm где надо и не надо.

Если писать программы на чистом C или C++, а затем выборочно преобразовывать отдельные функции на ассемблер, можно улучшить переносимость программы. После оптимизации, если необходимо перенести программу на другой компьютер, например на Macintosh с процессором 68000, относительно просто заменить оптимизированные ассемблерные модули исходным текстом программы на C, предусмотрительно сохраненным на диске. Затем, когда программа правильно работает на новом компьютере, можно начать оптимизацию участков программы на родном языке этого компьютера.

Внешние ассемблерные подпрограммы также могут упростить отладку. Можно ассемблировать и отлаживать внешние подпрограммы отдельно от основной программы, что является гораздо более легкой задачей, чем охота за маленькими чудовищами в джунглях 80K программного файла. Кроме того, вы сможете использовать свои внешние подпрограммы совместно с другими языками программирования. Несмотря на столь большие преимущества, имеется несколько "крючков", о которых следует помнить при использовании внешних подпрограмм:

- Вы больше не сможете использовать вперемежку инструкции C и операторы ассемблера, как это можно было делать с операторами asm. Вам придется функции полностью писать на ассемблере.
- Вы должны хорошо разбираться в сегментах и сегментных регистрах, режимах адресации, упрощенных моделях памяти и в директивах, имеющих к этому отношение. (Безусловно, вы самым внимательным образом прочли каждое слово в этой книге, и все эти нюансы не составят для вас ни малейшей проблемы.)
- Шаги по компиляции и компоновке программы могут оказаться более сложными, хотя компилятор и может помочь в этом, запуская ассемблер и компоновщик непосредственно.

Упрощенные модели памяти

Внешние подпрограммы хороши тем, что "запутанные" директивы SEGMENT оказываются совершенно ненужными. Имена сегментов, классов и других опций, связанных с сегментами, идентичны для моделей памяти C, C++ и Turbo Assembler. Это означает, что можно упростить директивы моделей памяти, такие как DATASEG, CODESEG, FARDATA и CONST, для организации сегментов данных и кодовых сегментов своего ассемблерного модуля. Если вам необходимо объявить сегменты вручную, вы можете это делать до тех пор, пока следуете соглашениям, принятым для компилятора и компоновщика. Однако сложно представить ситуацию, когда это необходимо, поэтому здесь не будут рассматриваться такие нюансы. Для получения более подробной информации обратитесь к руководствам и справочникам по C и C++.

Программа 13.3, CSHELL.ASM, показывает один из многих возможных способов организации внешних ассемблерных модулей. Вы сможете использовать CSHELL в качестве шаблона для своих собственных программ, помещая различные процедуры там, где это указано посредством комментариев в тексте программы. Нет смысла ассемблировать эту программу — она не выполняет никаких полезных действий, но вы можете ассемблировать ее с помощью команды

```
tasm /ml cshell
```

Опция /ml сообщает Turbo Assembler, что необходимо включить режим распознавания строчных и прописных символов. Это соответствует тому, как работают компиляторы C и C++, если рассматривать такие имена, как MyFunction и myfunCTion, в качестве *различных* идентификаторов.

Программа 13.3. CSHELL.ASM

```
1:  %TITLE "Shell for C .OBJ modules -- by Tom Swan"
2:
3:      IDEAL
4:
5:      MODEL small
6:
7:  DATASEG
8:
9:  ;----- Поместите здесь объявление данных PUBLIC
10:
11: ;----- Поместите здесь объявление данных EXTRN
```

И наоборот, переменная из программы может импортироваться ассемблерным модулем. Все символы в C и C++ имеют атрибут PUBLIC; таким образом, следующий текст только объявляет переменную обычным образом:

```
/* В программе C или C++. */
int NewValue;
main()
{
 NewValue = 500;
}
```

Далее, в ассемблерном модуле, чтобы импортировать NewValue, следует внутрь сегмента данных поместить директиву EXTRN:

```
DATASEG
EXTRN _NewValue Word
```

Теперь можно использовать _NewValue в ассемблерных операторах. Например, чтобы скопировать значение *импортированной переменной* _NewValue *в переменную типа слова* — AsmValue, *объявленную в ассемблерном* модуле, можно использовать следующие команды в кодовом сегменте:

```
CODESEG
mov ax, [_NewValue]     ; Загрузка переменной в ax
mov [_AsmValue], ax     ; Копирование переменной в ассемблерный модуль
```

Кодовый сегмент (строки 21–50) включает в себя место для внешней функции. Строка 32 объявляет имя функции, которое должно быть объявлено строкой общего пользования:

```
PUBLIC funcname
```

Предполагаемая функция объявляется NEAR (строка 32), указывая на то, что код будет храниться в том же сегменте, что и вызывающая инструкция для функции. Можно убрать или заменить NEAR на FAR, если предполагается вызывать функцию из другого сегмента.

Строки 33–34 и 46 готовят, сохраняют и восстанавливают bp *для адресации параметров в стек, используя метод*, который сейчас будет пояснен. Инструкции в строках 36–38 и 42–44 являются необязательными. Необходимо сохранять и восстанавливать si и di только в том случае, если эти регистры используются в функции. Кроме того, можно вычесть значение из sp, чтобы создать место для временных (локальных) переменных (см. строку 36), позже возвращая прежнее значение присвоением bp в sp (см. строку 44).

О символах подчеркивания

Как показывают некоторые из приведенных выше примеров, все символы PUBLIC и EXTRN должны начинаться символами подчеркивания. Это необходимо делать только в ассемблерном модуле (но не в исходной программе C или C++), поскольку компилятор добавляет подчеркивание ко всем глобальным символам, если только вы не используете опцию –u для компиляции программ. (Не применяйте эту опцию, если только вы не готовитесь также перекомпилировать всю используемую при выполнении C библиотеку, в которой предполагается, что все глобальные символы начинаются символом подчеркивания.) Если вы получаете во время компоновки сообщение об ошибке "undefined symbol" (неопределенный символ), то причиной может оказаться отсутствие подчеркивания в ассемблерном модуле.

Использование дальних данных

Если вы объявляете переменные в дальнем сегменте данных после ключевого слова FARDATA, то вы должны подготовить сегментный регистр к размещению переменных в памяти. (Подробнее этот вопрос рассмотрен в главе 11.) Прежде всего, вслед за директивой FARDATA необходимо объявить данные:

```
FARDATA
_OuterLimits dw ?
```

Затем, в кодовом сегменте, следует перед использованием переменной подготовить сегментный регистр. Одним из возможных подходов является использование оператора SEG для загрузки адреса дальнего сегмента данных:

```
mov ax, SEG _OuterLimits          ; Адресует дальний сегмент данных
mov es, ax                        ; посредством es
mov [es:_OuterLimits], dx         ; Резервируется dx для переменной
```

Можно также использовать заранее определенный символ @fardata:

```
mov ax, @fardata        ; Адресует дальний сегмент данных
mov es, ax              ; посредством es
mov [es:_OuterLimits], dx                ; Резервирует dx для переменной
```

Разделение кода

Вызов ассемблерных функций идентичен вызову функций C или C++. В качестве примера программа 13.4, CFILLSTR.C, объявляет внешнюю функцию, предназначенную для заполнения строки символами. Этот пример демонстрирует также, как можно заменить функции процедурами на языке ассемблера. Команды для выполнения компиляции и ассемблирования CFILLSTR я приведу несколько позже — как вы увидите, существует много способов проделать это.

Программа 13.4. CFILLSTR.C

```
1: /* Test CFILLSTR External Module -- by Tom Swan */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: extern void fillstring(unsigned char far * thestring,
7: int stringlength, char fillchar);
8:
9: char *test = "Filled to the brim";
10:
11: int main()
12: {
13: printf("Before fillstring. %s\n", test);
14: fillstring( test, strlen(test), '@' );
15: printf("After fillstring: %s\n", test);
16: return 0;
17: }
18:
19: /*
20: void fillstring( unsigned char far * thestring,
21: int stringlength, char fillchar )
22: {
23: int i;
24:
25: for (i = 0; i < stringlength; i++)
26: thestring[ i ] = fillchar;
27: }
28: */
```

Компиляция CFILLSTR.C

Временно удалите строки 19 и 28, активирующие функцию в строках 20–27. Позднее вы замените эту версию функции fillstring "на чистом C" оптимизированным ассемблерным модулем. Но сперва откомпилируйте и запустите программу с помощью команд:

```
tcc -v cfillstr
cfillstr
```

Если вы пользуетесь Borland C++, введите следующие команды (замените bcc на tcc для Turbo C++):

```
bcc -v cfillstr.c
cfillstr
```

Опцию –v используйте только в том случае, если хотите проверить программу с помощью Turbo Debugger. Вы можете также применить команду View|CPU для проверки машинного кода fillstring. Как иллюстрирует рис. 13.1, текст, сгенерированный компилятором, весьма компактный, но его можно сделать еще лучше. Обратите внимание, что в отличие от Inline операторов asm только si, но не di, сохраняется и восстанавливается — налицо небольшое улучшение. Но, несмотря на это, инструкции, подобные les внутри цикла for, не эффективны. Компилятор явно недостаточно сообразителен, чтобы "понять", что es больше нигде в цикле не меняется; следовательно, повторная инициализация регистра при каждом прохождении не нужна.

Замечание

В зависимости от версии Turbo C, Turbo C++ или Borland C++, реальный сгенерированный машинный код может отличаться от приведенного на рис. 13.1.

Вызов ассемблерных функций из C

Замените ограничители комментария /* и */ в строках 19 и 28 CFILLSTR.C, если вы удалили эти строки. Затем введите текст программы 13.5, CFILL.ASM, который содержит ассемблерную версию функции fillstring. Инструкции по ассемблированию модулей в законченную программу приведены после текста программы.

```
_fillstring: void fillstring( unsigned char far * thestring,
 cs:022c55   push   bp
 cs:022D 8BEC       mov    bp,sp
 cs:022F 56   push   si
CFILLSTR#37: for (i = 0; i < stringlength; i++)
 cs:0230 33F6 xor    si,si
 cs:0232 EB0A       jmp    023E
CFILLSTR#38: thestring[ i ] = fillchar;
 cs:0234 8F460F     mov    al,[bp + 0A]
 cs:0237 C45E04     les    bx,[bp + 04]
 cs:023A 268800     mov    es:[bx + si],al
 cs:023D 46   inc    si
 cs:023E 3B7608     cmp    si,[bp + 08]
 cs:0241 7CF1       jl CFILLSTR#38 (0234)
CFILLSTR#39:
 cs:0243 5E   pop    si
 cs:0244 5D   pop    bp
 cs.0245 C3   ret
```

Рис. 13.1. Функция fillstring из CFILLSTR.C после дизассемблирования с помощью Turbo Debugger

Программа 13.5. CFILL.ASM

```
1: %TITLE "Fill C Strings Demonstration -- by Tom Swan"
2:
3: IDEAL
4:
5: MODEL small
6:
7: CODESEG
8:
9: PUBLIC _fillstring
10:
11: ;-------------------------------------------------------
12: ; void fillstring( unsigned char far * thestring,
13: ; int stringlength, char fillchar)
14: ;-------------------------------------------------------
15: PROC _fillstring NEAR
16:
17: ARG thestring:Dword, stringlength:Word, fillchar:Byte
18:
19:       push   bp                      ; Сохранение старого указателя bp
20:       mov    bp, sp                  ; Адресация параметров
21:       mov    cx, [stringlength]      ; Присвоение длины строки cx
22:       jcxz   @@99                    ; Выход, если длина = 0
23:       push   di                      ; Сохранение di
24:       les    di, [thestring]         ; Адресация строки посредством es:di
25:       mov    al, [fillchar]          ; Присвоение заполняющего символа al
26:       repnz  stosb                   ; Накопление символов в строке
27:       pop    di                      ; Восстановление сохраненного di
28: @@99:
29:       pop    bp                      ; Восстановление сохраненного bp
30:       ret                            ; Возврат в точку вызова
31: ENDP _fillstring
32:
33:       END                            ; Конец модуля
```

Ассемблирование и компоновка внешних модулей

Теперь на диске у вас должны быть два файла — CFILLSTR C (с функцией fillstring, преобразованной обратно в комментарий) и CFILL.ASM, содержащий ассемблерный вариант этой же функции. Существует несколько методов ассемблирования, компиляции и компоновки отдельных модулей (и аналогичных многофайловых программ) для создания конечной .EXE программы. Проще всего предоставить проделать все это Turbo C:

```
tcc cfillstr cfill asm
```

Если вы используете Borland C++, введите следующую команду (замените bcc на tcc для Turbo C++):

```
bcc cfillstr c cfill asm
```

В любом случае команда сперва компилирует CFILLSTR.C в CFILLSTR.OBJ. Затем, распознав расширение имени файла .ASM как имя ассемблерного модуля, компилятор вызывает Turbo Assembler, чтобы ассемблировать CFILL.ASM в CFILL.OBJ. И наконец, компилятор вызывает Turbo Linker для объединения модулей с объектными кодами в CFILLSTR.EXE. Если компиляции и ассемблированию подлежит небольшое количество модулей, этот одношаговый метод наиболее прост для использования.

Замечание

Я умышленно не назвал ассемблерный модуль в этом примере CFILLSTR.ASM. Если программа на C содержит хотя бы один inline оператор asm или если вы определяете опцию -B либо используете раннюю версию Turbo C, компилятор преобразует всю программу в ассемблерный модуль с названием CFILLSTR ASM, удаляя таким образом файл с ассемблерным текстом без предварительного предупреждения Поэтому из предосторожности всегда используйте разные имена для ваших модулей на C и на ассемблере Другими словами, если файл вашеи основной программы называется KERMIT.C, не сохраняйте свои внешние подпрограммы в KERMIT.ASM

Раздельное ассемблирование и компоновка

Если у вас имеется большое количество модулей, вы сэкономите время, осуществляя ассемблирование и компоновку раздельно. Первым шагом следует ассемблировать все .ASM файлы. Поскольку пример fillsrting обладает только одним таким файлом, все можно проделать с помощью единственной команды

```
tasm /ml cfill
```

Опция /ml включает различение строчных и прописных символов, чтобы слова UpAndDown и upanddown рассматривались как различные — как это принято в программах C и C++. (Turbo Assembler обычно не обращает внимания на то, в каком регистре набраны символы, поэтому опция /ml необходима во избежание ошибок при компоновке.) После ассемблирования всех внешних модулей откомпилируйте основную программу. Опять же, поскольку в этом примере имеется только один файл .C, для этого необходима только одна команда

```
tcc -c cfillstr
```

Если вы пользуетесь Borland C++, примените следующую команду (замените bcc на tcc для Turbo C++):

```
bcc -c cfillstr c
```

Опция –c означает "только компилировать", вызывая создание CFILLSTR.OBJ, но не компоновку программы в законченный кодовый файл. Для включения всех модулей вам необходимо выполнить этот шаг самим, вызывая Turbo Linker для объединения файлов с объектным кодом с соответствующими библиотечными подпрограммами для создания CFILLSTR.EXE. Существует два метода компоновки. Сначала рассмотрим более сложный метод:

```
tlink c \tc\lib\c0s cfillstr cfill, cfillstr,, c \tc\lib\cs
```

Отредактируйте параметры в соответствии с требованиями инсталяции на вашем компьютере. Например, при использовании Borland C++4 можно применить следующую команду:

```
tlink c \bc4\lib\c0s cfillstr cfill, cfillstr,, c \bc4\lib\cs
```

Первый член после tlink специфицирует файл объектного кода в директории \LIB для соответствующей модели памяти, в данном случае — C0S.OBJ. Второй и третий члены перечисляют подлежащие компоновке файлы с объ-

ектным кодом .OBJ — они могут быть перечислены в любом порядке. Запятая отделяет список файлов .OBJ от имени, которое должно использоваться для конечного файла, в данном случае — CFILLSTR.EXE. Две запятые, следующие за этим, показывают место необязательного файла карты, не создающегося в данном примере. И наконец, специфицируется рабочая библиотека — также в каталоге \LIB.

Имена файла с объектным кодом C0S и библиотечным файлом CS должны соответствовать модели памяти, используемой программой. Последняя буква этих двух имен представляет одну из моделей, перечисленных в табл. 13.2.

Более простая компоновка

Упрощенный (но не очень быстрый) метод компоновки отдельных модулей состоит в использовании компилятора в качестве "первой части" Turbo Linker. Другими словами, вставляя различные команды компиляции, можно пропустить компиляцию и перейти прямо к компоновке. Это дает возможность избежать необходимости специфицировать имена рабочих библиотечных файлов и, следовательно, упрощает команду компоновки. Например, для ассемблирования, компиляции и компоновки CFILLSTR этим методом требуются три команды:

```
tasm /ml cfill
tcc -c cfillstr
tcc -ms cfillstr obj cfill.obj
```

Если вы используете Borland C++, введите следующие команды (замените bcc на tcc для Turbo C++):

```
tasm /ml cfill
bcc -c cfillstr c
bcc -ms cfillstr obj cfill.obj
```

Первые две команды — те же, что были описаны выше. Третья вызывает компилятор во второй раз, используя опцию -ms для определения модели памяти, в данном случае small (малая). (См. табл. 13.2 для определения букв остальных опций моделей памяти.) Вслед за опцией модели памяти идут файлы с объектным кодом, подлежащие компоновке. Хотя приходится включать расширение имен файлов .OBJ в перечисление каждого файла, этот не очень длинный метод компоновки упрощает большую часть черновой работы по непосредственному запуску Turbo Linker.

Таблица 13.2 **Имена файлов рабочей библиотеки**

Модель памяти	Объектный файл	Библиотечный файл	Опция TCC
Tiny (крошечная)	C0T.OBJ	CS.LIB	-mt
Small (малая)	C0S.OBJ	CS.LIB	-ms
Medium (средняя)	C0M OBJ	CM LIB	-mm
Compact (компактная)	C0C.OBJ	CL.LIB	-mc
Large (большая)	C0L.OBJ	CL.LIB	-ml
Huge (огромная)	C0H.OBJ	CH.LIB	-mh

Отладка многоязыковых программ

Существует несколько подходов к отладке программ, в которых совместно используются C или C++ и язык ассемблера. При первом подходе отладочная информация добавляется только к операторам C или C++. Чтобы сделать это, необходимо выполнить компиляцию одношаговой командой:

```
bcc -v cfillstr.c cfill.asm
```

Эта же команда приводилась ранее, но с опцией –v она подключает отладочную информацию в CFILLSTR.EXE. Затем можно провести отладку программы с помощью команды

```
td cfillstr
```

Проблема состоит в том, что эта команда не позволяет просмотреть исходный ассемблерный код — в основном окне отображаются только строки исходного C или C++ текста. Чтобы просмотреть также и ассемблерный текст, необходимо проводить ассемблирование и компоновку отдельно, используя более сложные методы, обсужденные в предыдущем разделе. Применительно к примерам CFILL.ASM и CFILLSTR.C полная процедура будет выглядеть следующим образом:

```
tasm /ml /zi cfill
tcc -c -v cfillstr
tcc -ms -lv cfillstr obj cfill obj
```

Если вы используете Borland C++, вместо этого введите следующие команды (замените bcc на tcc для Turbo C++):

```
tasm /ml/zi cfill
bcc -c -v cfillstr c
bcc -ms -lv cfillstr.obj cfill.obj
```

Сперва ассемблируется CFILL.ASM с применением опции /ml для включения режима различения строчных и прописных символов и /zi для включения отладочной информации в CFILL.OBJ. Затем компилятор компилирует CFILLSTR.C, учитывая только (-c) и дополняя CFILLSTR.OBJ отладочной информацией (-v). И наконец, компилятор вызывается в качестве первой части Turbo Linker. Опция -ms выбирает соответствующую модель памяти. Опция -lv передает букву опции, в данном случае v, в Turbo Linker, чтобы отладочная информация и из CFILLSTR.OBJ, и из CFILL.OBJ была перенесена в конечный файл CFILLSTR.EXE. Результирующий файл может быть затем загружен в Turbo Debugger с помощью

```
td cfillstr
```

После запуска Turbo Debugger многократно нажимайте <F7> для пошагового прохождения программы. Когда вы достигнете обращения к fillstring, Turbo Debugger переключится на исходный ассемблерный код, давая вам возможность пошагово выполнить инструкции внешнего модуля. Когда ассемблерный модуль завершится, вы снова увидите текст исходной программы. (Конечно, для этого и CFILLSTR.C, и CFILL.ASM должны находиться в текущем каталоге.)

Как работают CFILLSTR.C и CFILL.ASM

Теперь, когда вы знаете, как ассемблировать, компилировать и компоновать несколько модулей, написанных на языке ассемблера и C или C++, рассмотрим подробнее, как эти два файла работают вместе. Сперва рассмотрим строки 6–7 CFILLSTR.C (программа 13.4), которые объявляют функцию fillstring внешней, используя директиву extrn. Это дает возможность компилятору определить, что код обращения к fillstring будет сообщен ему позднее. (Если это не так, Turbo Linker выдает сообщение об ошибке.)

Программа 13.5, CFILL.ASM, заменяет функцию fillstring ассемблерным модулем. Строка 9 объявляет _fillstring public (общего пользования), добавляя к ее имени символ подчеркивания, чтобы привести в соответствие с правилом C и C++ для глобальных символов. Внутри функции директива ARG (строка 17) упрощает адресацию параметров, передаваемых в стек. Без ARG вам пришлось бы высчитывать смещения относительно bp и использовать инструкции, подобные

```
mov cx, [bp + 6]
```

исходя из предположения, что нужный вам параметр находится на 6 байт выше адреса, указываемого ss:bp. Вместо использования этого чреватого ошибками метода, ARG дает возможность перечислять параметры функции в том же порядке, в каком появляются идентификаторы в прототипе функции (см. строки 12–13). Для каждого из параметров, разделяемых запятыми, приводится его имя и размер, с использованием одного из определителей размера из табл. 13.1, но без суффикса ptr. Применение ARG подобным образом позволяет строкам 21 и 24–25 сослаться на параметры по именам. Конечно, вы по-прежнему должны быть внимательны при специфицировании верных размеров своих переменных.

После загрузки соответствующих регистров строка 26 использует повторяющуюся строчную инструкцию для накопления требуемого числа символов в строку. Эта длина никак не проверяется, поэтому будьте внимательны, чтобы не произвести запись в памяти поверх другого содержимого. Сравните полученный результат с откомпилированным текстом программы на рис. 13.1 для версии fillstring на чистом C. Не нужно большой исследовательской работы, чтобы понять, что однострочная инструкция выполняется быстрее, чем цикл for C, который требует восьми ассемблерных инструкций.

Ассемблерная версия fillstring также резервирует регистр di на тот случай, если регистровая переменная используется другой функцией, вызывающей fillstring. Однако обратите внимание, как строки 23 и 27 затем сохраняют и восстанавливают di, до тех пор пока предыдущая программа не проверит длину строки и не осуществит выход, если длина равна 0 (строки 21–22). Это не только улучшает программу, но и уменьшает время исполнения, если fillstring вызывается достаточно часто строками нулевой длины.

Вызов функций C из языка ассемблера

До сих пор мы рассматривали, как разделить переменные между C или C++ и ассемблером, а также как вызывать внешние ассемблерные функции из программ, написанных на C или C++. Теперь подойдем к этому с другой стороны, т.е. к вызову функций C или C++ из ассемблерного модуля — это также возможно, но требует большего внимания.

Если функция не обладает параметрами, процесс достаточно прост. Объявите функцию C или C++ в директиве EXTRN и воспользуйтесь инструкцией call:

```
CODESEG
EXTRN _cfuntion proc

call _cfunction
```

Здесь предполагается, что функция, названная _cfunction, существует в программе, подлежащей компоновке вместе с ассемблерным модулем. Напомню еще раз — символ подчеркивания добавляется в объявлениях ассемблера (но не в тексте программы на C или C++)

Если функции требуются параметры, процесс усложняется. Простые параметры, такие как символы и целые числа, часто передаются непосредственно в стек. Сложные переменные, такие как строки, структуры и множества, передаются посредством ссылок, т е по адресу. Кроме того, многие функции возвращают результат в специфичные регистры *При вызове функций C или C++ из языка ассемблера вы сами должны позаботиться о подобных нюансах.*

Сперва рассмотрим простейший случай вызова функции с одним целочисленным параметром:

```
void showscore( int thescore)
{
 printf(*\nThe score is  %d\n, thescore),
}
```

Чтобы вызвать функцию showscore из ассемблерного модуля, передавая значение переменной типа слова в качестве thescore, можно написать:

```
CODESEG
EXTRN     _showscore proc
mov  ax  76             , Присвоение score регистру
push ax                 , Передача параметра в стек
call _showscore         , Вызов функции C
pop  ax                 , Фиксация стека
```

Прежде всего, значение score присваивается ax (любой другой регистр точно так же подойдет для этого), а затем выталкивается в стек перед вызовом _showscore. После возврата из функции слово выталкивается из стека. Это необходимо потому, что в C и C++ вызывающая программа удаляет параметры из стека. (Если вы читали главу 12, то вспомните, что в Pascal процедуры и функции берут на себя заботу по удалению стековых параметров перед возвратом.) Если у вас имеется несколько параметров, может быть, будет лучше просто прибавить их общее число байтов к sp Например, чтобы вызвать функцию, которая оперирует четырьмя 16-битовыми параметрами, можно воспользоваться следующими командами:

```
push [v1]          , Выталкивание четырех переменных
push [v2]          , (не показанных) в стек
push [v3]
push [v4]
call _aCfunction          , Вызов функции C
add  sp  8                , Удаление параметров
```

Выталкивание нескольких параметров осуществляется в порядке, *обратном* тому, в каком они объявляются в функции C или C++. Исходя из предположения, что функция fillstring определена как

```
void fillstring( unsigned char far * the string,
 int stringLength, char fillchar ),
```

для вызова этой функции из языка ассемблера и заполнения строковой переменной пробелами требуются несколько шагов. Сперва ассемблерный модуль объявляется строковой переменной:

```
DATASEG
PUBLIC _astring
_astring db 80 dup (0)
```

Затем этот же модуль объявляет _fillstring в директиве EXTRN и вызывает функцию для заполнения строчной переменной пробелами:

```
CODESEG
EXTRN _fillstring:proc
...
xor ah, ah          ; Обнуление старшей половины ax
mov al, ' '         ; Присвоение пробела al
push ax           ; Проталкивание параметра fillchar
move ax, 79         ; Присвоение длины строки ax
push ax           ; Проталкивание параметра длины строки
push ds           ; Проталкивание сегмента адреса строки
mov ax, offset _astring; Присвоение смещения адреса ax
push ax           ; Проталкивание смещения адреса строки
call _fillstring        ; Вызов функции
add sp, 8         ; Удаление параметров из стека
```

Каждый из параметров — заполняющий символ, длина строки и 32-битовый указатель строковой переменной — проталкивается в стек в порядке, обратном перечисленному в определении функции. Применительно к указателю — сегмент адреса проталкивается перед смещением. После обращения к _fillstring к указателю стека sp добавляются 8 байт, удаляя параметры из стека.

Несмотря на то что в этом примере функция _fillstring в действительности написана на языке ассемблера, вызовы функций C и C++ ничем не отличаются. Если вы точно не уверены, как вызвать библиотечную подпрограмму (например, вездесущую printf()), запустите тестируемую программу в Turbo Debugger и просмотрите откомпилированный машинный код. Это подскажет вам, какие параметры требуются, а также даст много дополнительных сведений о том, как компиляторы преобразуют операторы C и C++ на ассемблер, — знания, которые вы сможете использовать при создании своих собственных внешних модулей.

Результаты выполнения функции

Многие из функций C и C++ возвращают значения в регистры или в верхушку стека математического сопроцессора (st(0)). Табл. 13.3 приводит регистры, используемые для возврата различных типов данных. Все 8-битовые типы возвращаются в al, 16-битовые — в ax, 32-битовые — в dx:ax.

Таблица 13.3. **Типы результатов функций**

Тип данных	Байты	Регистр(ы)
Unsigned char	1	al
Char	1	al
Enum	2	ax
Unsigned short	2	ax
Short	2	ax
Unsigned int	2	ax
Int	2	ax
Unsigned long	4	dx:ax
Long	4	dx:ax
Float	4	st(0) (стек 8087)
Double	8	st(0) (стек 8087)
Long double	10	st(0) (стек 8087)
Near *	2	ax
Far *	4	dx:ax

Локальные переменные

В дополнение к переменным, объявленным в сегменте данных либо общим с программой C или C++, можно использовать локальные переменные, помещенные в стек ваших ассемблерных модулей. Локальные переменные существуют только во время выполнения функции. Стековая область создается для переменных при запуске функции,

а затем очищается перед ее завершением. Таким образом, другие функции могут использовать эти же области памяти для своих собственных локальных переменных, снижая общий объем памяти, требуемой для всей программы Вам, вероятно, известно, как объявить локальные переменные в функциях C и C++, например управляющую переменную в цикле for:

```
void countup()
{
 int i,
 for (i = 0, i < 10, i++)
 printer( "%d ", i ),
}
```

Целая переменная i помещается в памяти в стек при запуске функции countup и существует только до тех пор, пока выполняется эта функция. В ассемблерном модуле можно проделать то же самое с помощью директивы LOCAL. Вот пример законченной функции:

```
PROC _cfunction NEAR
 LOCAL i Word =stacksize
 push bp
 mov  bp, sp
 sub  sp  stacksize
 mov  [i], 0
@@10
 inc  [i]
,
,------Код, использующий локальную переменную [i]
,
 cmp  [i], 10
 jne  @@10
 mov  sp, bp
 pop  bp
 ret       , Возврат в точку вызова
ENDP _cfunction
```

Директива LOCAL в этом примере подготавливает переменную i типа Word (слово). Указание =stacksize назначает общее число байтов, занимаемое всеми локальными переменными — в данном случае 2 байта. Это значение вычитается из sp после подготовки адресации переменных в стек. Затем, для ссылки на i, используются такие инструкции, как mov, inc и cmp. Благодаря директиве LOCAL ссылки типа [i] переводятся следующим образом:

```
mov  [bp - 2], 0
inc  [bp - 2]
```

и т.д. При использовании LOCAL нет необходимости вычислять отрицательные смещения относительно bp, чтобы определить местоположение переменных в стеке, — достаточно воспользоваться именами переменных.

Обратите внимание, что инструкция mov sp,bp, непосредственно перед окончанием функции в этом примере, восстанавливает sp. Поскольку bp не изменяется во время выполнения этой функции, можно восстановить sp посредством bp, удаляя область локальной переменной из стека, или прибавить stacksize к sp с помощью команды

```
add sp, stacksize
```

Подходят оба метода, но восстановление sp посредством bp — быстрее. Можно также объявить несколько локальных переменных операторами, подобными следующему:

```
LOCAL i Word, j Word, c Byte =stacksize
```

Теперь, после вычитания stacksize из указателя стека для резервирования области в стеке, можно использовать три локальные переменные — i, j и c. (Необходимо всегда делать LOCAL, что упрощает адресацию локальных переменных, это не создает область для переменных в памяти.)

Замечание

Я включил локальные переменные в этот раздел, поскольку вы должны знать, как их использовать. Но не забывайте, что одна из причин, по которой компиляция программ C и C++ может происходить медленнее, заключается в том, что адресация локальных переменных требует дополнительного времени. Это же справедливо для Pascal и других языков программирования. Одним из побудительных мотивов для подключения ассемблерных модулей к языкам высокого уровня является стремление увеличить быстродействие программы. Один из способов добиться этого — хранение переменных в быстром процессорном регистре, а не в стеке. Отсюда следует вывод: не применяйте методики, которые просто кажутся вам интересными; пользуйтесь теми методами, которые дают выигрыш в скорости.

Вызов функций C++ из ассемблера

C++ расширяет язык C посредством введения объектно-ориентированных классов и некоторых дополнительных синтаксических правил, которые в основном помогают программистам писать более удобные программы. Вся ранее приведенная информация относительно совместного использования языков C и ассемблера также относится и к C++, но имеется несколько нюансов, о которых вы должны знать — чтобы они не сбили вас с толку.

Этот раздел поясняет, как использовать совместно C++ и язык ассемблера, а также демонстрирует, как соединять ассемблерные модули с классами C++. Вы должны иметь в своем распоряжении Turbo C++ или Borland C++, чтобы осуществить компиляцию программ. Для проверки программ этого раздела я использовал Borland C++ и Turbo Assembler версий 4.0 и 4.5.

Искажение имен

C++ позволяет загрузку имен функций — одного поверх другого. Это означает, что две различные функции могут обладать одинаковыми именами, при условии, что они отличаются по меньшей мере одним параметром. Это программное средство весьма удобно, но создает проблему для обычных компоновщиков. Поэтому, чтобы компоновщики могли различить несколько функций, обладающих одинаковыми именами, C++ *искажает* их имена, объединяя с параметрами. Результатом является новое, хотя и непроизносимое, имя, уникальное для всех функций модуля.

Следующий пример показывает, как выглядят искаженные имена. Вы, наверно, знаете, что, используя потоки ввода-вывода C++, можно написать цепочку и начать новую строку оператором

```
cout << "Запишите меня в стандартный выход" << endl
```

Манипулятор endl осуществляет возврат каретки и переход на новую строку стандартного выхода. Когда вы подключаете эту программу к поточной библиотеке I/O (ввода-вывода), C++ искажает идентификатор endl, передавая компоновщику следующее объявление:

```
extrn @endl$qr7ostream:near
```

Символ @endl$qr7ostream:near является искаженным именем функции, которое создает компилятор, используя нигде не описанный алгоритм. Тем не менее я обнаружил это имя, компилируя тестовую программу на C++ с опцией –S, а затем просматривая результирующий текстовый .ASM файл.

К сожалению, искажение имен функций создает множество проблем для программистов, которым необходимо комбинировать языки C++ и ассемблера. При подключении к модулям C++ у вас имеется две возможности.

1. Скомпилировать свои C++ модули в текстовые файлы .ASM и скопировать искаженные имена для использования в ассемблерных модулях.
2. Отключить искажение имен для функций C++, вызываемых из ассемблера, или для подпрограмм, вызываемых из C++.

Второй подход обычно оказывается лучше, хотя и не всегда практичнее. Например, вам необходимо подключиться к существующей библиотеке функций C++ или к загружаемым одна поверх другой функциям. В этих случаях придется откомпилировать программу на C++, чтобы обнаружить искаженные имена, которые вы сможете использовать в директивах EXTRN в своих ассемблерных модулях. Это сделает вашу программу труднопереносимой, поскольку алгоритм искажения имен вполне может измениться в будущих версиях компиляторов.

В большинстве случаев при совместном использовании языка ассемблера и C++ лучше отключать искажение имен. Программы 13.6, CPPFUNC.CPP, и 13.7, CPPLOOP.ASM, показывают основную методику этого процесса. Как работает программа, я объясню после текста программ. Откомпилируйте, ассемблируйте, скомпонуйте и запустите программу с помощью следующих команд (замените bcc на tcc для Turbo C++):

```
bcc -c cppfunc
tasm /ml cpploop
bcc cppfunc.obj cpploop.obj
cppfunc
```

При запуске программы отобразятся следующие три строки:

```
Добро пожаловать в C++ и Ассемблер
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Это все!
```

Программа 13.6. CPPFUNC.CPP

```
1: // Calling C++ and assembly language functions -- by Tom Swan
2:
3: #include <iostream.h>
4:
5: extern "C" void Loop();                    // Функция-прототип в ассемблерном модуле
6: extern "C" void Terminate();              // Функция-прототип в модуле C++
7: extern int len;                           // Объявление данных в ассемблерном модуле
8: char c,                                   // Определение глобальных данных в модуле C++
9:
10: int main()
11: {
12:  cout << "Добро пожаловать в C++ и Ассемблер" << endl;
13:
14:  c = '@';                                // Присвоение значений глобальным данным C++
15:  len = 40;                               // Присвоение значений данным ассемблерного модуля
16:  Loop();                                 // Вызов функции ассемблерного модуля
17:  return 0;                               // Конец программы
18: }
19:
20: // Функция, вызываемая внешней функцией loop() ассемблерного модуля
21: extern "C"
22: void Terminate()
23: {
24:  cout << endl << "Это все!" << endl;
25: }
```

Программа 13.7. CPPLOOP.ASM

```
1:   %TITLE "C++ and Assembly Language External Function -- by Tom Swan"
2:
3:        IDEAL
4:        MODEL small
5:
6:   ;----- Data segment
7:
8:        DATASEG
9:
10:       EXTRN _c:BYTE                  ; Данные, объявленные в модуле C++
11:  _len      DW 0                      ; Данные, объявленные в ассемблерном модуле
12:       PUBLIC _len                    ; Делает данные доступными для C++
13:
14:  ;----- Code segment
15:
16:       CODESEG
17:
18:       EXTRN _Terminate:PROC          ; Функция в модуле C++
19:       PUBLIC _Loop                   ; Функция в ассемблерном модуле
20:
21:  ;-------------------------------------------------------
22:  ; void Loop();
23:  ;-------------------------------------------------------
24:  PROC _Loop NEAR
25:       mov    cx, [_len]             ; Получение длины от ассемблерного модуля
26:       jcxz   @@99                   ; Выход, если длина = 0
27:  @@10:
28:       mov    ah, 2                  ; Выбирает выходную функцию ДОС 2
29:       mov    dl, [_c]               ; Получение символа от модуля C++
30:       int    21h                    ; Вызов ДОС для вывода символа
31:       loop   @@10                   ; Цикл в cx
32:  @@99:
33:       call   _Terminate             ; Вызов функции в модуле C++
34:       ret                           ; Возврат в точку вызова
35:  ENDP _Loop
36:
37:       END                           ; Конец модуля
```

Вызов ассемблерных функций из C++

Для вызова ассемблерной функции из модуля C++ объявите прототип функции, как вы сделали бы это для программы на чистом C++, но предварите его extern "C", как показано в строке 5 CPPFUNC.CPP:

```
extern "C" void Loop()
```

Тем самым объявляется функция, названная Loop, не возвращающая никакого значения. Префикс extern сообщает компилятору, что функция располагается в другом модуле (компилятору не нужно знать, что функция будет написана на другом языке). Взятое в кавычки "C" отключает искажение имени, давая вам возможность использовать в ассемблерном модуле символ _Loop вместо искаженного имени. (Однако вы по-прежнему должны перед именем добавлять символ подчеркивания.)

Строка 16 в модуле C++ вызывает Loop. Ассемблерный модуль CPPLOOP.ASM обеспечивает выполнение этой функции. Чтобы компоновщик мог соединить оба модуля, ассемблерный модуль делает _Loop функцией общего пользования (public) в строке 19. Собственно функция в строках 24–35 выполняет функциональные действия. (Для целей отладки функция выводит символ @ определенное число раз. Это вызывает отображение строки символов @, которую вы и видите при запуске программы.)

Несколько внешних функций

При объявлении нескольких внешних ассемблерных функций можно использовать отдельные объявления extern, как в тексте программы, либо заключить несколько объявлений в скобки. Например, можно объявить три функции следующим образом:

```
extern "C" void f1();
extern "C" void f2();
extern "C" void f3();
```

Можно также использовать единственное объявление и перечислить все функции в скобках:

```
extern "C" {
 void f1(),
 void f2();
 void f3();
}
```

Оба формата дают один и тот же результат: три функции — f1, f2, f3 с неискаженными именами. В ассемблерном модуле можно ссылаться на эти функции, добавляя перед их именами символ подчеркивания, а именно: _f1, _f2, _f3.

Вызов функций C++ из ассемблера

Вызов функций C++ из ассемблерного модуля порождает такую же проблему с искажением имен. Для простоты обычно лучше отключать искажение имен, используя методику, описанную в предыдущих разделах. Программа 13.6, CPPFUNC.CPP, приводит пример объявления в строке 6:

```
extern "C" void Terminate();
```

Несмотря на тот факт, что функция объявлена extern, она выполняется в строках 21–25. Это может показаться странным, но не забывайте, что компилятору все равно, как выполняются функции. "Внешняя" функция может быть написана в другом модуле на C++, ассемблере или любом другом языке. Внешние функции могут быть написаны в *том же самом* модуле, в каком они объявляются, как показано в этом примере. Объявление extern предупреждает компилятор, чтобы он не ожидал исполнения функции в текущем модуле. Единственной причиной использования extern в данном случае является стремление отключить искажение имени.

Обратите внимание, что в строках, обеспечивающих выполнение функции, необходимо повторить префикс extern "C" (см. строку 21). Этот префикс является частью прототипа функции и, следовательно, должен быть повторен в выполнении функции. Контрольная функция, Terminate, отображает сообщение перед окончанием программы.

Модуль C++ не вызывает Terminate. Это происходит в ассемблерном модуле CPPLOOP.ASM. Поскольку функция существует в другом модуле, первым шагом является объявление ее EXTRN в кодовом сегменте модуля, как показано в строке 18:

```
EXTRN _Terminate:PROC
```

Замечание

В ассемблере используйте директиву EXTRN (без буквы E между T и R), в C++ —директиву extern (с буквой e).

Директива EXTRN специфицирует функцию (PROC), названную _Terminate, которая существует в другом модуле. Ассемблеру нет необходимости знать, как эта функция выполняется — достаточно того, что ее нет в текущем модуле. Объявление функции внешней позволяет программе обращаться к ней, как это показано в строке 33. Это все, что вам надлежит сделать для вызова функции C++ из ассемблерного модуля. Имеется, однако, несколько сложностей при передаче аргументов между функциями C++ и ассемблера туда и обратно. Я остановлюсь на этих проблемах несколько позже.

Совместные глобальные данные

Программы CPPFUNC.CPP и CPPLOOP.ASM демонстрируют также, как получить доступ к глобальным данным в модулях C++ и ассемблера. Демонстрационная программа использует две глобальные переменные — целое значение len и символьную переменную c. Ассемблерная функция Loop отображает назначенный символ len раз.

Просто ради интереса я определил переменную len в ассемблерном модуле, а сам символ — в программе на C++. Каждый модуль объявляет оба символа таким образом, что оба модуля имеют доступ к глобальным данным программы.

Замечание

Объявление символа присваивает ему имя и тип. Определение символа устанавливает местоположение объекта, на который ссылается символ. Различать *объявление* и *определение* очень важно, особенно при программировании на нескольких языках в нескольких модулях. Например, вы объявляете символ внешним в одном модуле, чтобы иметь доступ к объекту, определяемому этим символом в другом модуле. И что особенно важно — несколько модулей могут объявлять один и тот же символ (до тех пор, пока они делают это одинаково), но только один модуль может определять местоположение объекта.

Поскольку len определяется в ассемблерном модуле, модуль C++ должен объявить этот символ extern (см. строку 7):

```
extern int len,
```

В данном случае вам не нужно специфицировать "C", поскольку C++ искажает только имена функций, но не данных. (C++, однако, искажает имена классов, но об этом — позднее.)

Глобальный символ "c" объявляется и определяется в модуле C++ (см. строку 8):

```
char c,
```

Строки 14–15 в CPPFUNC.CPP присваивают значения этим двум глобальным переменным. Эти операторы ссылаются непосредственно на переменные — для C++ неважно, что одна переменная определяется в модуле C++, а другая — в ассемблерном компоненте программы. Глобальные данные используются таким же образом, независимо от того, где они определяются.

Ассемблерный модуль CPPLOOP.ASM также объявляет оба символа глобальных данных. Строка 10 использует директиву EXTRN в сегменте данных модуля для объявления объекта данных _c типа BYTE (обратите внимание, что к имени символа добавлен символ подчеркивания). Этот объект данных определяется в модуле C++.

Другое значение, len, объявляется и определяется в ассемблерном модуле. Для этого требуется два шага. Сперва необходимо определить слово, название _len, как показано в строке 11, а затем объявить этот символ public (общего пользования) (см. строку 12), чтобы другие модули могли пользоваться его значением.

Строки 25 и 29 показывают, как применять глобальные данные в языке ассемблера. Несмотря на то что _len определяется в ассемблерном модуле, а _c — в модуле C++, программа ссылается на оба символа, используя одинаковый синтаксис. Для ассемблера неважно, *где* определяется глобальная переменная.

Передача аргументов функции

Совместное использование C++ и ассемблера становится более сложным, когда к функциям добавляются аргументы. Приходится очень внимательно программировать, обращаясь к функциям из различных модулей и выбирая аргументы из стека. Две следующие программы, 13.8, CPPARG.CPP, и 13.9, ASMARG.ASM, демонстрируют основную методику. Как работает программа, я объясню в разделах, следующих за текстами программ. Откомпилируйте, ассемблируйте и запустите пример посредством команд (замените bcc на tcc для Turbo C++):

```
bcc -c cpparg
tasm /ml asmarg
bcc cpparg
```

Запуск программы сопровождается отображением трех строк:

```
xxxxxxxxxx
yyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Программа 13.8. CPPARG.CPP

```
1: // Pass arguments to/from assembly language -- by Tom Swan
2:
3: #include <iostream.h>
4:
5: extern "C" void CPPFunction(char c, int k);
6: extern "C" void ASMFunction(char c, int k);
7:
8: int main()
9: {
10: CPPFunction('x', 10); // Вызов функции C++
11: ASMFunction('y', 20); // Вызов ассемблерной функции
12: return 0;
13: }
14:
15: // Функция, вызываемая модулями - C++ и ассемблерным
16: extern "C"
17: void CPPFunction(char c, int k)
18: {
19: for (int i = 0; i < k; i++)
20: cout << c;
21: cout << endl;
22: }
```

Программа 13.9. ASMARG.ASM

```
1:   %TITLE "C++ and Assembly Language Arguments -- by Tom Swan"
2:
3:        IDEAL
4:        MODEL small
5:
6:   ;----- Макроопределения
7:
8:   cr   EQU    13                      ; Возврат каретки
9:   lf   EQU    10                      ; Перевод строки
10:
11:  ;----- Сегмент кода
12:
13:       CODESEG
14:
15:       EXTRN _CPPFunction:PROC         ; Функция в модуле C++
16:       PUBLIC _ASMFunction             ; Функция в ассемблерном модуле
17:
18:  ;-------------------------------------------------------
19:  ; void ASMFunction(char c, int k);
20:  ;-------------------------------------------------------
21:  PROC _ASMFunction NEAR
22:
23:       ARG    c_offset:byte, k_offset:word
24:
25:       push   bp                       ; Сохранение bp вызывающего модуля
26:       mov    bp, sp                   ; Установка для адресации аргументов
27:       mov    cx, [k_offset]           ; Получение счетчика цикла (k)
28:       jcxz   @@99                     ; Выход, если k = 0
29:  @@10:
30:       mov    ah, 2                    ; Выбор функции вывода ДОС 2
31:       mov    dl, [c_offset]           ; Получение символа (c) для отображения
32:       int    21h                      ; Вызов ДОС для вывода символа
33:       loop   @@10                     ; Цикл по cx
34:       mov    dl, cr                   ; Возврат каретки выходного устройства
35:       int    21h
36:       mov    dl, lf                   ; Перевод строки выходного устройства
37:       int    21h
```

```
38:  @@99:
39:        mov    ax, 30                    ; Выталкивание аргумента-счетчика
40:        push   ax                        ; в стек
41:        mov    al, 'z'                   ; Выталкивание аргумента-символа
42:        push   ax     ; в стек
43;        call   _CPPFunction              ; Вызов функции C++ и передача аргументов
44:        add    sp, 4                     ; Подготовка стека после возврата
45:
46:        pop    bp                        ; Восстановление bp вызывающего модуля
47:        ret                              ; Возврат в точку вызова
48:  ENDP _ASMFunction
49:
50:        END                              ; Конец модуля
```

Передача аргументов из C++ в язык ассемблера

Демонстрационная программа использует две функции — CPPFunction (определенную в модуле C++) и ASMFunction (определенную в ассемблерном модуле). Как и раньше, обе функции объявляются посредством extern "C" для подавления искажения имен, а в случае ASMFunction — еще и для того, чтобы указать, что выполнение этой функции осуществляется в отдельном модуле.

Замечание

Как и в случае с объектами данных, объявление функции просто присваивает ей имя, возвращаемый тип и перечисляет параметры. Определение функции подразумевает то, где написаны ее операторы. Необходимо различать это, поскольку объявлять функцию можно во многих модулях, но определять ее — только один раз.

Несмотря на то что функции написаны по-разному, как показывают строки 10–11, они используются идентично. Для C++ неважно, каким образом вы выполняете ассемблерные функции.

Обе функции выполняют одинаковую задачу — записывают определенное число символов в стандартный выводной файл. CPPFunction написана в C++, ASMFunction — на ассемблере. В отличие от предыдущего примера, который использовал глобальные данные, новые функции получают аргументы через стек. Строки 10–11 пересылают функциям аргументы символа и длины.

Функция ASMFunction в ассемблерном модуле ASMARG.ASM получает аргументы, используя директиву ARG, следующую за заголовком процедуры (строка 23). Аргументы перечисляются в том же порядке, что и в прототипе функции C++:

```
ARG c_offset:byte, k_offset.word
```

Аргументы, перечисленные таким образом, не являются объектами данных; они смещаются в стеке относительно регистра bp. Использование ARG подобным образом позволяет ассемблеру вычислить смещения вместо вас — но вы должны специфицировать правильные типы данных. Символьная переменная в C++ является байтом в ассемблере, целая в C++ — эквивалентом ассемблерного слова и т.д.

Строки 27 и 31 показывают, как загружать параметры в регистры. Однако чтобы эти операторы работали, необходимо зарезервировать и подготовить регистр bp, как показано в строках 25, 26. Эти инструкции сохраняют текущее значение bp, а потом устанавливают bp в соответствии с текущим указателем стека. Ассемблерная программа может затем использовать смещения ARG, c_offset и k_offset, чтобы получать доступ и передавать аргументы.

Не забудьте восстановить сохраненное значение bp, как показано в строке 46, перед возвратом из ассемблерной функции.

Передача аргументов из языка ассемблера в C++

Обратный процесс — передача аргументов из языка ассемблера в C++ — требует иной тактики. В приведенной программе строка 43 вызывает _CPPFunction модуля C++. Эта функция ожидает получения двух аргументов, что ассемблерный модуль и обеспечивает, выталкивая значения в стек.

Сделать это достаточно просто, как показывают строки 39–42, но нужно быть уверенным, что значения выталкиваются в правильном порядке. Стоит вытолкнуть сперва самый правый из аргументов, и программа может пойти по неверному пути. Например, строки 39, 40 выталкивают целое значение len, строки 41, 42 — символ. Этот порядок — обратный тому, в каком аргументы объявляются в прототипе функции (см. CPPARG.CPP, строку 5).

Замечание

Несмотря на то что символьный аргумент требует для своего хранения один байт, программа в строке 40 выталкивает в стек слово. Невозможно вытолкнуть в стек единственный байт.

Существует еще один чрезвычайно важный шаг, о котором вы не должны забывать. Поскольку функции C++ не очищают свои собственные стеки, вы должны удалить вытолкнутые аргументы после вызова функции C++. Строка 44 в ассемблерном модуле показывает, как выполнить эту важную задачу. Можно втолкнуть обратно вытолкнутые значения, но проще — прибавить соответствующее значение к указателю стека. (Убедитесь, что вы правильно вычислили размер. Поскольку программа выталкивает два слова, приведенный текст вычитает из sp четыре байта.)

Объявление аргументов процедуры автоматически

Используя альтернативную форму директивы ARG, можно упростить задачу получения аргументов, переданных операторами C++ ассемблерным функциям. Конечные результаты при этом те же, но вы можете сравнить эти две методики и выбрать ту, которая больше соответствует вашим вкусам. Метод, показанный здесь, исключает необходимость в подготовке и восстановлении регистра bp, но в остальном он аналогичен предыдущему.

Программа 13.10, ASMARG2.ASM, заменяет собой ASMARG.ASM. Сперва откомпилируйте, ассемблируйте и скомпонуйте программы, как объяснено в предыдущем разделе, а затем ассемблируйте и подключите новый модуль, используя следующие команды (замените bcc на tcc для Turbo C++):

```
tasm /ml asmarg2
bcc -ecpparg2.exe cpparg.obj asmarg2.obj
cpparg2
```

Запуск программы CPPARG2.EXE порождает такой же вывод, что и при запуске первоначального примера.

Программа 13.10. ASMARG2.ASM

```
1:  %TITLE "C++ Arguments Part 2 -- by Tom Swan"
2:
3:        IDEAL
4:        MODEL small
5:
6:  ;----- Макроопределения
7:
8:  cr    EQU   13                    ; Возврат каретки
9:  lf    EQU   10                    ; Перевод строки
10:
11: ;----- Кодовый сегмент
12:
13:       CODESEG
14:
15:       EXTRN _CPPFunction:PROC     ; Функция в модуле C++
16:       PUBLIC _ASMFunction         ; Функция в ассемблерном модуле
17:
18: ;-------------------------------------------------------
19: ; void ASMFunction(char c, int k);
20: ;-------------------------------------------------------
21: PROC _ASMFunction C c_arg:byte, k_arg:word
22:
23:       mov   cx, [k_arg]           ; Получение аргумента k
24:       jcxz  @@99                  ; Выход, если k = 0
25: @@10:
26:       mov   ah, 2                 ; Выбор функции вывода ДОС 2
27:       mov   dl, [c_arg]           ; Получение символа для отображения
28:       int   21h                   ; Вызов ДОС для вывода символа
29:       loop  @@10                  ; Цикл по cx
30:       mov   dl, cr                ; Возврат каретки устройства вывода
31:       int   21h
32:       mov   dl, lf                ; Перевод строки устройства вывода
33:       int   21h
34: @@99:
35:       mov   ax, 30                ; Выталкивание аргумента-счетчика
36:       push  ax                    ; в стек
37:       mov   al, 'z'               ; Выталкивание аргумента-символа
38:       push  ax                    ; в стек
39:       call  _CPPFunction          ; Вызов функции C++ и передача аргументов
40:       add   sp, 4                 ; Установка стека после возврата
41:
42:       ret                         ; Возврат в точку вызова
43: ENDP _ASMFunction
44:
45:       END                         ; Конец модуля
```

В строке 21 модифицированный текст программы объявляет _ASMFunction и ее аргументы посредством единственной директивы

```
PROC _ASMFunction C c_arg:byte, k_arg:word
```

"C" после имени функции указывает, что аргументы приводятся для языка C (т.е. они выталкиваются в стек в порядке справа налево). Остальные строки являются такими же, как и для директивы ARG.

В результате Turbo Assembler автоматически пишет инструкции для сохранения, инициализации и восстановления bp. При использовании этой альтернативной методики не приходится проводить точные операции по выталкиванию bp. За исключением этого отличия, в остальном процесс программирования остается тем же самым.

Использование классов C++ с языком ассемблера

Одной из главных причин использования C++ является написание объектно-ориентированных программ, содержащих классы.

Однако подключение языка ассемблера к тексту объектно-ориентированной программы является исключительно сложным по нескольким причинам

- Внутренние форматы объектов класса, групповых функций и в особенности виртуальных функций зависят от исполнения компилятора. Эти форматы, часть из которых документирована туманно либо недостаточно, могут также отличаться в зависимости от версии компилятора.
- В отличие от простых функций C++, нельзя отключить искажение имен классов C++ и групповых функций. Технически это возможно сделать в отношении ограниченных классов, но поскольку загрузка одного имени поверх другого является существенной для техники программирования на C++, то не имеет практического смысла отключать искажение имен для объектно-ориентированного кода. Это делает ссылку на имена класса и групповой функции в языке ассемблера исключительно трудной, так как вы вынуждены делать это, используя искаженные имена.
- Многие особенности C++, такие как обращение с выдержками, множественное наследование, загрузка операторов — одного оператора поверх другого и иные методы программирования, которые программисты на C++ считают достоинством, требуют высочайшего мастерства, чтобы выполнить их в языке ассемблера.
- В связи с тем, что язык C++ продолжает развиваться, все, что вы пишите сегодня, может оказаться устаревшим к тому моменту, когда вы будете ассемблировать свою программу. Написание переносимых ассемблерных интерфейсов для C++ является, по всем практическим соображениям, неосуществимой мечтой.

Каково же решение этой проблемы? Как известно любому регбисту, ответ прост: если не можешь идти вперед, бей по мячу.

Создание класса C++

Программа 13.11, CPPOOP.CPP, демонстрирует первый шаг простого метода совместного использования классов C++, объектно-ориентированного программирования и языка ассемблера. Эта методика гарантированно работает со всеми версиями C++ и является полностью переносимой (за исключением, конечно, самого ассемблерного кода).

Как я повсеместно подчеркиваю в этой книге, при совместном использовании нескольких языков обычно лучше сперва писать программу на языке высокого уровня, а затем, после того как вы добились, что она работает, преобразовывать отдельные функции на ассемблер.

В данном случае, поскольку трудно непосредственно вызвать язык ассемблера из C++, используется другая тактика — в виде дополнительных функций, служащих интерфейсом между классом и ассемблерным модулем. Групповые функции класса вызывают эти дополнительные функции, которые, в свою очередь, обращаются к ассемблерному коду. Несмотря на то что этот метод добавляет одно дополнительное обращение к функции и тем самым несколько уменьшает преимущество использования языка ассемблера (хотя и ненамного), результирующую программу легко написать и скомпоновать.

Откомпилируйте, ассемблируйте, скомпонуйте и запустите приведенные программы посредством следующих команд (замените bcc на tcc для Turbo C++):

```
bcc -c cppoop
tasm /ml asmfill
bcc cppoop.obj asmfill.obj
cppoop
```

Выполнение демонстрационной программы вызывает вывод таких строк:

```
Buffer : b1, size = 10 byte(s)
Contents: @@@@@@@@@@
```

```
Buffer : b2, size = 15 byte(s)
Contents: ###############

Buffer : b3, size = 25 byte(s)
Contents: *************************

Buffer : b1, size = 10 byte(s)
Contents: 1111111111

Buffer : b2, size = 15 byte(s)
Contents: 222222222222222

Buffer : b3, size = 25 byte(s)
Contents: 3333333333333333333333333
```

Для демонстрационных целей приведенная программа объявляет класс, TBuffer, для создания буферных объектов, заполненных определенными значениями байтов. Программа отображает размер каждого буфера, который создается и управляется динамически классом с использованием оператора new (новый) C++. Ассемблерный модуль заполняет объекты класса буфера, применяя быструю цепочечную инструкцию loop. Чтобы выполнить это, ассемблерный модуль должен вызвать групповые функции класса буферных объектов для определения размеров буфера и используемого символа-заполнителя. Эти действия демонстрируют также, как передавать объекты класса между модулями на языке ассемблера и на C++.

Программа 13.11. CPPOOP.CPP

```
1  // Object-oriented C++ and assembly language -- by Tom Swan
2:
3: #include <iostream.h>
4.
5: class TBuffer {
6:
7: // Конструктор и деструктор
8. public:
9:   TBuffer(char c, int bs);
10:  ~TBuffer();
11:
12. // Групповые функции
13: public:
14:  void SetFillChar(char c)
15:    { fillChar = c; }
16:  char GetFillChar()
17:    { return fillChar; }
18:  int GetFillSize()
19:    { return fillSize; }
20:  void FillBuffer();
21:  void ShowBuffer(const char *s);
22:
23: // Личные члены данных
24: private:
25:  char fillChar;     // Символ для помещения в буфер
26:  int fillSize;      // Размер буфера в байтах
27:  char far *buffer;  // Указатель на буфер
28: };
29:
30: // Объявление внешней функции ассемблерного модуля
31: extern "C" void ASMFillBuffer(TBuffer far &bo, char far *buffer);
32:
33: // Объявления внешней функции модуля cpp
34: extern "C" char CPPGetFillChar(TBuffer &bo);
35: extern "C" int CPPGetFillSize(TBuffer &bo);
36:
37: int main()
38: {
39:  TBuffer b1('@', 10); // Конструирование объектов
40:  TBuffer b2('#', 15);
41:  TBuffer b3('*', 25);
42:
43:  b1.ShowBuffer("b1"); // Отображение объектных буферов
44:  b2.ShowBuffer("b2");
45:  b3.ShowBuffer("b3");
46:
```

```
47:  b1.SetFillChar('1'); // Установка символа-заполнителя и перезаполнение буфера
48:  b1.FillBuffer();
49:  b2.SetFillChar('2');
50:  b2.FillBuffer();
51:  b3.SetFillChar('3');
52:  b3.FillBuffer();
53:
54:  b1.ShowBuffer("b1"); // Отображение объектных буферов
55:  b2.ShowBuffer("b2");
56:  b3.ShowBuffer("b3");
57:
58:  return 0; // Конец программы
59: }
60:
61: // Исполняемый конструктор TBuffer
62: TBuffer::TBuffer(char c, int bs)
63: {
64:  fillChar = c;       // Сохранение символа-заполнителя
65:  fillSize = bs;      // Сохранение размера буфера
66:  buffer = 0; // Инициализация указателя буфера
67:  if (fillSize <= 0) return;              // Выход, если размер<=нулю
68:  buffer = new char[fillSize];            // Выделение памяти под буфер
69:  FillBuffer();      // Заполнение буфера символами
70: }
71:
72: // Исполняемый деструктор TBuffer
73: TBuffer::~TBuffer()
74: {
75:  delete buffer;      // Удаление выделенной памяти
76: }
77:
78: // Исполняемая групповая функция заполнения буфера
79: // Вызов внешней ассемблерной функции
80: void TBuffer:.FillBuffer()
81: {
82:  ASMFillBuffer(*this, buffer);          // Вызов функции ассемблерного модуля
83:
84: /* Эквивалентный код C++ для приведенного выше вызова функции
85:  if (buffer == 0) return,
86:  for (int i = 0; i < GetFillSize(); i++)
87:    buffer[i] = GetFillChar();
88: */
89: }
90:
91: // Исполняемая групповая функция отображения буфера
92: void TBuffer::ShowBuffer(const char *s)
93: {
94:  cout << endl;
95:  cout << "Buffer : " << s;
96:  cout << ", size = " << GetFillSize() << " byte(s)" << endl;
97:  cout << "Contents: ",
98:  for (int i = 0; i < GetFillSize(); i++)
99:    cout << buffer[i];
100: cout << endl;
101: }
102.
103: // Возврат символа заполнителя для объекта bo
104: // Вызывается внешней ассемблерной функцией
105: extern "C"
106: char CPPGetFillChar(TBuffer &bo)
107: {
108: return bo GetFillChar();
109: }
110:
111: // Возврат размера буфера для объекта bo
112: // Вызывается внешней ассемблерной функцией
113: extern "C"
114: int CPPGetFillSize(TBuffer &bo)
115: {
116: return bo.GetFillSize();
117: }
```

Строки 5–28 объявляют класс TBuffer. Это C++ в чистом виде. Обратите внимание, что некоторые групповые функции исполняются "inline" (строки 14–19), а остальные — обычным образом (строки 20, 21). При использовании техники интерфейса, рассмотренной в этом разделе, групповые функции могут быть и виртуальными, хотя таковых нет в данном примере. Можно также использовать множественное наследование и все остальные методы программирования на C++.

Строки 30, 31 объявляют внешнюю ассемблерную функцию, используемую классом TBuffer. Она объявляется директивой extern "C" совершенно так же, как и в предыдущих примерах. Кроме отключения искажения имени функции ASMFullBuffer, определение сообщает компилятору, что выполнение функции осуществляется в отдельном модуле.

Остальные две функции C++ аналогичны объявленной в строках 34, 35. Ассемблерный модуль обращается к ним для получения составляющих данных от объекта класса TBuffer.

КРАТКИЙ ОБЗОР

Строка 31 объявляет ассемблерную функцию, которую нужно вызвать из C++.

Доступ к объектам класса из языка ассемблера

Программа 13.12, ASMFILL.ASM, представляет собой исполнение ассемблерной функции _ASMFillBuffer, вызываемой классом TBuffer. Кроме того, программа демонстрирует, как передавать и получать аргументы, на которые имеются ссылки, объектам класса. (К объектным указателям применима идентичная методика, поскольку ссылки C++ физически, если не синтаксически, идентичны указателям.)

Программа 13.12. ASMFILL.ASM

```
1:  %TITLE "External function for a C++ class object -- by Tom Swan"
2:
3:        IDEAL
4:        MODEL small
5:
6:  ;----- Code segment
7:
8:        CODESEG
9:
10:       EXTRN _CPPGetFillChar:PROC        ; Функция в модуле C++
11:       EXTRN _CPPGetFillSize:PROC        ; Функция в модуле C++
12:       PUBLIC _ASMFillBuffer             ; Функция в ассемблерном модуле
13:
14: ;-------------------------------------------------------
15: ; void ASMFillBuffer(TBuffer far &bo, char far *buffer);
16: ;-------------------------------------------------------
17: PROC _ASMFillBuffer NEAR
18:
19:       ARG    bo_offset:DWORD, buffer_offset:DWORD
20:
21:       push   bp                         ; Сохранение bp вызывающего модуля
22:       mov    bp, sp                     ; Установка для адресации аргументов
23:       push   di                         ; Сохранение di, так как используется
24:                                         ; для регистровых переменных
25:       les    di, [bo_offset]            ; Получение адреса объекта bo в es:di
26:       push   es                         ; Выталкивание сегмента адреса объекта bo
27:       push   di                         ; Выталкивание смещения адреса объекта bo
28:       call   _CPPGetFillChar            ; Вызов функции C++, передача объекта arg
29:       add    sp, 4                      ; Настройка указателя стека для удаления arg
30:       push   ax                         ; Сохранение символьного результата al в стеке
31:
32:       les    di, [bo_offset]            ; Получение адреса объекта bo в es:di
33:       push   es                         ; Выталкивание сегмента адреса объекта bo
34:       push   di                         ; Выталкивание смещения адреса объекта bo
35:       call   _CPPGetFillSize            ; Вызов функции C++, передача объекта arg
36:       add    sp, 4                      ; Настройка указателя стека для удаления arg
37:       mov    cx, ax                     ; Копирование целого результата из ax в cx
38:       les    di, [buffer_offset]        ; Получение адреса буфера в es:di
39:       pop    ax                         ; Вызывается символ-заполнитель из стека
40:
41: ; al = fill character
42: ; cx = buffer size
43: ; es:di = buffer address
```

```
44:
45:         jcxz    @@99                    ; Не выполняется никаких действий, если cx=0
46:         cld                             ; Установка направления заполнения вперед
47:         rep     stosb                   ; Заполнение буфера: al -> es:di в cx
48:   @@99:
49.         pop     di                      ; Восстановление di
50.         pop     bp                      ; Восстановление bp вызывающего модуля
51:         ret                             ; Возврат в точку вызова
52:   ENDP _ASMFillBuffer
53:         END                             ; Конец модуля
```

Кодовый сегмент модуля объявляет две внешние функции — _CPPGetFillChar и _CPPGETFillSize. Эти функции определяются в модуле C++, который соединяется с объектом TBuffer. Ключевой концепцией в данном случае является то, что ассемблерный модуль не вызывает непосредственно групповые функции класса. Вместо этого ассемблер вызывает интерфейсные функции, которые и выполняют это подключение.

Кроме того, строка 12 делает ассемблерную функцию _ASMFILLBuffer доступной для общего пользования (public), чтобы ее мог вызывать модуль C++.

Строки 17–52 выполняют функцию, которая передает два аргумента в стек. Директива ARG в строке 19 готовит два смещения типа DWORD для получения доступа к этим аргументам. Они имеют тип DWORD, поскольку используются 32-битовые указатели. (Аргументы объявляются far в модуле C++.)

Как я объяснял, при использовании ARG необходимо сохранять и инициализировать регистр bp для адресации аргументов, переданных в стек. Строки 21, 22 выполняют эту задачу. Я также выталкиваю регистр di, поскольку функция использует этот регистр.

Замечание

Turbo C, Turbo C++ и Borland C++ используют si и di для регистровых переменных. Если только вы не исключаете из своих программ регистровые переменные, вы должны сохранять и восстанавливать si и di в функциях, использующих эти регистры.

Строка 25 показывает, как получить адрес объекта класса, переданный ассемблерной функции. Инструкция les загружает в регистры es:di адрес из стека, на который имеется ссылка относительно bp. Другими словами, после этого шага es:di адресует объект TBuffer, переданный _ASMFillBuffer посредством ссылки.

Адрес этого объекта необходим для того, чтобы вызвать его групповую функцию GetFillChar. Но, как я уже отмечал, непосредственно вызвать групповую функцию слишком трудно, и, кроме того, это сделало бы программу в высшей степени зависимой от выполнения. Чтобы избежать подобных досадных осложнений, следует просто вызвать интерфейсную функцию вроде _CPPGetFillChar, которая вызывает уже саму групповую функцию класса. Интерфейсная функция требует адреса объекта TBuffer, который ассемблерная функция выталкивает в стек в строках 26, 27.

Вслед за вызовом функции, как в случае всех обращений к функциям в C и C++, программа удаляет вытолкнутый аргумент прибавлением соответствующего значения к указателю стека (см. строку 29).

Интерфейсная функция _CPPGetFillChar возвращает символ-заполнитель в регистр ax. Нам понадобится это значение немного позднее, поэтому строка 30 выталкивает его в стек для хранения.

Затем программа вызывает вторую интерфейсную функцию _CPPGetFillSize. Сперва в строке 32 les загружает адрес буфера, который выталкивается в стек перед вызовом интерфейсной функции в строке 35. Указатель стека восстанавливается после вызова этой функции (строка 36), и возвращенный размер заполнителя в виде целого числа помещается в регистр cx (строка 37).

Строка 38 снова использует les для загрузки es:di адресом буфера, переданного ассемблерной функции в качестве ее второго аргумента. И наконец, строка 39 выталкивает сохраненный символ обратно в ax.

В программах, использующих несколько параметров, я посчитал полезным помещать комментарии, описывающие состояние различных регистров в стратегически важных точках. Комментарии в строках 41–43 отражают значения, хранимые в al, cx и es:di в этот момент выполнения программы. Рационально просмотреть в этом месте предшествующий код, чтобы проверить, все ли регистры подготовлены надлежащим образом.

После того как вся черновая работа проделана, ассемблерная функция может приступить к заполнению буфера заданными символами. Это самая простая часть всего процесса. Строка 45 пропускает следующие две инструкции, если длина буфера равна нулю. Строка 46 подтверждает, что направление заполнения — вперед (по возрастанию адресов). Строка 47 выполняет заполнение практически мгновенно, используя сверхскоростную повторяющуюся инструкцию stosb.

И наконец, строки 49–51 восстанавливают сохраненные значения регистров di и bp перед возвратом в точку вызова функции.

Резюме

Основными причинами подключения языка ассемблера к программам на C и C++ являются стремление увеличить быстродействие программы и обеспечить доступ к аппаратному обеспечению на низком уровне. Компиляторы Borland C и C++ предоставляют два метода для подключения языка ассемблера в программы: inline операторы asm и внешние функции. Операторы inline просты в использовании, но не такие универсальные, как внешние функции.

Поскольку большинство программ 90% времени тратят на исполнение 10% инструкций, обнаружение и оптимизация критичных 10% часто дает поразительное увеличение быстродействия. Переписывание остальных 90% может оказаться пустой тратой времени. Не переписывайте операторы C и C++, которые уже и так работают достаточно быстро.

Регистры bp, cs, ds, sp и ss должны быть восстановлены перед окончанием работы ассемблерного модуля. Регистры ax, cx, dx, di, si и es могут применяться свободно. Однако, поскольку компиляторы C и C++ могут использовать di и si для регистровых переменных, необходимо сохранять эти два регистра.

Введение inline операторов asm заставляет ранние версии Turbo C генерировать ассемблерный текст всей программы. Этот файл затем может быть ассемблирован и скомпонован для создания конечной программы. Можно сэкономить время, с самого начала используя опцию -B для компиляции программы, либо вставить в программу эквивалентный оператор #pragma inline. Другая опция, -S, дает возможность просмотреть ассемблерный текстовый файл, который обычно удаляется.

Inline операторы asm, помещенные внутри функций, попадают в кодовый сегмент программы, а помещенные снаружи функций — в сегмент данных программы. Можно разделить данные и код с C и C++ и получить доступ к структурам C и C++ в ассемблерных операторах.

Написание внешних ассемблерных функций требует большей работы, чем помещение операторов asm непосредственно в программу C или C++, но результат часто стоит затраченных усилий. Внешние модули экономят время компиляции, позволяя разрабатывать программу по кускам — при этом нет никакой необходимости компилировать всю программу в ассемблерный текст. Можно также в ассемблерных модулях использовать режим Ideal. Что ценнее всего, упрощенные модели памяти упрощают написание функций по сравнению с тем, если бы вам пришлось объявлять сегменты "жестким образом" (что вы по-прежнему можете делать, если пожелаете). Ассемблирование, компиляция и компоновка многомодульных программ является сложным делом, но использование компилятора в качестве "первой части" Turbo Linker может сэкономить время и усилия.

Вызов ассемблерных функций из C или C++ идентичен вызову других функций. Движение в обратном направлении — вызов функций из языка ассемблера — требует, чтобы вы вытолкнули параметры функции в стек, а затем, после возврата из функции, удалили эти параметры. Можно также объявить в функциях локальные переменные, хотя программа может исполняться быстрее, если у вас имеется возможность использовать регистры для хранения временных значений.

Искажение имен в C++ усложняет задачу совместного использования C++ и ассемблера. Отключайте искажения имен посредством объявлений extern "C" для ассемблерных функций, вызываемых из C++, а также для функций C++, вызываемых ассемблерными модулями.

Непосредственное объединение классов C++ и языка ассемблера слишком сложно и зависимо от выполнения программы. Вместо попытки вызвать групповые функции класса непосредственно из языка ассемблера можно применить более практичный метод, продемонстрированный в этой главе, — использовать интерфейсные функции, которые и вызывают собственно составляющие класса. Передача ссылок на объекты этим функциям делает совместное использование ассемблера и C++ относительно простым. И, что самое ценное, конечные результаты являются переносимыми и не зависящими от нюансов исполнения.

Упражнения

13.1. Какие существуют два способа подключения ассемблера к программам C? Как отличается компиляция при этих двух методах?

13.2. Когда обязательно сохранять и восстанавливать регистры si и di? Когда нет необходимости это делать?

13.3. Напишите inline ассемблерную функцию для отображения значений флагов 8086. Единственный оператор C, который можно использовать, — это обращение к printf для отображения результатов — остальные инструкции должны быть операторами asm. Подсказка: см. рис. 4.2 для определения позиций битов флага.

13.4. Предположим, что у вас имеется имя структуры C Things и переменная этой структуры, названная MyThings. Какой inline оператор asm можно использовать для загрузки адреса поля структуры, названного OneThing?

13.5. Какую опцию командной строки можно использовать для компиляции программы в ассемблерный текст? Какая при этом возникает опасность?

13.6. Предположим, у вас имеются две внешние функции, названные FUNC1.ASM и FUNC2.ASM. Какие команды требуются для ассемблирования, компиляции и компоновки внешних модулей с основной про-

граммой C, названной MAIN.C, для создания конечной программы, названной MAIN.EXE? Предположите, что программа использует малую модель памяти.

13.7. Какую директиву ARG можно применить для адресации параметров следующей функции-прототипа:

```
extern void copystring (unsigned char far * source,
 unsigned char far * destination,
 int sourcelen)
```

13.8. Какие операторы C необходимы для вызова внешней функции, определенной, как в вопросе 13.7?

13.9. Напишите внешний модуль, чтобы завершить функцию copystring, приведенную в вопросах 13.7 и 13.8 Модуль должен копировать символы sourcelen из исходной строки в строку назначения.

13.10. Дана внешняя функция из вопроса 13.9. Какие ассемблерные операторы требуются, чтобы вызвать функцию для передачи адреса и длины двух строк — string1 и string2, объявленных во внешнем сегменте данных?

Задания

13.1. Скомпилируйте различные C или C++ программы (возможно, из общеиспользуемой библиотеки) с применением опции -S, создавая .ASM файлы, которые вы сможете просмотреть. Отыщите операторы, где inline asm код мог бы улучшить быстродействие Перекомпилируйте программы, запустите их, замерьте время выполнения и оцените результаты своей оптимизации.

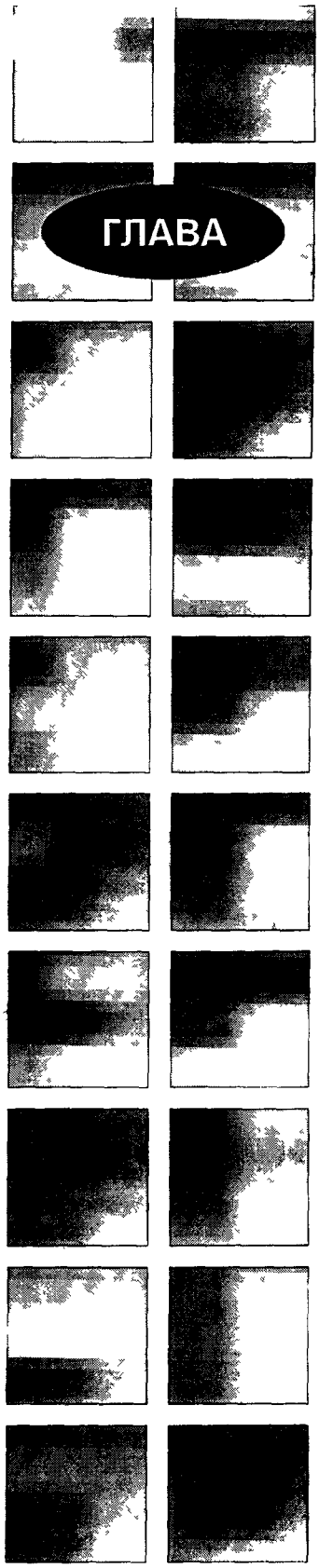
13.2. Преобразуйте процедуры в модуле ASYNCH.ASM из главы 10 (либо в каком-либо другом модуле) во внешние функции C или C++.

13.3. Стандартная C функция printf действительно является весьма универсальной — она способна записывать в стандартный выводной файл любые строки, символы и цифровые данные. Но программирование подобной универсальной функции требует времени. Напишите ряд упрощенных функций для записи строк и целых чисел.

13 4. Разработайте библиотеку внешних функций C быстрого непосредственного обращения к видеотерминалу для отображения текста на экране персонального компьютера PC.

13.5. Напишите программу C или C++ для преобразования всех символов текстового файла в строчные, включая, быть может, также и предложения, начинающиеся с заглавных символов. После отладки своей программы выборочно преобразуйте части программы на ассемблер для увеличения быстродействия.

13.6. Используйте Turbo Debugger, чтобы проследить обращения функции к различным подпрограммам из рабочей библиотеки Borland C++ или Turbo C. Задокументируйте как можно больше программ. (Это будет полезно для изучения того, как стандартные функции выполняются в языке ассемблера.)



ГЛАВА 14

Программирование с объектами

- **Объектно-ориентированное программирование (ООП) посредством TASM**
- **Для чего используется ООП**
- **Основы объектов TASM**
- **Полиморфизм**
- **Другие узкие места и "завлекалочки" ООП**
- **Резюме**
- **Упражнения**
- **Задания**

Объектно-ориентированное программирование (ООП) посредством TASM

Объектно-ориентированное программирование, или ООП, становится основной концепцией языков высокого уровня, таких как C++ и Borland Pascal. До недавнего времени, если вы хотели применить ООП, то должны были писать программу на одном из этих языков либо использовать менее известную прикладную систему, вроде Smalltalk или Actor.

Однако, начиная с TASM 3.0, теперь можно писать объектно-ориентированные программы на языке ассемблера. Точное представление о том, *почему* вы захотите это сделать, является одним из наиболее трудных аспектов изучения использования ООП, поэтому перед тем, как углубиться в объектно-ориентированные свойства TASM, прочтите следующие разделы, посвященные обзору ООП и его значению для программистов.

Для чего используется ООП

Говоря кратко, ООП дает возможность писать компьютерные программы в значительной степени путем конструирования *объектов*. Объект является просто структурой, которая объединяет данные и код (рис. 14.1), вместе известные как *члены* объекта, которые *заключены* в один удобный пакет.

Рис. 14.1. Объект является структурой, которая заключает в себе данные и код

Ниже приведено несколько ключевых положений, касающихся объекта, представленного на рис. 14.1.

- Код, заключенный в объекте, обычно выполняет какие-либо операции с данными объекта. Это не обязательно, но обычно именно так. Код объекта состоит из подпрограмм, называемых *методами*, которые пишутся так же, как обычные подпрограммы.
- Данные, заключенные в объекте, *скрыты*. Только код объекта имеет к ним прямой доступ. Как обычно бывает при использовании языка ассемблера, можно легко нарушить это правило, но тогда вам придется отклониться от законов ООП на свой собственный страх и риск. Данными объекта могут быть любые переменные (байты, слова, множества, структуры, указатели и даже другие объекты), которые можно определить.
- Для того чтобы использовать объект, необходимо создать для него хранилище. Такое хранилище называется *инстанцией* — это объектно-ориентированный эквивалент переменных данных, таких как байты или слова. Можно определить столько инстанций объекта, сколько необходимо.
- Можно конструировать новые объекты на базе уже существующих — эта методика носит название *наследования*. Используя наследование, можно писать программы путем расширения библиотеки существующих объектов.

Программирование с объектами предоставляет несколько преимуществ по сравнению с обычными методами, однако при этом имеется и несколько подводных камней, которые необходимо учитывать. Следующие разделы описывают многие особенности ООП — как преимущества, так и недостатки.

Преимущества ООП

Чтобы понять значение объектов, рассмотрим, как большинство программистов пишут обычную программу. Сначала они определяют данные программы, резервируя хранилище для байтов, слов и других структур. Затем пишут подпрограммы для действий операций над данными или некоторые операторы, которые либо передают данные подпрограммам, либо помещают адреса в регистры, либо выталкивают значения в стек для использования их подпрограммами.

Нет ничего плохого в такой концептуальной модели написания компьютерных программ. Но когда программы разрастаются сверх некоторого среднего уровня сложности, одна из частей программы может случайно изменить данные, которые требуются для другой части, приводя к досадным сбоям и таким поворотам в процессе исполнения программы, которые трудно устранить.

Даже программисты высшей квалификации удивляются, обнаружив, как просто создать подобную путаницу. Например, вы можете определить глобальную переменную count, используемую в цикле, который прокручивается заданное число раз. В этом цикле происходит обращение к еще одной подпрограмме, которая, в свою очередь, обращается к другим подпрограммам — достаточно обычная ситуация. Но всегда существует угроза, что какой-нибудь оператор где-нибудь внутри программы может также использовать count для своих собственных нужд. Эта роковая, но легко допускаемая ошибка приводит к дефектному циклу, который модифицирует свой собственный управляющий параметр и в результате вызывает сбой всей программы.

ООП может помочь предотвратить такого рода конфликты. Поскольку объекты *заключают* в себе код и данные, использование данных ограничено определенным кругом подпрограмм. Такое капсулирование предоставляет программистам два важных преимущества:

- Когда сбой происходит из-за ошибки в данных, вы *можете быть уверены*, что найдете *решение проблемы в коде объекта*. Особенно это касается больших программ, где возможность ограничить отладку относительно небольшим разделом является огромным преимуществом при практическом применении и выявлении мест, вызывающих проблемы.
- В объектно-ориентированные программы проще добавлять новый код. Поскольку использование данных ограничено подпрограммами объекта, то можно спокойно применять данные в новом программном модуле, не опасаясь вызвать конфликт с другими модулями. *Вам всегда известны пределы использования данных*. Этот аспект ООП имеет исключительное значение для разработчиков, особенно применительно к прикладному программному обеспечению, которое пишут группы программистов.

Недостатки ООП

Несмотря на радужные перспективы, ООП обладает и несколькими колючками. Изначально, разрабатывать объектно-ориентированные программы труднее. Если вы относитесь к тому типу программистов, которым, осененным свежей идеей, не терпится начать набирать инструкции, тогда ООП может оказаться для вас неподходящей моделью. При использовании ООП тщательное планирование является существенным условием достижения серьезных результатов.

Значение ООП более ощутимо в больших программах, чем в маленьких. Примеры программ приведенные в этой главе, могут создать впечатление, что в них используются излишне сложные методы для выполнения относи-

тельно простых действий. Если вы пишите обеспечение для маленьких программ, ООП может *увеличить* сложность вашего кода. (Но даже маленькие программы часто могут с успехом использовать библиотеки существующих объектов.)

ООП и Turbo Assembler

Свойства ООП Turbo Assembler сходны с аналогичными особенностями Pascal и C++, хотя имеются некоторые важные отличия, которые я опишу в этой главе. В языке ассемблера, например, вы должны сами конструировать различные таблицы, указатели и выполнять такие операции, как загрузка регистров, что в других языках выполняется автоматически.

Кроме того, при использовании ООП в Turbo Assembler проще попасть в неприятную ситуацию, чем в других языках, где имеются встроенные стражи для предотвращения ошибок. Например, компиляторы C++ и Borland Pascal всегда проверяют использование в операторах правильных типов объектов. В языке ассемблера все ограничения сняты и нарушить правила ООП сравнительно просто (как и нарушить правила обычного программирования).

Еще одним недостатком ООП в Turbo Assembler является то, что инстанции объектов (т.е. переменные определенного типа данных) несовместимы с классами C++ и объектами Pascal. Если вы намерены применять совместно ассемблерный и код ООП высокого уровня, вероятно, лучше использовать язык высокого уровня для конструирования своих объектно-ориентированных модулей. Для принятия решения о совместном использовании языка ассемблера высокоуровневым ООП C++ еще раз просмотрите главу 13 "Использование языка ассемблера при программировании на C и C++".

Замечание

Объектно-ориентированные команды Turbo Debugger (например, View|Hierarchy — Просмотр|Иерархия) не распознают объекты TASM. В Turbo Debugger можно просмотреть инстанции объектов, но они отображаются структурами, а не объектами.

Несмотря на эти оговорки при использовании ООП в Turbo Assembler, имеется много веских причин для выбора этой модели программирования для написания программ на языке ассемблера. Как я убежден, ООП замечательно подходит для больших прикладных программ, особенно написанных бригадами программистов. Кроме того, отладка, настройка и будущие пересмотры программ потенциально упрощаются благодаря ООП.

Еще одной веской причиной использования ООП является преобразование существующей объектно-ориентированной программы C++ или Pascal на чистый язык ассемблера. Если вам необходимо преобразовать код ООП высокого уровня в язык ассемблера, особенности ООП TASM в огромной степени упростят преобразование.

ООП в своих собственных понятиях

Подобно другим методикам, ООП использует свои собственные понятия, со многими из которых вы встретитесь в этой главе. Просмотрите эти термины, чтобы ознакомиться с ними, но не слишком обращайте внимание, если некоторые из них покажутся недостаточно ясными.

Замечание

Следующий словарь поясняет различия и сходства между терминологией ООП C++, Borland Pascal и Turbo Assembler. Термины Turbo Assembler более сходны с терминами, используемыми в Borland Pascal, чем с применяемыми в C++.

Базовый объект — объект, используемый для получения другого объекта. Полученный объект наследует свойства базового объекта. Свойства одного и того же базового объекта может наследовать более чем один полученный объект. Например, графическая программа может объявить объект общего назначения TGraphics, а затем использовать его в качестве базы для получения объектов специального назначения, таких как TCircle и TRectangle. Базовый объект обеспечивает код и данные, которые являются общими для всех связанных объектов. Объекты, полученные на основе базового, дополняют данные и код тем, что является специфичным для их собственных нужд. Любой объект может быть базовым. См. также *Полученный объект*.

Класс — термин C++ для *объектов*, используемый в Turbo Assembler и Borland Pascal. См. *Объект*.

Конструктор — специальный метод инициализации объектной инстанции. Turbo Assembler не поддерживает концепцию конструктора, хотя, как я покажу в этой главе, можно запрограммировать его эквивалент. (В C++ конструкторы могут быть вызваны автоматически. В языке ассемблера вызов конструктора объекта целиком зависит от вас.)

Полученный объект — объект, который наследует свойства (данные и код) другого, базового объекта. Полученный объект может быть использован в качестве базового, от которого, в свою очередь, можно получить другой объект (см. *Базовый объект*). Набор базового и полученных объектов в объектно-ориентированной программе создает иерархию взаимосвязанных объектов. Типичный код ООП состоит из множества таких объектных иерархий. В Turbo Assembler полученный объект может наследовать свойства только одного базового объекта (см. также *Одиночное* и *Множественное наследование*.)

Деструктор — специальный метод, который используется для разрушения объектной инстанции. Turbo Assembler не поддерживает концепцию деструктора.

Инкапсуляция — процесс связывания данных и кода в объект. Хотя это и не обязательно, код объекта (т.е. ассемблерные подпрограммы) обычно выполняют некоторые действия над или с закапсулированными в объекте данными. Инкапсуляция ограничивает использование данных определенным набором подпрограмм, что может упростить отладку, настройку и пересмотр программ.

Наследование — содержимое объекта, которое получено от другого объекта. Полученный объект *наследует* данные и код базового объекта. Используя наследование, можно изменять существующие объекты быстро и легко. См. также *Базовый объект, Полученный объект, Одиночное наследование* и *Множественное наследование*.

Инстанция — хранилище объектов, называемая также *Объектной инстанцией*. Инстанция объекта аналогична переменным таких типов, как байт и слово. В Turbo Assembler инстанции определяют, используя тот же синтаксис, что и для структур. (Инстанция является эквивалентом *объекта класса* C++.)

Член — любой компонент объекта. Например, метод является членом объекта, переменная в объекте — членом данных.

Метод — второй термин для подпрограмм объекта. См. также *Статический метод* и *Виртуальный метод*. (Метод является эквивалентом для *групповой функции* C++.)

Множественное наследование — свойство некоторых языков ООП, позволяющее получать новые объекты, используя наследование от более чем одного объекта. TASM не поддерживает множественное наследование (см. *Одиночное наследование*).

Объект — специальная структура, связывающая данные и код. Важно понять, что объект является просто *описанием в исходном тексте* взаимосвязанных данных и кода. Объекты существуют только в тексте программы; они не существуют во время ее исполнения. Чтобы использовать объект в программе, необходимо создать его *инстанцию*, аналогично тому, как создаются переменные других типов данных, таких как байты и слова. (Объект в Turbo Assembler является эквивалентом *класса* C++.)

Объектная инстанция — то же, что Инстанция.

Полиморфизм — процесс, посредством которого объектная инстанция может определить те действия, которые нужно выполнить над данным объектом или для него. Действие выполняется как виртуальный метод. Указатель (например, регистров ds:di) может адресовать инстанцию графического объекта, полученную из общей базы. Вызов виртуального метода Draw этой инстанции приводит к рисованию окружности, если указатель адресует инстанцию Circle, или прямоугольника — если инстанцию Rectangle. Нужная функция выбирается в процессе выполнения без точного программного определения типа объекта в вызывающей инструкции. Используя *полиморфизм*, можно модифицировать действия существующих кодов путем написания новых объектов и виртуальных методов. См. также *Виртуальный метод*.

Одиночное наследование — техника построения полученных объектов из одного базового объекта. Все языки ООП, включая Turbo Assembler, поддерживают одиночное наследование. См. также *Множественное наследование*.

Статический метод — подпрограмма объекта. Обращение к статическим методам аналогично обращениям к необъектно-ориентированным подпрограммам. Адреса статических методов объединяются в вызывающие инструкции во время компоновки.

Виртуальный метод — подпрограмма объекта. Обращение к виртуальным методам осуществляется непосредственно по адресам, хранящимся в таблице виртуальных методов объекта. Адреса виртуальных методов объединяются в вызывающую инструкцию во время выполнения. См. также *Полиморфизм*.

Таблица виртуальных методов, или TBM *(Virtual method table — VMT)* — таблица адресов виртуальных методов. Каждый объект, который обладает одним или более виртуальным методом, должен иметь соответствующую таблицу виртуальных методов. Создание этой таблицы и инициализация указателя на TBM в каждой объектной инстанции целиком лежит на вас.

Основы объектов TASM

Чтобы научиться пользоваться ООП в программах Turbo Assembler, вам необходимо овладеть тремя фундаментальными методиками:

- инкапсуляцией
- наследованием
- виртуальными методами

Вам нужно также научиться комбинировать эти методики, используя *полиморфизм*, для создания объектов, которые могут определять свои собственные действия. Далее в данной главе освещаются эти вопросы. Сперва я поясню инкапсуляцию, наследование и виртуальные методы в общих чертах, а затем покажу, как применять эти методики к объектам Turbo Assembler. В заключение я поясню, как создавать и использовать список объектов, который демонстрирует удивительный мир программирования с использованием полиморфизма.

Замечание

Руководство пользователя Borland написано исходя из предположения применения режима Ideal для объектно-ориентированных программ, но по непонятным причинам все примеры в этом руководстве применяют режим MASM И что еще хуже, многие из напечатанных примеров содержат ошибки и не работают надлежащим образом Не стоит и говорить, что эти факты помешали многим программистам, использующим язык ассемблера, применять объектно-ориентированные свойства TASM Все примеры программ в этой главе применяют режим Ideal Поскольку отсутствует какая-либо официальная документация по режиму Ideal и ООП, большинство синтаксических конструкций и примеров программ этой главы я получил экспериментальным путем

Инкапсуляция

Объекты аналогичны структурам, созданным посредством директивы STRUC. На тот случай, если вам необходимо освежить сведения по использованию структур ассемблера, следующий материал является кратким обзором по этому вопросу.

STRUC связывает несколько переменных под одним именем. Например, чтобы создать структуру, названную Point, можно использовать объявление, подобное следующему:

```
STRUC Point
 x dw ?
 y dw ?
ENDS Point
```

Это объявление создает структуру, названную Point и содержащую две переменные типа слова — x и y. Структура является просто *описанием* типа данных, она не занимает памяти во время выполнения. Чтобы использовать структуру, вы должны определить переменную, имеющую ее тип. Например, можно поместить в сегмент данных следующие инструкции:

```
DATASEG
p1   Point      <>
p2   Point      <45, 68>
```

Первая строка служит началом сегмента данных. Вторая строка определяет переменную структуры Point — иначе говоря, p1 является областью памяти, которая состоит из двух переменных типа слова, названных p1.x и p1.y. Третья строка также объявляет переменную p2 структуры Point. Кроме того, эта строка инициализирует ее две переменные типа слова значениями 45 и 68 соответственно.

Объекты создаются посредством использования специальной формы директивы STRUC. Действительно, объекты *являются* структурами — но, кроме того, что они содержат данные, объекты также специфицируют подпрограммы, названные *членами*, которые обычно выполняют действия над или с этими данными. Типичным действием членов является присваивание значений данным объекта. Другие члены могут возвращать значения данных. Члены могут выполнять также и другие дополнительные задачи.

Следующее объявление является примером объекта — TPoint, который объявляет четыре метода: два — для изменения переменных объекта x и y, два — для возвращения их значений:

```
STRUC TPoint METHOD {
 getx dword = TPoint_getx
 gety dword = TPoint_gety
 setx dword = TPoint_setx
 sety dword = TPoint_setx
 }
 x dw ?
 y dw ?
ENDS TPoint
```

Сравните эту директиву STRUC с необъектно-ориентированной структурой Point. Ключевое слово METHOD "говорит" ассемблеру, что эта структура специфицирует имена подпрограмм, которые должны быть связаны с объектом. Объявления подпрограмм в скобках следуют за ключевым словом METHOD. Каждое объявление осуществляется в форме

```
getx dword = TPoint_getx
```

Тем самым утверждается, что объект обладает связанным с ним методом, названным getx, и адрес этого метода должен храниться в указателе двойного слова (dword) (32 бита). (Программы, применяющие малую модель памяти, могут использовать смещение word вместо dword.) Указатель метода (getx) инициализируется по адресу действительной подпрограммы (TPoint_getx), которую вы должны написать где-нибудь в программе, используя директиву

PROC, как вы делаете это для других подпрограмм (естественно, законченный пример потребовал бы дополнительных инструкций):

```
PROC TPoint_getx PASCAL
     ret
ENDP TPoint_getx
```

Использованное мною соглашение относительно имен может показаться спорным, но оно достаточно хорошо работает. Я начинаю имена объектов с буквы T — это указывает на то, что объект является данными Типа. Имя метода (например, getx) описывает назначение подпрограммы объекта — в данном случае получение (get) значения переменной объекта x. Действительное имя подпрограммы в директиве PROC состоит из имени объекта, символа подчеркивания и имени метода (TPoint_getx). Эти соглашения помогают мне распознавать отношения между объектами, методами и подпрограммами.

Другие методы объекта TPoint — gety, setx и sete объявляются таким же образом. Каждый из указателей dword инициализируется по адресу действительной подпрограммы, исполняющейся где-нибудь в другом месте.

После методов объекта следуют все связанные с ними переменные, в данном случае два неинициализированных слова — x и y. Инстанции (т.е. переменные) объекта TPoint состоят из этих двух слов, точно так же, как в обычной структуре. Объект TPoint следует использовать, как любую другую структуру. Следующие операторы, например, определяют две инстанции TPoint:

```
p1 TPoint <>
p2 TPoint <12, 34>
```

Важно понять, что методы объекта TPoint *не* хранятся в самом объекте. Объект просто *устанавливает взаимосвязь* между кодом и данными — в действительности он не хранит код и данные в одном месте. Предшествующие две инстанции — p1 и p2 занимают по четыре байта каждая — как раз столько, сколько требуется для каждой из двух переменных инстанции типа слова — x и y.

Замечание

Предыдущий раздел станет для вас понятнее, если вы будете думать об объектах как о типах данных, аналогичных тем, что встроены в язык ассемблера — например, байтах и словах. Байт является типом данных, который просто описывает природу и размер определенного вида информации. Чтобы использовать байт, необходимо определить переменную этого типа, применяя директиву DB (define byte — определить байт). Операции, подобные сложению и вычитанию, которые можно производить над байтами, не хранятся внутри байтовых переменных. Они пишутся в подпрограммах, в которые вы пересылаете значения байтов. Отличие объектно-ориентированного программирования заключается в том, что вместо того, чтобы пересылать данные в подпрограммы, вы обращаетесь к методам для инстанций объектов. В этом смысле инстанция "знает", как выполнять операции над самой собой.

Перечисленные факты ведут к важному выводу: *объекты и структуры действительно являются одним и тем же*. Однако они отличаются способом использования. Структуры применяются, как любые другие переменные, но при использовании объектов необходимо вызывать *методы* для действий над данными инстанциями объекта. Чтобы помочь вам понять, как это происходит в действительности, следующие две программы приводят полную конструкцию объекта TPoint.

Программа-модуль 14.1, TPOINT.INC, показывает, как объявить и исполнить объект Turbo Assembler. Модуль создан таким образом, чтобы его можно было включать в программу с помощью директивы INCLUDE, поэтому я пока не пытаюсь ассемблировать его. Позднее я объясню, как это сделать. Теперь просмотрите TPOINT.INC, и мы строка за строкой рассмотрим текст модуля.

Замечание

Borland предполагает, что объявление объектов осуществляется в файлах, имеющих расширение .ASO (для ассемблерных объектов). Я вместо этого использую расширение .INC, поскольку мои текстовые редакторы запрограммированы распознавать это расширение. Вы можете называть свои файлы с модулями объектов, используя любое другое расширение, если желаете.

Программа 14.1. TPOINT.INC

```
1:  %TITLE  TPoint object -- by Tom Swan
2:
3:  GLOBAL TPoint_getx:PROC
4:  GLOBAL TPoint_gety PROC
5:  GLOBAL TPoint_setx.PROC
6:  GLOBAL TPoint_sety.PROC
```

```
7:
8:   STRUC TPoint METHOD {                    ; Начало объявления объекта TPoint
9:   getx:dword = TPoint_getx                 ; Возвращение данных объекта x
10:  gety:dword = TPoint_gety                 ; Возвращение данных объекта y
11:  setx:dword = TPoint_setx                 ; Изменение данных объекта x
12:  sety:dword = TPoint_sety                 ; Изменение данных объекта y
13:  }                                        ; Конец объявлений метода
14:  x dw ?                                   ; Данные объекта x
15:  y dw ?                                   ; Данные объекта y
16:  ENDS TPoint                              ; Конец объявления объекта TPoint
17:
18:  CODESEG
19:
20:  %NEWPAGE
21:  ;-----------------------------------------------------
22:  ; TPoint_getx Метод getx объекта TPoint
23:  ;-----------------------------------------------------
24:  ; Input:
25:  ; ds:si = адресу инстанции
26:  ; Output:
27:  ; ax = значению инстанции x
28:  ; Registers:
29:  ; ax
30:  ;-----------------------------------------------------
31:  PROC TPoint_getx PASCAL
32:       mov ax, [(TPoint PTR si).x]          ; Помещение значения инстанции x в ax
33:       ret                                  ; Возврат в точку вызова
34:  ENDP TPoint_getx
35:  %NEWPAGE
36:  ;-----------------------------------------------------
37:  ; TPoint_gety Метод gety объекта TPoint
38:  ;-----------------------------------------------------
39:  ; Input:
40:  ; ds:si = адресу инстанции
41:  ; Output:
42:  ; ax = значению инстанции y
43:  ; Registers:
44:  ; ax
45:  ;-----------------------------------------------------
46:  PROC TPoint_gety PASCAL
47:       mov ax, [(TPoint PTR si).y]          ; Помещение значения инстанции y в ax
48:       ret                                  ; Возврат в точку вызова
49:  ENDP TPoint_gety
50:  %NEWPAGE
51:  ;-----------------------------------------------------
52:  ; TPoint_setx Метод setx объекта TPoint
53:  ;-----------------------------------------------------
54:  ; Input:
55:  ; ds:si = адресу инстанции
56:  ; параметр x (word) (слово)
57:  ; Output:
58:  ; none
59:  ; Registers:
60:  ; ax
61:  ;-----------------------------------------------------
62:  PROC TPoint_setx PASCAL
63:       ARG @@x:word                         ; Создание смещения стека для параметра x
64:       USES ax                              ; Резервирование ax (не обязательно)
65:       mov ax, [@@x]                        ; Помещение значения параметра x в ax
66:       mov [(TPoint PTR si).x], ax          ; Помещение значения параметра x в инстанцию x
67:       ret                                  ; Возврат в точку вызова
68:  ENDP TPoint_setx
69:  %NEWPAGE
70:  ;-----------------------------------------------------
71:  ; TPoint_sety Метод sety объекта TPoint
72:  ;-----------------------------------------------------
73:  ; Input:
74:  ; ds:si = адресу инстанции
75:  ; параметр y (word) (слово)
76:  ; Output:
77:  ; none
78:  ; Registers:
79:  ; ax
80:  ;-----------------------------------------------------
```

```
81:  PROC TPoint_sety PASCAL
82:       ARG @@y:word                    ; Создание смещения стека для параметра y
83:       USES ax                         ; Резервирование ax (не обязательно)
84:       mov ax, [@@y]                   ; Помещение значения параметра y в ax
85:       mov [(TPoint PTR si).y], ax     ; Помещение значения параметра y в инстанцию y
86:       ret                             ; Возврат в точку вызова
87:  ENDP TPoint_sety
```

Строки 8–16 объявляют объект TPoint, содержащий четыре метода и две переменные. Модуль содержит также четыре оператора GLOBAL в строках 3–6, которые объявляют имена подпрограмм методов типа TPoint_getx таким образом, чтобы другие модули могли обращаться к ним.

Замечание

При использовании для экспорта символа, как это сделано здесь для методов TPoint, GLOBAL интерпретируется в качестве директивы PUBLIC. При использовании для импорта символа, как это может быть сделано другим модулем, которому необходимо воспользоваться объектом TPoint, GLOBAL интерпретируется как директива EXTRN. Применительно к методам объектов можно использовать директивы PUBLIC, EXTRN, но более принято применение директивы двойного назначения GLOBAL.

После этих объявлений в строке 18 модуль начинает или продолжает кодовый сегмент программы. За этим следуют воплощения методов объекта — иначе говоря, его подпрограммы, которые хранятся вместе с другими кодами программы. Метод TPoint_getx, например, выполнен в виде подпрограммы в строках 31–34.

Эта подпрограмма обладает только двумя инструкциями. Строка 32 помещает значение переменной объектной инстанции x в регистр ax. Строка 33 осуществляет возврат в точку вызова метода. Как демонстрирует эта часть распечатки, вы пишете методы объекта таким же образом, как обычные подпрограммы.

Однако здесь имеется одно главное отличие между TPoint_getx и обычным кодом. Как и все методы, TPoint_getx должен вызываться в ссылке на инстанцию объекта TPoint. По соглашению регистр ds:si адресует эту инстанцию.

Строка 32, например, получает значение переменной инстанции x, адресуя объектную инстанцию посредством ds:si. Внимательно просмотрите синтаксис этой строки — он отличается от синтаксиса, приведенного в руководстве пользователя Borland, которое не объясняет, как использовать режим Ideal совместно с возможностями ООП TASM. Необходимо применять квадратные скобки вокруг субвыражения (TPoint PTR si), чтобы ассемблер воспринял его как единое целое. Следует также сообщить ассемблеру тип объекта, адресуемого посредством ds:si (в этом примере TPoint). И наконец, вы должны включить директиву PTR, чтобы указать на непрямую ссылку к памяти.

Замечание

Вызов TPoint_getx требует особой формы инструкции call, обеспечивающейся директивой CALL...METHOD, которая является уникальной для Turbo Assembler. При обсуждении следующей программы я объясню, как пользоваться этой директивой.

Следующий метод в TPOINT.INC, TPoint_gety, идентичен TPoint_getx, но возвращает значение переменной объектной инстанции y (см. строки 46–49).

Еще два метода, TPoint_setx и TPoint_sety, завершают воплощение методов TPoint. Метод в строках 62–68 демонстрирует, как получить аргументы, передаваемые инструкциями, которые обращаются к методу. В данном случае TPoint_setx требует, чтобы вызывающая его процедура передала 16-битовое слово данного, хранимого в переменной объектной инстанции x (строка 63).

Вы можете передавать информацию методу, используя любую методику по своему желанию: например, с помощью регистра как глобальную переменную или посредством стека. В примере используется стековый аргумент, объявленный как

```
ARG @@x:word
```

Директива сообщает ассемблеру, что следует вычислить смещение в стеке 16-битового слова параметра и присвоить этому смещению имя @@x. Вы можете использовать любое имя по выбору — благодаря своему локальному префиксу символа (@@) использование символа ограничено текущей процедурой PROC. Это означает, что другая процедура PROC может определять аргумент с именем @@x, не вступая в конфликт с данной.

Замечание

При использовании ARG очень важно выбрать, кроме модели памяти, еще и соответствующий язык. Все методы в объекте TPoint (как и остальные в этой главе) используют модель PASCAL, которая возлагает на подпрограммы ответственность за очистку своих собственных стеков.

Следующая за директивой ARG TPoint_setx также сообщает ассемблеру, что она использует регистр ax (строка 64). Директива USES автоматически вставляет инструкции push и pop для сохранения и восстановления регистров. Вам не обязательно использовать USES, но принято делать это, чтобы быть уверенным, что подпрограмма сохраняет и восстанавливает критичные регистры. Несколько регистров разделяются запятыми, как в следующем примере:

```
USES ax, cx, si, es
```

Благодаря директиве ARG можно сослаться на аргументы, переданные в стек. Например, чтобы загрузить значение аргумента x в ax, подпрограмма исполняет следующую инструкцию в строке 65:

```
mov ax, [@@x]
```

Строка 66 сохраняет это значение в переменной объектной инстанции x. Метод TPoint_sety в строках 81–87 сходен с методом TPoint_setx, но помещает 16-битовый аргумент в переменную объектной инстанции y.

Следующим шагом является использование объекта Tpoint — путем включения его модуля в основную программу. Использование объекта подразумевает применение трех ключевых методик:

- Определение объектных инстанций
- Адресация объектных инстанций
- Вызов объектных методов

Программа 14.2, ENCAPSUL.ASM, демонстрирует эти методики. Теперь можно ассемблировать программу, содержащую модуль TPOINT.INC. Для этого вы можете ввести следующие две инструкции. В этом случае вы добавите отладочную информацию к программе ENCAPSUL.EXE, которая, подобно многим примерам этой книги, не вызывает никакого вывода на экран. Вам потребуется использование Turbo Debugger, как описано после текста программы, чтобы исследовать, как она работает:

```
tasm /zi encapsul
tlink /v encapsul
```

Программа 14.2. ENCAPSUL.ASM

```
1       %TITLE "TPoint object demonstration -- by Tom Swan"
2:
3:          IDEAL                       ; Выбор синтаксиса режима Ideal
4:
5:          JUMPS                       ; Разрешение автоматических условных переходов
6:
7:          LOCALS @@                   ; Разрешение ограничительных блочных меток
8:
9       MODEL large, PASCAL             ; Выбор модели памяти и языка
10.
11:         STACK 1000h                 ; Размер стека программы
12:
13:         INCLUDE "tpoint.inc"        ; Включение модуля объекта TPoint
14:
15          DATASEG                     ; Начало сегмента данных
16:
17      exCode DB 0                     ; Код выхода программы
18:
19:     ;----- Определение инстанций TPoint
20.
21.     p1 TPoint <>                    ; Первоначальная инстанция TPoint
22.     p2 TPoint <01h, 02h>            ; Инициализированная инстанция TPoint
23.
24          CODESEG                     ; Начало кодового сегмента
25:
26:     Start:
27:         mov ax, @data               ; Инициализация DS по адресу
28:         mov ds, ax                  ; сегмента данных
29:
30:     ;----- Вызов методов TPoint
31:
32:         mov si, offset p1           ; Адресация инстанции посредством ds:si
33:         CALL si METHOD TPoint:getx  ; Вызов объектного метода
34:
35:         mov si, offset p2           ; Адресация инстанции посредством ds:si
36:         CALL si METHOD TPoint:gety  ; Вызов объектного метода
```

```
37:
38:     ;----- Передача численных аргументов методам
39:
40:          mov si, offset p1                , Адресация инстанции посредством ds:si
41:          CALL si METHOD TPoint:setx, 03h  ; Передача аргумента методу
42:
43:          mov si, offset p1                , Адресация инстанции посредством ds:si
44:          CALL si METHOD TPoint.sety, 04h  ; Передача аргумента методу
45:
46:     ;----- Передача регистровых аргументов методам
47:
48:          mov si, offset p2                ; Адресация инстанции посредством ds:si
49:          mov dx, 05h                      ; Загрузка аргумента в dx
50:          CALL si METHOD TPoint:setx, dx   ; Передача dx в стеке методу
51:
52:          mov si, offset p2                ; Адресация инстанции посредством ds:si
53:          mov cx, 06h                      ; Загрузка аргумента в cx
54:          CALL si METHOD TPoint:sety, cx   ; Передача cx в стеке методу
55:
56:     Exit:
57:          mov ah, 04Ch                     ; Функция ДОС: Выход из программы
58:          mov al, [exCode]                 ; Возвращение значения кода выхода
59:          int 21h                          ; Вызов ДОС. Прерывание программы
60:
61:          END Start                        ; Конец программы / точка входа
```

В начале объектно-ориентированной ассемблерной программы требуются несколько директив. Вы можете поэкспериментировать, варьируя типами и количеством директив, но я нахожу, что в большинстве случаев лучше всего работают следующие директивы:

```
IDEAL
JUMPS
LOCALS @@
MODEL large,PASCAL
STACK 1000h
```

Вы обнаружите их и в других программах этой главы (см. строки 3–11 в программе 14.2). Первая строка выбирает режим Ideal Turbo Assembler. Кроме других преимуществ (обсуждаемых в этой книге), которыми обладает режим Ideal, он делает символы структуры локальными для этой структуры. В режиме MASM символы структуры являются глобальными и должны быть уникальными во всей программе. По этой причине режим Ideal требует директив GLOBAL, но несмотря на это дополнительное усложнение, локальные структурные символы упрощают программирование, исключая возможность конфликта между различными структурами.

Директива JUMPS разрешает автоматические условные переходы, давая возможность ассемблеру сгенерировать более эффективный код. Директива LOCALS объявляет @@ в качестве префикса локальных символов. В ООП вам придется применять много локальных символов. Использование локального префикса предотвратит конфликты, которые могли возникнуть, если бы вы объявили символы, подобные @@x и @@y, глобальными. Кроме того, некоторые директивы ООП генерируют код, который требует префикс локального символа.

Директива MODEL в этом примере (строка 9) выбирает большую модель памяти. Поскольку объектно-ориентированные программы имеют тенденцию к увеличению, использование этой модели правомерно. Можно, однако, писать текст программы, применяя и малую, и огромные модели памяти, как я поясню далее в этой главе, но нюансы адресации при использовании малой модели могут оказаться сложными. Для получения лучших результатов следует применять большую модель, пока вы не будете уверены, что это можно обойти.

Замечание

Библиотека MTA.LIB, созданная вами ранее при изучении предыдущих разделов этой книги, ассемблирована для малой модели памяти. Если вы подключаете к этой библиотеке объектно-ориентированную программу, применяющую большую модель памяти, то вам сначала необходимо путем редактирования директив MODEL создать версии модулей библиотеки, использующие большую модель. Например, чтобы создать версию модуля STRINGS, использующую большую модель, замените в STRINGS.ASM MODEL на large, затем реассемблируйте и поместите модуль в MTA.LIB.

Директива MODEL в строке 9 также специфицирует язык Pascal. Это не означает, что программа написана для Pascal. Она просто меняет код, вставляемый ассемблером для директив PROC и ENDP. При использовании модели языка Pascal вы объявляете и передаете аргументы в стек в том же самом порядке. Например, если метод требует аргументов x и y, вы должны объявить и передать их в этом же порядке. Кроме того, модель Pascal приводит к то-

му, что ассемблер удаляет аргументы из стека путем вставки особой формы инструкции ret, которая настраивает указатель стека — sp. Другие модели (например, модель C) требуют того, чтобы *вызывающая процедура* очищала стек. В основном это не принято, и поскольку подпрограммы в ООП имеют тенденцию использовать множество аргументов, передаваемых методам, Pascal является наилучшим выбором.

И наконец, программа определяет стек (строка 11). И снова, из-за интенсивного применения в программе ООП стековых аргументов, может потребоваться стек больших размеров, чем обычно. Для всех программ в этой главе я использовал 1000h. Вы можете увеличить это значение для больших программ со многими объектами.

Чтобы можно было использовать объект TPOint, приведенная в качестве примера программа включает в себя модуль TPOINT.INC (строка 13). Если ваша программа применяет более одного объекта, то в этом месте должны содержаться все модули.

Вслед за этим программа определяет глобальные переменные, две из которых — объектные инстанции. Сперва директива DATASEG в строке 15 начинает глобальный сегмент данных программы. Переменная кода выхода в строке 17 является той же, что и используемая в большинстве программ этой книги. Строки 21, 22 демонстрируют два способа определения объектных инстанций.

Первая строка (21) создает инстанцию с именем p1 для объекта TPoint. Поскольку угловые скобки в этом операторе пусты, значения переменных x и y инстанции неинициализированные. При просмотре в Turbo Debugger они устанавливаются в нуль, но при нормальном использовании программы они могут оказаться равными любым значениям, загруженным в память.

Вторая строка (22) определяет вторую объектную инстанцию, но при этом специфицирует начальные значения для переменных инстанции. Эта строка создает инстанцию с x, установленным в 01h, и y, установленным в 02h.

Далее программа демонстрирует, как адресовать объектные инстанции и как вызывать объектные методы. Существует одно очень важное правило, которое следует запомнить: *объектный метод надлежит вызывать ссылкой на объектную инстанцию* Другими словами, никогда нельзя вызывать метод по контексту; вместо этого необходимо специфицировать объектную инстанцию, над которой производит действия метод.

Имеется много способов адресовать объектные инстанции — можно передавать их адреса в стековые переменные либо адресовать их, используя любую комбинацию регистров по своему выбору. Регистровая адресация, вероятно, предпочтительнее и рациональнее использовать одни и те же регистры для адресации объектных инстанций во всей программе. Лично я использую ds:si.

Поскольку инстанции приведенной в качестве примера программы находятся в сегменте данных, регистр ds уже инициализирован подготовительными инструкциями в строках 27, 28. Для адресации инстанции p1 требуется только один шаг:

```
mov si, offset p1
```

Эта инструкция помещает смещение адреса инстанции p1 в si. Теперь ds:si адресует инстанцию объекта TPoint надлежащим образом, и программа может вызвать любой из объектных методов для выполнения действий над этой инстанцией или для нее. Например, чтобы вызвать метод TPoint_getx, который возвращает переменную x инстанции, строка 33 выполняет особую форму инструкции call:

```
CALL si METHOD TPoint getx
```

Действительно, это не является инструкцией ассемблера — данная директива CALL...METHOD уникальна для Turbo Assembler. Чтобы отличать эту директиву от обычных вызовов подпрограмм call, я набираю ее прописными буквами, но вы, если желаете, можете использовать строчные. Синтаксис директивы CALL...METHOD несколько сложнее:

```
CALL <instance_ptr> METHOD {<имя объекта> }
 <имя_метода> {USES {segreg }offsreg} {<расширенные_параметры_вызова>}
```

Первый элемент, <instance_ptr>, может быть адресом объекта или ссылкой на регистр. Поскольку я всегда адресую инстанцию посредством ds:si, то помещаю si между ключевыми словами CALL и METHOD. Это удовлетворяет требованиям синтаксиса, но в данном случае регистр больше никак не используется. (Позднее в этой главе, когда вы приступите к изучению виртуальных методов, эта часть синтаксиса CALL...METHOD будет иметь большее значение.)

Далее, CALL...METHOD дает возможность специфицировать имя объекта. *Всегда* делайте это. Вы должны сослаться на объект по имени (особенно в режиме Ideal), чтобы сослаться, также на любой из членов этого объекта. В данном случае вам необходимо вставить имя метода, который вы хотите вызвать, — метод TPoint:getx, например, как показано в строке 33.

Замечание

В операторе CALL ..METHOD специфицируйте метод набором имени объекта, двоеточия и имени метода. Не используйте действительное имя подпрограммы. Например, как показывает строка 33, правильным способом ссылки на метод getx объекта TPoint является TPoint:getx. В действительности подпрограмма названа TPoint_getx в модуле TPOINT.INC.

Как показывает синтаксис CALL...METHOD, можно специфицировать предложение USES в этой директиве для резервирования любых регистров, которые изменяет метод. Я предпочитаю, чтобы сами методы резервировали все регистры, кроме тех, что используются для передачи информации обратно вызывающим процедурам, поэтому редко вставляю USES в оператор CALL...METHOD. Однако если вы хотите использовать эту опцию, наберите ее, подобно следующей:

```
CALL si METHOD TPoint·getx USES cx, di
```

И наконец, в инструкции CALL...METHOD вы можете перечислить любые аргументы, которые должны быть вытолкнуты в стек. Эти аргументы могут быть численными значениями, ссылками к памяти или регистрами. Независимо от формы *они всегда передаются в стек*. Например, чтобы передать значение 03h методу setx объекта TPoint, строка 41 использует инструкцию

```
CALL si METHOD TPoint:setx, 03h
```

Следуя этой директиве, Turbo Assembler генерирует инструкции для выталкивания 03h в стек. Метод setx, как я объяснил для модуля TPOINT.INC, использует директиву ARG для доступа к этому аргументу.

Можно также передавать методу регистровые значения. Например, можно поместить значение в cx (или другой регистр) и передать это значение посредством инструкции:

```
mov cx, 04h
CALL si METHOD TPoint:setx, cx
```

Однако, несмотря на видимость, вторая строка *не* передает значение регистра в cx методу setx. Она выталкивает значение cx в стек, и метод по-прежнему должен использовать директиву ARG для доступа к этому значению. (См. также строки 48–50 в качестве другого примера передачи регистровых значений методу.)

Замечание

Методы могут использовать значения, переданные в регистры. Однако если вы специфицируете эти регистры как аргументы CALL...METHOD, то они по-прежнему будут вытолкнуты в стек и вы должны *объявить* директиву ARG для использования этих аргументов. Это дает возможность Turbo Assembler сгенерировать инструкцию возврата, которая удаляет „вытолкнутые" байты аргумента путем настройки указателя стека. Если вы где-то не используете ARG, стек будет заполняться повторно и вам все равно придется проверять, все ли методы специфицируют директивы ARG для *каждого* аргумента в директивах CALL...METHOD.

На данном этапе весьма полезно запустить демонстрационную программу ENCAPSUL в Turbo Debugger. Последуйте некоторым советам, чтобы изучить, как работает программа.

1. Создайте программный файл ENCAPSUL.EXE, если вы еще не сделали этого. Введите td encapsul, чтобы запустить Turbo Debugger и загрузить демонстрационную программу.
2. Используйте клавиши стрелок для перемещения мерцающего курсора к инстанции p2 и нажмите <Ctrl>+<W> для установки инсталяции ее в окно Watches. Проделайте то же самое для p1. Окно Watches должно теперь иметь два входа для TPoint. Они отображаются как переменные "struc", чем в действительности и являются объектные инстанции. Обратите внимание также, что переменные x и y в p2 инициализируются значениями в угловых скобках в определении инстанции.
3. Трижды нажмите <F7>, чтобы выполнить инструкции, инициализирующие ds и адресующие p1 посредством ds:si. Снова нажмите <F7>, чтобы выполнить первую инструкцию CALL...METHOD. На экране отобразится модуль TPOINT.INC и курсор установится на инструкции mov в методе TPoint_getx.
4. Нажмите <Alt>+<V><R>, чтобы вызвать окно Registers, затем нажмите <F7>, чтобы выполнить инструкцию mov. Обратите внимание, что ax изменяется на значение адресуемой переменной инстанции x. Снова нажмите <F7>, чтобы выполнить инструкцию метода ret, которая завершает этот CALL...METHOD.
5. Нажмите <F7> еще четыре раза, чтобы выполнить оператор CALL...METHOD, и наблюдайте, как используются модифицированные регистры, которые Turbo Debugger подсвечивает. Эти шаги вернут значение переменной y инстанции p2.
6. Теперь программа приостановлена на инструкции, которая помещает смещение инстанции p1 в si. Нажмите <F7>, чтобы выполнить эту инструкцию. Теперь регистры ds:si адресуют инстанцию p1.
7. Перед выполнением следующего оператора CALL...METHOD откройте окно CPU (нажмите <Alt+V, C> и затем <F5>, чтобы распахнуть окно на весь экран). Вы обнаружите инструкции, которые выглядят наподобие следующих:

```
pushax
pushbp
mov bp,sp
mov word ptr [bp+02],0003
pop bp
pushcs
call tpoint_setx
nop
```

Вы видите реальные инструкции, которые Turbo Assembler генерирует для команды CALL...METHOD (в строке 41 текста программы). Первые пять инструкций "сверлят отверстие" в стеке, создавая пространство для аргумента, который должен быть передан методу. Инструкция push cs вызывает *дальний* вызов (far call), после которого *ближний* вызов (near call) осуществляет действительное обращение к подпрограмме метода. Nop является заполнителем, оставшимся после оптимизации, которую выполняет TASM для преобразования дальних вызовов в эффективные инструкции push cs и ближний вызов. Этот nop занимает байт, но конечный результат все же быстрее, чем эквивалентный дальний вызов. (Ассемблер осуществляет эту модификацию для всех дальних вызовов подпрограмм, а не только для объектно-ориентированных директив CALL...METHOD.)

Используйте клавишу <F7> Turbo Assembler для запуска остающихся инструкций. Вы можете сделать это, просматривая окна Module или CPU. В окне Module вы выполняете CALL...METHOD и другие инструкции как отдельные команды, даже если они содержат несколько шагов. В окне CPU вы выполняете эти шаги по отдельности. Проведите выполнение программы обоими способами, чтобы побольше узнать о том, как она работает. Нажмите <Alt+X>, чтобы выйти из отладчика.

Перед тем как приступить к чтению следующего раздела, проверьте, все ли вы поняли из предыдущего материала.

- Как объявлять объект и использовать директивы GLOBAL для его методов (первая часть текста программы 14.1).
- Как воплотить объектный метод и адресовать объектные инстанции (подпрограммы во второй части программы 14.1).
- Как определять объектную инстанцию (сегмент данных в программе 14.2).
- Как адресовать объектную инстанцию и вызывать ее методы (кодовый сегмент текста программы 14.2).

Наследование

Используя наследование, вы создаете новые объекты из существующих. Новый, или *полученный объект*, наследует методы и переменные своего родителя, или *базового объекта*. Другими словами, полученный объект является *копией* базового объекта, в которую можно добавлять новые методы и переменные.

Эти добавленные методы могут быть полностью новыми либо *замещать* методы с этими же именами в базовом объекте. Кроме того, замещающие методы в полученном объекте могут обращаться к тем методам базового объекта, которые они замещают. Нельзя замещать члены данных объекта — можно только его методы. Однако можно добавлять новые члены данных в полученные объекты.

В этом разделе вы узнаете, как использовать наследование для создания полученных объектов в Turbo Assembler. Следующие три программы демонстрируют:

- как получать новый объект, базирующийся на существующем,
- как вызывать методы полученного объекта,
- как замещающий метод может обращаться к методу базового объекта.

Программа 14.3 объявляет базовый объект, названный для простоты TBase. (Для своих объектов вы можете использовать любое имя по своему усмотрению.) Объект не обладает никакой практической ценностью, но может быть полезным в качестве шаблона для ваших собственных экспериментов с ООП. Сам я часто создаю подобные объекты для экспериментирования с ними перед тем, как воплотить идеи в реальную программу. Не пытайтесь пока ассемблировать эту программу — я объясню, как это сделать, в надлежащее время.

Программа 14.3. TBASE.INC

```
1:  %TITLE  TBase object -- by Tom Swan"
2:
3:  GLOBAL TBase_init PROC
4:  GLOBAL TBase_getData·PROC
5
6:  STRUC TBase METHOD {                ; Объявление базового объекта
7:  init:dword = TBase_init             ; Метод объекта TBase
8:  getData:dword = TBase_getData       ; Метод объекта TBase
9:  }                                   ; Конец объектных методов
10: TBase_data dw ?                     ; Данные объекта TBase
```

```
11:  ENDS TBase                          ; Конец базового объекта
12:
13:  CODESEG
14:
15:  ;-----------------------------------------------------
16:  ; TBase_init Метод init объекта TBase
17:  ;-----------------------------------------------------
18:  ; Input:
19:  ; ds:si = адресу инстанции
20:  ; arg1 = слову, которое должно храниться в инстанции
21:  ; Output:
22:  ; arg1 -> инстанцию TBase_data
23:  ; Registers:
24:  ; none
25:  ;-----------------------------------------------------
26:  PROC TBase_init PASCAL
27:       ARG @@data:word                 ; Создание смещения аргумента в стеке
28:       USES ax                         ; Резервирование регистра ax (не обязательно)
29:       mov ax, [@@data]                ; Помещение аргумента в ax
30:       mov [(TBase PTR si).TBase_data], ax ; Сохранение ax в инстанции
31:       ret
32:  ENDP TBase_init
33:
34:  ;-----------------------------------------------------
35:  ; TBase_getData Метод getData объекта TBase
36:  ;-----------------------------------------------------
37:  ; Input:
38:  ; ds:si = адресу инстанции
39:  ; Output:
40:  ; ax = инстанции TBase_data
41:  ; Registers:
42:  ; ax
43:  ;-----------------------------------------------------
44:  PROC TBase_getData PASCAL
45:       mov ax, [(TBase PTR si).TBase_data]   ; ax <- базовые данные
46:       ret
47:  ENDP TBase_getData
```

Объект TBase объявляет одну переменную (TBase_data) и два метода (строки 3, 4 и 6–11). Первый метод, TBase_init, инициализирует инстанцию объекта TBase, т.е. присваивает переменным инстанции или другим переменным определенные значения. Второй метод, TBase_getData, возвращает переменные инстанции или другие переменные.

Затем в модуле реализуются объектные методы. Строки 26–32 программируют метод TBase_init, которому требуется аргумент типа слово, переданный посредством инструкции CALL...METHOD. Метод размещает этот аргумент в переменной TBase_data.

Метод TBase_getData возвращает переменную TBase_data объектной инстанции в регистр ax.

Программа 14.4, TDERIVED.INC, показывает, как получить новый объект, используя TBase. В тексте программы показывается отношение между базовым и полученным объектом, а также демонстрируется несколько имеющих к этому отношение технических приемов. Перед использованием модуля вам придется изучить еще одну программу, поэтому пока не спешите ассемблировать эту программу.

Программа 14.4. TDERIVED.INC

```
1:   %TITLE "TDerived object -- by Tom Swan"
2:
3:   GLOBAL TDerived_init:PROC
4:   GLOBAL TDerived_getData:PROC
5:
6:   STRUC TDerived TBase METHOD {          ; Объявление получения объекта из базового
7:        init:dword = TDerived_init        ; Метод объекта TDerived
8:        getData:dword = TDerived_getData  ; Метод объекта TDerived
9:   }                                      ; Конец объектных методов
10:  TDerived_data dw ?                     ; Данные объекта TDerived
11:  ENDS TDerived                          ; Конец полученного объекта
12:
13:  CODESEG
14:
15:  ;-----------------------------------------------------
16:  ; TDerived_init Метод init объекта TDerived
```

```
17:  ;-------------------------------------------------------
18:  ; Input:
19:  ; ds:si = адресу инстанции
20:  ; arg1 = слову, подлежащему хранению в данных базовой инстанции
21:  ; arg2 = слову, подлежащему хранению в данных полученной инстанции
22:  ; Выход:
23:  ; arg1 -> в инстанцию.TBase_data
24:  ; arg2 -> в инстанцию.TDerived_data
25:  ; Регистры:
26:  ; никаких
27:  ;-------------------------------------------------------
28:  PROC TDerived_init PASCAL
29:       ARG @@data1:word, \
30:            @@data2:word
31:       USES ax
32:       mov ax, [@@data1]                    ; Помещение arg1 в ax *
33:       CALL si METHOD TBase:init, ax        ; Вызов базового метода init *
34:       mov ax, [@@data2]                    ; Помещение arg2 в ax
35:       mov [(TDerived PTR si).TDerived_data], ax ; Сохранение в инстанции
36:       ret
37:  ENDP TDerived_init
38:
39:  ; ----- * Эти операторы - mov и call можно также записать следующим образом:
40:  ; CALL si METHOD TBase:init, [@@data1]
41:
42:  ;-------------------------------------------------------
43:  ; TDerived_getData Метод getData объекта TDerived
44:  ;-------------------------------------------------------
45:  ; Вход:
46:  ; ds:si = адресу инстанции
47:  ; Выход:
48:  ; ax = инстанции.TBase_data
49:  ; dx = инстанции.TDerived_data
50:  ; Регистры:
51:  ; ax, dx
52:  ;-------------------------------------------------------
53:  PROC TDerived_getData PASCAL
54:       CALL si METHOD TBase:getData              ; ax <- базовые данные
55:       mov dx, [(TDerived PTR si).TDerived_data] ; dx <- полученные данные
56:       ret
57:  ENDP TDerived_getData
```

Вы объявляете полученный объект, как и любой другой, но с одним отличием. После имени нового объекта вставляется имя базового объекта. Например, строка 6 объявляет объект TDerived следующим образом:

```
STRUC TDerived TBase METHOD
```

Это объявление приводит к тому, что TDerived появляется в виде копии TBase. Другими словами, TDerived наследует переменные и методы от TBase. В дополнение к унаследованному, TDerived объявляет два замещающих метода и новый член данных.

Член данных, TDerived_data, добавляется к переменной TBase_data, унаследованной от базового объекта. Другими словами, инстанции объекта TDerived теперь содержат *две* переменные — одну, названную TBase_data, и одну — TDerived_data.

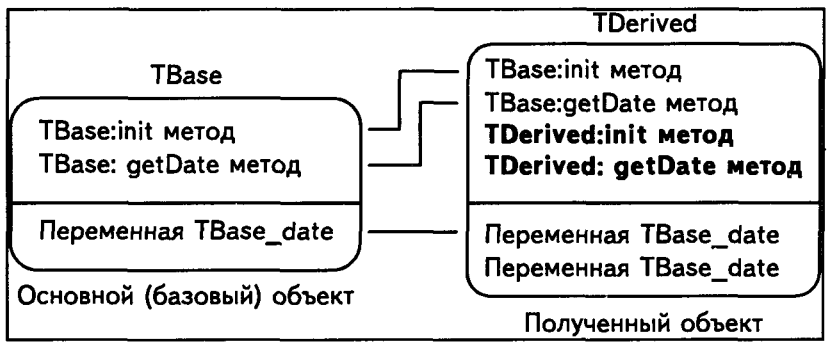


Рис. 14.2. Базовый и полученный объекты

Концептуально, базовый и полученный объекты имеют структуру, изображенную на рис. 14.2. Полученный объект *наследует* члены своего базового объекта. Новые члены полученного объекта показаны выделенным шрифтом.

Два метода, init и getData, *замещают* унаследованные от TBase. Можно было бы полученному объекту объявить совершенно новые методы, просто дав им уникальные имена, но приведенный объект этого не делает.

Как иллюстрирует рис. 14.2, новые методы не уничтожают методы, которые они замещают, — подпрограммы TBase по-прежнему "живут и здравствуют" в своем первоначальном модуле. Когда программа обращается к замещающим полученным методам, она вызывает замещающий код. Зачастую этот код, в свою очередь, обращается к методу базового объекта для выполнения части требуемых действий в дополнение к новой программе, осуществляемой замещающим методом. Это не является обязательным, и замещающие методы иногда не обращаются к унаследованным методам.

Например, строки 28–37 осуществляют замещение метода init для объекта TDerived. Замещающий метод требует двух 16-битовых аргументов, которые должны быть сохранены в переменных инстанции — инстанции TDerived теперь содержат *две* такие переменные.

Чтобы инициализировать унаследованную переменную TBase_data, строки 32, 33 вызывают метод init базового объекта. Регистры ds:si уже адресуют инстанцию, поэтому они не требуют инициализации. Программа просто загружает ax первым из двух аргументов подпрограммы и вызывает метод init объекта TBase.

Эти шаги инициализируют унаследованную часть объектной инстанции. Чтобы довершить дело, строки 34, 35 сохраняют второй аргумент в новой переменной TDerived_data, которую TDerived прибавляет к своим унаследованным членам. Теперь обе переменные инстанции данных инициализированы.

Метод TDerived_getData аналогично вызывает метод своего базового объекта с тем же именем (getData) для получения данных инстанции (строка 54). Следующая инструкция помещает данные инстанции полученного объекта в dx. Таким способом замещающий метод возвращает две переменные инстанции в регистры ax и dx. Особо обратите внимание на то, как методы полученного объекта обращаются к методам своего базового объекта для построения уже существующего кода. Эти приемы — расширение объектов посредством наследования и замещения методов — сердце и душа объектно-ориентированного программирования.

Программа 14.5, INHERIT.ASM, показывает, как использовать базовый и полученный объекты из предыдущих двух распечаток программ. Теперь введите тексты этих программ и можете ассемблировать и компоновать демонстрационную программу, которая включает в себя модули TBASE.INC и TDERIVED.INC. Выполните следующие отдельные команды:

```
tasm /zi inherit
tlink /v inherit
```

Замечание

Демонстрационная программа не производит никакой информации для вывода. Чтобы лучше разобраться, как работает программа, загрузите ее в Turbo Debugger с помощью командыtd inherit. Прибавьте переменные программы к окну *Watches* и используйте клавишу <F7> отладчика для пошагового прохождения программы, читая строчку за строчкой объяснения, следующие за приведенной ниже распечаткой текста программы.

Программа 14.5. INHERIT.ASM

```
1:  %TITLE "Inheritance demonstration -- by Tom Swan"
2:
3:      IDEAL
4:
5:      JUMPS
6:
7:      LOCALS @@
8:
9:      MODEL large, PASCAL
10:
11:     STACK 1000H
12:
13:     INCLUDE "tbase.inc"
14:
15:     INCLUDE "tderived.inc"
16:
17:     DATASEG
18:
19: exCode db 0                         ; Код выхода программы
20:
21: b1 TBase <>                         ; Определение инстанции базового объекта
22:
23: d1 TDerived <>                      ; Определение инстанции полученного объекта
```

```
24:
25:       CODESEG
26:
27:  Start:
28:       mov ax, @data                    ; Инициализация DS по адресу
29:       mov ds, ax                       ; сегмента данных
30:
31:       mov si, offset b1                ; Адресация инстанции b1
32:       CALL si METHOD TBase:init, \     ; Вызов базового метода init
33:             0001h                      ; Передача аргумента методу
34:
35:       mov si, offset d1                ; Адресация инстанции d1
36:       CALL si METHOD TDerived:init, \  ; Вызов полученного метода init
37:             0002h, 0003h               ; Передача аргументов методу
38:
39:       mov si, offset b1                ; Адресация инстанции b1
40:       CALL si METHOD TBase:getData     ; Получение данных в ax
41:
42:       mov si, offset d1                ; Адресация инстанции d1
43:       CALL si METHOD TDerived:getData  ; Получение данных в ax, dx
44:
45:  Exit:
46:       mov ah, 04Ch                     ; Функция DOC: Выход из программы
47:       mov al, [exCode]                 ; Возвращение значения кода выхода
48:       int 21h                          ; Вызов DOC. Прерывание программы
49:
50:       END Start                        ; Конец программы / точка входа
```

Как показывают строки 21-23, вы определяете инстанции полученного объекта аналогично инстанциям базового. Полученный объект используется так же, как и любой другой. В действительности, как вы увидите позже, полученный объект может сам быть базовым для другого объекта. Практически не существует ограничения на число объектов, которые можно получить из других.

Строки 31–33 вызывают метод init базового объекта, которому оператор CALL...METHOD передает значение 0001h. Когда вы проследите этот код в Turbo Debugger, то увидите, что метод сохраняет и передает значение аргумента в переменную инстанции TBase_data.

Замечание

Инструкции CALL...METHOD могут быть достаточно длинными. Чтобы они лучше читались, можно писать их отдельными строками, как показано здесь. Заканчивайте каждую предыдущую строку "символом продолжения" — обратной косой чертой (\) (см., например, строки 32 и 36).

Строки 35–37 выполняют аналогичную задачу, но вызывают метод init полученного объекта. Этот метод требует *двух* аргументов, в данном случае — численных значений 0002h и 0003h. При прослеживании этой инструкции в Turbo Debugger сначала вы натолкнетесь на метод init объекта TDerived. Этот метод вызывает init объекта TBase, который сохраняет первый аргумент в переменной инстанции TBase_data. Затем полученный init сохраняет второй аргумент в TDerived_data. Таким образом, два метода инициализируют две переменные объекта.

Строки 39–43 вызывают методы getData объектов TBase и TDerived для восстановления значений переменных инстанций. Откройте окно Registers (Регистры) (нажмите <Alt>+V, R>), чтобы просмотреть эти значения при прослеживании этой части программы.

Прежде чем идти дальше, убедитесь, что вы разобрались в следующих вопросах.

- Как получить объект из базового.
- Как определять базовые и полученные инстанции.
- Как обращаться к базовым и полученным методам.
- Как вызывать метод базового объекта из замещающего метода полученного объекта.

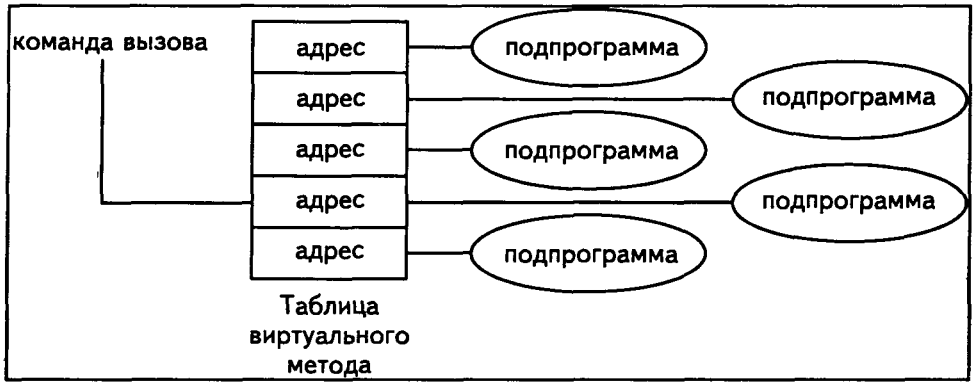


Рис. 14.3. Обращение к виртуальным методам осуществляется косвенно путем просмотра адресов подпрограмм в таблице виртуальных методов (ТВМ) во время выполнения программы

Виртуальные методы

Объектные методы, с которыми вы знакомились, являются *статическими методами*. Их адреса фиксируются в памяти на долгое время, и, следовательно, Turbo Assembler может создавать обычные инструкции call для подпрограмм объекта.

Виртуальные методы отличаются от статических методов способом их адресации. Вместо того чтобы вычислять адрес виртуального метода во время ассемблирования, ассемблер генерирует инструкции, которые извлекают адрес из *таблицы виртуальных методов* (ТВМ) во время выполнения программы. Обращения к виртуальным методам — косвенные, они осуществляются путем ссылки на вход в ТВМ (см. рис. 14.3).

Каждый объект, обладающий одним (или более) виртуальным методом, должен иметь ТВМ, и каждая инстанция объекта должна иметь указатель ТВМ, адресующий эту ТВМ. Директивы CALL...METHOD автоматически извлекают адреса, но вы должны сами создать ТВМ и подключить ее к каждой объектной инстанции путем инициализации ее указателя в ТВМ.

Эти основополагающие факты поясняют, что из себя представляют виртуальные методы, но они не объясняют, зачем они могут вам понадобиться. Виртуальные методы являются наиболее мощным инструментом в ООП, но их значение может быть сперва неочевидным. Вкратце, виртуальные методы дают возможность программам использовать *полиморфизм* — мудреное слово, за которым скрывается относительно простое понятие, поясняемое программой в этом разделе.

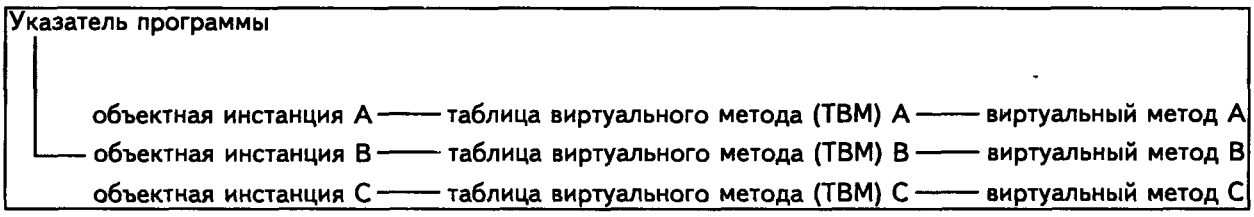


Рис. 14.4. Когда указатель адресует объектную инстанцию, обращения к виртуальным методам объекта осуществляется путем их просмотра в таблице виртуальных методов

Рис. 14.4 наглядно иллюстрирует полиморфизм. На диаграмме инструкция CALL...METHOD делает попытку вызвать метод для объектной инстанции, адресуемой программным указателем (ds:si, например). Поскольку метод виртуальный, его адрес берется из ТВМ, которая адресуется указателем, хранящимся в объектной инстанции. Посмотрим, что произойдет, если программная инструкция изменит этот указатель, чтобы адресовать другую объектную инстанцию. Та же самая инструкция CALL...METHOD в этом случае обратится к другому виртуальному методу. В этом и состоит полиморфизм — объектные инстанции определяют во время выполнения, какие виртуальные методы вызывать. Другими словами, инстанции *определяют свои собственные действия*.

Воплощаются виртуальные методы так же, как и статические. Только способ обращения к виртуальным методам отличает их от статических методов.

Для дополнения объекта виртуальными методами требуется одно новое ключевое слово и три новые директивы. Проделайте следующие шаги.

1. Введите один (или более) виртуальный метод в объявление объекта, предварив объявление метода словом VIRTUAL.

2. Введите указатель TBM в раздел данных объекта, воспользовавшись директивой TBLPTR.
3. Определите область памяти для TBM, применив директиву TBLINST.
4. Инициализируйте указатель TBM для каждой объектной инстанции, воспользовавшись директивой TBLINIT.

Несколько фрагментов программы иллюстрируют эти четыре шага. Пусть имеется объект, TBase, который объявляет виртуальный метод и указатель TBM:

```
STRUC TBase METHOD {
 construct:dword = TBase_construct          ; Статический метод
 VIRTUAL action:dword = TBase_action        ; Виртуальный метод (шаг #1)
}
 TBLPTR                                     ; Указатель TBM (шаг #2)
 TBase_data dw ?                            ; Другие данные объекта
ENDS TBase
```

Объект может содержать другие статические методы, и он может содержать столько виртуальных методов, сколько требуется. Каждый из виртуальных методов должен предваряться ключевым словом VIRTUAL.

Замечание

Объект, полученный из TBase, наследует указатель TBM, созданный посредством TBLPTR. Полученный объект не должен определять свой собственный указатель TBM. Только базовый объект в иерархии родственных объектов должен использовать эту директиву.

После объявления объекта необходимо объявить таблицу виртуальных методов, в которой хранятся адреса виртуальных методов объекта. Turbo Assembler автоматически вставляет надлежащие адреса в эту таблицу. Все, что вы должны сделать, — это создать ее. Но вы *должны* создать TBM для каждого объекта, который обладает одним (или более) виртуальным методом, — это правило одинаково относится как к базовым, так и к полученным объектам. Если базовый объект определяет TBM, объект, полученный из этого базового, также должен определять свою собственную TBM. Полученный объект наследует только *указатель* TBM базового объекта, но не таблицу виртуальных методов.

Чтобы определить TBM вслед за объявлением объекта, поместите директиву TBLINST. Эта директива создает TBM для самого последнего объявленного объекта. Вы можете также открыть сегмент для хранения TBM. Например, после предыдущего объявления объекта вы можете поместить такие инструкции:

```
, шаг #3
DATASEG
TBLINST
```

Вслед за этими инструкциями можно также поместить директиву CODESEG, чтобы воплотить объектные методы (описать процедуры объектных методов), но это не обязательно. Вы можете, например, объявить несколько объектов, определить их TBM, а затем воплотить их методы в другом модуле. Возможно множество комбинаций файлов, объявлений и модулей, хотя я предпочитаю объявления каждого объекта и его методы помещать в одном файле, который затем необходимо вставить в конечную программу. Такой подход упрощает использование объектов в различных программах и при небольшом дополнительном вмешательстве (как я поясню в следующем разделе) дает возможность создавать объекты, которые работают и с малой, и с большой, и с огромной моделями памяти.

И наконец, напишем статический метод, который инициализирует объектный указатель TBM (объявленный директивой TBLPTR). Я называю этот метод *конструктором,* хотя, строго говоря, объекты Turbo Assembler не обладают эквивалентом конструкторов C++ или Pascal. Используем в конструкторе директиву TBLINIT, чтобы инициализировать указатель TBM объектной инстанции. При предположении, что ds:si адресует объектную инстанцию, предлагаю один из способов написания конструктора объекта TBase:

```
PROC TBase_construct PASCAL
 TBLINIT TBase PTR si                       ; Инициализация указателя TBM
 ret
ENDP TBase_construct
```

Объектный конструктор необходимо вызывать для *каждой* инстанции объекта TBase. Каждая такая инстанция обладает своим собственным указателем TBM, который должен быть инициализирован в индивидуальном порядке. Обращение к конструктору осуществляется так же, как и к любому другому статическому методу. Например, в сегменте данных следует сначала определить инстанцию TBase:

```
b1 TBase <>
```

Затем, в кодовом сегменте, необходимо адресовать b1 посредством ds:si и вызвать метод конструктора объекта TBase:

```
mov  si, offset b1
CALL si METHOD TBase·construct
```

Замечание

Конструктор не может быть виртуальным, поскольку для того, чтобы обратиться к виртуальным методам, указателю TBM объектной инстанции нужно присвоить адрес TBM объекта. Целью конструктора как раз и является выполнение этой задачи, хотя он может выполнять и другие операции инициализации.

Для вызова виртуального метода используется та же директива CALL...METHOD, которая вызывает статические методы, но с одним отличием, требуемым в режиме Ideal. В дополнение к специфицированию si в качестве указателя инстанции, необходимо сообщить ассемблеру, на какой базовый объект указывает si. Например, следующие команды адресуют инстанцию b1 посредством ds:si и вызывают объектный виртуальный метод action:

```
mov  si, offset b1
CALL TBase PTR si METHOD TBase:action
```

Вторая строка генерирует инструкции, отыскивающие расположение подпрограммы action согласно TBM, адресуемой указателем TBM инстанции TBase. Влияние этой инструкции по сути состоит в том, что si может адресовать *полученную* объектную инстанцию, и в этом случае будет вызываться подпрограмма action полученного объекта. Предположим, например, что вы объявляете объект TDerived, полученный из TBase. Пусть вы также объявляете в новом объекте замещающий метод action. Затем вы определяете объект, адресуя его посредством ds:si, и вызываете подпрограмму action:

```
DATASEG
d1 TDerived <>                          ; Определение инстанции полученного объекта
CODESEG
. .
mov si, offset d1                       ; Адресация инстанции посредством ds:si
CALL TBase PTR si METHOD Tbase:action   ; Вызов TDerived:action!
```

Даже несмотря на то что инструкция CALL...METHOD специфицирует TBase, в действительности инструкция вызывает виртуальный метод action объекта TDerived. Теперь сравните последнюю строчку в этом программном фрагменте с последней строкой предыдущего примера. Инструкции идентичны — единственное, что изменилось, это объект, адресуемый посредством ds:si. Можете назвать это *проявлением полиморфизма* — объект сам определяет, какой вызвать виртуальный метод.

Следующие несколько распечаток программ демонстрируют эти понятия. Вначале рассмотрим программу 14.6, OOMACROS.INC, представляющую несколько макросов, упрощающих работу с виртуальными методами. Макрос, который я модифицировал и преобразовал в режим Ideal, используя аналогичный макрос режима MASM, поставляемый на дисках Turbo Assembler 4.0, дает возможность писать программы ООП для малой, большой и огромной моделей памяти. Пока не ассемблируйте ее текст — вам нужно будет включить ее в другую программу, как я позже поясню. (Поскольку инструкции Turbo Assembler используют буквы только латинского алфавита, в тексте программы и далее, помимо TBM (таблица виртуальных методов), будет использоваться VMT (virtual method table) — примечание редактора).

Программа 14.6. OOMACROS.INC

```
1.   %TITLE "TASM OOP VMT macros -- by Tom Swan"
2:
3:   ;---- Макросы для малой (small) модели памяти и макроопределения
4:
5:   IF (@CodeSize EQ 0)
6:
7:        MACRO VMTSeg                    ;; Сохранение TBM (VMT) в кодовом сегменте
8:        CODESEG
9:        ENDM VMTSeg
10:
11:       @VMTSeg = @code                 ;; Приравнивание VMTSeg с именем кодового сегмента
12:
13:       MACRO LoadVMTSeg reg            ;; Подготовка регистра, адресующего сегмент VMT (TBM)
14:       push   cs                       ;; Выталкивание адреса кодового сегмента в стек
15:       pop    reg                      ;; Установка cs на требуемый сегментный регистр
16:       ENDM
```

```
17:
18:       MPtr EQU <WORD>                    ;; Виртуальные функции имеют 16-битовую адресацию
19:
20:  ELSE
21:
22:  ;----- Макросы для большой (large) и огромной (huge) моделей памяти и макроопределения
23:
24:       SEGMENT VMT_Seg PUBLIC             ;; Сохранение VMT в отдельном сегменте данных
25:       ENDS VMT_Seg
26:
27:       MACRO VMTSeg
28:       SEGMENT VMT_Seg                    ;; Использование макроса VMTSeg для создания
29:       ENDM VMTSeg                        ;; сегмента VMT
30:
31:       @VMTSeg = VMT_Seg                  ;; Уравнивание VMTSeg сегментом данных
32:
33:       MACRO LoadVMTSeg reg               ;; Подготовка регистра, адресующего сегмент VMT
34:       push   bx                          ;; Сохранение регистра, используемого макросом
35:       mov    bx, @VMTSeg                 ;; Помещение адреса сегмента в bx
36:       mov    reg, bx                     ;; Помещение bx в требуемый сегментный регистр
37:       pop    bx                          ;; Восстановление сохраненного bx
38:       ENDM LoadVMTSeg
39:
40:       MPtr EQU <DWORD>                   ;; Виртуальные функции с 32-битовой адресацией
41:
42:  ENDIF
43;
44:  ;----- Определение макроса таблицы виртуальных методов (все модели памяти)
45:
46        MACRO Make_VMT
47:             VMTSeg                       ;; Начало нового сегмента для большой и огромной моделей
48:             TBLINST                      ;; Создание таблицы виртуальных методов
49:             ENDS                         ;; Окончание сегмента, начатого макросом VMTSeg
50:             CODESEG                      ;; Результирующий кодовый сегмент
51:       ENDM Make_VMT
```

Модуль OOMACROS.INC использует обычные директивы для предотвращения своей программной зависимости от действующей модели памяти. Строка 5 проверяет символ CodeSize. Если он равен нулю, значит, используется малая модель памяти, в противном случае — большая либо огромная модели памяти.

Замечание

Я проверял этот макрос только при использовании с малой, большой и огромной моделями памяти. Если вы используете другие модели, тщательно перепроверьте работу макроса с ними.

Макрос VMTSeg является символом, который приравнен к CODESEG для программ, использующих малую модель памяти (строки 7–9), либо к отдельному сегменту данных VMT_Seg для большой и огромной моделей (строки 24, 25 и 27–29). Вам не обязательно использовать макрос VMTSeg в программе.

Для инициализации сегментного регистра, адресующего сегмент, хранящий TBM, воспользуйтесь макросом LoadVMTSeg. Макрос дает возможность писать программу, не зависящую от модели памяти. Например, перед вызовом одного или более виртуальных методов, при использовании любой модели памяти, вставьте следующую инструкцию, чтобы инициализировать es для адресации сегмента TBM:

```
LoadVMTSeg es
```

При малой модели памяти это использование макроса LoadVMTSeg (строки 13–16) приводит к исполнению следующих инструкций:

```
push cs
pop  es
```

Эти две инструкции приравнивают es и cs. По соглашению, в программах, использующих малую модель памяти, TBM хранятся в кодовом сегменте. (Это не обязательно, но является результатом использования макроса в OOMACROS.INC.)

При использовании большой и огромной моделей макрос LoadVMTSeg генерирует следующие инструкции:

```
push bx
mov  bx, @VMTSeg
mov  es, bx
pop  bx
```

Таким образом, es устанавливается для адресации отдельного сегмента TBM (VMT), названного @VMTSeg (строки 31 и 33–38).

Кроме того, OOMACROS.INC определяет символ MPtr, приравненный к WORD для программ с малой моделью памяти (строка 18) или к DWORD для большой и огромной моделей (строка 40). Вы можете использовать этот макрос в объявлениях объекта для создания не зависящих от моделей памяти объектов. При использовании малой модели памяти адреса методов имеют 16-битовое смещение; при большой и огромной моделях сегменты и смещения адресов — 32-битовые. Для автоматизации выбора необходимого размера указателя замените в своем объявлении объекта dword на MPtr, как показано в следующем фрагменте:

```
STRUC TBase METHOD {
 construct:MPtr = TBase_construct
 VIRTUAL action:MPtr = TBase_action
}
 TBLPTR
...
ENDS TBase
```

Поместите также файл OOMACROS.INC перед объявлением объекта. Теперь можно ассемблировать объект для малой, большой и огромной моделей памяти.

И наконец, в OOMACROS.INC имеется макрос Make_VMT, который нужно применять для определения TBM. Поместите макрос там, где обычно использовали бы директиву TBLINST, после каждого объявления объекта:

```
STRUC TBase METHOD      ; Объявление объекта
...                     ; Объявления объектных метода и данных
ENDS TBase              ; Конец объявления объекта
Make_VMT                ; Применяйте этот макрос вместо TBLINST
```

При использовании малой модели памяти Make_VMT осуществляет переключение на кодовый сегмент, вставляет TBM (посредством TBLINST) и затем переустанавливает кодовый сегмент с помощью директивы CODESEG (которая не является необходимой в данном случае, но не причиняет вреда).

При использовании большой и огромной моделей Make_VMT осуществляет переключение на отдельный сегмент, вставляет TBM (VMT) (опять-таки посредством TBLINST) и затем продолжает кодовый сегмент директивой CODESEG. Результатом является объект, который можно применять в программах с малой, большой и огромной моделями памяти.

Следующие три программы приводят в действие предыдущие установки и макросы. Программа является модифицированной версией примера наследования, приведенного в этой главе. Эта версия, однако, дополняет объекты TBase и TDerived виртуальными методами. Демонстрационная программа определяет инстанции этих объектов и вызывает их виртуальные методы. (Эти файлы также могут стать полезными шаблонами для разработки новых объектно-ориентированных программ — достаточно скопировать их в другую директорию и воспользоваться командами глобального поиска и замены вашего редактора для замены имен объектов.)

Программа 14.7, TBASE.INC, объявляет и воплощает объект TBase. Модуль исходит из предположения, что файл OOMACROS.INC уже включен в основную программу. Не начинайте ассемблировать программу — я подскажу, когда можно будет это сделать. Построчный разбор следует за текстом программы.

Программа 14.7. TBASE.INC

```
1:  %TITLE "TBase object -- by Tom Swan"
2:
3:  GLOBAL TBase_construct:PROC
4:  GLOBAL TBase_init:PROC
5:  GLOBAL TBase_getData:PROC
6:  GLOBAL TBase_action:PROC
7:
8:  STRUC TBase METHOD {
9:       construct:mptr = Tbase_construct     ; Конструктор инстанции
10:      init:mptr = TBase_init               ; Инициализатор инстанции
11:      getData:mptr = TBase_getData         ; Статический метод
12:      VIRTUAL action:mptr = TBase_action   ; Виртуальный метод
13: }
14: TBLPTR                                    ; Указатель таблицы виртуального метода
15: TBase_data dw ?                           ; Данные объекта TBase
16: ENDS TBase
```

```
17:
18:  Make_VMT                                  ; Определение VMT (TBM) TBase
19:
20:  CODESEG
21:
22:  ;-------------------------------------------------------
23:  ; TBase_construct Конструктор TBase (инициализирует указатель VMT)
24:  ;-------------------------------------------------------
25:  ; Вход:
26:  ; ds:si = адресу инстанции
27:  ; Выход:
28:  ; VMT ptr инициализирован
29:  ; Регистры:
30:  ; нет
31:  ;-------------------------------------------------------
32   PROC TBase_construct PASCAL
33:       TBLINIT TBase PTR si                 ; Инициализация указателя VMT
34:       ret
35:  ENDP TBase_construct
36:
37:  ;-------------------------------------------------------
38:  ; TBase_init Метод init TBase (инициализирует данные инстанции)
39:  ;-------------------------------------------------------
40:  ; Вход:
41:  ; ds:si = адресу инстанции
42:  ; arg1 = слову, подлежащему хранению в инстанции
43:  ; Выход:
44:  ; arg1 -> в инстанцию.TBase_data
45:  ; Регистры:
46:  ; нет
47:  ;-------------------------------------------------------
48.  PROC TBase_init PASCAL
49:       ARG @@data:word                      ; Создание смещения аргумента в стеке
50:       USES ax                              ; Резервирование регистра ax (не обязательно)
51:       mov ax, [@@data]                     ; Помещение аргумента в ax
52:       mov [(TBase PTR si).TBase_data], ax  ; Сохранение ax в инстанции
53:       ret
54:  ENDP TBase_init
55:
56:  ;-------------------------------------------------------
57.  ; TBase_getData Метод getData объекта TBase
58:  ;-------------------------------------------------------
59:  ; Вход.
60:  ; ds:si = адресу инстанции
61:  ; Выход:
62:  ; ax = инстанции.TBase_data
63:  ; Регистры:
64:  ; ax
65:  ;-------------------------------------------------------
66:  PROC TBase_getData PASCAL
67:       mov ax, [(TBase PTR si).TBase_data]  ; ax <- базовые данные
68:       ret
69:  ENDP TBase_getData
70.
71:  ;-------------------------------------------------------
72:  ; TBase_action Виртуальный метод action объекта TBase
73:  ;-------------------------------------------------------
74:  ; Вход:
75:  ; ds:si = адресу инстанции
76:  ; Выход:
77:  ; ax = 0000 (произвольная операция для демонстрации)
78:  ; Регистры:
79:  ; ax
80:  ;-------------------------------------------------------
81:  PROC TBase_action PASCAL
82:       xor ax, ax                           ; ax <- 0000h
83:       ret
84.  ENDP TBase_action
```

Я поясню только то, что является новым для TBase. Я добавил метод конструктора (строка 9), названный construct. Как я уже упоминал, этот метод должен быть статическим, поскольку его задачей является инициализация указателя TBM объектной инстанции. До тех пор пока это не произойдет, программа не должна обращаться ни к

одному виртуальному методу этой инстанции. Я также добавил виртуальный метод action (строка 12). Объект TBase объявляет указатель TBM с помощью директивы TBLPTR (строка 14).

Все статические и виртуальные методы должны содержать соответствующие директивы GLOBAL, как показано в строках 3–6. В директивах GLOBAL отсутствуют какие-либо синтаксические отличия между статическими и виртуальными методами.

Вслед за объявлением объекта макрос Make_VMT определяет TBM (VMT) для TBase. Всегда используйте этот макрос (либо директиву TBLINST, если вы не применяете макросы в OOMACROS.INC) *сразу* после каждого объявления объекта.

Строка 20 начинает или продолжает кодовый сегмент программы. Поскольку макрос Make_VMT уже выполнил этот шаг, строка 20 не нужна, но я все равно включил ее для единообразия со всеми программами этой главы. Нет ничего страшного в повторном выполнении директив CODESEG.

Строки 22–35 воплощают конструктор TBase. Как я пояснял, этот метод имеет одно назначение — инициализацию указателя таблицы виртуальных методов (VMT) для каждой инстанции объекта. Строка 33 выполняет этот шаг посредством директивы TBLINIT. Метод исходит из предположения, что ds:si адресует объектную инстанцию.

Замечание

Следует разделять процессы конструирования и инициализации объектных инстанций. Объектный *конструктор* инициализирует указатель таблицы виртуальных методов объектной инстанции. Объектный *инициализатор* присваивает значения переменным инстанции. *Не пытайтесь* объединять эти шаги в одном методе. Как вам станет понятно из следующей программы, полученный объект должен обладать своим собственным конструктором, но обычно он вызывает инициализатор своего базового объекта для присвоения значений унаследованным переменным. Помните также, что я использую слово *"конструктор"* в смысле, отличном от C++ и Pascal, и что это является моим соглашением, но не Turbo Assembler.

И наконец, в TBASE.INC содержится воплощение виртуального метода объекта TBase, TBase_action (строки 71–84). Реальная подпрограмма ничем не отличается от статического метода либо от любой другой подпрограммы. В этом примере TBase_action не выполняет никаких полезных действий, но просто ради того, чтобы метод что-нибудь выполнял, строка 82 устанавливает регистр ax в нуль посредством выполнения инструкции xor. Позже, когда вы будете прогонять этот участок программы через Turbo Debugger, это действие поможет убедиться в том, что была вызвана TBase_action, а не какая-то другая подпрограмма.

Следующая программа 14.8, TDERIVED.INC, объявляет и воплощает полученный из TBase объект TDerived. *Этот модуль является пересмотренной версией аналогичного файла и объекта, приведенных в этой главе при обсуждении наследования.* Как и для предыдущей распечатки программы, я поясню только то, что является новым и усовершенствованным в модифицированном файле.

Программа 14.8. TDERIVED.INC

```
1:  %TITLE "TDerived object -- by Tom Swan"
2:
3:  GLOBAL TDerived_construct:PROC
4:  GLOBAL TDerived_init:PROC
5:  GLOBAL TDerived_getData:PROC
6:  GLOBAL TDerived_action:PROC
7:
8:  STRUC TDerived TBase METHOD {
9:       construct:mptr = TDerived_construct   ; Конструктор инстанции
10:      init:mptr = TDerived_init             ; Инициализатор инстанции
11:      getData:mptr = TDerived_getData       ; Замещающий статический метод
12:      VIRTUAL action:mptr = TDerived_action ; Замещающий виртуальный метод
13: }
14: TDerived_data dw ?                         ; Данные объекта TDerived
15: ENDS TDerived
16:
17: Make_VMT                                   ; Определение VMT объекта TDerived
18:
19: CODESEG
20:
21: ;-------------------------------------------------------
22: ; TDerived_construct Конструктор объекта TDerived
23: ;-------------------------------------------------------
24: ; Вход:
25: ; ds:si = адресу инстанции
26: ; Выход:
27: ; VMT ptr инициализирована
28: ; Регистры:
29: ; нет
```

```
30:  ;-----------------------------------------------------
31:  PROC TDerived_construct PASCAL
32:       TBLINIT TBase PTR si                    ; Инициализация указателя VMT инстанции
33:       ret
34:  ENDP TDerived_construct
35:
36:  ;-----------------------------------------------------
37:  ; TDerived_init Метод init объекта TDerived
38:  ;-----------------------------------------------------
39:  ; Вход:
40:  ; ds:si = адресу инстанции
41:  ; arg1 = слову, подлежащему хранению в данных базовой инстанции
42:  ; arg2 = слову, подлежащему хранению в данных полученной инстанции
43:  ; Выход:
44:  ; arg1 -> в инстанцию.TBase_data
45:  ; arg2 -> в инстанцию.TDerived_data
46:  ; Регистры:
47:  ; нет
48:  ;-----------------------------------------------------
49:  PROC TDerived_init PASCAL
50:       ARG @@data1:word, \                     ; Создание смещений стека для аргументов
51:            @@data2:word
52:       USES ax                                 ; Резервирование ax (не обязательно)
53:       mov ax, [@@data1]                       ; Помещение аргумента arg1 в ax
54:       CALL si METHOD TBase:init, ax           ; Вызов базового метода init
55:       mov ax, [@@data2]                       ; Помещение arg2 в ax
56:       mov [(TDerived PTR si).TDerived_data], ax ; Сохранение в инстанции
57:       ret
58:  ENDP TDerived_init
59:
60:  ; ----- Предыдущие операторы mov и call могут быть описаны и так:
61:  ; CALL si METHOD TBase:init, [@@data1]
62:
63:  ;-----------------------------------------------------
64:  ; TDerived_getData Метод getData объекта TDerived
65:  ;-----------------------------------------------------
66:  ; Вход:
67:  ; ds:si = адресу инстанции
68:  ; Выход:
69:  ; ax = инстанции.TBase_data
70:  ; dx = инстанции.TDerived_data
71:  ; Регистры:
72:  ; ax, dx
73:  ;-----------------------------------------------------
74:  PROC TDerived_getData PASCAL
75:       CALL si METHOD TBase:getData            ; ax <- базовые данные
76:       mov dx, [(TDerived PTR si).TDerived_data]      ; dx <- полученные данные
77:       ret
78:  ENDP TDerived_getData
79:
80:  ;-----------------------------------------------------
81:  ; TDerived_action Виртуальный метод action объекта TDerived
82:  ;-----------------------------------------------------
83:  ; Вход:
84:  ; ds:si = адресу инстанции
85:  ; Выход:
86:  ; ax = 0ffffh (произвольная операция для демонстрации)
87:  ; Регистры:
88:  ; ax
89:  ;-----------------------------------------------------
90:  PROC TDerived_action PASCAL
91:       mov ax, 0ffffh                          ; ax <- 0ffffh
92:       ret
93:  ENDP TDerived_action
94:
95:  ;----- Использует это для вызова родительской функции в TDerived_action:
96:  ; call TBase_action
```

Объект TDerived наследует методы и переменные от TBase (строки 8–15). Новый объект объявляет замещающий конструктор (строка 9) — важнейший шаг, о котором вы не должны забывать во всех полученных объектах, использующих виртуальные методы. Конструктор инициализирует указатель на таблицу виртуальных методов полученного объекта, который наследуется от TBase. Обратите внимание на то, что TDerived *не* объявляет этот указа-

тель посредством директивы TBLPTR — только базовый объект в иерархии родственных объектов может использовать эту директиву.

TDerived объявляет также методы, замещающие init, getData, статический и виртуальный методы action. Кроме того, в строке 14 объект объявляет переменную.

На данном этапе вам должно быть понятно, что TDerived содержит три члена данных — указатель TBM, слово, унаследованное от TBase, и новую переменную, объявленную в строке 14.

Указатели TBM (VMT) являются унаследованными; самих же TBM — нет, и, как показывает строка 17, вам необходимо использовать макрос Make_VMT (или директиву TBLINST, если вы не используете макросы в OOMACROS.INC), чтобы создать TBM для полученного объекта.

После объявления объекта и определения TBM в модуле реализуются методы TDerived. Строки 21–34 реализуют объектный конструктор, который, как и в TBase, использует директиву TBLINIT (строка 32) для инициализации указателя TBM для объектных инстанций TDerived.

Замечание

Не вызывайте конструктор базового объекта из полученного конструктора. Инстанции полученного объекта должны адресовать свои собственные TBM, а не TBM какого-либо базового объекта.

Статические методы TDerived, TDerived_init и TDerived_getData — неизмененные. Описание этих подпрограмм вы можете найти в разделе данной главы, посвященном наследованию. (Замечание: строки 60, 61 демонстрируют альтернативную технику вызова базового объектного метода и передачи аргумента, переданного полученному объектному методу. Вместо того чтобы загружать аргумент в регистр и передавать его базовому методу, как показано в строках 53, 54, можно передать его непосредственно в инструкции CALL...METHOD, как показано в строке 61.)

Как и в TBase, виртуальный метод action полученного объекта (строки 80–93) не выполняет никаких полезных действий. Просто ради того, чтобы подпрограмма что-то делала, инструкция mov в строке 91 устанавливает значение ax в 0ffffh. Когда вы запустите тестовую программу в Turbo Debugger, это значение поможет вам различить обращения к базовому и к полученному методам action.

Обратите внимание, что, как и в TBase, TDerived_action является простой обычной подпрограммой, сходной с любой другой. Только обращение к виртуальным методам выделяет их, а не объявление этих методов.

Программа 14.9, VIRTUAL.ASM, приводит предыдущие три программы в действие. Теперь введите эти программы, после чего можно ассемблировать и компоновать демонстрационную программу. Для этого введите следующие команды:

```
tasm /zi virtual
tlink /v virtual
```

Замечание

Программа VIRTUAL.ASM не создает никакой выходной информации. Запустите программу в Turbo Debugger, чтобы проверить, как она работает.

Программа 14.9. VIRTUAL.ASM

```
1:  %TITLE "Virtual function demonstration -- by Tom Swan"
2:
3:          IDEAL
4:
5:          JUMPS
6:
7:          LOCALS @@
8:
9:          MODEL large, PASCAL
10:
11:         STACK 1000H
12:
13:         INCLUDE "..\oomacros.inc"
14:
15:         INCLUDE  tbase.inc
16:
17:         INCLUDE "tderived.inc"
18:
19:         DATASEG
20:
21: exCode db 0 ; Код выхода программы
22:
23: ;----- Определение объектов без первоначальных значений
```

```
24:
25:  b1 TBase <>
26:  d1 TDerived <>
27:
28:  ;----- Определение объектов с определенными первоначальными значениями
29:  ; и помещение рамок (0) для указателей VMT
30:
31:  b2 TBase <0, 987>
32:  d2 TDerived <<0, 654>, 321>
33:
34:  ;----- Определение объектов с определенными первоначальными значениями
35:  ; и определенными указателями VMT
36:
37:  b3 TBase <@TableAddr_TBase, 987>
38:  d3 TDerived <<@TableAddr_TDerived, 654>, 321>
39:
40:
41:       CODESEG
42:
43:  Start:
44:       mov ax, @data                        ; Инициализация DS по адресу
45:       mov ds, ax                           ; сегмента данных
46:
47:       mov si, offset b1                    ; Адресация инстанции b1
48:       LoadVMTSeg es                        ; Инициализация es (сегмент VMT)
49:       CALL si METHOD TBase.construct       ; Подготовка VMT ptr b1
50:       CALL si METHOD TBase:init, 01h       ; Инициализация данных инстанции
51:       CALL TBase PTR si METHOD TBase:action ; Вызов виртуальной функции
52:
53:       mov si, offset d1                    ; Адресация инстанции d1
54:       LoadVMTSeg es                        ; Инициализация es
55:       CALL si METHOD TDerived.construct    ; Вызов статической функции
56:       CALL si METHOD TDerived.init, 02h, 03h ; Вызов статической функции
57:       CALL TBase PTR si METHOD TBase.action ; Вызов виртуальной функции
58:
59:       mov si, offset d1
60:       LoadVMTSeg es
61:       CALL TBase PTR si METHOD TBase.action , Вызов TDerived:action
62:
63:       mov si, offset b2
64:       LoadVMTSeg es
65:       CALL si METHOD TBase:construct
66:       CALL TBase PTR si METHOD TBase:action ; Вызов TBase:action
67:
68:       mov si, offset d2
69:       LoadVMTSeg es
70:       CALL si METHOD TDerived:construct
71:       CALL TBase PTR si METHOD TBase:action ; Вызов TDerived:action
72:
73:       mov si, offset b3
74:       LoadVMTSeg es
75:       CALL TBase PTR si METHOD TBase:action ; Вызов TBase:action
76:
77:       mov si, offset d3
78:       LoadVMTSeg es
79:       CALL TBase PTR si METHOD TBase:action ; Вызов TDerived:action
80:
81:  Exit:
82:       mov ah, 04Ch                         ; Функция ДОС: Выход из программы
83:       mov al, [exCode]                     ; Возврат значения кода выхода
84:       int 21h                              ; Вызов ДОС. Прерывание программы
85:
86:       END Start                            ; Конец программы / точка входа
```

Приведенная программа демонстрирует несколько ключевых методик виртуальных методов. Она показывает:

- как писать программы, не зависящие от модели памяти,
- три способа определения инстанций объектов, использующих виртуальные методы,
- правильный способ инициализации инстанций объектов, использующих виртуальные методы,
- как вызывать виртуальные методы для объектных инстанций, адресуемых указателями (полиморфизм).

Вы можете заменить модель памяти в строке 9 на small, large (ее настоящее значение) или huge. Другие модели памяти также могут работать, но я проверил только эти три. Настоятельно советую вам проверить по крайней мере малую и большую модели и просмотреть объектные инстанции в Turbo Debugger. Возможно, вы захотите воспользоваться командой Turbo Debugger View:Data, чтобы определить расположение таблицы виртуальных методов, которая конфигурируется по-разному и хранится в различных сегментах, в зависимости от модели памяти.

Строки 13–17 включают файлы OOMACROS.INC, TBASE.INC и TDERIVED.INC. При использовании макросов TBM не забудьте вставить OOMACROS.INC перед объявлением любых объектов.

Строки 25, 26 показывают стандартный способ определения объектных инстанций. Эти определения аналогичны тем, с которыми вы уже познакомились — тот факт, что объекты содержат виртуальные методы, не оказывает никакого влияния на то, как вы определяете *неинициализированные* инстанции этих объектов.

Однако, если вы желаете переопределить первоначальные значения переменных в определениях объекта, вы должны также пересчитать указатель TBM в каждой из этих инстанций. Строки 31, 32 демонстрируют один способ выполнения этого правила. Первое определение определяет инстанцию b2 объекта TBase. Переменной этой инстанции TBase_data присвоено значение 987. Указателю TBM инстанции присвоено значение 0 — шаблон, который будет изменен, когда программа вызовет объектный конструктор для b2.

Строка 32 показывает, как, используя угловые скобки, инициализировать полученный объект, его унаследованные переменные и указатель TBM. Внутреннее выражение, <0, 654>, присваивает нуль унаследованному указателю TBM и 654 — унаследованной переменной TBase_data. Внешнее выражение <..., 321> присваивает 321 переменной TDerived_data. Как и в случае с b2, указатель TBM инстанции унаследованного объекта будет инициализирован, когда программа вызовет объектный конструктор для d2.

Строки 37, 38 демонстрируют альтернативную методику инициализации объектных инстанций. Когда вы объявляете объект и связанную с ним TBM, Turbo Assembler создает символ, представляющий адрес TBM. Символу присваивается имя:

```
@TableAddr_<имя объекта>
```

где <имя объекта> является именем объявленного объекта. Например, @TableAddr_TBase представляет адрес TBM объекта TBase, @TableAddr_TDerived — адрес TBM объекта TDerived.

Можете использовать эти символы для полной инициализации объектных инстанций, как показано в строках 37, 38. Строка 37 инициализирует базовый объект, строка 38 — полученный объект, используя угловые скобки аналогично строке 32.

При определении объекта этим способом нет необходимости вызывать объектные конструкторы для инициализации указателей TBM. Они *уже* инициализированы. Но по-прежнему нужно писать конструктор для объектов, применяющих виртуальные методы, поскольку эту альтернативную методику можно использовать только применительно к статическим объектам, определенным в тексте программы. Нельзя, например, использовать этот метод при конструировании объекта, для которого вы выделяете некоторую область памяти, может быть, обращаясь к функции ДОС. Поэтому я не рекомендую использование методики, показанной в строках 37, 38. Вместо этого для определения объектных инстанций лучше использовать методику, приведенную в строках 25, 26 или 31, 32, и всегда вызывать объектный конструктор для *каждой* из этих инстанций.

Строки 47–51 демонстрируют, как осуществить это для первой инстанции, объекта Tbase — b1. Строка 47 адресует инстанцию посредством ds:si. Строка 48 новая — она использует макрос LoadVMTSeg для инициализации регистра es сегментом, в котором хранится TBM. (Этот макрос обязателен только в программах, использующих малую модель памяти. Однако его применение гарантирует результат, независимый от модели памяти.)

Строки 49, 50 конструируют и инициализируют объектную инстанцию b1. Вы *должны* вызвать конструктор, как демонстрируется в строке 49, прежде чем вызывать любой из виртуальных методов. Если этого не сделать, то программа, вероятно, разрушится и может привести к зависанию операционной системы вашего компьютера.

Сравните инструкции CALL...METHOD в строках 50, 51. Первая из этих двух строк вызывает для объекта TBase статический метод init, вторая строка — виртуальный метод action. Turbo Assembler использует специфицированный регистр (в данном случае si) для размещения TBM объекта (посредством указателя TBM, хранящегося в инстанции) и для вызова подпрограммы, адресуемой посредством TBM. Между прочим, можно использовать тот же синтаксис для вызова статических методов. Другими словами, можно переписать строку 50 следующим образом:

```
CALL TBase PTR si METHOD TBase:init, 01h
```

Вы можете вызвать *все* методы таким образом (предваряя si именем своего объекта и PTR). Затем можно изменить методы со статических на виртуальные простым пересмотром объявлений их объектов. Однако, если методы всегда будут статическими, префикс PTR не является необходимым. Он нужен только для того, чтобы позволить Turbo Assembler сгенерировать инструкции, обеспечивающие доступ ко входам TBM.

Очень полезно просмотреть код, который Turbo Assembler генерирует для обращения к подпрограммам виртуальных методов, подобных строке 51. Чтобы выполнить это посредством Turbo Debugger, ассемблируйте и скомпонуйте программу, затем наберите td virtual для загрузки ее в отладчик. Нажимайте <F8> до тех пор, пока курсор не

достигнет строки 51. Затем воспользуйтесь командой View:CPU, чтобы просмотреть сгенерированные инструкции. (Нажмите <F5>, чтобы распахнуть окно на весь экран.)

При использовании большой модели памяти вы обнаружите следующие инструкции:

```
les bx,[si]
call es:far [bx]
```

Инструкция les загружает 32-битовый адрес, хранящийся в ds:si, в регистры es:bx. Другими словами, поскольку указатель TBM является первым членом данных в инстанции, инструкция копирует значение указателя TBM в es:bx. Вот почему указатель TBM должен быть первым членом в объектной инстанции.

Вторая инструкция выполняет косвенное обращение к другому адресу. Этим адресом, в данном случае, является первый вход в объектной TBM. Если бы объект содержал другие виртуальные методы, сгенерированная инструкция call, возможно, выглядела бы так:

```
call es:far [bx+04]
```

Адреса виртуальных методов хранятся в TBM, и косвенное обращение использует es:bx, для загрузки адреса подпрограммы из TBM, по которому и осуществляется действительное обращение. Таким образом, обращения к виртуальным методам во время выполнения программы направляются к соответствующим областям памяти.

При использовании малой модели памяти Turbo Assembler генерирует другую последовательность для инструкции CALL...METHOD в строке 51:

```
mov bx,[si]
call es.[bx]
```

Первая инструкция загружает смещение адреса объектной TBM в bx, но вторая использует *неинициализированный* сегментный регистр es для размещения адреса виртуального метода. Это может оказаться неприятным "жучком" в Turbo Assembler, ассемблер может просто ожидать, что es должен быть инициализирован сегментом, хранящим TBM. Учитывая это, макрос LoadVMTSeg в OOMACROS.INC инициализует es надлежащим образом независимо от модели памяти. Если вы не желаете хранить свою TBM для малой модели в кодовом сегменте, то прежде чем вызывать виртуальные методы, вы должны убедиться в том, что es инициализирован по надлежащему адресу сегмента. Но, вероятно, проще использовать большую модель — в этом случае нет необходимости применять LoadVMTSeg.

Оставшаяся часть программы, начиная со строки 53 (снова обратитесь к тексту программы 14.9, VIRTUAL.ASM) демонстрирует, как вызвать конструкторы, статические и виртуальные методы для других объектных инстанций, определенных в сегменте данных программы. Запустите программу в Turbo Debugger и загрузите программные инстанции b1, d1, b2, d2, b3 и d3 в окно Watches (перемещайте курсор к каждой из инстанций и нажимайте <Ctrl>+<W>).

Затем нажимайте <F7>, чтобы проследить каждое обращение к подпрограмме. Повторяйте это в окне Module до тех пор, пока не узнаете код, а затем просмотрите инструкции в окне CPU, чтобы проследить действительные инструкции. Можете распахнуть с помощью команды View окно Registers, чтобы просмотреть значения регистров.

Уделите особое внимание инструкциям CALL...METHOD в строках 57, 61, 66, 71, 75 и 79. Хотя все эти инструкции идентичны, программа вызывает TBase_action или TDerived_action, в зависимости от типа объектной инстанции, адресуемой посредством ds:si. Это полиморфизм в действии. Объектные инстанции сами, посредством своих TBM, во время выполнения программы определяют, какие виртуальные методы вызывать.

Весьма полезно повторить эти эксперименты в Turbo Debugger для различных моделей памяти. Замените в строке 9 large на small, перекомпилируйте и проследите результаты в Turbo Debugger. Обратите внимание на различные размеры указателей TBM и просмотрите также код, сгенерированный для инструкций CALL...METHOD.

Полиморфизм

Программы этого раздела применяют полиморфизм на практике. Программы реализуют объект — TList, который может хранить перечни инстанций другого объекта — TItem. После описания этих двух объектов я объясню, как получить из TItem новые объекты и как вставить их в скомпонованный перечень.

Для работы с программами этого раздела создайте новый подкаталог, например LIST, где можно располагать файлы программ данного раздела.

Замечание

Техника программирования подключаемых списков на языке ассемблера является одним из предметов, вызвавших наибольшее число вопросов от читателей первого издания этой книги. Можно, конечно, программировать подключаемые списки с помощью обычного кода, но использование объектов делает эту задачу значительно проще, как показывают приведенные ниже модули и пример программы.

Создание списочного объекта

Первой задачей при создании списочного объекта является определение общей темы той информации, которая должна храниться в списке. Объект TItem в программе-модуле 14.10, TITITEM.INC, называется *абстрактным* объектом, поскольку он никогда не используется для определения объектных инстанций. Для хранения объекта в списке из TItem получают новый объект. Таким образом, можно получать столько различных видов объектов, сколько необходимо, и хранить их в одном и том же списке. Список может оперировать *любым* видом информации — все что нужно, это получить свои объекты из TItem.

Программа 14.10. TITEM.INC

```
%TITLE "TItem object -- by Tom Swan"

GLOBAL TItem_construct:PROC
GLOBAL TItem_init:PROC
GLOBAL TItem_print:PROC

STRUC TItem METHOD {
     construct:mptr = TItem_construct  ; Конструктор TItem
     init:mptr = TItem_init            ; Инициализатор TItem
     VIRTUAL print:mptr = TItem_print  ; Печать, отображение темы
}
TBLPTR                                 ; Указатель таблицы виртуального метода
next dw ?                              ; Указатель на следующую тему
ENDS TItem

Make_VMT                               ; Определение VMT (TBM) TItem

CODESEG

;-------------------------------------------------------
; TItem_construct Конструктор TItem
;-------------------------------------------------------
; Вход:
; ds:si = адресу инстанции TItem
; Выход:
; VMT ptr инициализировано
; Регистры:
; нет
;-------------------------------------------------------
PROC TItem_construct PASCAL
     TBLINIT TItem PTR si              ; Инициализация указателя TBM
     ret
ENDP TItem_construct

;-------------------------------------------------------
; TItem_init Инициализация указателя темы "next"
;-------------------------------------------------------
; Вход:
; ds:si = адресу инстанции TItem
; Выход:
; поле next<- nil (0000)
; Регистры:
; нет
;-------------------------------------------------------
PROC TItem_init PASCAL
     mov [(TItem PTR si).next], 0      ; Установка поля next в нуль
     ret
ENDP TItem_init

;-------------------------------------------------------
; TItem_print Печать темы VIRTUAL
;-------------------------------------------------------
; Вход:
; ds:si = адресу инстанции TItem
; Выход:
; нет
; Регистры:
; нет
;-------------------------------------------------------
PROC TItem_print PASCAL
     ret                               ; Инструкции, обеспечиваемые реальными темами
ENDP TItem_print
```

Строки 7–14 в TITEM.INC объявляют объект TItem, который обладает конструктором (construct), инициализатором (init) и виртуальным методом print. Объект определяет указатель TBM (строка 12), а также переменную next (строка 13). Переменная next представляет 32-битовый адрес следующей темы в списке. Если эта переменная равна нулю, значит, тема является последним (или единственным) перечисленным значением.

Строка 16 создает TBM для TItem. Помните: объект, полученный из TItem, требует своей собственной TBM, но он наследует указатель TBM от TItem. Не следует применять в объектах, полученных из TItem, директиву TBLPTR. Для определения TBM полученных объектов используйте макрос Make_VMT.

Замечание

Для объектов этого раздела требуется файл OOMACROS.INC, приведенный в данной главе ранее.

Метод конструктора (строки 30–33) инициализирует указатель TBM объектной инстанции TItem. Этот код аналогичен кодам других конструкторов, с которыми вы встречались в данной главе.

Инициализатор TItem (строки 45–48) устанавливает переменную next объектной инстанции в нуль — значение, указывающее на то, что в списке отсутствует следующий объект. В полученных объектах не забудьте вызвать TItem:init для инициализации унаследованной переменной next.

Виртуальный метод print TItem (строки 60–62) не выполняет никаких действий. Предполагается, что в полученных объектах этот метод будет замещен кодом, соответствующим типу хранимой информации. TItem_print называется *абстрактным методом*. Он служит просто в качестве шаблона для действий, которые должны быть определены в полученных объектах.

Программа 14.11, TLIST.INC, объявляет и реализует объект TList, который организует подключаемый список объектных инстанций TItem (либо любых инстанций объектов, полученных из TItem). Методы объекта TList несколько сложнее остальных, приведенных в этой главе. Если у вас возникнут проблемы при построчном разборе, следующем за распечаткой, загрузите пример основной программы (последняя программная распечатка этой главы) в Turbo Debugger и проследите методы в TList.

Программа 14.11. TLIST.INC

```
1:  %TITLE "TList object -- by Tom Swan"
2:
3:  GLOBAL TList_construct:PROC
4:  GLOBAL TList_init:PROC
5:  GLOBAL TList_getCount:PROC
6:  GLOBAL TList_insertItem:PROC
7:  GLOBAL TList_printAll:PROC
8:
9:  STRUC TList METHOD {
10:     construct:mptr = TList_construct  ; Конструктор TList
11:     init:mptr = TList_init            ; Инициализатор TList
12:     getCount:mptr = TList_getCount    ; Возвращение числа тем
13:     VIRTUAL insertItem:mptr = TList_insertItem    ; Вставка TItem в список
14:     VIRTUAL printAll:mptr = TList_printAll        ; Печать, отображение всех тем
15: }
16:     TBLPTR                            ; Указатель TBM
17:     root dw ?                         ; Ptr для первой темы в списке
18:     num dw ?                          ; Число тем, помещенных в список
19: ENDS TList
20:
21: Make_VMT                              ; Определение TBM объекта TList
22:
23: CODESEG
24:
25: ;-----------------------------------------------------
26: ; TList_construct Конструктор TList
27: ;-----------------------------------------------------
28: ; Вход:
29: ; ds:si = адресу инстанции TList
30: ; Выход:
31: ; VMT ptr инициализировано
32: ; Регистры:
33: ; никаких
34: ;-----------------------------------------------------
35: PROC TList_construct PASCAL
36:     TBLINIT TList PTR si              ; Инициализация указателя TBM
37:     ret
38: ENDP TList_construct
39:
```

```
40:  ;-----------------------------------------------------
41:  ; TList_init Инициализация списка пустым состоянием
42:  ;-----------------------------------------------------
43:  ; Вход:
44:  ; ds:si = адресу инстанции TList
45:  ; Выход:
46:  ; root field <- nil (0000) (empty list)
47:  ; Регистры:
48:  ; нет
49:  ;-----------------------------------------------------
50:  PROC TList_init PASCAL
51:       mov [(TList PTR si).root], 0       ; Установка root в нуль (пусто)
52:       mov [(TList PTR si).num], 0        ; Установка num (числа тем) в нуль
53:       ret
54:  ENDP TList_init
55:
56:  ;-----------------------------------------------------
57:  ; TList_getCount Возвращение числа тем, помещенных в список
58:  ;-----------------------------------------------------
59:  ; Вход:
60:  ; ds:si = адресу инстанции TList
61:  ; Выход:
62:  ; ax = числу тем в списке
63:  ; Регистры:
64:  ; ax
65:  ;-----------------------------------------------------
66:  PROC TList_getCount PASCAL
67:       mov ax, [(TList PTR si).num]       ; Получение из списка поля num (числа тем)
68:       ret
69:  ENDP TList_getCount
70:
71:  ;-----------------------------------------------------
72:  ; TList_insertItem Помещение темы в список VIRTUAL
73:  ;-----------------------------------------------------
74:  ; Вход:
75:  ; ds:si = адресу инстанции TList
76:  ; arg = TItem 16-битовому адресу (смещение в сегменте данных)
77:  ; Выход:
78:  ; Инстанция Item, подключенная к списку
79:  ; Регистры:
80:  ; нет
81:  ;-----------------------------------------------------
82:  PROC TList_insertItem PASCAL
83:       ARG @@item:word                    ; Смещение стека для аргумента
84:       USES ax, bx                        ; Резервирование ax и bx
85:       mov ax, [(TList PTR si).root]      ; Установка ax в ptr root списка
86:       mov bx, [@@item]                   ; Установка bx в ptr темы
87:       mov [(TItem PTR bx).next], ax      ; Установка next темы = root
88:       mov [(TList PTR si).root], bx      ; Установка root списка = ptr темы
89:       inc [(TList PTR si).num]           ; Увеличение числа тем
90:       ret
91:  ENDP TList_insertItem
92:
93:  ;-----------------------------------------------------
94:  ; TList_printAll Вызов print для всех тем списка VIRTUAL
95:  ;-----------------------------------------------------
96:  ; Вход:
97:  ; ds:si = адресу инстанции TList
98:  ; Выход:
99:  ; зависит от методов print тем
100: ; Регистры:
101: ; нет
102: ;-----------------------------------------------------
103: PROC TList_printAll PASCAL
104:      USES si, es                        ; Резервирование регистров
105:      LoadVMTSeg es                      ; Инициализация es (не обязательно*)
106:      mov si, [(TList PTR si).root]      ; Установка si в ptr root списка
107: @@10:
108:      or si, si                          ; Проверка si на 0
109:      jz @@99                            ; Переход к выходу, если si = 0
110:      CALL TItem PTR si METHOD TItem:print  ; Вызов метода print темы!
111:      mov si, [(TItem PTR si).next]      ; Установка si в ptr следующей темы
112:      jmp @@10                           ; Цикл до завершения
113: @@99:
```

```
114:      ret
115: ENDP TList_printAll
116:
117: ; * Не обязательно, в зависимости от модели памяти
```

Объект TList (строки 9–19) объявляет три статических и два виртуальных метода. Объект также определяет указатель TBM (строка 16) и две переменные: root, которая адресует инстанцию первой темы в списке (либо равна нулю, если список пуст), и num, которая содержит число объектов в списке. Строка 21 определяет TBM для TList.

Замечание

TList не получен из TItem — эти два объекта являются отдельными и различными.

Как и во всех объектах этой главы, имеющих по крайней мере один виртуальный метод, конструктор TList (строки 35–38) инициализирует указатель TBM инстанции TList посредством использования директивы TBLINIT.

Инициализатор TList (строки 50–54) устанавливает две переменные, root и num, в нуль. Программа должна определить объектную инстанцию TList, адресовать эту инстанцию посредством ds:si и вызвать объектные конструктор и инициализатор, прежде чем помещать в список любые инстанции TItem.

Метод TList_getCount (строки 66–69) возвращает в регистр ax число тем в списке. Это значение используется для написания циклов, выполняющих действия над помещенными в список темами.

Метод TList_insertItem (строки 82–91) помещает инстанцию объекта, полученного из TItem, в список. Смещение адреса полученной из TItem инстанции в качестве аргумента передается методу. Строка 85 устанавливает ax в значение root текущего списка, которое указывает на первую (если имеется хоть одна) тему списка. Строка 86 присваивает регистру bx переданное в качестве аргумента смещение новой инстанции, которая должна быть помещена в список.

Строки 87, 88 помещают инстанцию TItem в список. Это осуществляется путем установки унаследованного темой указателя next на адрес первой текущей темы списка (или на нуль, если список пуст). После этого root устанавливается на значение адреса новой темы, которая становится новой первой темой в списке. И наконец, строка 89 увеличивает num для ведения счета тем в списке.

Виртуальный метод TList_printAll демонстрирует удачное использование полиморфизма. Этот метод применяет ds:si для адресации первой темы списка (строки 105, 106). (Макрос LoadVMTSeg в строке 105 необходим только для программ, использующих малую модель памяти, но он включен в программу, чтобы объекты TList и TItem могли использоваться с любыми моделями памяти.)

Цикл в строках 107–112 адресует каждую тему в списке. Сперва строка 108 проверяет, не равен ли si нулю, показывая, что была обработана последняя тема или что список пуст. Если si равен нулю, строка 109 осуществляет переход на выход из цикла, завершая метод. В противном случае строка 110 вызывает виртуальный метод print темы.

Поскольку этот метод виртуальный, действительный вызываемый метод print *зависит от типа тем в списке.* В объект, полученный из TItem, вы должны поместить свой собственный метод print для выполнения желательных вам действий. Метод printAll объекта TList вызовет метод print вашего объекта из строки 110.

Строка 111 присваивает значение переменной next текущей темы si, после чего в строке 112 осуществляется переход в начало цикла. Таким образом, все темы обрабатываются вслед за своими указателями next.

Использование объекта TList

Для использования объекта TList, прежде всего необходимо, чтобы какие-либо объекты хранились в списке. Все подобные темы должны быть получены из TItem, но, кроме этого, нет никаких других серьезных ограничений. Можно свободно хранить в списке любые типы данных и смешивать различные типы объектных инстанций в *одном и том же* списке — особенность, которую трудно запрограммировать, используя обычные методики ассемблера.

Программа 14.12, TINTOBJ.INC, приводит пример объекта TIntObj, полученного из TItem. Новый объект хранит целочисленное значение. Используйте его для создания списков 16-битовых целых чисел.

Программа 14.12. TINTOBJ.INC

```
1.  %TITLE 'TIntObj object -- by Tom Swan
2:
3:  GLOBAL TIntObj_construct:PROC
4:  GLOBAL TIntObj_init.PROC
5:  GLOBAL TIntObj_print:PROC
6:
7:  STRUC TIntObj TItem METHOD {
8:       construct:mptr = TIntObj_construct    ; Конструктор TIntObj
9:       init:mptr = TIntObj_init              ; Инициализатор TIntObj
10:      VIRTUAL print:mptr = TIntObj_print    ; Печать, отображение темы
11:  }
```

```
12:       data_i dw ?                          ; 16-битовые целочисленные данные
13:  ENDS TIntObj
14:
15:  Make_VMT                                  ; Определение TBM TIntObj
16:
17:  DATASEG
18:
19:  TIntObj_buffer db 20 DUP (0)
20:  TIntObj_msg db 'Integer item = ', 0
21:
22:  CODESEG
23:
24:  ;----- Из BINASC.OBJ, STRIO.OBJ
25:       EXTRN BinToAscHex:Proc, NewLine:Proc, StrWrite:Proc
26:
27:  ;-------------------------------------------------------
28:  ; TIntObj_construct Конструктор TIntObj
29:  ;-------------------------------------------------------
30:  ; Вход:
31:  ; ds:si = адресу инстанции TIntObj
32:  ; Выход:
33:  ; VMT ptr инициализировано
34:  ; Регистры:
35:  ; нет
36:  ;-------------------------------------------------------
37:  PROC TIntObj_construct PASCAL
38:       TBLINIT TIntObj PTR si               ; Инициализация указателя TBM
39:       ret
40:  ENDP TIntObj_construct
41:
42:  ;-------------------------------------------------------
43:  ; TIntObj_init Инициализация указателя "next" темы
44:  ;-------------------------------------------------------
45:  ; Вход:
46:  ; ds:si = адресу инстанции TIntObj
47:  ; arg = 16-битовому целому числу, подлежащему хранению в инстанции
48:  ; Выход:
49:  ; инстанция.data_i <- arg
50:  ; Регистры:
51:  ; нет
52:  ;-------------------------------------------------------
53:  PROC TIntObj_init PASCAL
54:       ARG @@data:word
55:       USES ax
56:       CALL si METHOD TItem:init            ; Вызов унаследованного метода init объекта TItem
57:       mov ax, [@@data]                     ; Получение аргумента из стека
58:       mov [(TIntObj PTR si).data_i], ax    ; Присвоение значения arg инстанции
59:       ret
60:  ENDP TIntObj_init
61:
62:  ;-------------------------------------------------------
63:  ; TIntObj_print Печать темы VIRTUAL
64:  ;-------------------------------------------------------
65:  ; Вход:
66:  ; ds:si = адресу инстанции TIntObj
67:  ; Выход:
68:  ; нет
69:  ; Регистры:
70:  ; нет
71:  ;-------------------------------------------------------
72:  PROC TIntObj_print PASCAL
73:       USES ax, cx, di, es                  ; Резервирование регистров
74:       push ds                              ; Установка es равным ds
75:       pop es                               ; для внешних подпрограмм
76:       mov di, offset TIntObj_msg           ; Адресация строки-метки
77:       call StrWrite                        ; Отображение строки
78:       mov ax, [(TIntObj PTR si).data_i]    ; Получение целочисленных данных инстанции
79:       mov cx, 1                            ; Минимальное число цифр на выход
80:       mov di, offset TIntObj_buffer        ; Адресация рабочей строки
81:       call BinToAscHex                     ; Преобразование целого числа в строку
82:       call StrWrite                        ; Отображение строки
83:       call NewLine                         ; Начало новой отображаемой строки
84:       ret
85:  ENDP TIntObj_print
```

Объект TIntOBJ (строки 7–13) наследует члены от TItem. Новый объект обеспечивает свои собственные методы конструктора и инициализатора (строки 8, 9), а также замещает виртуальный метод print. Помните, что метод print объекта TItem является просто шаблоном — метод ptint в полученном объекте выполняет реальное действие, когда программа вызывает метод printAll из списка.

TIntObj в строке 12 определяет переменную data_i для хранения целочисленного значения темы. TIntObj наследует указатель TBM от TItem, поэтому нет необходимости в объявление объекта вставлять директиву TBLPTR (в действительности это было бы ошибкой). TIntObj наследует также от TItem переменную next, таким образом инстанция TIntObj может быть подключенной к списку.

Строка 15 использует макрос Make_VMT для создания TBM для TIntObj. Я говорил об этом раньше, но я буду повторять это снова и снова. Полученный объект наследует указатель TBM, но не саму TBM. Все объекты, включая и полученные, и базовый, обладающие одним или более виртуальными методами, должны определять свои собственные TBM.

Строки 19, 20 определяют строчный буфер и строку сообщения для использования в объектных методах. Эти значения собраны в сегменте данных основной программы.

Строка 22 продолжает кодовый сегмент модуля, после которого строки 24, 25 объявляют три внешние подпрограммы, используемые объектными методами. Эти подпрограммы взяты из модулей BINASC.OBJ и STRIO.OBJ, приведенных в этой книге. Ассемблированные модули находятся в библиотечном файле MTA.LIB, созданном при изучении предыдущих глав книги. Любая программа, использующая TIntObj, должна быть подключена к библиотеке. (Пример программы в конце данной главы показывает необходимые для этого шаги.)

Конструктор TIntObj (строки 37–40) инициализирует указатель TBM объектной инстанции — аналогичная задача выполняется всеми конструкторами в примерах объектов этой главы.

Инициализатор TIntObj демонстрирует важную для полученных объектов методику ООП. Метод init полученного объекта имеет две задачи: вызвать метод init базового объекта для инициализации переменных, объявленных для TItem (и унаследованных полученным объектом), и инициализировать свои собственные данные.

Первая задача — вызов метода объекта-родоначальника — выполняется в строке 56. TItem:init не требует никаких аргументов, и, поскольку ds:si уже адресует объектную инстанцию, единственная директива CALL...METHOD удовлетворяет этому требованию.

Вторая задача — инициализация собственных данных полученного объекта — выполняется в строках 57, 58. Сначала переданный аргумент присваивается ax, который затем присваивается переменной data_i объектной инстанции. Между прочим, можно заменить эти две строки одной инструкцией:

```
mov [(TIntObj PTR si) data_i], [@@data]
```

Замещающий виртуальный метод TIntObj_print объекта TIntObj использует строчную подпрограмму и подпрограмму преобразования двоичных чисел в код ASCII, приведенные в этой книге ранее, чтобы отобразить целое значение объектной инстанции. Программа демонстрирует также, что можно легко использовать совместно объектно-ориентированные и обычные подпрограммы. Необходимо быть внимательным при резервировании значений регистров — особенно ds:si и es, которые адресуют объектные инстанции и сегменты TBM, но в остальном программирование является очевидным.

Строки 76, 77 отображают строку "Integer item =" ("Целочисленная тема ="), определенную в начале модуля (строка 20). Строка 78 загружает ax целочисленным значением data_i объектной инстанции, которое преобразуется в строчную форму посредством BinToAscHex и хранится в буфере (строка 19). Строки 82, 83 отображают этот буфер и начинают новую строку отображения.

Список не ограничивается хранением целочисленных данных — просто, образовав из Titem новый объект, можно с таким же успехом хранить любые типы данных. Например, в тексте программы 14.13, TSTROBJ.INC, показано, как создать строчный объект. Используя объект TstrObj из этого модуля и TIntObj — из предыдущего раздела, можно создать списки строк и целых чисел.

Программа 14.13. TSTROBJ.INC

```
1:   %TITLE 'TStrObj object -- by Tom Swan'
2:
3:   GLOBAL TStrObj_construct:PROC
4:   GLOBAL TStrObj_init PROC
5:   GLOBAL TStrObj_print:PROC
6
7:   STRUC TStrObj TItem METHOD {
8:        construct.mptr = TStrObj_construct    , Конструктор TStrObj
9:        init:mptr = TStrObj_init              ; Инициализатор TStrObj
10:       VIRTUAL print.mptr = TStrObj_print    ; Печать, отображение темы
11:  }
12.       data_s dw ?                           ; Ptr для нулевой ограничительной строки
13:  ENDS TStrObj
14:
```

```
15:  Make_VMT                              ; Определение TBM TStrObj
16:
17:  DATASEG
18:
19:  TStrObj_msg db 'String item = ', 0
20:
21:  CODESEG
22:
23:  ;----- Из STRIO.OBJ
24:       EXTRN NewLine:Proc, StrWrite:Proc
25:
26:  ;-----------------------------------------------------
27:  ; TStrObj_construct Конструктор TStrObj
28:  ;-----------------------------------------------------
29:  ; Вход:
30:  ; ds:si = адресу объектной инстанции
31:  ; Выход:
32:  ; Указатель VMT (TBM) инициализирован
33:  ; Регистры:
34:  ; нет
35:  ;-----------------------------------------------------
36:  PROC TStrObj_construct PASCAL
37:       TBLINIT TStrObj PTR si             ; Инициализация указателя TBM
38:       ret
39:  ENDP TStrObj_construct
40:
41:  ;-----------------------------------------------------
42:  ; TStrObj_init Инициализация указателя "next" темы
43:  ;-----------------------------------------------------
44:  ; Вход:
45:  ; ds:si = адресу инстанции TStrObj
46:  ; arg = 16-битовому смещению строки в сегменте данных
47:  ; Выход:
48:  ; инстанция.data_s <- arg
49:  ; Регистры:
50:  ; нет
51:  ;-----------------------------------------------------
52:  PROC TStrObj_init PASCAL
53:       ARG @@data.word
54:       USES ax
55:       CALL si METHOD TItem:init          ; Вызов метода init объекта-родоначальника TItem
56:       mov ax, [@@data]                   ; Получение аргумента из стека
57:       mov [(TStrObj PTR si).data_s], ax ; Присвоение аргумента инстанции
58:       ret
59:  ENDP TStrObj_init
60:
61:  ;-----------------------------------------------------
62:  ; TStrObj_print Печать темы VIRTUAL
63:  ;-----------------------------------------------------
64:  ; Вход:
65:  ; ds:si = адресу инстанции TStrObj
66:  ; Выход:
67:  ; нет
68:  ; Регистры:
69:  ; нет
70:  ;-----------------------------------------------------
71:  PROC TStrObj_print PASCAL
72:       USES di, es                        ; Резервирование регистров
73:       push ds                            ; Установка es равным ds
74:       pop es                             ; для внешних подпрограмм
75:       mov di, offset TStrObj_msg         ; Адресация строки-метки
76:       call StrWrite                      ; Отображение строки
77:       mov di, [(TStrObj PTR si).data_s] ; Получение ptr строки инстанции
78:       call StrWrite                      ; Отображение строки
79:       call NewLine                       ; Начало новой строки отображения
80:       ret
81:  ENDP TStrObj_print
```

TStrObj (строки 7–13) отражает создание TIntObj. Оба объекта получены из TItem, и оба объявляют одинаковые статические и виртуальные методы. Строка 12 определяет переменную data_s, представляющую смещение строки. (Можно также хранить строчные данные непосредственно в объекте — TStrObj демонстрирует только одну из несчетного множества методик хранения данных в объектах.)

Строка 15 использует макрос Make_VMT для создания TBM для TStrObj. Обещаю, что в последний раз я напоминаю это правило: *полученные объекты наследуют только указатели TBM; они не наследуют сами TBM.*

Строка 19 определяет строку сообщения, которое должно отображаться при печати объектных инстанций TStrObj. Строка 24 импортирует две библиотечные подпрограммы из MTA.LIB для отображения начала новой строки и для записи строки в стандартный выходной файл.

Замечание

Несколько модулей, таких как TINTOBJ.INC и TSTROBL.INC, могут без всякого вреда дублировать одни и те же объявления EXTRN. Только одна копия каждой подпрограммы компонуется в заключительную программу.

Конструктор TStrObj (строки 36–39) выполняет обычную задачу инициализации указателя TBM объектной инстанции, используя директиву TBLINIT.

Инициализатор TStrObj (строки 52–59) вызывает метод init TItem для инициализации унаследованных данных и присвоения переменной data_s переданного аргумента word, представляющего смещение адреса строки (строки 56, 57).

И наконец, в новом модуле строки 71–81 осуществляют виртуальный метод print объекта TStrObj. Как и в методе print TIntObj, требуются некоторые манипуляции с регистрами для резервирования es и si, но в остальном программирование достаточно очевидно. Строки 75, 76 отображают сообщение, определенное в начале модуля. Строки 77–79 отображают строчные данные инстанции TStrObj и начинают новую строку вывода.

На этом этапе у вас теперь имеются все компоненты, необходимые для создания списка строчных и целочисленных данных. Образец основной программы, приведенный в тексте программы 14.14, LIST.ASM, демонстрирует, как объединять предыдущие элементы в конечную прикладную программу. Теперь можно ассемблировать и запустить пример программы. Введите следующие команды (если нужно, модифицируйте путь в каталоге для библиотечного файла MTA.LIB, созданного ранее):

```
tasm /zi list
tlink /v list,,, ..\..\MTA.LIB
```

Программа 14.14. LIST.ASM

```
1:  %TITLE "List object demonstration -- by Tom Swan"
2:
3:        IDEAL
4:
5:        JUMPS
6:
7:        LOCALS @@
8:
9:        MODEL small, PASCAL
10:
11:       STACK 1000H
12:
13:       INCLUDE "..\oomacros.inc"
14:
15:       INCLUDE "titem.inc"
16:
17:       INCLUDE "tlist.inc"
18:
19:       INCLUDE "tintobj.inc"
20:
21:       INCLUDE "tstrobj.inc"
22:
23:       DATASEG
24:
25: exCode db 0 ; Код выхода программы
26:
27: ;----- Определение инстанции list
28:
29: list TList <>
30:
31: ;----- Определение инстанций целочисленной темы
32:
33: i1 TIntObj <>
34: i2 TIntObj <>
35: i3 TIntObj <>
36:
37: ;----- Определение инстанций строчной темы
38:
39: s1 TStrObj <>
40: s2 TStrObj <>
```

```
41:
42:  ;----- Определение статических строк для строчных инстанций
43:
44:  str1        db 'Some colors: Red, White, Blue', 0
45:  str2        db 'Some days: Monday, Tuesday, Friday', 0
46:
47:  ;----- Определение различных программных строк
48:
49:  str3        db 'After initializing list...', 0
50:  str4        db 'After inserting integer items...', 0
51:  str5        db 'After inserting string items...', 0
52:  strNum      db 'Number of items in list = ', 0
53:  strBuf      db 20 DUP (0)
54:
55:       CODESEG
56:
57:  ;----- Из BINASC.OBJ, STRIO.OBJ
58:       EXTRN BinToAscDec.Proc, NewLine:Proc, StrWrite:Proc
59:
60:  Start:
61:       mov    ax, @data                    ; Инициализация DS по адресу
62:       mov    ds, ax ; сегмента данных
63:
64:  ;----- Инициализация инстанции list
65:
66:       mov    si, offset list
67:       LoadVMTSeg es
68:       CALL   si METHOD TList:construct
69:       CALL   si METHOD TList:init
70:
71:       mov    di, offset str3
72:       call   DisplayItems
73:
74:  ;----- Инициализация инстанций целочисленной темы
75:
76:       mov    si, offset i1
77:       LoadVMTSeg es
78:       CALL   si METHOD TIntObj:construct
79:       CALL   si METHOD TIntObj:init, 01h
80:
81:       mov    si, offset i2
82:       LoadVMTSeg es
83:       CALL   si METHOD TIntObj:construct
84:       CALL   si METHOD TIntObj:init, 02h
85:
86:       mov    si, offset i3
87:       LoadVMTSeg es
88:       CALL   si METHOD TIntObj:construct
89:       CALL   si METHOD TIntObj.init, 03h
90:
91:  ;----- Инициализация инстанций строчной темы
92:
93:       mov    si, offset s1
94:       LoadVMTSeg es
95:       CALL   si METHOD TStrObj:construct
96:       CALL   si METHOD TStrObj:init, offset str1
97:
98:       mov    si, offset s2
99:       LoadVMTSeg es
100:      CALL   si METHOD TStrObj:construct
101:      CALL   si METHOD TStrObj:init, offset str2
102:
103: ;----- Помещение инстанций целочисленной темы в список
104:
105:      mov    si, offset list
106:      LoadVMTSeg es
107:      mov    ax, offset i1
108:      call   InsertItem
109:      mov    ax, offset i2
110:      call   InsertItem
111:      mov    ax, offset i3
112:      call   InsertItem
113:
114:      mov    di, offset str4
115:      call   DisplayItems
116:
117: ;----- Помещение инстанций строчной темы в список
```

```
118:
119:      mov    ax, offset s1
120:      call   InsertItem
121:      mov    ax, offset s2
122:      call   InsertItem
123:
124:      mov    di, offset str5
125:      call   DisplayItems
126:
127: Exit:
128:      mov    ah, 04Ch                    ; Функция ДОС: Выход из программы
129:      mov    al, [exCode]                ; Возвращение значения кода выхода
130:      int    21h                         ; Вызов функции ДОС. Прерывание программы
131:
132: ;-----------------------------------------------------
133: ; InsertItem Помещение объектной инстанции в список
134: ;-----------------------------------------------------
135: ; Вход:
136: ; ax = смещению инстанции в сегменте данных
137: ; Выход:
138: ; нет
139: ; Регистры:
140: ; si
141: ;-----------------------------------------------------
142: PROC InsertItem PASCAL
143:      mov    si, offset list
144:      LoadVMTSeg es
145:      CALL   TList PTR si METHOD TList:InsertItem, ax
146:      ret
147: ENDP InsertItem
148:
149: ;-----------------------------------------------------
150: ; DisplayItems Отображение всех тем списка
151: ;-----------------------------------------------------
152: ; Вход:
153: ; di = адресу строки сообщения
154: ; Выход:
155: ; нет
156: ; Регистры:
157: ; нет
158: ;-----------------------------------------------------
159: PROC DisplayItems PASCAL
160:      USES   es
161:
162:      mov    si, offset list             ; Адресация инстанции list
163:      LoadVMTSeg es                      ; Подготовка регистра es
164:      CALL   si METHOD TList:getCount    ; Получение количества тем в списке
165:      push   ax                          ; Сохранение результата в стеке
166:
167:      push   ds                          ; Устанавливает es = ds для
168:      pop    es                          ; внешних подпрограмм
169:      call   NewLine                     ; Начинает новую строку отображения
170:      call   StrWrite                    ; Отображает сообщение на дисплее
171:      call   NewLine                     ; Начало новой строки отображения
172:
173:      mov    di, offset strNum           ; Адресация метки числа тем
174:      call   StrWrite                    ; Отображение метки
175:
176:      pop    ax                          ; Получение числа помещенных в список тем
177:      mov    cx, 1                       ; Минимальное число цифр на выход
178:      mov    di, offset strBuf           ; Адресация рабочей строки
179:      call   BinToAscDec                 ; Преобразование целого числа в строку
180:      call   StrWrite                    ; Отображение строки
181:      call   NewLine                     ; Начало новой строки отображения
182:
183:      mov    si, offset list             ; Адресация инстанции list
184:      LoadVMTSeg es                      ; Подготовка регистра es
185:      CALL   TList PTR si METHOD TList:printAll ; Отображение всех тем
186:      ret
187: ENDP DisplayItems
188:
189:      END    Start                       ; Конец программы / точка входа
```

При запуске демонстрационной программы LIST на экране появляется следующая выходная информация:

```
After initializing list...                    После инициализации list...
Number of items in list = 0                   Число тем в списке

After inserting integer items...              После помещения целочисленных тем...
Number of items in list = 3                   Число тем в списке
Integer item = 3 Целочисленная тема
Integer item = 2 Целочисленная тема
Integer item = 1 Целочисленная тема

After inserting string items...               После помещения строчных тем
Number of items in list = 5                   Число тем в списке
String item = Some days· Monday, Tuesday, Friday  Строчная тема
String item = Some colors. red,White, Blue    Строчная тема
Integer item = 3                              Целочисленная тема
Integer item = 2                              Целочисленная тема
Integer item = 1                              Целочисленная тема
```

Эти строки иллюстрируют тот факт, что список может быть пустым, содержать объекты одного типа (целочисленные) или содержать объекты различных типов (целые числа и строки). Число объектов в списке в каждом случае сообщается.

Все эти операции осуществляются почти полностью объектами и методами TItem, TList, TIntObj и TStrObj. Кроме того, сообщения, которые вы видите на экране, отображаются виртуальными методами print, которые демонстрируют, каким образом полиморфизм меняет действия программы простым помещением объектов в список.

Замечание

Для лучшего понимания программирования в этом разделе запустите LIST в Turbo Debugger (введите td list) и проследите код в объектных методах TItem, TList, TIntObj и TStrObj. Поместите также list и другие объектные инстанции в окно *Watces* или просмотрите их значения командой Data:Inspect (подведите к ним курсор и нажмите <Ctrl+W> или <Ctrl+I>).

Строка 9 выбирает для демонстрационной программы малую модель памяти. Хотя вы можете ассемблировать программу для большой и огромной моделей (TItem и TList работают с любыми моделями памяти), поскольку демонстрационная программа обращается к подпрограммам из MTA.LIB, она должна быть ассемблирована для малой модели.

Замечание

Если вы хотите использовать для демонстрации LIST большую модель, вам необходимо сначала реассемблировать все модули в MTA.LIB и перестроить библиотеку, модифицировав директивы MODEL в библиотечных модулях.

Строка 13 содержит файл OOMAKROS.INC, который определяет не зависящие от модели памяти макросы, используемые объектами программы. Программа включает объектные модули в строках 15–21.

Строка 29 определяет инстанцию list объекта TList. В этом примере содержится только один список, но нет никаких ограничений на число списков, которые могут быть определены программой.

Программа нуждается также в том, чтобы в список были помещены несколько инстанций. Строки 33–35 определяют три целочисленные инстанции объекта TIntObj, строки 39, 40 — две строчные инстанции объекта TStrObj. Объектные инстанции могут также храниться в буферах памяти — достаточно использовать ds:si для адресации области соответствующего размера и вызвать объектные методы конструктора и инициализатора для подготовки этой области. Можно получить область памяти, вызвав функцию ДОС либо начав новый сегмент данных. Размещение объектных инстанций целиком зависит от вас.

Строки 49–53 определяют несколько различных строк и строчный буфер для сообщений, отображаемых во время выполнения программы. Эти сообщения показывают, какой участок программы выполняется в данный момент.

Строки 66–69 инициализируют объектную инстанцию list, адресуя ее посредством ds:si и вызывая методы конструктора и инициализатора TList. Теперь list готов к приему инстанций объектов, полученных из TItem.

Строки 71, 72 отображают текущее состояние list, который на этом этапе пуст. На экране вы видите следующие сообщения (нажмите <Alt+F5>, если вы запустили Turbo Debugger, затем нажмите любую клавишу для возврата):

```
After initializing list...                    После инициализации list
Number of items in list = 0                   Число тем в списке
```

Для простоты локальная подпрограмма DisplayItems (строки 159–187) осуществляет это отображение. В подпрограмме строки 162–165 вызывают метод getCount объекта TList для получения числа тем в списке. Это значение преобразуется в строку и отображается в строках 167–181. Затем строки 183–185 вызывают метод printAll списка, который вызывает каждый виртуальный метод print для каждой из тем списка. *Объекты, помещенные в список, сами определяют, какой виртуальный метод print вызвать.* Воспользуйтесь Turbo Debugger, чтобы проследить инструкцию CALL...METHOD в строке 185 для наглядной демонстрации полиморфизма.

Вернемся к строкам 76–89 основной программы, которые инициализируют три целочисленные объектные инстанции программы. Строки 93–101 аналогично инициализируют две строчные объектные инстанции. Как я упоминал, вы должны вызывать методы конструктора и инициализатора для *каждой* объектной инстанции. Каждая инстанция обладает своим собственным указателем ТВМ, который должен индивидуально инициализироваться по адресу ТВМ объекта. Каждая инстанция также содержит свои собственные переменные. Все эти элементы должны быть инициализированы надлежащим образом, прежде чем объектные инстанции смогут использоваться каким-либо образом.

Строки 105–112 помещают в список три целочисленные инстанции. Это делается простой передачей смещения адреса каждой инстанции методу InsertItem списка. После этих инструкций строки 114, 115 снова вызывают локальную подпрограмму DisplayItems — на этот раз, однако, список содержит три целочисленные темы и на экране вы видите сообщения:

```
After inserting integer items...
Number of items in list = 3
Integer item = 3
Integer item = 2
Integer item = 1
```

Вам должно быть понятно, что последние три строчки отображаются виртуальным методом print объекта TIntObj. Программа, однако, не обращается к этому методу непосредственно — вместо этого *объектные инстанции сами определяют, какой метод вызвать*

Строки 119–122 продолжают демонстрацию, помещая в список строчные инстанции программы. И снова это делается просто передачей смещения адресов каждой инстанции объектному методу InsertItem объекта TList. Строки 124, 125 затем опять вызывают DisplayItems для отображения текущего состояния списка. На экране вы видите:

```
After inserting string items...
Number of items in list = 5
String item = Some days: Monday, Tuesday, Friday
String item = Some colors: red,White, BLue
Integer item = 3
Integer item = 2
Integer item = 1
```

Теперь список содержит пять тем — две строчные и три целочисленные. Вы можете прибавить к списку новые инстанции и можете получить из TItem другие типы объектов для помещения в список различной информации. Все, что вам необходимо сделать, — это получить из TItem новый объект и воплотить его методы. Как минимум, объект требует конструктор, инициализатор и виртуальный метод print.

Другие узкие места и "завлекалочки" ООП

До сих пор я умышленно ограничивал эту главу информацией, необходимой для написания прикладных программ ООП на ассемблере. Узкие места и "завлекалочки", о которых пойдет речь ниже, предназначены для квалифицированных программистов, которые хотят продвинуться за пределы основ, а также для тех, кто хочет лучше понять, как Turbo Assembler создает и использует объектные инстанции.

"Жучки" в Debugger

При отладке кода ООП опасайтесь очевидных "жучков" в Turbo Debugger, которые могут разрушить вашу систему. "Жучок" может привести к зависанию компьютера и остановке DOS или Windows.

Вы можете столкнуться с такой проблемой, если TD останавливается кодом останова 0D. Это, очевидно, является следствием того, что отладчик не устанавливает надлежащим образом регистр es при просмотре некоторых видов объектных инстанций. Ошибка, похоже, происходит при просмотре инстанций полученных объектов, которые не объявляют никаких новых членов данных. Попытка просмотреть или проследить за инстанцией такого полученного объекта вызывает остановки.

Чтобы обойти эту проблему, определите в полученном объекте холостой байт или слово данных. Например, создайте свой полученный объект наподобие следующего:

```
STRUC TDerived TBase METHOD {
; methods
}
 dummy_data dw ?          ; Временно: удалить из заключительного кода
ENDS TDerived
```

После отладки можно удалить dummy_data. Можно также использовать обычные ассемблерные директивы для автоматического удаления объявления данных из заключительной программы.

Замечание

Может быть, какой-то талантливый молодой программист фирмы Borland — в действительности все программисты, с которыми я когда-либо встречался в Borland, талантливые и молодые — уже нашел жучка, пока вы читаете этот раздел. Если вы не сможете сами решить проблему, то не стоит и беспокоиться о ней.

Еще немного об указателях TBM

Указатель TBM является 16- или 32-битовой переменной, адресующей TBM объекта. Каждая инстанция объекта, обладающего одним (или более) виртуальным методом, должна иметь указатель TBM. Однако имеется только одна TBM для каждого отдельного объекта. Все инстанции одного и того же объекта пользуются одной и той же TBM. Рис. 14.5 иллюстрирует, как объектные инстанции, указатели TBM и сами TBM располагаются в памяти.

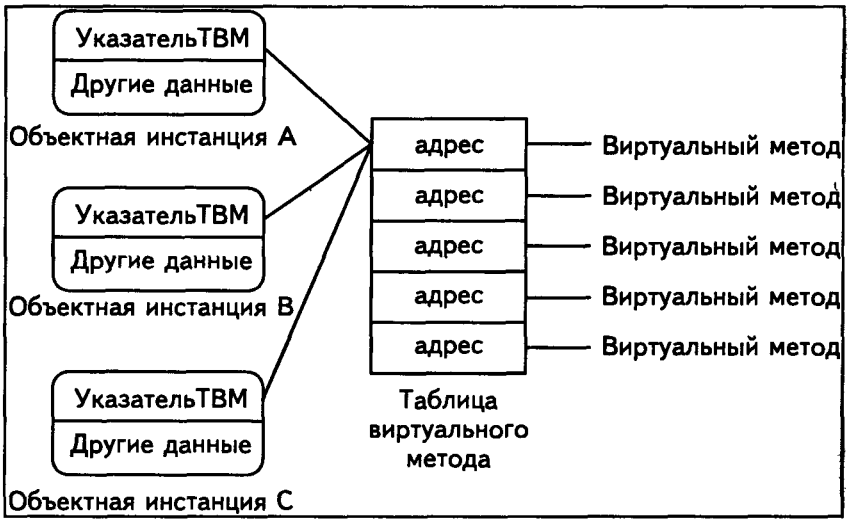


Рис. 14.5. Каждая инстанция объекта обладает указателем TBM, который адресует первый вход объектной TBM. Входы в TBM адресуют объектные виртуальные методы

Директива TBLPTR Turbo Assembler присваивает указателю TBM имя @MPTR_<имя_объекта>. Например, указатель TBM объекта TItem называется @MPTR_TITEM. Любые полученные объекты (TIntObj и TStrObj, к примеру) наследуют этот указатель.

ПОДСКАЗКА

Ассемблируйте программу LIST.ASM с опцией /la и просмотрите результирующий файл LIST.LST для усвоения символов, которые Turbo Assembler генерирует для объектно-ориентированных программ.

Сгенерированные символы указателя представляют смещения в структуре объекта. Реальные указатели TBM являются 16-битовыми словами в программах, использующих малую модель памяти, и 32-битовыми двойными словами в программах, использующих большую и огромную модели памяти. Следующий фрагмент из файла листинга показывает структуру объектов TItem и TStrObj. Обратите внимание, как TStrObj наследует указатель TBM и переменные NEXT от TItem. Значения слов являются смещениями в объектной структуре, представляемыми следующими символами:

```
TITEM
     @MPTR_TITEM Word   0000
     NEXT Word   0002
TSTROBJ
     @MPTR_TITEM        Word              0000
     NEXT Word   0002
     DATA_S      Word   0004
```

Из этой информации нужно извлечь два важных факта. Первый — указатель TBM должен быть первым элементом данных в объекте, второй — объект состоит *только* из данных. Несмотря на тот факт что объект заключает в себе и код, и данные, в действительности объектная инстанция содержит только данные. Связь с кодом осуществляется только в исходном тексте программы (для статических методов) и посредством TBM (для виртуальных методов).

Инициализация указателя TBM

Вы можете использовать информацию предыдущего раздела для инициализации указателя TBM методом, отличным от стандартного, применяющего директиву TBLINIT. Пусть, например, имеется объект TBase, тогда можно использовать код наподобие следующего (ds:si адресует инстанцию, которая должна быть инициализирована):

```
    mov word ptr [si.@Mptr_TBase], offset @TableAddr_TBase
if @CodeSize eq 1
    mov word ptr [si.@Mptr_TBase+2], seg @TableAddr_TBase
endif
```

В малых моделях памяти первая инструкция mov присваивает указателю TBM объектной инстанции адрес TBM, представленный как @TableAddr_TBase.

В большой и огромной моделях первая инструкция mov инициализирует смещение указателя TBM. Третья строчка, следующая после обычной директивы if, инициализирует значение сегмента указателя. Предшествующий код генерируется директивой

```
TBLINIT [si]
```

Замечание

В Руководстве пользователя Turbo Assembler не используются скобки вокруг аргумента TBLINIT. Это приводит к тому, что пример в документации не работает в режиме Ideal.

Вызов виртуальных методов объекта-родоначальника

Для вызова виртуального метода базового объекта из виртуального метода полученного объекта всегда используется статическая функция call. Не используйте CALL...METHOD. Например, для вызова базового виртуального метода action из виртуального метода action полученного объекта применяет код наподобие следующего:

```
PROC TDerived_action PASCAL
 call TBase_action , Call замещает виртуальный метод базового объекта
 ret
ENDP TDerived_action
```

В данном случае существенно то, что виртуальные методы являются всего только подпрограммами, сходными с любыми другими. Однако они адресуются входами в объектной TBM, которая адресуется указателем TBM объектной *инстанции*. Если вместо предшествующих строк вы вставите что-либо наподобие следующей строки, то ваша программа, скорее всего, зависнет:

```
CALL TBase PTR si METHOD TBase.action ; ???
```

Это может показаться верным способом вызова базового виртуального метода, но из-за того, что регистры ds:si адресуют инстанцию *полученного* объекта, инструкция на самом деле осуществляет рекурсивное обращение к полученному методу — к тому, который и делает попытку вызвать базовый метод. И как результат — стек быстро перегружается адресами возврата и программа сбивается.

Если вы сталкиваетесь с перегрузкой стека в объектно-ориентированных программах, вероятной причиной является виртуальный метод, делающий попытку вызвать базовый метод с таким же названием, используя директиву CALL...METHOD. Замените этот код статической инструкцией call для подпрограммы метода.

TBM и сегментная адресация

По соглашению объектные инстанции адресуются посредством ds:si. Это не является жестким правилом, но это соглашение для данной главы, и я советую вам адресовать *все* инстанции аналогично. Использование множества комбинаций регистров для адресации объектных инстанций слишком чревато ошибками.

Имейте в виду, что Turbo Assembler использует сегментный регистр es для адресации TBM. По этой причине трудно использовать es для адресации объектных инстанций.

Как обычно в языке ассемблера, назначения регистров зависят от вас. Однако я нахожу, что применение ds:si для адресации объектных инстанций и резервирование es для адресации ТВМ приводят к наилучшим результатам.

Вызов виртуальных методов без использования CALL...METHOD

Можно вызывать виртуальные методы, не используя CALL...METHOD. Например, в программах, применяющих малую модель памяти, можно воспользоваться кодом вроде следующего:

```
push cs          ; Выталкивание регистра кодового сегмента
pop  es          ; Делает es равным cs
mov  bx,[si]     ; Присвоение указателя ТВМ bx
call es:[bx]     ; Непрямое обращение к подпрограмме с использованием входа ТВМ
```

Программа предполагает, что ds:si адресует объектную инстанцию и указатель ТВМ инстанции является первым пунктом в структуре инстанции. Первые две строки инициализируют es по адресу кодового сегмента — исходя из предположения, что ТВМ хранятся в этом сегменте. Если вы храните их где-то в другом месте — в сегменте данных, например — инициализируйте es соответствующим образом.

Третья строчка перемещает 16-битовый указатель ТВМ из объектной инстанции в bx. Заключительная инструкция косвенно вызывает первый метод, находящийся по адресу, указанному в ТВМ. Чтобы вызвать второй виртуальный метод, к bx необходимо прибавить 02, третий метод — прибавить 04, и т.д. Например:

```
call es:[bx+02] ; Вызов второго виртуального метода (малая модель)
call es:[bx+04] ; Вызов третьего виртуального метода (малая модель)
```

В программах, использующих малую модель памяти, указатели ТВМ являются 16-битовыми смещениями, но вы должны адресовать ТВМ, применяя регистры es и bx (полный 32-битовый указатель). Входы в ТВМ являются 16-битовыми смещениями и предполагается, что весь код размещается в кодовом сегменте программы.

В программах с большой и огромной моделями памяти указатели ТВМ являются 32-битовыми адресами, как и входы ТВМ, поэтому вызов виртуальных методов требует несколько больших усилий. Например, можно использовать код наподобие следующего:

```
mov  bx, 1F62h          ; Помещение сегмента ТВМ в bx
mov  es, bx             ; Установка es в сегмент ТВМ
les  bx, [si]           ; Загрузка es:bx указателем VMT из инстанции
call es:far [bx]        ; Непрямой вызов первого виртуального метода в ТВМ
```

Первые две строки инициализируют регистр es сегментом, в котором хранятся ТВМ. Действительное значение адреса, 1F62, будет отличаться в ваших программах (лучше его заменить именем сегмента). Третья строка использует инструкцию les для загрузки es:bx 32-битовым указателем в объектной инстанции, адресуемой посредством ds:si. Инструкция call в заключительной строке вызывает подпрограмму, хранимую по адресу первого входа ТВМ. Поскольку эти входы являются полными 32-битовыми адресами, вы должны прибавлять 04, а не 02, чтобы получить доступ к следующему виртуальному методу. Например, чтобы вызвать второй виртуальный метод объекта, необходимо использовать следующую инструкцию:

```
call es:far [bx+04] ; Вызов второго виртуального метода (большая и огромная модели)
```

Во всех предшествующих примерах предполагалось, что указатели ТВМ — первые элементы данных объектных инстанций. Возможно создание инстанций, которые хранят указатели ТВМ где-либо в другом месте (или использующие несколько указателей ТВМ), но эта методика требует внимательного программирования и отладки. В таких случаях вместо обращения к CALL...METHOD используйте фрагменты программ этого раздела в качестве руководства по вызову виртуальных методов.

Оптимизированная хвостовая рекурсия

Разновидность директивы CALL...METHOD может оптимизировать некоторые виды методов (виртуальных или статических). Когда метод заканчивается инструкцией CALL...METHOD, можно немного выиграть в быстродействии, используя хорошо известную оптимизирующую методику, названную *оптимизированной хвостовой рекурсией*.

Эта техника не ограничивается объектно-ориентированным программированием. В целом, когда любая подпрограмма заканчивается инструкцией call, за которой следует возврат, эта инструкция call может быть заменена на jmp. Например, рассмотрим, что произойдет в подпрограмме, заканчиваемой следующими двумя инструкциями:

```
call OtherSubroutine
ret
```

Когда OtherSubroutine осуществляет возврат, следуя своей собственной инструкции ret, программа просто выполняет еще одну ret. Два возврата, очевидно, избыточны, и предшествующие две инструкции могут быть заменены следующей:

```
jmp OtherSubroutine
```

Это отсекает хвост подпрограммы, заставляя ее перескакивать к OtherSubroutine, которая осуществляет возврат к первоначальной точке вызова. Ничего не теряя, вы экономите область стека и уменьшаете число инструкций ret с двух до одной. Вот почему эта техника названа оптимизированной хвостовой рекурсией. (Как и многие технические термины, это звучит более экзотично, чем есть на самом деле.)

Объектно-ориентированные программы могут использовать директиву JMP...METHOD Turbo Assembler для выполнения аналогичной оптимизации для некоторых видов методов. Первым шагом является выявление любых методов, заканчивающихся инструкцией CALL...METHOD, или тех, что могут быть модифицированы для этого. Например, загрузите в свой редактор файл TDERIVED.INC (либо снова обратитесь к тексту программы 14.4).

Метод TDerived_getData вблизи конца файла начинается инструкцией CALL...METHOD. Поскольку порядок инструкций метода в этой подпрограмме не критичен, директиву CALL...METHOD можно перемещать к самому ret. Если вы хотите продолжать, преобразуйте метод, чтобы он выглядел следующим образом:

```
PROC TDerived_getData PASCAL
     mov  dx, [(Tderived PTR si) TDerived_data]
     CALL si METHOD TBase getData              ; Переместите сюда эту строку
     ret
ENDP TDerived_getData
```

Свою программу ассемблируйте, скомпонуйте и проверьте, чтобы убедиться, что модифицированный метод работает правильно. Если все нормально, то можете заменить последние две инструкции метода JMP.. METHOD — просто замените CALL на JMP и удалите ret. Оптимизированный метод теперь выглядит следующим образом:

```
PROC TDerived_getData PASCAL
     mov  dx [(Tderived PTR si) TDerived_data]
     JMP  si METHOD TBase getData
ENDP TDerived_getData
```

Резюме

Объектно-ориентированное программирование, или ООП, использует объекты для заключения данных и кода. Особенности Turbo Assembler дают возможность писать объектно-ориентированные программы на языке ассемблера.

Преимущества ООП включают в себя потенциальную простоту отладки, настройки и пересмотра — особенно в больших программах. Недостатками являются начальные трудности создания объектно-ориентированной программы и тот факт, что существует весьма мало стандартных процедур для объектных структур.

ООП характеризуется тремя ключевыми методиками: инкапсуляцией, наследованием и виртуальными методами. В Turbo Assembler данные и код инкапсулируются в специальных объявлениях STRUC, названных *объектами*.

Объект (полученный объект) может наследовать члены кода и данных другого объекта (базового объекта). Подпрограммы объекта, или *методы*, могут быть статическими (вызываемыми непосредственно) или виртуальными (вызываемыми посредством просмотра адресов из таблицы виртуальных методов (ТВМ).

Все объекты, обладающие одним или более виртуальными методами, должны определять указатель ТВМ (либо наследовать указатель от базового объекта). Все такие объекты должны также определять ТВМ. Объекты наследуют указатели ТВМ, но не сами ТВМ. Указатель ТВМ в каждой объектной инстанции нужно инициализировать для указания на ТВМ объекта.

Для вызова объектных методов и передачи им аргументов используйте директиву CALL...METHOD. В некоторых случаях можно заменять CALL...METHOD на JMP...METHOD для оптимизации методов, оканчивающихся обращениями к другим методам.

Полиморфизм является процессом создания объектов, использующих виртуальные методы для выбора действий во время выполнения программы. Объекты TList и TItem в этой главе демонстрируют, как применять полиморфизм для создания списков различных видов объектных инстанций.

Упражнения

14.1. Добавьте к ENCAPSUL.ASM (программа 14.2) операторы CALL...METHOD для установки обоих членов данных объекта p2 в нуль.

14.2. *Объявите новый объект TRect, определяющий координаты левого верхнего и правого нижнего угла воображаемого прямоугольника.* Думайте об этом объекте как о средстве для создания прямоугольной области на текстовом или графическом экране. Включите некоторые методы, которые, по вашему мнению, потребуются объекту.

14.3. *Создайте метод, получающий два аргумента-слова. Вставьте в метод операторы, загружающие аргументы в регистры* cx и dx. Метод должен резервировать регистры, которые он использует. (Результат этого упражнения дает полезный шаблон для начала новых методов.)

14.4. Покажите шаги, требуемые для вызова воображаемого статического метода AnyStatic, объявленного объектом TAnyObject.

14.5. Покажите шаги, необходимые для вызова воображаемого виртуального метода AnyVirtual, объявленного объектом TAnyObject.

14.6. *Усложненное* Создайте новый объект, TDateObj, полученный из объекта этой главы — TItem (программа 14.10). Ваш объект должен обладать членами данных, которые могут хранить дату (год, месяц, день). (*Подсказка*: TItem объявляет виртуальный метод. Убедитесь, что включили в TDateObj все необходимые для виртуального метода компоненты.)

14.7. *Усложненное* Используя объект TDateObj из упражнения 14.6, модифицируйте LIST.ASM (программа 14.14) для хранения и отображения двух инстанций TDateObj в программном списке.

Задания

14.1. Модифицируйте TList, полученный из Titem, так, чтобы инстанции TList сами могли храниться в списке Напишите демонстрационную программу, показывающую, как программа может создать список списков, которые могли бы содержать списки других списков, и т.д. — другими словами, многомерные множества.

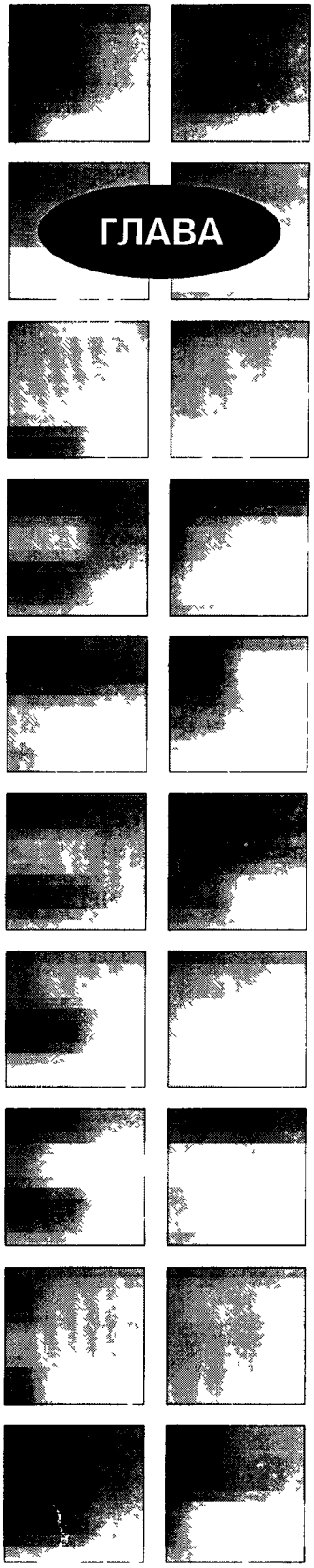
14.2. *Усложненное*. Расширьте TList, чтобы он содержал другие методы для поиска, вставки, удаления и реорганизации инстанций TItem в списках.

14.3. Преобразуйте одну или более объектно-ориентированных программ из C++ или Pascal на язык Ассемблера.

14.4. Преобразуйте объектно-ориентированный пример списка этой главы и связанные с ним модули на C++ или Pascal.

14.5. *Усложненное*. Перекомпилируйте библиотеку MTA.LIB для большой и огромной моделей памяти и пересмотрите пример списка (программа 14.14) в этой главе для использования большой модели. Создайте библиотечные файлы для малой, большой и огромной моделей памяти и придумайте систему для компоновки библиотеки.

14.6. *Усложненное*. Напишите программу, отображающую значения адресов в таблице виртуальных методов.



ГЛАВА 15

Программирование для Windows

- **Введение в программирование для Windows с применением TASM**
- **Развитие прикладных программ для Windows с использованием TASM**
- **Резюме**
- **Упражнения**
- **Задания**

Введение в программирование для Windows с применением TASM

Несмотря на то что большинство разработчиков Windows пишут свои программы на C, для написания программного обеспечения под эту популярную операционную систему можно применять и Turbo Assembler. Вы получаете полное управление над инструкциями запуска и прерывания прикладной программы, можете эффективно использовать регистры и выполнять другие оптимизирующие действия, недоступные программистам на языках высокого уровня.

Однако написание программ для Windows на ассемблере — дело не простое. По этому вопросу имеется мало доступной информации, а то, что публикуется, — плохого качества. Например, документация Borland по TASM и по Windows носит отрывочный характер и содержит множество ошибок. И что еще хуже, примеры программ для Windows на дисках TASM являются неполными либо содержат серьезные дефекты, которые могут разрушить среду Windows и привести к потере информации. (Не используйте примеры TASM в качестве основы для своих программ!)

Чтобы помочь исправить эти недочеты, данный раздел представляет программирование для Windows с применением TASM в режиме Ideal. В последующих разделах вы найдете построчное описание двух полных прикладных программ, которые демонстрируют следующие ключевые моменты программирования для Windows на ассемблере.

- Как писать запускающие модули
- Как инициализировать прикладную программу для Windows
- Как вызвать функции Windows
- Как регистрировать классы окна
- Как создать и отобразить окно

- Как писать цикл для сообщений
- Как получать, ссылаться и пересылать сообщения
- Как создавать и использовать диалоговые окна
- Как создавать и использовать команды поднимающегося меню
- Как рисовать в окне графические изображения

Несмотря на то что в этой главе содержится много информации, она все же не является полным учебником по программированию для Windows. Я раскрываю этот вопрос, насколько это возможно сделать в одной главе, но чтобы написать на ассемблере законченную прикладную программу для Windows, вам нужны также такие учебники, как *Programming Windows 3.1*, Charles Petzold, либо одна из моих книг — *Mastering Windows Programming with Borland C++* или *Type and Learn Windows Programming Using WinScope*. Кроме того, для получения информации, выходящей за пределы этой главы, вам потребуется справочник по Windows API (application programming interface — интерфейс прикладного программирования), а также другие утилиты и файлы, поставляемые с системой Windows.

Замечание

Turbo Assembler не обеспечивает всех необходимых средств и файлов, требующихся для ассемблирования и компоновки файлов с программами для Windows. В дополнение к файлам, которые вы получаете посредством Turbo Assembler, вам нужны такие утилиты, как компиляторы ресурсов и системной помощи, импортируемая библиотека (для подключения программ к функциям Windows) и Turbo Debugger для Windows (TDW). Для программ этой главы требуется Turbo Assembler 4.0 (обычно инсталируется в каталоге C:\TASM), а также Borland C++ 4.0, 4.5 либо более поздних версий (инсталируется в каталоге C:\BC4 либо C:\BC45). Кроме того, каталоге C:\BC4\BIN должна быть указана в пути системы, чтобы можно было подключать примеры программ этой главы.

Минимальное приложение для Windows

О прикладных программах для Windows всегда нужно помнить следующее: они являются просто программами DOS, которые запускаются под управлением операционной системы Windows. В отличие от обычных прикладных программ DOS-программы для Windows должны подчиняться множеству новых правил и ограничений, поэтому овладение необходимыми методиками весьма — сложный процесс.

Программа 15.1, WHELLO.ASM, демонстрирует основные требования к ассемблерным программам для Windows. Ассемблирование и компоновка программы требуют двух дополнительных файлов, приведенных в текстах программ 15.2, WHELLO.DEF, и 15.3, (WHELLO.RC). Назначение каждого из этих файлов я поясню после распечаток. Если я не указываю другой вариант, номера строк в ссылках относятся к WHELLO.ASM.

Программа 15.1. WHELLO.ASM

```
1:  %TITLE "Bare Windows program in assembly language -- by Tom Swan"
2:
3:      IDEAL
4:
5:      JUMPS
6.
7:      P286
8:
9:      LOCALS @@
10:
11:     MODEL  large, WINDOWS PASCAL
12:
13: ;----- Включает объявления Windows (требуется режим MASM)
14.
15:     %NOINCL
16:
17:     MASM
18:
19.     INCLUDE windows.inc
20.
21:     IDEAL
22:
23: ;----- Определение внешних функций, импортируемых из Windows
24:
25: EXTRN      InitTask:PROC
26: EXTRN      WaitEvent PROC
27. EXTRN      InitApp:PROC
28: EXTRN      LoadIcon:PROC
29: EXTRN      LoadCursor.PROC
30: EXTRN      CreateWindow:PROC
```

```
31:  EXTRN       ShowWindow:PROC
32:  EXTRN       UpdateWindow:PROC
33:  EXTRN       RegisterClass:PROC
34:  EXTRN       GetMessage:PROC
35:  EXTRN       TranslateMessage:PROC
36:  EXTRN       DispatchMessage:PROC
37:  EXTRN       PostQuitMessage:PROC
38:  EXTRN       DefWindowProc:PROC
39:
40:  ;----- Определение глобальных программных процедур, вызываемых внутри программного модуля
41:
42:  GLOBAL      PASCAL  WinMain:PROC
43:  GLOBAL      PASCAL  AppInit:PROC
44:  GLOBAL      PASCAL  AppRun:PROC
45:  GLOBAL      PASCAL  RegisterWin:PROC
46:
47:  ;----- Определение программных процедур, экспортируемых в Windows
48:
49:  PUBLIC      WndProc
50:
51:  ;----- Определение исходных тождеств
52:
53:  ID_ICON     EQU    100
54:
55:  ;----- Глобальные инициализированные переменные
56:
57:       DATASEG
58:
59:  ;----- Следующий 16-битовый буфер должен быть первым в программном
60:  ;     сегменте данных. Windows использует эту область для своих целей
61:
62:               DB     16 DUP (0)          ; Резервирование для Windows
63:  exCode       DB     0                   ; Код выхода, возвращаемый DOS
64:  szAppName    DB     'WHello', 0         ; Имя программы или название окна
65:  szWndName    DB     'WHelloWin', 0      ; Имя класса окна
66:
67:  ;----- Глобальные неинициализированные переменные
68:
69:       UDATASEG
70:
71:  psp          DW     ?                   ; Префикс программного сегмента
72:  pszCmdLine   DW     ?                   ; Указатель на командную строку
73:  hPrevInst    DW     ?                   ; Дескриптор предыдущей инстанции
74:  hInstance    DW     ?                   ; Дескриптор данной инстанции
75:  cmdShow      DW     ?                   ; Стиль отображения окна
76:  msg          MSGSTRUCT ?                ; Структура цикла сообщения
77:
78:       CODESEG
79:  Start:
80:
81:  ;----- Начало требуемых инициализаций
82:
83:       call   InitTask                    ; Инициализация данной задачи
84:       or     ax, ax                      ; Проверка результата в ax
85:       jnz    @@InitTaskOk                ; Продолжение, если ax не нуль
86:       jmp    @@InitFail                  ; В противном случае выход с кодом ошибки
87:
88:  @@InitTaskOk:
89:
90:  ;----- Сохранение различных тем, возвращаемых InitTask
91:
92:       mov    [psp], es                   ; Префикс программного сегмента
93:       mov    [pszCmdLine], bx            ; Указатель на командную строку (es:bx)
94:       mov    [hPrevInst], si             ; Дескриптор предыдущей программной инстанции
95:       mov    [hInstance], di             ; Дескриптор данной программной инстанции
96:       mov    [cmdShow], dx               ; Стиль отображения окна
97:
98:  ;----- Продолжение требуемых инициализаций
99:
100:      push   0                           ; Выталкивание ID задачи (0 = текущей задаче)
101:      call   WaitEvent                   ; Очистка любых ожидающих событий
102:      push   di                          ; Выталкивание управления программной инстанции
103:      call   InitApp                     ; Инициализация очереди прикладной программы
104:      or     ax, ax                      ; Проверка результата в ax
105:      jnz    @@InitAppOk                 ; Продолжение, если InitApp успешно
106:      jmp    @@InitFail                  ; В противном случае выход с кодом ошибки
107:
```

```
108: @@InitAppOk:
109:
110:      call   WinMain                    ; Инициализация выполнена - начало прикладной программы
111:      jmp    Exit                       ; Переход к выходу
112:
113: @@InitFail:
114:
115:      mov    [exCode], 0ffh             ; Начальный код ошибки = -1
116:
117: Exit:
118:      mov    ah, 04Ch                   ; Функция DOS: Вызов из программы
119:      mov    al, [exCode]               ; Возвращение значения кода выхода
120:      int    21h                        ; Вызов DOS.  Прерывание программы
121:
122: ;-------------------------------------------------------
123: ; WinMain          Эквивалент WinMain в программе C
124: ;-------------------------------------------------------
125: ; Вход:
126: ;   нет
127: ;   Замечание:эта процедура не требуется, но она позволяет Turbo
128: ;         Debugger пропустить начальный участок программы и начать
129: ;         трассировку с этого места. Очевидно, это происходит потому, что TD
130: ;         распознает в WinMain точку входа в прикладную программу
131: ; Выход:
132: ;   нет
133: ; Регистры:
134: ;   нет
135: ;-------------------------------------------------------
136: PROC   WinMain PASCAL
137:      call   AppInit                    ; Инициализация прикладной программы
138:      call   AppRun                     ; Выполнение цикла сообщения
139:      ret
140: ENDP   WinMain
141:
142: ;-------------------------------------------------------
143: ; AppInit           Регистрация и создание окна прикладной программы
144: ;-------------------------------------------------------
145: ; Вход:
146: ;   hPrevInst  Дескриптор предыдущей инстанции (глобальная)
147: ;   hInstance  Дескриптор данной инстанции (глобальная)
148: ;   cmdShow    Стиль отображения окна (глобальная)
149: ; Выход:
150: ;   нет
151: ; Регистры:
152: ;   ax
153: ;-------------------------------------------------------
154: PROC   AppInit PASCAL
155:      USES   di, si
156:
157:      call   RegisterWin                ; Регистрация основного окна программы
158:      mov    si, [hInstance]            ; Использование si для содержания управляющей
159:                                        ; переменной инстанции
160: ;----- Создание элемента окна из зарегистрированного класса окна
161:
162:      push   ds                         ; Сегмент для szWndName
163:      push   OFFSET szWndName           ; Имя класса окна
164:      push   ds                         ; Сегмент для szAppName
165:      push   OFFSET szAppName           ; Заголовок для титульной полосы
166:      push   WS_OVERLAPPEDWINDOW        ; Стиль окна
167:      push   0                          ; Нижнее слово в Style
168:      push   CW_USEDEFAULT              ; Начальная координата x
169:      push   CW_USEDEFAULT              ; Начальная координата y
170:      push   CW_USEDEFAULT              ; Начальная ширина
171:      push   CW_USEDEFAULT              ; Начальная высота
172:      push   0                          ; Дескриптор для родительского окна (нет)
173:      push   0                          ; Дескриптор для меню (нет)
174:      push   si                         ; Дескриптор программной инстанции
175:      push   0                          ; Необязательные параметры пользователя (нет)
176:      push   0                          ; Необязательные параметры пользователя (нет)
177:      call   CreateWindow               ; Создание элемента окна
178:      mov    di, ax                     ; Сохранение управляющей переменной окна в di
179:
180: ;----- Начало процесса отображения основного окна
181:
182:      push   di                         ; Выталкивание управляющей переменной окна
183:      push   [cmdShow]                  ; Выталкивание стиля окна
184:      call   ShowWindow                 ; Делает окно видимым
```

```
185:
186: ;----- Вызывает немедленное рисование содержания окна
187:
188:      push   di                          ; Выталкивание управляющей переменной окна
189:      call   UpdateWindow                ; Дополнение содержимого окна
190:
191:      ret
192: ENDP   AppInit
193:
194: ;--------------------------------------------------------
195: ; AppRun            Запуск прикладной программы ("цикл сообщения")
196: ;--------------------------------------------------------
197: ; Вход:
198: ;  нет
199: ; Выход:
200: ;  нет
201: ; Регистры:
202: ;  ax
203: ;--------------------------------------------------------
204: PROC   AppRun  PASCAL
205: @@10:
206:      push   ds                          ; Выталкивание сегмента адреса сообщения
207:      push   OFFSET msg                  ; Выталкивание смещения адреса сообщения
208:      push   NULL                        ; Не используется
209:      push   NULL                        ; Не используется
210:      push   NULL                        ; Не используется
211:      call   GetMessage                  ; Получение следующего сообщения
212:      or     ax, ax                      ; GetMessage возвращает нуль?
213:      jz     @@99                        ; Если да, то выход из цикла
214:      push   ds                          ; Выталкивание сегмента адреса сообщения
215:      push   OFFSET msg                  ; Выталкивание смещения адреса сообщения
216:      call   TranslateMessage            ; Перевод сообщений клавиатуры
217:      push   ds                          ; Выталкивание сегмента адреса сообщения
218:      push   OFFSET msg                  ; Выталкивание смещения адреса сообщения
219:      call   DispatchMessage             ; Пересылка сообщения оконной процедуре
220:      jmp    @@10                        ; Цикл до окончания прикладной программы
221: @@99:
222:      ret
223: ENDP   AppRun
224:
225: ;--------------------------------------------------------
226: ; RegisterWin      Регистрация класса основного окна программы
227: ;--------------------------------------------------------
228: ; Вход:
229: ;  hPrevInst  Дескриптор предыдущей инстанции (глобальная)
230: ;  hInstance  Дескриптор данной инстанции (глобальная)
231: ; Выход:
232: ;  нет
233: ; Регистры:
234: ;  ax
235: ;--------------------------------------------------------
236: PROC   RegisterWin  PASCAL
237:      LOCAL  @@wc:WNDCLASS               ; Размещение структуры в стеке
238:      USES   di, si                      ; Резервирование регистров
239:
240:      cmp    [hPrevInst], 0              ; Предыдущая инстанция выполняется?
241:      jne    @@99                        ; Если да, то переход к выходу
242:      mov    si, [hInstance]             ; Использование si для хранения дескриптора инстанции
243:
244: ;----- Присвоение значений глобальной структуре класса окна @@wc
245:
246:      mov    [@@wc.clsStyle], NULL
247:      mov    [WORD PTR @@wc.clsLpfnWndProc     ], OFFSET WndProc
248:      mov    [WORD PTR (@@wc.clsLpfnWndProc) + 2], SEG WndProc
249:      mov    [@@wc.clsCbClsExtra], 0
250:      mov    [@@wc.clsCbWndExtra], 0
251:      mov    [@@wc.clsHInstance], si
252:
253: ;----- Получение и присвоение дескриптора пиктограммы из ресурсов прикладной программы
254:
255:      push   si                          ; Дескриптор программной инстанции
256:      push   0                           ; Верхнее слово из исходного ID
257:      push   ID_ICON                     ; Нижнее слово из исходного ID
258:      call   LoadIcon                    ; Загрузка пиктограммы из ресурсов прикладной программы
259:      mov    [@@wc.clsHIcon], ax         ; Сохранение дескриптора пиктограммы
260:
261: ;----- Получение и присвоение дескриптора курсора
```

```
262:
263:        push   0                          ; Дескриптор инстанции (нет)
264:        push   0                          ; Верхнее слово исходного ID
265:        push   IDC_ARROW                  ; Нижнее слово исходного ID
266:        call   LoadCursor                 ; Загрузка стандартного курсора
267:        mov    [@@wc.clsHCursor], ax      ; Сохранение дескриптора курсора
268:
269: ;----- Присвоение остающихся значений структуры класса окна
270:
271:        mov    [@@wc.clsHbrBackground], COLOR_WINDOW + 1
272:        mov    [WORD PTR @@wc.clsLpszMenuName      ], NULL
273:        mov    [WORD PTR (@@wc.clsLpszMenuName) + 2 ], NULL
274:        mov    [WORD PTR @@wc.clsLpszClassName     ], OFFSET szWndName
275:        mov    [WORD PTR (@@wc.clsLpszClassName) + 2], ds
276:
277: ;----- Регистрация класса окна
278:
279:        push   ss                         ; Выталкивание сегмента класса окна
280:        lea    ax, [@@wc]                 ; Загрузка ax смещением класса окна
281:        push   ax                         ; Выталкивание смещения класса окна
282:        call   RegisterClass              ; Регистрация класса окна
283: @@99:
284:        ret
285: ENDP   RegisterWin
286:
287: ;-------------------------------------------------------
288: ; WndProc           Основная процедура Window (Вызывается Windows)
289: ;-------------------------------------------------------
290: ; Вход:
291: ;   hWnd     WORD (стек)   Дескриптор окна
292: ;   uMsg     WORD (стек)   Идентификатор сообщения
293: ;   wp       WORD (стек)   Необязательный параметр типа слова
294: ;   lp       DWORD (стек)  Необязательный параметр типа двойного слова
295: ; Выход:
296: ;   Зависит от сообщения
297: ; Регистры:
298: ;   ax, dx
299: ;-------------------------------------------------------
300: PROC   WndProc WINDOWS PASCAL FAR
301:        ARG    hWnd:WORD, uMsg:WORD, wp:WORD, lp:DWORD
302:        USES   si
303:
304:        mov    si, [uMsg]                 ; Использование si для хранения сообщения
305:        cmp    si, WM_DESTROY             ; Сообщение - WM_DESTROY?
306:        je     @@WMDESTROY                ; Если да, то переход к формированию сообщения
307:        jmp    @@DEFWINDOWPROC            ; В противном случае переход к первоначальному процессу
308:
309: @@WMDESTROY:
310:        push   0                          ; Выталкивание определяемого пользователем кода выхода
311:        call   PostQuitMessage            ; Вызов Windows для фиксирования сообщения WM_QUIT msg
312:        xor    dx, dx                     ; Возвращение 0L (DWORD ноль) в ax:dx
313:        xor    ax, ax
314:        jmp    @@99
315:
316: @@DEFWINDOWPROC:
317:        push   [hWnd]                     ; Выталкивание дескриптора окна
318:        push   si                         ; Выталкивание значения сообщения
319:        push   [wp]                       ; Выталкивание необязательного параметра типа слова
320:        push   [lp]                       ; Выталкивание необязательного параметра длинного типа
321:        call   DefWindowProc              ; Вызов дескриптора первоначального сообщения
322: @@99:
323:        ret
324: ENDP   WndProc
325:
326:        END    Start                      ; Конец программы / точка входа
```

Программа 15.2. WHELLO.DEF

```
1:   NAME             WHELLO
2:   DESCRIPTION      'WHello v1.00a (C) 1995 -- by Tom Swan'
3:   EXETYPE          WINDOWS
4:   STUB             'WINSTUB.EXE'
5:   CODE             PRELOAD MOVEABLE DISCARDABLE
6:   DATA             PRELOAD MOVEABLE MULTIPLE
7:   HEAPSIZE         1024
```

```
8    STACKSIZE        8192
9    EXPORTS          WndProc
```

Программа 15.3 WHELLO RC

```
1    #define ID_ICON 100
2
3    ID_ICON ICON whello ico
```

Как ассемблировать WHello

Для работы с программой этой главы создайте новый каталог (например, WIN), а в ней создайте подкаталог (например, WHELLO) для работы с программами этого подраздела. Для ассемблирования и компоновки демонстрационной программы WHello воспользуйтесь файлом MAKEFILE, в котором перечислите порядок ассемблирования и компоновки программы. Если вы получите какое-либо сообщение об ошибке, модифицируйте параметры в этом файле. На вашем жестком диске должны быть инсталированы Turbo Assembler 4.0 и Borland C++ 4.0, 4.5 или более поздних версий. C:\TASM\BIN и C:\BC4\BIN либо эквивалентные каталоги должны быть указаны в операторе PATH системы.

При написании MAKEFILE можете воспользоваться соответствующей документацией.

Имеется много способов ассемблирования и компоновки WHello, но желательно добавить в конечный программный файл отладочную информацию — давая возможность запустить его посредством TDW, а также сгенерировать файл листинга программы — WHELLO.LST. Смените подкаталог на WIN\WHELLO, а затем наберите команду

```
make
```

Эта команда ассемблирует и компонует программу, используя созданный ранее файл MAKEFILE.

Вставка в команду опции -B вызывает полную перестройку всей программы. Это может потребоваться, например, после изменения модуля, когда вы хотите реассемблировать программу с использованием отладочной информации.

После ассемблирования и компоновки программы откройте Windows File Manager (Диспетчер файлов Windows) и смените каталог на WIN\WHELLO. Выберите WHELLO.EXE, чтобы запустить программу. Она отобразит окно, показанное на рис. 15.1. Попытайтесь подвигать окно, распахнуть его на весь экран и выполнить другие обычные действия. Эти тесты, хотя и простые, показывают, что WHello полностью подходит для применения с Windows. Воспользуйтесь привычным для себя методом, чтобы завершить программу — например, нажмите <Alt+F4>, дважды нажмите на кнопку в верхнем левом углу либо нажмите на нее один раз, чтобы открыть системное меню и выбрать команду Close (Закрыть).

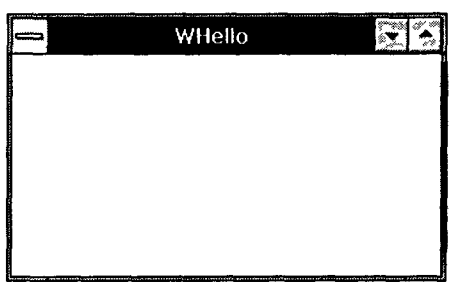


Рис. 15.1. Простое окно WHello

Предварительные действия

Имеется несколько подготовительных шагов, которые необходимо выполнить перед тем, как приступать к написанию первой инструкции программы для Windows. Строки 3–11 в WHELLO.ASM выбирают режим Ideal Turbo Assembler, разрешают оптимизацию переходов, специфицируют инструкции 80286 (Windows 3.1 требует наличия процессора 80286 либо более поздних моделей) и подключают локальные символы, предваряемые @@. Строка 11 является наиболее важной в этом ряду. Она выбирает большую модель памяти, а также специфицирует опции WINDOWS и PASCAL.

Можно применять и другие модели памяти, но большая модель, вероятно, наиболее подходит для ассемблера. Функции Windows должны вызываться с использованием дальних, 32-битовых адресов, поэтому немного можно выиграть, используя программы, написанные для малой модели памяти. (Однако есть и преимущества в обращении к локальным подпрограммам с использованием 16-битовых смещений адресов. В этом случае вы сможете применить малую модель памяти.)

Необходимо специфицировать WINDOWS и PASCAL в директиве MODEL, чтобы Turbo Assembler прибавил необходимые начальные и завершающие инструкции к подпрограммам. Эти опции преобразуют инструкции, генерируемые для директив PROC и ENDP, которые необходимо использовать в начале и конце подпрограмм.

Замечание

Более опытные программисты могут использовать другие соглашения по обращению к внутренним подпрограммам — не обязательно к каждой процедуре добавлять вступительную инструкцию Windows Однако в этом случае обязательно нужно специфицировать WINDOWS, PASCAL и FAR в функциях обратного вызова (callback functions), как это показано в WINHELLO.ASM, в строке 300. (Функция callback является подпрограммой, вызываемой Windows Но подробнее об этом дальше)

Строки 15–21 включают файл WINDOWS.INC, обычно находящийся в C:\TASM\INCLUDE. Если вы располагаете Borland C++, то обнаружите аналогичную копию этого файла в C:\BC4\INCLUDE. Он же поставляется Microsoft Windows Software Development Kit (SDK) В этом файле содержатся различные объявления структур и символов —ассемблерные эквиваленты объявлениям в заглавном файле WINDOWS H для программ C. Объявления этого файла сделаны для режима MASM, поэтому нужно выполнить переключение на этот режим (строка 17) перед тем, как включать его в свою программу. Из-за этого переключения вы не должны использовать кавычки вокруг имени файла (строка 19). Строка 21 осуществляет обратное переключение на режим Ideal для остальной части модуля Строка 15 предотвращает запись WINDOWS.INC в файл листинга программы.

Замечание

Не используйте директиву STACK в прикладных программах для Windows. Windows выделяет область под стек вашей программы в соответствии со значением, специфицированным в файле определения компоновщика (WHELLO DEF в данном примере).

Объявления External и Public

Программа Windows в значительной степени состоит из обращений к Windows API, которая содержит сотни функций Прежде чем вы сможете вызвать функцию Windows, нужно объявить ее внешней (EXTRN) для вашей программы, как показано в строках 25–38 Просто прибавьте новые функции к этому списку, используя следующий стиль:

```
EXTRN ShowWindow PROC
```

Если вы не применили большую модель памяти и опцию PASCAL в директиве MODEL, тогда вместо этого используйте полную форму.

```
EXTRN PASCAL ShowWindow far
```

Строки 42–45 объявляют собственные подпрограммы программы как GLOBAL, а также выбирают для обращения к ним соглашение PASCAL. Вы можете использовать для локальных подпрограмм и другие соглашения по обращению, но PASCAL обещает простоту передачи аргументов в стек. Четыре директивы GLOBAL не являются необходимыми — они просто объявляют подпрограммы таким образом, чтобы операторы могли вызвать их из любого места. В большой программе со множеством модулей вы можете собрать директивы GLOBAL в отдельном файле и подключать его к другим модулям.

Строка 49 объявляет особый тип подпрограммы, известный как *функция обратного вызова (callback function)*. Вы сами никогда не вызываете функцию обратного вызова. Вместо этого нужно передать адрес этой подпрограммы в Windows, которая вызывает эту подпрограмму *обратно* в соответствующие моменты. В данном случае WHello содержит только одну функцию обратного вызова — WndProc, которая должна быть объявлена PUBLIC (общего пользования) (строка 49). Функции обратного вызова нужно писать в надлежащей форме.

Строка 53 объявляет символ ID_ICON, равный значению 100. Символ идентифицирует *источник*, являющийся в данном случае системной "пиктограммой" программы, которую Windows отображает на экране, когда вы минимизируете окно.

Сегменты данных

Программы для Windows могут определять инициализированные и неинициализированные данные. Инициализированные данные хранятся в кодовом файле .EXE программы и загружаются в память во время выполнения. Неинициализированные данные на время выполнения программы занимают байты, не содержащие заранее определенных значений.

Для создания области под инициализированные данные программы применяйте директиву DATASEG. Вы *должны* использовать эту директиву даже в том случае, если ваша программа не содержит глобальных переменных, и резервировать первые 16 байт сегмента данных для собственного использования Windows (строки 57–62).

Вслед за этими требуемыми 16 байтами объявите все глобальные переменные, используемые вашей программой. В данном случае WHello содержит три инициализированные переменные: exCode — значение, возвращаемое DOS при прерывании программы; szAppName представляет имя прикладной программы (отображаемое также в виде заголовка основного окна); zsWndName представляет имя *класса окна*, который описывает характеристики окна. Классы окон должны обладать уникальными именами во всей прикладной программе. Обычно их имена образуются прибавлением "Win" к имени модуля, как сделал я в строке 65, но вы можете использовать и другое имя, если желаете.

Замечание

Префикс sz в szAppName и szWndName означает „zero-terminated string" (цепочка, прерываемая нулем). Вы встретите много подобных префиксов среди символов Windows. Например, lpfn означает „long pointer to a function" (длинный указатель на функцию), wp — „word parameter" (параметр типа слова), lp — „long parameter" (длинный параметр) и т.д. Типичная прикладная программа для Windows содержит тысячи символов, и эти соглашения по поводу имен помогают сохранять „читабельность" программ и, следовательно, упрощают их модификацию и настройку.

В дополнение к инициализации данных программа Windows может выделить память для переменных, которым будут присвоены значения во время выполнения. Перед всеми подобными объявлениями должна стоять директива UDATASEG, как показано в строке 69.

Строки 71–79 объявляют неинициализированные данные WHello. Приведенная программа не использует первые две переменные, содержащие префикс программного сегмента (т.е. смещение адреса, по которому DOS ожидает обнаружить различные темы, относящиеся к этой программе), и смещение адреса цепочки командной строки, если таковая будет передана программе. Эти значения могут оказаться полезными в более сложных прикладных программах.

Строки 73–75 требуются во всех программах для Windows. Переменные hPrevInst и hInstance являются *дескрипторами программных инстанций*. Знакомясь с программированием для Windows, вы часто будете встречаться со словом *дескриптор*. Например, дескриптор — это просто целочисленное значение, представляющее внутренний объект какого-либо вида. Дескриптор программной инстанции идентифицирует задачу, заключающуюся в выполнении кода прикладной программы. Дескриптор окна представляет структуру данных окна, создаваемую Windows. Дескрипторы передаются различным функциям — для отображения специфического окна, например, или для рисования графических образов внутри его границ.

Дескриптор hPrevInst в строке 73 относится к предыдущей программной инстанции, если вы запускаете более одной копии WHello, дескриптор hInstance в строке 74 — к инстанции текущей программы. Попытайтесь теперь запустить более одной копии WHello. Каждая инстанция использует один и тот же код, хранящийся в памяти, но получает свой собственный сегмент данных. Вы можете запустить столько программных инстанций, сколько позволяет память. (Некоторые прикладные программы, однако, не допускают своего исполнения более одного раза.)

Переменная cmdShow в строке 75 представляет стиль основного окна. Обычно это значение устанавливается равным единице, означающей, что окно должно отображаться нормально. Но оно может быть любым из следующих значений, объявленных в WINDOWS.INC:

```
SW_SHOWNORMAL        = 1  (нормальное)
SW_SHOWMINIMIZED     = 2  (минимизированное)
SW_SHOWMAXIMIZED     = 3  (максимизированное)
```

Строка 76 в неинициализированном сегменте данных программы определяет структуру сообщения, имеющую тип MSGSTRUCT, объявленный в WINDOWS.INC. Windows использует много структур для описания различных предметов. В данном случае msg представляет *сообщение*, которое формируется и выполняется посредством программного цикла сообщения (подробнее об этом позже).

Замечание

Вы можете повторять директивы DATASEG и UDATASEG столько раз, сколько необходимо. Вам не обязательно определять все переменные в одном месте или даже в одном модуле. Убедитесь, однако, что вы зарезервировали первые *16 байт в инициализированном сегменте данных для собственного использования* Windows.

Код запуска

Программисты С знают, что функция WinMain — это функция, с которой начинается исполнение программ Windows. Но это верно только по отношению к С операторам программы. Перед WinMain следует критичный код низкого уровня, который должен быть исполнен всеми программами для Windows.

Большинство разрабатывающих систем Windows обеспечивает наличие этого критичного кода в *запускающем модуле*. Если вы располагаете, например, Borland C++, то обнаружите запускающий модуль для Windows в файле C0W.ASM, находящемся в каталоге C:\BC4\LIB\STARTUP. Другие разрабатывающие системы, такие как Borland Pascal для Windows, автоматически добавляют запускающий код в откомпилированные программы. Как правило, запускающий модуль вызывает несколько функций Windows, инициализирует некоторые требуемые переменные и вызывает WinMain.

На языке ассемблера вы должны сами обеспечить все запускающие инструкции, как это показано в строках 78–120 в WHELLO.ASM (загляните в текст программы 15.1). Несмотря на свое имя, запускающий код ответствен также и за прерывание прикладной программы Windows. Вы должны правильно запрограммировать все запускающие процедуры — любая ошибка в них наверняка вызовет серьезные проблемы. Однако одним из преимуществ написания своего собственного запускающего кода является избавление от лишнего багажа. Запускающий модуль Borland C++, например, готовит различные таблицы, вызывает конструкторы статического класса объектов и выполняет другие инициализации, требуемые для функций стандартной библиотеки, но не имеющие значения для программистов на ассемблере.

Первым шагом должен быть вызов InitTask — функции Windows API, объявленной EXTRN в строке 25. Эта функция не требует аргументов, поэтому, как показывает строка 83, вы просто вызываете ее.

Строка 84 проверяет результат InitTask, возвращенный в регистр ax. Если это значение не равно нулю, строка 85 продолжает программу, осуществляя переход к метке в строке 88. Если InitTask возвратило нуль, Windows не может инициализировать задачу (обычно из-за недостатка памяти). В этом случае строка 86 осуществляет переход к метке InitFail.

Замечание

Большинство документации по Windows написано на С, следовательно, вам может потребоваться знание С для написания ассемблерного кода для специфичных операций. Например, вы должны передать в стек все аргументы, требуемые для функций Windows, и вам необходимо ссылаться на значения, возвращаемые в различные регистры, но официальная документация объясняет эти шаги только для С. *Подсказка:* воспользуйтесь опцией Borland C++ -s для компиляции приведенных программ С в эквивалентный ассемблерный текст .ASM. Затем можно просмотреть сгенерированный для руководства текст.

В основном функции Windows возвращают 16-битовые значения в регистр ax. Они возвращают 32-битовые значения в ax:dx. Если эти значения представляют адрес, то сегментная часть находится в dx, смещение — в ax. С некоторыми исключениями (InitTask, например) функции Windows резервируют регистры di, si, bp и ds. Если для хранения переменных вы используете другие регистры, вытолкните их в стек перед вызовом функции Windows, затем, после обращения, вытолкните их из стека.

Теперь вернемся к приведенной программе (снова обратитесь к распечатке программы 15.1). Строки 92–96 сохраняют значения, возвращаемые InitTask. Среди них — дескрипторы инстанции программы и стиль основного окна.

Строки 100–106 продолжают процесс инициализации. Во-первых, сроки 100, 101 вызывают WaitEvent, которая очищает любые ожидающие *события* для данной задачи. Строка 100 выталкивает требуемый аргумент, равный ID задачи (нуль представляет текущую задачу). Всегда имеется одно такое событие — то, которое начинает эту задачу. Обращение в строке 101 очищает его, а также проверяет очередь Windows на предмет других задач, которые могут ожидать своей очереди на выполнение.

Строки 102, 103 выполняют третью и заключительную инициализацию — вызывая функцию InitApp, которая инициализирует очередь сообщений прикладной программы, небольшую область памяти, содержащую сообщения прикладной программы. Регистр ax показывает, успешно ли была выполнена InitApp. Если ax равен нулю — программа должна закончиться немедленно, поскольку она не содержит никакой очереди сообщений. В противном случае строка 110 вызывает WinMain, где начинается действие прикладной программы.

Строки 100–103, кроме того, демонстрируют привлекательный аспект программирования для Windows на ассемблере. Если вы просмотрите дизассемблированную прикладную программу на языке высокого уровня (как поступил я, разрабатывая эту главу), то сможете в запускающем коде обнаружить следующие инструкции:

```
xor  ax, ax
push ax
call WaitEvent
mov  ax, [hInstance]
push ax
call InitApp
```

Этот фрагмент является логическим эквивалентом инструкций, приведенных в строках 100–103 WHELLO.ASM. Однако вместо того, чтобы обнулять ax, а затем выталкивать его в стек, проще вытолкнуть численное нулевое значение. И еще, поскольку di содержит дескриптор программной инстанции, возвращенный предыдущим обращением к InitTask, строка 102 выталкивает это значение, вместо того чтобы перезагружать его из глобальной переменной hInstance.

В данном случае эти небольшие усовершенствования оказывают крошечный и, вероятно неощутимый эффект на быстродействие программы. Но другие маленькие улучшения могут иметь далеко идущие последствия. В языке ассемблера *вы сами* решаете, как использовать регистры и память. В C, Pascal и других языках за вас многие из этих решений принимает компилятор.

Запускающий код ответствен также за прерывание программы Windows, что равносильно окончанию программы DOS. Строки 118–120 прерывают программу WHello после возврата ее функции WinMain. Строка 115 сохраняет –1 в глобальной переменной exCode, которая обычно равна 0, если не было обнаружено никаких ошибок. Инструкции в строках 118–120 копируют это значение в ax вместе с кодом 04Ch функции DOS, а затем выполняют прерывание 21h для возврата в Windows. Операционная система Windows забирает это и другие прерывания у DOS, так что хотя и кажется, что строка 120 осуществляет возврат в DOS, в действительности она передает управление обратно в Windows.

Инициализация регистра сегмента данных

Каждая инстанция программы, другими словами, каждая новая копия программы, исполняемой вами, получает от Windows свой собственный сегмент данных. Вы должны дать возможность Windows обеспечить адресацию сегмента данных и инициализировать регистр ds. Это происходит, когда Windows загружает прикладную программу, и, таким образом, ds оказывается уже инициализированным перед тем, как программа выполняет первую свою инструкцию.

Никогда не устанавливайте ds в @data, как это делается для программ DOS — и как неверно показывают примеры программ Borland на дисках TASM. *Не начинайте свой запускающий код следующими инструкциями:*

```
mov  ax, @data   ; Как легко разрушить прикладную программу
mov  ds, ax      ; для Windows двумя простыми шагами
```

Эта очень грубая ошибка приводит к тому, что все программные инстанции ссылаются на один и тот же сегмент данных. Когда одна из этих инстанций заканчивается, ссылки остальных на данные оказываются в незащищенной области памяти. Подобные ссылки могут привести к GPF (general protection faults — общие сбои защиты) и к тому, что DOS и Windows станут нестабильными. И, что еще хуже, системный организатор памяти может вслед за этим завершить Windows и вернуть вас к приглашению DOS, что может привести к окончательной потере любых несохраненных документов.

ПРЕДУПРЕЖДЕНИЕ

Инициализация ds может плачевно отразиться на здоровье вашей программы. Всегда позволяйте Windows присваивать значение ds.

Функция WinMain

Как я упоминал, строка 110 в тексте программы 15.1, WHELLO.ASM, вызывает подпрограмму WinMain. Этот шаг не обязателен — подпрограмма просто вызывает две другие (см. строки 136–140), таким образом, действие WinMain сводится к нулю. Вы можете удалить WinMain, заменив строку 110 двумя инструкциями:

```
call AppInit
call AppRun
```

Единственная причина включения WinMain — необходимость убедить Turbo Debugger для Windows в том, чтобы он обрабатывал код так, как будто это программа C. Совершенно очевидно, что TDW распознает WinMain в качестве стартовой области прикладной программы.

Запуская TDW, выберите File:Open и активируйте окно Execute startup (выполнение запуска), чтобы начать трассировку программы с первого оператора в WinMain. Отключите это окно, чтобы начать трассировку программы с запускающих инструкций программы (по вызову InitTask в WHello). За исключением того, что WinMain делает возможной эту уловку, она не имеет никакого практического назначения в ассемблерной программе.

Регистрация окна

Кроме запускающего и завершающего кода, прикладная программ для Windows может быть разделена на две стадии. На первой стадии выполняется *регистрация* класса окна и создается инстанция этого класса — как основ-

ное окно программы. На второй стадии выполняется программный *цикл сообщения*, который получает сообщения, предназначенные для программы, и передает эти сообщения вместе с их заключительным местом назначения.

Замечание

Цикл сообщения необходим, потому что Windows применяет многозадачную систему *без приоритетов*. Это означает, что каждая программная инстанция несет ответственность за создание препятствий для выполнения других программ. Приоритетные многозадачные операционные системы сами отвечают за распределение времени для исполняемых задач (разделение времени). Если бы программа Windows исполняла длинные последовательности инструкций без периодических возвратов к циклу сообщений, другие программы не могли бы нормально запускаться.

Вторую стадию я опишу в следующем разделе. Первая стадия, в свою очередь, делится на две операции: регистрацию класса окна и создание окна. В WHello, когда WinMain вызывает AppInit, строка 157 немедленно вызывает RegisterWin — подпрограмму, описанную в строках 236–285. Эта подпрограмма регистрирует класс основного окна программы.

Класс окна — описание характеристик окна (стиля его границ, цвета, пиктограмм, формы курсора и т.п.). В дополнение к этим элементам данных класс окна специфицирует адрес подпрограммы — так называемой *оконной процедуры*, которая отвечает за распоряжение сообщениями, пересылаемыми в окно.

Необходимо зарегистрировать класс окна, после чего можно создавать инстанции (называемые также элементами) этого класса. Можно создавать столько элементов оконного класса, сколько требуется программе. Один и тот же класс используется многими программными инстанциями, поэтому только первая инстанция должна *регистрировать* класс. Последующие инстанции должны использовать тот же зарегистрированный класс для создания элементов своего основного окна.

Прежде чем регистрировать оконный класс, всегда проверяйте, нет ли каких-либо предшествующих программных инстанций. Строки 240, 241 делают это путем проверки hPrevInst, которая хранит дескриптор предыдущей программной инстанции, возвращенный InitTask. Если это значение не равно нулю, то оно идентифицирует предшествующую задачу; следовательно, текущая задача не является первой и можно спокойно считать, что оконный класс уже был зарегистрирован. В противном случае строки 242–282 инициализируют составляющие структуры WINDCLASS и передают адрес структуры функции RegisterClass (строка 282).

Как я упоминал, эта глава не является полным руководством по программированию для Windows, поэтому я поясню только несколько ключевых моментов по регистрации окна. Полное описание значений оконных классов вы можете найти в учебнике по Windows. Обратите внимание, что строка 242 использует si для хранения дескриптора программной инстанции — маленькое, но важное усовершенствование. В основном регистры si и di доступны для часто используемых значений, но для подстраховки зарезервируйте эти регистры, вытолкнув их в стек, либо, как это сделано в WHello — перечислите регистры в директиве USES (строка 238), которая автоматически вставляет необходимые инструкции push и pop в пролог и эпилог подпрограммы.

Для демонстрационных целей я объявил структуру WINDCLASS, @@wc, как локальную переменную в стеке (строка 237). Вам не обязательно делать это — вы можете определить глобальную область для wc, возможно в неинициализированном сегменте данных программы. Однако, поскольку она находится в стеке, wc удаляется после возврата из подпрограммы RegisterWin. После этого структура не нужна, поэтому нет смысла хранить ее в глобальном сегменте данных.

Строки 247, 248 присваивают адрес функции WndProc структуре @@wc. Благодаря этому присвоению элементы окна, созданные из класса, получают свои сообщения, выполненные этой важной подпрограммой, которую я поясню в деталях немного позже.

Строки 255–259 определяют системную пиктограмму для основного окна программы путем выталкивания в стек двух аргументов и вызова функции Windows LoadIcon. Эта последовательность демонстрирует важный, и часто раздражающий, аспект программирования для Windows на языке ассемблера. Некоторые аргументы, такие как дескриптор программной инстанции, выталкиваемый из si в строке 255, являются 16-битовыми значениями, другие, как исходный ID системной пиктограммы, — 32-битовыми. Следовательно, требуется *две* инструкции push, как показано в строках 256, 257. Во всех случаях, вызывая функции Windows, необходимо вытолкнуть в стек правильное число битов, иначе позднее могут возникнуть серьезные проблемы. Всегда дважды проверьте количество и размеры аргументов функций путем сверки с описанием API Windows и дизассемблирования небольших контрольных программ, написанных на C, Pascal либо на другом языке высокого уровня.

Замечание

Если вы используете цепочки для идентификации источников, вытолкните сперва сегмент адреса цепочки, а за ним — его смещение. Например, вы могли бы использовать инструкции push ds и push OFFSET id_icon в строках 256, 257. Я предпочитаю идентифицировать источники целыми значениями, а для таких значений необходимо выталкивать нулевое значение флага, которое представляет адрес мнимого сегмента. Адрес сегмента данных прикладной программы не может быть нулевым, и, следовательно, Windows распознает это значение в качестве индикатора того, что смещение является идентификатором целочисленного источника, а не адресом. Если вы знакомы с программированием для Windows на C, то узнаете в инструкциях в строках 256, 257 эквивалент макроса MAKEINTERSOURSE.

Строки 263–266 вызывают еще одну функцию Windows, для которой также требуется два аргумента — 16-битовый дескриптор инстанции и 32-битовый идентификатор источника. Инструкция call в строке 266 специфицирует форму курсора для курсора мыши при его перемещениях в рабочей области окна (пространстве внутри границ окна). Затем, после вызова LoadCursor, программа сохраняет возвращаемое в регистре ax значение в поле clsHCursor (сокращение для "window class cursor handle" —дескриптор курсора оконного класса) в структуре оконного класса.

После этого и других присвоений структуре оконного класса программа вызывает RegisterClass для регистрации информации оконного класса. Строки 279–281 показывают, как передавать функции локальные стековые переменные посредством адреса. Очевидно, адрес сегмента переменной является тем же, что и сегмент стека, поэтому строка 279 просто выталкивает ss. Строка 280 загружает смещение переменной в ax, используя инструкцию lea (load effective adress — загрузка эффективного адреса). Строка 281 выталкивает значение смещения в стек, а затем строка 282 вызывает RegisterClass. Вы можете использовать аналогичный код для передачи функциям других базирующихся на стеке аргументов.

Вам не обязательно использовать локальные переменные так, как я сделал это для wc. Например, если программа определяет глобальную переменную для структуры WNDCLASS, просто передайте адрес этой структуры в RegisterClass. Чтобы сделать это изменение, сперва определите wc с помощью UDATASEG, как показано ниже:

```
UDATASEG
wc   WNDCLASS  ?
```

Затем в подпрограмме RegisterWin присвойте значения уже не локальной переменной структуры @@wc, а полям wc. (В остальном код является идентичным.) И наконец, передайте в RegisterClass адрес переменной посредством следующих инструкций:

```
push ds                 ; Выталкивание сегмента адреса wc в сегмент данных
push OFFSET wc          , Выталкивание смещения адреса wc
call RegisterClass      , Регистрация оконного класса с использованием значения в wc
```

Замечание

От вас зависит, где хранить переменные вашей программы. При использовании глобальных переменных недостатком является то, что переменная занимает память даже после того, как она больше не нужна. Преимуществом в данном случае является более простой код.

Создание окна

После того как программа регистрирует свой оконный класс, следующей задачей является создание элемента этого класса для использования его в качестве основного окна программы. Выполните это, вызвав другую функцию Windows — CreateWindow. Запомните следующее правило: только первая программная инстанция должна регистрировать оконный класс, но каждая программная инстанция должна создавать свой собственный элемент окна этого класса.

Замечание

Если вы читали предшествующую главу, посвященную объектно-ориентированному программированию, или у вас имеется некоторый опыт ООП, то вы можете заметить некоторое сходство между оконными классами и элементами окна при сравнении их с объектами и объектными инстанциями (или применительно к C++ — с классами и объектами класса). Можно думать об оконном классе как о типе данных, описывающих характеристики окна. Элемент окна — это инстанция оконного класса; он представляет одно или более реальных окон, обладающих характеристиками класса. Самое важное — каждый оконный класс обладает связанной с ним оконной процедурой, которая является грубым эквивалентом объектных методов (или в C++ — групповых функций). Оконные классы заключают в себе данные и код, точно так же, как объекты и классы C++ в ООП. Это не означает, что программы Windows являются объектно-ориентированными, но они используют концепцию инкапсуляции и, до некоторых пределов, наследования оконных классов.

Функция CreateWindow требует набора аргументов, выталкиваемых в стек инструкциями в строках 162–176. В данном случае я снова не буду раскрывать каждое значение, которое объясняется в большинстве учебников по Windows. Однако я прокомментировал каждую строку, так что у вас имеется возможность сравнить эти инструкции с программой C, осуществляющей обращение к CreateWindow, которая выглядит примерно так:

```
WINDCLASS wc,
if (!hPrevInst) {
 wc.style         = NULL;
 wc.lpfnWndProc   = WndProc;
 wc cbClsExtra    = 0,
```

```
wc.cbWndExtra    = 0;
wc.hInstance     = hInst;
wc.hIcon         = LoadIcon(hInst,MAKEINTERSOURCE(ID_ICON));
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = COLOR_WINDOW + 1;
wc.lpszMenuName  = MAKEINTERSOURCE(ID_MENU),
wc lpszClassName = WndName;
RegisterClass(&wc);
}
```

В случае успешного завершения CreateWindow возвращает дескриптор только что созданному оконному элементу. Строка 178 сохраняет этот дескриптор в di, но вы можете сохранить его в глобальной переменной для дальнейшего использования. Затем, в строках 182, 183, программа выталкивает дескриптор окна вместе со значением глобальной переменной cmdShow (возвращаемой InitTask во время старта программы) в стек. Строка 184 потом вызывает ShowWindow, чтобы сделать окно видимым. В C эквивалентный оператор выглядит следующим образом:

```
ShowWindow(hWnd, nCmdShow);
```

СУЩЕСТВЕННАЯ ДЕТАЛЬ

При вызове функций Windows выталкивайте значения аргументов в таком же порядке, как и в эквивалентных операторах C — слева направо. Это срабатывает благодаря использованию соглашения по вызову PASCAL. Кроме того, по этому соглашению функции по возвращении удаляют из стека все аргументы, следовательно, вам не нужно убирать из стека каждый аргумент, который вы туда вытолкнули. Напротив, делая это, можно разрушить стек и вызвать серьезные сбои в программе.

И наконец, в подпрограмме AppInit строки 188, 189 вызывают UpdateWindow, которая не является необходимой, но приводит к тому, что содержание окна дополняется немедленно посредством генерации сообщения WM_PAINT. (Программа WinApp в конце этой главы показывает, как использовать это сообщение.)

Со временем содержание окна так или иначе будет отображено надлежащим образом, и вы можете удалить строки 188, 189. Но ведь это замечательно, когда окно новой программы появляется на экране немедленно. Вызов UpdateWindow, как это показано здесь, — один из способов подтверждения, что именно так и произойдет.

Цикл сообщения

До настоящего момента вся часть программы, с которой вы знакомились, могла быть озаглавлена "Прелюдия к симфонии в WA" (WA — Windows Application — прикладное программирование Windows). Программа инициализирована надлежащим образом, ее оконный класс зарегистрирован (если это первая программная инстанция) и создан элемент этого оконного класса для использования в качестве основного окна программы. Окно сделано видимым, а его содержимое, если таковое имеется, отображено. Наступает момент, когда должны "зазвенеть цимбалы". Так пусть же зазвучит истинная музыка программы!

Возможно, это выглядит излишне драматично применительно к компьютерной программе, но по отношению к Windows главный аккорд прикладной программы — в ее *цикле сообщения*. Именно здесь программа получает сообщения, которые представляют такие *события*, как щелчки мыши и выборы меню. Цикл сообщения направляет сообщения по их истинному назначению — обычно это одна или более оконных процедур, выполняющие действия событий.

Несмотря на свою важность, цикл сообщения — простой кусок программы, как это демонстрирует WHello в подпрограмме AppRun в строках 204–223. Даже если вы не являетесь программистом на C, полезно сравнить ассемблер с обычной версией, написанной на C:

```
MSG msg;
while (GetMessage(&msg, NULL, NULL, NULL)) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

В WHELLO.ASM строки 206–211 передают в GetMessage адрес переменной структуры сообщения — msg, вместе с тремя нулями, представляющими неиспользуемые параметры. GetMessage получает следующее сообщение, если таковое имеется в очереди сообщений прикладной программы. Если функция возвращает нуль, значит, полученное сообщение было WM_QUIT и цикл сообщения должен завершиться (см. строки 212, 213). В противном случае строки 214—216 вызывают TranslateMessage, которая конвертирует сообщения виртуальных клавиш, типа WM_KEYDOWN и WM_KEYUP, в эквивалентные символьные сообщения WM_CHAR. И наконец, строки 217–219 вызывают DispatchMessage, которая пересылает сообщение соответствующей оконной процедуре для обработки.

Цикл сообщения использует глобальную переменную — msg, типа MSGSTRUCT (определенную в строке 76), который является ассемблерным эквивалентом структуры MSG в C. Следует использовать для структуры сообще-

ния глобальную неинициализированную переменную. Например, сравните ссылки на msg в цикле сообщения с локальной структурой оконного класса wc в подпрограмме RegisterWin в строках 236–285.

Замечание

Обратите внимание, как инструкция jmp цикла сообщения в строке 220 осуществляет переход к локальной метке @@10 в строке 205. Может показаться подходящим осуществление перехода к началу подпрограммы — к метке AppRun, но не делайте этого! При использовании директивы MODEL, подобной PASCAL и WINDOWS, Turbo Assembler вставляет различные начальные и конечные инструкции на место директив PROC и ENDP. Например, если вы используете команду View|CPU TDW, то обнаружите инструкции, сохраняющие регистр bp, а затем присваивающие ему текущий указатель стека — sp для адресации стековых параметров. Если инструкция jmp возвращает к началу подпрограммы, программа снова исполняет начальные инструкции, что может вызвать серьезный сбой Чтобы избежать этой проблемы, всегда осуществляйте переход к локальной метке, даже в том случае, если в исходном тексте метка оказывается в начале подпрограммы.

Оконная процедура

Каждое сообщение, полученное циклом сообщения программы и переданное функции DispatchMessage, должно со временем быть обработано оконной процедурой либо первоначальным дескриптором сообщения в среде Windows. Как я объяснял, оконная процедура — это подпрограмма, адресуемая структурой оконного класса (обратитесь снова к строкам 247, 248 в WHELLO.ASM). Например, если вы перемещаете курсор мыши в окно и нажимаете правую кнопку, Windows пересылает оконной процедуре (т.е. циклу сообщения программы) сообщение WM_LBUTTONDOWN.

Типичная оконная процедура может содержать программы для десятков связанных и несвязанных сообщений. По этой причине написание оконных процедур иногда называют *управляемым программирующим событием*. При таком подходе, вместо того, чтобы писать инструкции, зависящие от порядка их исполнения, пишут код, который относится непосредственно к конкретным событиям, — щелчки мыши или нажатие клавиш. Порядок исполнения такого кода зависит от того, как пользователь запускает программу, а не от расположения инструкций в программе.

Ниже приведены несколько ключевых моментов, о которых следует помнить при написании оконных процедур управляемых событиями (номера строк относятся к WINHELLO.ASM):

- Оконная процедура должна допускать повторные входы в нее, т.е. вызываться рекурсивно. Это правило необходимо, поскольку если оконная процедура вызывает функцию Windows, как это часто происходит, функция может осуществить переключение события и сгенерировать сообщение, которое ведет к еще одному обращению к той же самой оконной процедуре перед тем, как осуществится возврат из текущей подпрограммы. Возможно, что множество таких рекурсивных обращений превратится в пачку, подобную самолетам в тумане над дежурным аэропортом, написание глобальных переменных в подпрограмме повторного входа будет подобно приказу этим самолетам летать в одном и том же воздушном пространстве — в таком случае катастрофа почти неизбежна. Для безопасного приземления используйте локальные переменные и аргументы и резервируйте любые используемые регистры (см. строки 301, 302).
- Оконная процедура должна возвращать 32-битовое значение в регистры ax:dx. Для большинства сообщений устанавливайте оба регистра в нуль, если ваша программа обрабатывает сообщение (см. строки 312, 313).
- Передавайте необработанные сообщения DefWindowProc, как это демонстрирует оконная процедура WHELLO.ASM (строки 317–321). Осуществляйте возврат из оконной процедуры немедленно после вызова этой функции, передавая таким образом результат DefWindowProc, находящийся в ax:dx, обратно в точку вызова оконной процедуры.
- Процедура основного окна должна реализовать по крайней мере одно сообщение: WM_DESTROY. В ответ на это сообщение вызовите PostQuitMessage, как это демонстрирует WHELLO.ASM (строки 310, 311).
- Обратитесь к справочнику по Windows API, чтобы выяснить потребность в специальных сообщениях и значения параметров wp и lp. Назначение и использование этих параметров варьируется в широких пределах для различных сообщений.

Помня об этих моментах, просмотрите оконную процедуру WHELLO (обратитесь к строкам 300-324 в WHELLO.ASM). Эта процедура объявляется несколько иначе, чем другие в приведенной распечатке:

```
PROC  WndProc WINDOWS PASCAL FAR
```

Вы должны использовать директиву PROC и сконфигурировать процедуру для WINDOWS. Подпрограмма должна использовать соглашение PASCAL относительно вызова и быть FAR (дальней). Независимо от модели памяти, Windows вызывает функцию, применяя полный 32-битовый адрес, и подпрограмма должна осуществлять возврат, используя инструкцию для возврата из дальней процедуры. Поскольку директива PROC включает в себя ключевое слово FAR, Turbo Assembler автоматически включает эту инструкцию вместо ret в строке 323.

Строка 301 также необходима. Она объявляет четыре параметра, которые Windows передают в стек оконной процедуре:

```
ARG hWnd:word, uMsg.WORD, wp:WORD, lp:DWORD
```

- hWnd:WORD является дескриптором оконного элемента, для которого это сообщение предназначено.
- uMsg:WORD — неприсвоенное целочисленное значение сообщения.
- wp:WORD — необязательное 16-битовое значение, передаваемое вместе с сообщением. Его назначение и смысл зависят от сообщения.
- lp:DWORD — необязательное 32-битовое значение, передаваемое вместе с сообщением. Его назначение и смысл зависят от сообщения.

Строка 302 применяет директиву USES для автоматического сохранения и восстановления регистра si, который оконная процедура использует для хранения копии значения сообщения, находящегося в стековой переменной msg. Если ваша программа использует также и di, обязательно дополните им директиву USES:

```
USES si, di
```

Вам не обязательно хранить значения в si и di, но использование регистров всегда эффективнее, чем ссылки к памяти. Поскольку длинная оконная процедура может обрабатывать десятки сообщений, обычно лучше копировать значение сообщения в si (или другой регистр), как демонстрирует строка 304. Затем сравните это значение с тем, которое обрабатывает ваш код. В данном случае приведенная оконная процедура обрабатывает только одно сообщение — WM_DESTROY. Если si равен значению этого сообщения, строка 306 осуществляет переход к участку, который его обрабатывает; в противном случае строка 307 осуществляет переход к первоначальному процессу.

Замечание

По соглашению я использую локальные метки, такие как @@WMDESTROY, для кода, который обрабатывает сообщение WM_DESTROY. Вы можете назвать метки своей оконной процедуры по своему желанию, но мой выбор помогает идентифицировать, какое сообщение обрабатывает каждый участок.

Строки 309–314 обрабатывают сообщение WM_DESTROY путем вызова функции Windows API PostQuitMessage. После вызова этой функции оконная процедура устанавливает ax:dx в нуль. Это показывает Windows, что сообщение успешно обработано. Строка 314 затем осуществляет переход к инструкции ret, завершающей процедуру.

Строки 316–321 демонстрируют правильный способ обработки сообщений. Сперва вытолкните дескриптор окна, значение сообщения (в si, если вы используете этот регистр), необязательные параметры типа слова и двойного слова. Затем вызовите DefWindowProc для обработки сообщения. Сразу после этого обращения осуществите возврат из оконной процедуры, передавая таким образом возвращаемое значение из DefWindowProc в ax:dx.

В полностью разработанной прикладной программе оконная процедура будет значительно сложнее, чем относительно простая процедура в WHELLO.ASM. Приведенная программа тем не менее демонстрирует основную структуру всех оконных процедур. На примере программы, приведенной в следующем разделе, вы узнаете, как расширить эту структуру для обработки других типов сообщений.

Файл определения компоновщика

Прежде чем обратиться к более сложному примеру программы, необходимо просмотреть два дополнительных файла, требуемых для создания исполняемого кодового файла Windows. Первый файл, WHELLO.DEF (обратитесь к тексту программы 15.2), называется *файлом определения компоновщика*. Он специфицирует размер кучи стека (той области памяти, где могут храниться динамические переменные, создаваемые во время выполнения программы), и ряд других установок.

Большинство руководств по Windows и разрабатывающих систем поясняет опции файла определения компоновщика. Я раскрою только те, которые используются здесь. Номера строк относятся к распечатке 15.2.

1. NAME — имя прикладной программы; обычно оно должно совпадать с именем кодового файла без какого-либо расширения. Это имя не берется в кавычки.
2. DESCRIPTION (ОПИСАНИЕ) — любая цепочка, ограниченная одинарными кавычками. Большинство программистов помещают здесь имя прикладной программы, версию и замечания по поводу авторских прав. Цепочка вставляется в исполняемый кодовый файл.
3. EXETYPE — должен быть WINDOWS. Некоторые справочники указывают на то, что здесь может использоваться номер версии (Windows 3.1, например), но Turbo Linker не распознает этот формат.
4. STUB — специфицирует имя программы DOS, которая исполняется, если пользователь предпринимает попытку запустить программу в ответ на приглашение DOS (что не допускается). Обычно STUB просто отображает сообщение "This program must be run under Microsoft Windows" (Эта программа должна запускаться в среде Microsoft Windows) и на этом заканчивает свою работу, но stub может выполнять любые действия, которые вам требуются. В сущности, вы можете комбинировать прикладные программы для Windows с *лю-*

быми программами DOS, просто указывая различные stub, и, таким образом, вы создадите двойной исполняемый DOS/Windows кодовый файл. Некоторые инсталяционные утилиты используют эту уловку, чтобы они могли исполняться как в программах DOS, так и в программах Windows. Кодовый файл WINSTUB.EXE, показанный здесь, поставляется с Borland C++ и обычно находится в каталоге C:\BC4\BIN. Этот каталог должен быть указан в PATH системы, чтобы компоновщик мог найти stub.

5. CODE — специфицирует атрибуты кодового сегмента программы. Большинство прикладных программ должно использовать три опции, показанные здесь, которые предварительно загружают код в память при исполнении программы, давая возможность Windows перемещать его, при необходимости — освобождать место в памяти, а также позволяя временно удалять сегмент из памяти, когда программа не является активной.
6. DATA — специфицирует атрибуты сегмента данных программы. Большинство прикладных программ должно использовать три опции, показанные здесь, которые предварительно загружают переменные при запуске программы, давая Windows возможность перемещать сегмент данных, чтобы освободить место в памяти, а также создавать новый сегмент данных для каждой программной инстанции.
7. HEAPSIZE — выбирает размер кучи в байтах. Windows хранит динамические переменные, такие как графические кисти, и некоторые другие элементы в куче. (Использование куч в ассемблере выходит за рамки этой главы, но в основном оно такое же, как в C или Pascal.) Используйте значение по крайней мере не меньше приведенного здесь.
8. STACKSIZE — специфицирует размер программного стека. Используйте значение по крайней мере не меньше приведенного здесь.
9. EXPORTS — перечисляет все подпрограммы, экспортируемые в Windows, например оконную процедуру, вызываемую Windows. Поскольку WndProc объявляется в исходном тексте как подпрограмма общего пользования (см. строку 49 в WHELLO.ASM), директива EXPORTS является повторной и ее можно удалить.

Файл описания ресурсов

Конечный файл содержит инструкции, описывающие ресурсы (источники), которые конфигурируют ресурсы программы. Ресурсами являются любые двоичные данные, хранящиеся в исполняемом кодовом файле программы. Инструкции программы загружают ресурсы в память из образа кодового файла и используют их для различных целей.

Например, команды программного меню обычно хранятся в ресурсе меню. Системная пиктограмма может храниться в ресурсе пиктограмм. Диалоговое окно со всеми своими кнопками и элементами управления хранится в диалоговом ресурсе и т.д. Вы можете также создать свои собственные ресурсы, которые могут иметь любые значения по вашему желанию.

Программная распечатка 15.3, WHELLO.RC, приводит единственный ресурс прикладной программы — пиктограмму, хранящуюся в файле WHELLO.ICO. Описатели ресурсов компилируются компилятором ресурсов, обычно утилитой RC.EXE фирмы Microsoft, поставляемой с большинством разрабатывающих систем Windows (но не с Turbo Assembler). В дополнение к RC.EXE Borland C++ поставляет также свою собственную утилиту BRC.EXE, которая функционально эквивалентна RC, но выполняется быстрее.

При компиляции файла описания ресурсов создается двоичная версия описателя в файле, заканчиваемом расширением .RES. Например, компиляция WHELLO.RC посредством следующей команды создает WHELLO.RES:

```
brc whello.rc
```

Инструкция в строке 3 WHELLO.RC копирует файл пиктограммы WHELLO.ICO в WHELLO.RES, который компоновщик подключает к кодовому файлу программы WHELLO.EXE. Файлы WHELLO.ICO и WHELLO.RES, таким образом, не требуются во время выполнения — исполняемый кодовый файл содержит всю прикладную программу, включая и ее ресурсы.

Файл описателя (см. строку 1 WHELLO.RC) определяет также ID_ICON, используя стиль записи C для представления этого ресурса. Одной из проблем при программировании на ассемблере для Windows является то, что компилятор описателей не распознает директив EQU. Поэтому приходится определять каждый идентификатор ресурсов дважды — используя директивы #define C-стиля в описателе ресурсов, и еще раз, используя директивы EQU в языке ассемблера (см., например, строку 53 в WHELLO.ASM).

Развитие прикладных программ для Windows с использованием TASM

Следующие распечатки дают дальнейшее представление о написании прикладных программ для Windows на ассемблере. Приведенная программа создает диалоговые окна, имеет раскрывающееся меню, отображает в окне графику и использует окно сообщений для запроса пользователя о том, заканчивать ли программу.

Я выбрал эти свойства потому, что они демонстрируют типичный код, который потребуется вам в большинстве программ для Windows. Программирование для Windows требует гораздо большего, чем поясняется здесь, но вы сможете использовать следующие тексты программы в качестве пособия при разработке многих прикладных программ различных типов.

Запуск прикладной программы для Windows

Все файлы WinApp в этом разделе приведены вместе. Вслед за текстами программ я поясняю, как работает программа. Если не указано иначе, все номера строк относятся к WINAPP.ASM. Далее следует описание файлов программы. Рис. 15.2 показывает экран программы, включая простейшую графику (прямоугольник внутри границ окна) и раскрывающееся меню.

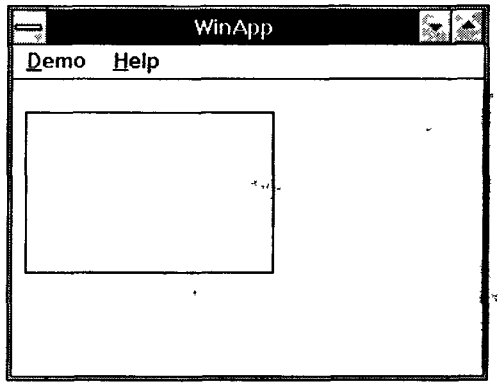


Рис. 15.2. Окно WinApp с простейшей графикой и раскрывающимся меню

- Программа 15.5, WINAPP.DEF — файл определений компоновщика, конфигурирующий сегменты кода и данных и обеспечивающий компоновщик разнообразной информацией.
- Программа 15.6, WINAPP.RC — описать программных ресурсов. Она специфицирует пиктограмму, меню и диалоговое окно. Для компиляции описателя ресурсов вам требуется компилятор ресурсов и заглавный файл WINDOWS.H, поставляемые со всеми C и C++ разрабатывающими системами Windows. Turbo Assembler этого не обеспечивает.
- Программа 15.7, WINAPP.RH, определяет идентификаторы ресурсов, используя директивы #define C-стиля. Этот файл включается в описатель ресурсов.
- Программа 15.8, WINAPP.RI, определяет эквивалентные идентификаторы ресурсов, используя директивы ассемблера EQU. Этот файл включается в WINAPP.ASM.

Программа 15.4. WINAPP.ASM

```
1:   %TITLE "Windows application shell in assembly language -- by Tom Swan"
2:
3:        IDEAL
4:
5:        JUMPS
6:
7:        P286
8:
9:        LOCALS  @@
10:
11:       MODEL   large, WINDOWS PASCAL
12:
13:  ;----- Вставка объявлений Windows  (требуется режим MASM)
14:
15:       %NOINCL
16:
17:       MASM
18:
19:       INCLUDE windows.inc
20:
21:       IDEAL
22:
```

```
;----- Вставка идентификаторов ресурсов

    INCLUDE "winapp.ri"

;----- Определение внешних функций, импортируемых из Windows

EXTRN       InitTask:PROC
EXTRN       WaitEvent:PROC
EXTRN       InitApp:PROC
EXTRN       LoadIcon:PROC
EXTRN       LoadCursor:PROC
EXTRN       CreateWindow:PROC
EXTRN       ShowWindow:PROC
EXTRN       UpdateWindow:PROC
EXTRN       RegisterClass:PROC
EXTRN       GetMessage:PROC
EXTRN       TranslateMessage:PROC
EXTRN       DispatchMessage:PROC
EXTRN       PostQuitMessage:PROC
EXTRN       DefWindowProc:PROC
EXTRN       SendMessage:PROC
EXTRN       MakeProcInstance:PROC
EXTRN       FreeProcInstance.PROC
EXTRN       DialogBox:PROC
EXTRN       EndDialog:PROC
EXTRN       MessageBox:PROC
EXTRN       DestroyWindow:PROC
EXTRN       BeginPaint:PROC
EXTRN       EndPaint:PROC
EXTRN       Rectangle:PROC

;----- Определение глобальных программных процедур, вызываемых внутри программы

GLOBAL      PASCAL  WinMain:PROC
GLOBAL      PASCAL  AppInit:PROC
GLOBAL      PASCAL  AppRun:PROC
GLOBAL      PASCAL  RegisterWin:PROC
GLOBAL      PASCAL  WinAppCommands:PROC
GLOBAL      PASCAL  HelpAbout:PROC

;----- Определение программных процедур, экспортируемых в Windows

PUBLIC      WndProc
PUBLIC      DlgProc

;----- Глобальные инициализированные переменные

    DATASEG

;----- Следующий 16-байтовый буфер должен быть первым в программном
;     сегменте данных. Windows использует эту область для своих целей

            DB      16 DUP (0)          ; Резервируется для Windows
exCode      DB      0                   ; Код выхода, возвращаемый в DOS
szAppName   DB      'WinApp', 0         ; Имя прикладной программы, или заголовок окна
szWndName   DB      'WinAppWin', 0      ; Имя оконного класса
szDlgString DB      'End program?', 0   ; Цепочка окна сообщений

;----- Глобальные неинициализированные переменные

    UDATASEG

psp         DW      ?                   ; Префикс программного сегмента
pszCmdLine  DW      ?                   ; Указатель на цепочку командной строки
hPrevInst   DW      ?                   ; Дескриптор предыдущей инстанции
hInstance   DW      ?                   ; Дескриптор данной инстанции
cmdShow     DW      ?                   ; Стиль отображения окна
msg         MSGSTRUCT ?                 ; Структура цикла сообщения
ps          PAINTSTRUCT ?               ; Структура WM_PAINT

    CODESEG

Start:
```

```
97:  ;----- Начало требуемых инициализаций
98:
99:       call   InitTask                  ; Инициализация данной задачи
100:      or     ax, ax                    ; Проверка результата в ax
101:      jnz    @@InitTaskOk              ; Продолжение, если ax не нуль
102:      jmp    @@InitFail                ; В противном случае - выход с кодом ошибки
103:
104: @@InitTaskOk:
105:
106: ;----- Сохранение различных тем, возвращаемых InitTask
107:
108:      mov    [psp], es                 ; Префикс программного сегмента
109:      mov    [pszCmdLine], bx          ; Указатель на командную строку (es:bx)
110:      mov    [hPrevInst], si           ; Дескриптор предыдущей программной инстанции
111:      mov    [hInstance], di           ; Дескриптор данной программной инстанции
112:      mov    [cmdShow], dx             ; Стиль отображения окна
113:
114: ;----- Продолжение требуемых инициализаций
115:
116:      push   0                         ; Выталкивание ID задачи (0 = текущей задаче)
117:      call   WaitEvent                 ; Очистка любых ожидающих событий
118:      push   di                        ; Выталкивание дескриптора программной инстанции
119:      call   InitApp                   ; Инициализация очереди прикладной программы
120:      or     ax, ax                    ; Проверка результата в ax
121:      jnz    @@InitAppOk               ; Продолжение, если InitApp успешно
122:      jmp    @@InitFail                ; В противном случае - выход с кодом ошибки
123:
124: @@InitAppOk:
125:
126:      call   WinMain                   ; Инициализация выполнена - начало прикладной программы
127:      jmp    Exit                      ; Переход к выходу
128:
129: @@InitFail:
130:
131:      mov    [exCode], 0ffh            ; Начальный код ошибки = -1
132:
133: Exit: mov   ah, 04Ch                  ; Функция DOS: Выход из программы
134:      mov    al, [exCode]              ; Возвращение значения кода выхода
135:      int    21h                       ; Вызов DOS.  Прерывание программы
136: %NEWPAGE
137: ;-------------------------------------------------
138: ; WinMain          Эквивалент WinMain в программе C
139: ;-------------------------------------------------
140: ; Вход:
141: ;   нет
142: ;   Замечание:  эта процедура не обязательна, но она позволяет Turbo
143: ;         Debugger пропустить начальный код и начать
144: ;         трассировку с этого места. Очевидно, это происходит потому, что TD
145: ;         распознает WinMain в качестве точки входа
146: ; Выход:
147: ;   нет
148: ; Регистры:
149: ;   нет
150: ;-------------------------------------------------
151: PROC   WinMain PASCAL
152:      call   AppInit                   ; Инициализация прикладной программы
153:      call   AppRun                    ; Выполнение цикла сообщения
154:      ret
155: ENDP   WinMain
156: %NEWPAGE
157: ;-------------------------------------------------
158: ; AppInit          Регистрация и создание окна программы
159: ;-------------------------------------------------
160: ; Вход:
160: ;   hPrevInst      Дескриптор предыдущей инстанции (глобальная)
162: ;   hInstance      Дескриптор данной инстанции (глобальная)
163: ;   cmdShow        Стиль отображения окна (глобальная)
164: ; Выход:
165: ;   нет
166: ; Регистры:
167: ;   ax
168: ;-------------------------------------------------
169: PROC   AppInit PASCAL
170:      USES   di, si
```

```
171:
172:      call   RegisterWin              ; Регистрация основного окна программы
173:      mov    si, [hInstance]          ; Использование si для хранения дескриптора инстанции
174:
175: ;----- Создание элемента окна из зарегистрированного оконного класса
176:
177:      push   ds                       ; Сегмент для szWndName
178:      push   OFFSET szWndName         ; Имя оконного класса
179:      push   ds                       ; Сегмент для szAppName
180:      push   OFFSET szAppName         ; Заголовок для титульной строки
181:      push   WS_OVERLAPPEDWINDOW      ; Стиль окна
182:      push   0                        ; Нижнее слово стиля
183:      push   CW_USEDEFAULT            ; Начальная координата x
184:      push   CW_USEDEFAULT            ; Начальная координата y
185:      push   CW_USEDEFAULT            ; Начальная ширина
186:      push   CW_USEDEFAULT            ; Начальная высота
187:      push   0                        ; Дескриптор для родительского окна (нет)
188:      push   0                        ; Дескриптор для меню (нет)
189:      push   si                       ; Дескриптор программной инстанции
190:      push   0                        ; Необязательные параметры пользователя (нет)
191:      push   0                        ; Необязательные параметры пользователя (нет)
192:      call   CreateWindow             ; Создание элемента окна
193:      mov    di, ax                   ; Сохранение дескриптора окна в di
194:
195: ;----- Начало процесса отображения основного окна
196:
197:      push   di                       ; Выталкивание дескриптора окна
198:      push   [cmdShow]                ; Выталкивание стиля окна
199:      call   ShowWindow               ; Делает окно видимым
200:
201: ;----- Вызывает немедленное рисование содержания окна
202:
203:      push   di                       ; Выталкивание дескриптора окна
204:      call   UpdateWindow             ; Дополнение содержимого окна
205:
206:      ret
207: ENDP   AppInit
208: %NEWPAGE
209: ;-------------------------------------------------------
210: ; AppRun          Запуск прикладной программы ("цикл сообщения")
211: ;-------------------------------------------------------
212: ; Вход:
213: ;   нет
214: ; Выход:
215: ;   нет
216: ; Регистры:
217: ;   ax
218: ;-------------------------------------------------------
219: PROC   AppRun  PASCAL
220: @@10. push  ds                      ; Выталкивание сегмента адреса сообщения
221:      push   OFFSET msg              ; Выталкивание смещения адреса сообщения
222:      push   NULL                    ; Не используется
223:      push   NULL                    ; Не используется
224:      push   NULL                    ; Не используется
225:      call   GetMessage              ; Получение следующего сообщения
226:      or     ax, ax                  ; GetMessage возвращает нуль?
227:      jz     @@99                    ; Если да, то выход из цикла
228:      push   ds                      ; Выталкивание сегмента адреса сообщения
229:      push   OFFSET msg              ; Выталкивание смещения адреса сообщения
230:      call   TranslateMessage        ; Перевод сообщений клавиатуры
231:      push   ds                      ; Выталкивание сегмента адреса сообщения
232:      push   OFFSET msg              ; Выталкивание смещения адреса сообщения
233:      call   DispatchMessage         ; Пересылка сообщения оконной процедуре
234:      jmp    @@10                    ; Цикл до завершения программы
235: @@99:   ret
236: ENDP   AppRun
237: %NEWPAGE
238: ;-------------------------------------------------------
239: ; RegisterWin      Регистрация класса основного окна программы
240: ;-------------------------------------------------------
241: ; Вход:
242: ;   hPrevInst      Дескриптор предыдущей инстанции (глобальный)
243: ;   hInstance      Дескриптор данной инстанции (глобальный)
244: ; Выход:
```

```
;  нет
; Регистры:
;  ax
;--------------------------------------------------
PROC   RegisterWin PASCAL
     LOCAL  @@wc:WNDCLASS              ; Размещение структуры в стеке
     USES   di, si                     ; Резервирование регистров

     cmp    [hPrevInst], 0             ; Предыдущая инстанция выполняется?
     jne    @@99                       ; Если да, то переход к выходу
     mov    si, [hInstance]            ; Использование si для хранения дескриптора инстанции

;----- Присвоение значений глобальной структуре оконного класса @@wc

     mov    [@@wc.clsStyle], NULL
     mov    [WORD PTR @@wc.clsLpfnWndProc     ], OFFSET WndProc
     mov    [WORD PTR (@@wc.clsLpfnWndProc) + 2], SEG WndProc
     mov    [@@wc clsCbClsExtra], 0
     mov    [@@wc.clsCbWndExtra], 0
     mov    [@@wc.clsHinstance], si

;----- Получение и присвоение дескриптора пиктограммы из ресурсов прикладной программы

     push   si                         ; Дескриптор программной инстанции
     push   0                          ; Верхнее слово  ID ресурса
     push   ID_ICON                    ; Нижнее слово ID ресурса
     call   LoadIcon                   ; Загрузка пиктограммы из ресурсов прикладной программы
     mov    [@@wc.clsHicon], ax        ; Сохранение дескриптора пиктограммы

;----- Получение и присвоение дескриптора курсора

     push   0                          ; Дескриптор инстанции
     push   0                          ; Верхнее слово ID ресурса
     push   IDC_ARROW                  ; Нижнее слово ID ресурса
     call   LoadCursor                 ; Загрузка стандартного курсора
     mov    [@@wc.clsHCursor], ax      ; Сохранение дескриптора курсора

;----- Присвоение остающихся значений структуры оконного класса

     mov    [@@wc.clsHbrBackground], COLOR_WINDOW + 1
     mov    [WORD PTR @@wc.clsLpszMenuName     ], ID_MENU
     mov    [WORD PTR (@@wc clsLpszMenuName) + 2 ], 0
     mov    [WORD PTR @@wc.clsLpszClassName    ], OFFSET szWndName
     mov    [WORD PTR (@@wc clsLpszClassName) + 2], ds

;----- Регистрация оконного класса

     push   ss                         ; Выталкивание сегмента wc
     lea    ax, [@@wc]                 ; Загрузка ax смещением wc
     push   ax                         ; Выталкивание смещения wc
     call   RegisterClass              ; Регистрация оконного класса
@@99:
     ret
ENDP   RegisterWin
%NEWPAGE
;--------------------------------------------------
; WndProc          Основная оконная процедура (вызывается Windows)
;--------------------------------------------------
; Вход:
;  hWnd     WORD (стек)   Дескриптор окна
;  uMsg     WORD (стек)   Идентификатор сообщения
;  wp       WORD (стек)   Необязательный параметр типа слова
;  lp       DWORD (стек)  Необязательный параметр типа двойного слова
; Выход:
;  Зависит от сообщения
; Регистры:
;  ax, dx
;--------------------------------------------------
PROC   WndProc WINDOWS PASCAL FAR
     ARG    hWnd:WORD, uMsg:WORD, wp:WORD, lp:DWORD
     USES   di, si

     mov    di, [hWnd]                 ; Использование di для хранения дескриптора окна
     mov    si, [uMsg]                 ; Использование si для хранения сообщения
```

```
319:      cmp   si, WM_DESTROY              ; Сообщение - WM_DESTROY?
320:      je    @@WMDESTROY                 ; Если да, то переход к обработке сообщения
321:      cmp   si, WM_CLOSE                ; Сообщение - WM_CLOSE?
322:      je    @@WMCLOSE                   ; Если да, то переход к обработке сообщения
323:      cmp   si, WM_PAINT                ; Сообщение - WM_PAINT?
324:      je    @@WMPAINT                   ; Если да, то переход к обработке сообщения
325:      cmp   si, WM_COMMAND              ; Сообщение - WM_COMMAND?
326:      je    @@WMCOMMAND                 ; Если да, то переход к обработке сообщения
327:      jmp   @@DEFWINDOWPROC             ; В противном случае - переход к первоначальному процессу
328:
329: ;----- Обработка сообщения WM_DESTROY
330:
331: @@WMDESTROY:
332:      push  0                           ; Выталкивание определяемого пользователем кода выхода
333:      call  PostQuitMessage             ; Вызов Windows для установки сообщения WM_QUIT
334:      jmp   @@RETURNZERO                ; Возвращение 0L (длинного 32-битового нуля)
335:
336: ;----- Обработка сообщения WM_CLOSE
337:
338: @@WMCLOSE:
339:      push  di                          ; Выталкивание дескриптора окна
340:      push  ds                          ; Выталкивание сегмента диалоговой цепочки
341:      push  OFFSET szDlgString          ; Выталкивание смещения диалоговой цепочки
342:      push  ds                          ; Выталкивание сегмента титульной цепочки
343:      push  OFFSET szAppName            ; Выталкивание смещения титульной цепочки
344:      push  MB_ICONQUESTION OR MB_YESNO ; Выталкивание стилей окна сообщений
345:      call  MessageBox                  ; Вызов функции Windows
346:      cmp   ax, IDYES                   ; Пользователь выбрал кнопку Yes?
347:      jne   @@RETURNZERO                ; Если нет, то выход из процесса WM_CLOSE
348:      push  di                          ; В противном случае - выталкивание дескриптора окна
349:      call  DestroyWindow               ; Разрушение окна и окончание программы
350:      jmp   @@RETURNZERO                ; Возвращение 0L (длинного 32-битового нуля)
351:
352: @@WMPAINT:
353:      push  di                          ; Выталкивание дескриптора окна
354:      push  ds                          ; Выталкивание сегмента структуры ps
355:      push  OFFSET ps                   ; Выталкивание смещения структуры ps
356:      call  BeginPaint                  ; Обработка рисования GDI
357:      mov   si, ax                      ; Сохранение контекстного дескриптора в si
358:
359: ;----- Вызов функции GDI
360:
360:      push  si                          ; Выталкивание HDC
362:      push  10                          ; Выталкивание координат прямоугольника
363:      push  25
364:      push  200
365:      push  150
366:      call  Rectangle                   ; Рисование прямоугольника
367:
368: ;----- Вставьте здесь обращения к другим функциям GDI
369:
370:      push  di                          ; Выталкивание дескриптора окна
371:      push  ds                          ; Выталкивание сегмента структуры ps
372:      push  OFFSET ps                   ; Выталкивание смещения структуры ps
373:      call  EndPaint                    ; Окончание рисования GDI
374:      jmp   @@RETURNZERO                ; Возвращение 0L (длинного 32-битового нуля)
375:
376: ;----- Обработка сообщения WM_COMMAND
377:
378: @@WMCOMMAND:
379:      push  di                          ; Выталкивание дескриптора окна
380:      push  [wp]                        ; Выталкивание параметра типа слова
381:      push  [lp]                        ; Выталкивание длинного параметра
382:      call  WinAppCommands              ; Вызов обработчика наших команд
383:      jmp   @@RETURNZERO                ; Возвращение 0L (длинного 32-битового нуля)
384:
385: ;----- Вызов обработчика первоначального сообщения системы Windows
386:
387: @@DEFWINDOWPROC:
388:      push  di                          ; Выталкивание дескриптора окна
389:      push  si                          ; Выталкивание значения сообщения
390:      push  [wp]                        ; Выталкивание необязательного параметра типа слова
391:      push  [lp]                        ; Выталкивание необязательного параметра длинного типа
392:      call  DefWindowProc               ; Вызов обработчика первоначального сообщения
```

```
393:      jmp    @@99                    ; Возвращение результата DefWindowProc
394:
395: @@RETURNZERO:
396:      xor    ax, ax                  ; Возвращение 0L (двойное слово нуля) в ax:dx
397:      xor    dx, dx
398: @@99:
399:      ret
400: ENDP   WndProc
401: %NEWPAGE
402: ;-------------------------------------------------
403: ; DlgProc          Диалоговая процедура (вызывается Windows)
404: ;-------------------------------------------------
405: ; Вход:
406: ;  hWndDlg  WORD (stack)   Дескриптор для диалогового окна
407: ;  uMsg     WORD (stack)   Идентификатор сообщения
408: ;  wp       WORD (stack)   Идентификатор WM_COMMAND
409: ;  lp       DWORD (stack)  Необязательный параметр типа двойного слова
410: ; Выход:
411: ;  Зависит от сообщения
412: ; Регистры:
413: ;  ax
414: ;-------------------------------------------------
415: PROC   DlgProc WINDOWS PASCAL FAR
416:      ARG    hWndDlg:WORD, uMsg:WORD, wp:WORD, lp:DWORD
417:      USES   si
418:
419:      mov    si, [uMsg]              ; Использование si для хранения сообщения
420:      cmp    si, WM_INITDIALOG       ; Это WM_INITDIALOG?
421:      je     @@WMINITDIALOG          ; Если да, то переход к обработке сообщения
422:      cmp    si, WM_COMMAND          ; Это WM_COMMAND?
423:      je     @@WMCOMMAND             ; Если да, то переход к обработке сообщения
424:      jmp    @@RETURNFALSE           ; Другие сообщения - выход
425:
426: ;----- Обработка сообщения WM_INITDIALOG
427:
428: @@WMINITDIALOG:                    ; Вставьте здесь любые инициализирующие процедуры
429:      jmp    @@RETURNTRUE            ; Выход и возвращение TRUE (ИСТИННО)
430:
431: ;----- Обработка сообщения WM_COMMAND (для диалоговых кнопок и т.п.)
432:
433: @@WMCOMMAND:
434:      mov    si, [wp]                ; Использование si для хранения идентификатора WM_COMMAND
435:      cmp    si, IDOK                ; Это IDOK?
436:      je     @@IDOK                  ; Если да, то переход к выполнению команды
437:      jmp    @@RETURNFALSE           ; Иначе выход и возвращение FALSE (ЛОЖНО)
438:
439: ;----- Обработка команды IDOK (т.е. когда пользователь выбирает кнопку OK)
440:
441: @@IDOK:
442:      push   [hWndDlg]               ; Выталкивание дескриптора диалогового окна
443:      push   0                       ; Выталкивание значения для возврата в точку вызова
444:      call   EndDialog               ; Завершение диалога
445: ;    jmp    @@RETURNTRUE            ; Разрешение добавлять команды
446:
447: @@RETURNTRUE:
448:      mov    ax, TRUE                ; Возвращение булевого значения TRUE (ИСТИННО)
449:      jmp    @@99
450:
451: @@RETURNFALSE:
452:      mov    ax, FALSE               ; Возвращение булевого значения FALSE (ЛОЖНО)
453: @@99:  ret
454: ENDP   DlgProc
455: %NEWPAGE
456: ;-------------------------------------------------
457: ; WinAppCommands      Подпрограмма команд меню
458: ;-------------------------------------------------
459: ; Вход:
460: ;  hWnd     WORD (stack)   Дескриптор для окна
461: ;  wp       WORD (stack)   Параметр типа слова (ID команды)
462: ;  lp       DWORD (stack)  Необязательный параметр типа двойного слова
463: ; Выход:
464: ;  нет
465: ; Регистры:
466: ;  нет
```

```
467: ;-----------------------------------------------------
468: PROC   WinAppCommands PASCAL
469:       ARG    hWnd:WORD, wId:WORD, lp:DWORD
470:       USES   di, si
471:
472:       mov    di, [hWnd]                ; Перемещение дескриптора окна в di
473:       mov    si, [wId]                 ; Перемещение ID команды в si
474:       cmp    si, CM_DEMO_EXIT          ; Команда - CM_DEMO_EXIT?
475:       je     @@CMDEMOEXIT              ; Если да, то переход к выполнению
476:       cmp    si, CM_HELP_ABOUT         ; Команда - CM_HELP_ABOUT?
477:       je     @@CMHELPABOUT             ; Если да, то переход к выполнению
478:       jmp    @@99                      ; неопознанная команда - выход
479:
480: ;----- Выполнение команды меню Demo:Exit
481:
482: @@CMDEMOEXIT:
483:       push   di                        ; Выталкивание дескриптора окна
484:       push   WM_CLOSE                  ; Выталкивание сообщения, подлежащего пересылке
485:       push   0                         ; Выталкивание неиспользованного параметра типа слова
486:       push   0                         ; Выталкивание неиспользованного длинного параметра (1)
487:       push   0                         ; Выталкивание неиспользованного длинного параметра (2)
488:       call   SendMessage               ; Пересылка сообщения WM_CLOSE
489:       jmp    @@99
490:
491: ;----- Выполнение команды меню Help:About
492:
493: @@CMHELPABOUT:
494:       push   di                        ; Выталкивание дескриптора окна
495:       call   HelpAbout                 ; Вызов нашей подпрограммы about-box
496: ;     jmp    @@99                      ; Позволяет добавлять команды
497:
498: @@99:  ret
499: ENDP   WinAppCommands
500: %NEWPAGE
501: ;-----------------------------------------------------
502: ; HelpAbout          Подпрограмма About box
503: ;-----------------------------------------------------
504: ; Вход:
505: ;  hWnd     WORD (stack)   Дескриптор для окна, ведущего диалог
506: ; Выходт:
507: ;  нет
508: ; Регистры:
509: ;  нет
510: ;-----------------------------------------------------
511: PROC   HelpAbout PASCAL
512:       ARG    hWnd:WORD
513:       USES   di, si
514:
515:       push   SEG DlgProc               ; Выталкивание сегмента диалоговой процедуры
516:       push   OFFSET DlgProc            ; Выталкивание смещения диалоговой процедуры
517:       push   [hInstance]               ; Выталкивание дескриптора программной инстанции
518:       call   MakeProcInstance          ; Создание процедурной инстанции
519:       mov    di, dx                    ; Сохранение сегмента адреса в di
520:       mov    si, ax                    ; Сохранение смещения адреса в si
521:
522:       push   [hInstance]               ; Выталкивание дескриптора программной инстанции
523:       push   0                         ; Выталкивание значения сегмента (0 = флагу)
524:       push   ID_ABOUT                  ; Выталкивание ID ресурса диалогового окна
525:       push   [hWnd]                    ; Выталкивание дескриптора окна
526:       push   di                        ; Выталкивание сегмента процедурной инстанции
527:       push   si                        ; Выталкивание смещения процедурной инстанции
528:       call   DialogBox                 ; Создание диалогового окна
529:
530:       push   di                        ; Выталкивание сегмента процедурной инстанции
531:       push   si                        ; Выталкивание смещения процедурной инстанции
532:       call   FreeProcInstance          ; Освобождение процедурной инстанции
533:
534:       ret
535: ENDP   HelpAbout
536:
537:       END    Start          ; Конец программы / точка входа
```

Программа 15.5. WINAPP.DEF

```
1:  NAME            WINAPP
2:  DESCRIPTION     'WinApp v1.00 (C) 1995 -- by Tom Swan'
3:  EXETYPE         WINDOWS
4:  STUB            'WINSTUB.EXE'
5:  CODE            PRELOAD MOVEABLE DISCARDABLE
6:  DATA            PRELOAD MOVEABLE MULTIPLE
7:  HEAPSIZE        1024
8:  STACKSIZE       8192
9:  EXPORTS         WndProc
10:                 DlgProc
```

Программа 15.6. WINAPP.RC

```
1:  #include <windows.h>
2:  #include "winapp.rh"
3:
4:  ID_ICON ICON winapp.ico
5:
6:  ID_MENU MENU
7:  BEGIN
8:    POPUP "&Demo"
9:    BEGIN
10:     MENUITEM "E&xit", CM_DEMO_EXIT
11:   END
12:   POPUP "&Help"
13:   BEGIN
14:     MENUITEM "&About...", CM_HELP_ABOUT
15:   END
16: END
17:
18: ID_ABOUT DIALOG 6, 15, 180, 98
19: STYLE WS_DLGFRAME | WS_POPUP
20: CAPTION "About WAbout"
21: {
22:  DEFPUSHBUTTON "OK", IDOK, 13, 47, 16, 40
23:  ICON ID_ICON, -1, 12, 16, 18, 16
24:  CONTROL "", -1, "static", SS_BLACKFRAME, 42, 8, 133, 81
25:  LTEXT "WinApp v1.00", -1, 55, 18, 112, 8
26:  LTEXT "Copyright \251 1995 by Tom Swan", -1, 54, 35, 112, 8
27:  LTEXT "All rights reserved", -1, 55, 52, 112, 8
28:  LTEXT "From Mastering Turbo Assembler 2nd Ed", -1, 55, 69, 112, 8
29: }
```

Программа 15.7. WINAPP.RH

```
1:  // ================================================
2:  // winapp.rh -- Константы ресурса (исходный или C-модули)
3:  // ================================================
4:
5:  // Идентификаторы ресурса
6:
7:  #define ID_ICON    100
8:  #define ID_MENU    100
9:  #define ID_ABOUT   100
10:
11: // Идентификаторы команд меню
12:
13: #define CM_DEMO_EXIT                101
14: #define CM_HELP_ABOUT               999
```

Программа 15 8 WINAPP RI

```
1     ;================================================
2     ; winapp ri -- Константы ресурса (ассемблерные модули)
3     ;================================================
4
5     ; Идентификаторы ресурса
6
7     ID_ICON           EQU             100
8     ID_MENU           EQU             100
9     ID_ABOUT          EQU             100
10
11    ; Идентификаторы команд меню
12
13    CM_DEMO_EXIT      EQU             101
14    CM_HELP_ABOUT     EQU             999
```

Как ассемблировать WinApp

Для работы с программами этого подраздела создайте в вашем каталоге WIN подкаталог WINAPP. Используйте те же команды для ассемблирования и компоновки WinApp, которые вы применяли для WHello Смените каталог на WIN\WINAPP и наберите make, предварительно создав аналогичный MAKEFILE для WinApp.

Замечание

Читая об инструкциях WinApp, используйте TDW для трассировки Выполните ассемблирование и компоновку, запустите TDW и используйте команду File|Open для открытия WINAPP EXE.

Обзор WinApp

Программа WinApp (см. текст программы 15.4) в значительной степени аналогична WHello, поэтому я поясню только те ее фрагменты, которые имеют существенные отличия. Строка 25 включает идентификаторы ресурсов из WINAPP.RI Типичные прикладные программы Windows содержат множество ресурсов, и обычно лучше объявить идентификаторы в отдельном файле, как показано здесь.

Строки 29–52 объявляют ряд функций Windows API, но, если не обращать внимания на количество функций, этот кусок аналогичен WHello. Этот список можно дополнить любыми функциями Windows, однако если он при этом станет значительно больше, то вы можете поместить объявления в отдельном файле. (Подсказка: рассортируйте файл в алфавитном порядке, чтобы иметь возможность легко определить, помещена ли в список какая-либо функция В остальном порядок объявлений не имеет значения.)

Строки 56–66 объявляют глобальные процедуры программы, а также делают доступными для общего пользования ее две функции обратного вызова, WndProc и DlgProc, чтобы Windows мог обращаться к ним. Благодаря операторам, помещенным в строках 65, 66, файл определения компоновщика не должен экспортировать имена подпрограмм. Однако сделав это, вы не причините программе никакого вреда, как показано в строках 9, 10 WINAPP.DEF (программа 15.5).

Сегмент данных программы (обратитесь снова к тексту программы 15.4) аналогичен WHello, но объявляет еще несколько переменных, назначение которых я поясню позже. Запускающий код одинаков в обеих программах. Как и в WHello, WinMain (строки 151–154) не являются обязательными, но они дают возможность TDW правильно *выполнить запускающий код*.

Операции по регистрации и отображению окна одинаковы в WHello и WinApp, но имеется одно существенное отличие. Строки 285, 286 присваивают значение ресурса меню, идентифицируемого ID_MENU, переменной clsLpszMenuName структуры оконного класса. По английски clsLpsz является аббревиатурой "window class long pointer to a zero-terminated string" (длинный указатель на ограниченную нулем цепочку оконного класса). Для создания в окне раскрывающегося меню просто вставьте команды меню в файл описания ресурсов и присвойте идентификатор ресурса меню структуре оконного класса.

Замечание

Имеются и другие способы активирования раскрывающихся меню в программах Windows, но присвоение идентификатора ресурса оконному классу — самый простой Хорошие учебники по Windows могут содержать альтернативные методы, которые, однако, выходят за рамки этой ознакомительной главы.

Цикл сообщения в WinApp (строки 219–236) аналогичен такому же коду в WHello. Эти две программы, однако, начинают различаться, начиная со строки 313 WINAPP.ASM, в начале основной оконной процедуры — WinProc.

Как и в WHello, WndProc обрабатывает сообщения, предназначенные для основного окна программы. В данном случае процедура использует регистр di для хранения дескриптора этого оконного элемента (строка 317) и si — для хранения значения сообщения (строка 318). Строки 319–327 просматривают значение сообщения в si и осуществляют переход к соответствующему участку оконной процедуры, обрабатывающему сообщение, либо к первоначальному обработчику. WndProc обрабатывает четыре сообщения:

- WM_DESTROY показывает, что элемент окна должен быть разрушен. Поскольку это основное окно программы, данное сообщение прерывает программу, вызывая PostQuitMessage. Код этого сообщения одинаков в WHello и WinApp.
- WM_CLOSE показывает, что пользователь предпринял попытку закрыть окно — нажав <Alt+F4>, например Программа WinApp использует это сообщение для отображения подсказки, подтверждающей закрытие окна, давая возможность пользователю продолжить программу, вместо того чтобы ее закончить. (Такой подход может предотвратить потерю информации — например, можно предложить пользователю сохранить и отредактировать файл, прежде чем программа будет завершена.)
- WM_PAINT показывает, что содержимое окна требует рисования. Windows генерируют это сообщение в ответ на множество событий. Например, максимизация окна на весь экран генерирует сообщение WM_PAINT при открытии окна путем перемещения в сторону другого окна.
- WM_COMMAND показывает, что пользователь выбрал команду из раскрывающегося меню окна. В WinApp эти команды программы обрабатываются дополнительной подпрограммой — но об этом позднее.

Для полного понимания приведенной программы необходимо просмотреть и выполнить трассировку кода для каждого из этих сообщений. Инструкции, следующие за локальной меткой @@WMCLOSE, обрабатывают сообщение WM_CLOSE; код, помеченный @@WM_PAINT, обрабатывает WM_PAINT и т.д. Инструкции, обозначенные @@RETURNZERO, устанавливают ax:dx в нуль перед тем, как осуществить возврат из оконной процедуры. Это является корректным завершением для большинства сообщений, но вам необходимо сверить информацию о действиях, требуемых для каждого сообщения, в справочнике по Windows API.

Для отображения окна сообщений, запрашивающего вас, нужно ли прерывать программу, WndProc в строке 345 вызывает функцию MessageBoks Windows. Этой функции нужно все множество аргументов, как показано в предыдущих инструкциях в строках 339–344. Результатом является диалоговое окно сообщения, которое действует самостоятельно до тех пор, пока не закрывается одной из своих кнопок (рис. 15.3).

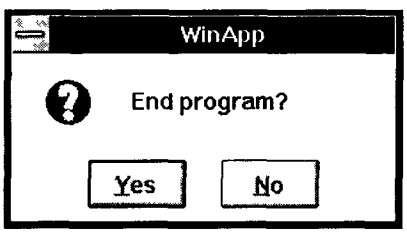


Рис. 15.3. Диалоговое окно WinApp, спрашивающее пользователя, заканчивать ли программу

Когда оконная процедура получает сообщение WM_PAINT, она должна отреагировать, дополняя содержимое окна. Первым шагом этого процесса является вызов функции API BeginPaint, как демонстрируется в строках 353-356. Этой функции вы должны передать адрес рисующей структуры, которая в данном случае является глобальной переменной в сегменте данных. (Этот участок не может вызываться рекурсивно, поэтому глобальная переменная допустима.) BeginPaint заполняет рисующую структуру ps значениями, которые можно использовать для отображения объектов окна, вызывая функцию интерфейса графического устройства — Graphics Device Interface (GDI).

Имеется гораздо больше возможностей программирования GDI, чем я в состоянии отразить в одной главе, поэтому я привожу только один пример функции, вызываемой здесь. Однако вы можете вызвать большинство функций GDI, используя аналогичную методику, а во многих учебниках по Windows вы найдете дополнительные примеры.

Обычно первым аргументом, передаваемым функции GDI, является *дескриптор контекста устройства* — device context handle (HDC), получаемый от BeginPaint. Дескриптор идентифицирует *контекст*, к которому относится вывод. В большинстве случаев этим контекстом является видеодисплей, отображающий окно, но он может также представлять и другие выходные устройства, как, например, графопостроитель или принтер.

BeginPaint сохраняет HDC в рисующей структуре, но для уверенности функция возвращает этот же дескриптор в регистр ax. Строка 357 извлекает пользу из этого факта, присваивая ax регистру si. Поскольку оконная процедура уже выбрала сообщение для обработки, si снова доступно для использования — оптимизация, которую едва ли можно выполнить в каком-нибудь языке высокого уровня.

Большинство функций GDI требует таких аргументов, как значения координат, цепочки и другие значения Строки 362–365, например, выталкивают в стек численные координаты прямоугольника. И наконец, строка 366 вызывает функцию Rectangle, отображающую прямоугольник, показанный на рис. 15.2.

Заключительным шагом при обработке сообщения WM_PAINT является вызов EndPaint, соответствующей предшествующему обращению к BeginPaint. Передайте дескриптор окна и адрес инициализированной рисующей структуры, а затем возвратите ax:dx, равные нулю (см. строки 370–374).

Строки 378–383 обрабатывают команды меню программы. Здесь мог быть помещен код каждой программы, но для сохранения модульной структуры и для упрощения настройки я предпочитаю вызывать локальные подпрограммы, такие как WinAppCommands. WndProc передает этой подпрограмме дескриптор окна, значения аргументов типа слова и длинного типа, выталкивая эти значения в стек. (В следующем разделе я поясню, что делает WinAppCommands).

И наконец, строки 388–393 вызывают первоначальный дескриптор сообщения, DefWindowProc, для сообщений, не обработанных оконной процедурой программы. Это дает уверенность в том, что обычные операции, такие как изменение размеров окна, перемещения, раскрытия меню и т.п., выполняются нормально.

Меню

Раскрывающееся меню WinApp содержит только две команды — *File|Exit* (завершающую программу) и *Help|About* (отображающую информационное диалоговое окно). Будучи относительно простой, программа демонстрирует основные шаги по выполнению команд меню гораздо более сложных, чем эти.

Прежде всего, создайте меню в виде описателя ресурса, как это делается в строках 6–16 WINAPP.RC (обратитесь снова к тексту программы 15.6). Большинство разрабатывающих систем, таких как Resource Workshop фирмы Borland, снабжены редактором меню, осуществляющим написание операторов описателя ресурсов, но можно набрать инструкции в файле .RC и вручную, как я поступил в данном случае.

После создания меню присвойте его идентификатор ресурса оконному классу (строки 285, 286 в WINAPP.ASM, программа 15.4). Windows затем отобразит меню у верхней границы окна и передаст сообщение WM_COMMAND оконной процедуре, когда пользователь выберет команду меню.

Параметр wp этого сообщения идентифицирует команду, которая была выбрана, и устанавливается в значение из ресурса меню, соответствующее этой команде. Например, когда вы выбираете команду *File\Exit* WinApp, wp устанавливается в значение идентификатора ресурса для этой команды — CM_DEMO_EXIT (см. строку 13 в файлах WINAPP.RH и WINAPPRI, а также строку 10 в файле описания ресурсов WINAPP.RC).

В соответствии с WM_COMMAND оконная процедура WinApp вызывает WinAppCommands (строки 468–499). В этой подпрограмме в строках 472–478 осуществляется копирование дескриптора окна hWnd в регистр di и идентификатора команды в si. После этих шагов строки 474–478 осуществляют переход к соответствующей локальной метке для выполнения конкретной команды. Например, когда вы выбираете *File|Exit*, строка 475 осуществляет переход к локальной метке @@CMDEMOEXIT.

Инструкции, помеченные этой меткой (строки 482–489), демонстрируют, как пересылать сообщение окну посредством вызова SendMessage из Windows API. В соответствии с командой "выйти из программы" WinApp пересылает окну сообщение WM_CLOSE. Это приводит к тому, что Windows вызывает оконную процедуру, отображающую, как вы, вероятно, помните, окно сообщения, которое просит подтвердить решение завершить программу.

WinAppCommands обрабатывает вторую команду меню программы, вызывая другую подпрограмму — HelpAbout (строки 494, 495). Эта подпрограмма отображает информационный диалог программы, рассматриваемый в следующем разделе.

Замечание

При использовании WinApp в качестве отправной точки для разработки своих собственных программ вы можете разрешить описанный переход в строке 496 и добавить после этой инструкции дополнительные командные инструкции.

Диалоговые окна

Диалоговое окно — это специализированное окно, которое обычно содержит набор элементов управления для выбора программных опций, отображения информации, вставки данных и выполнения других интерактивных операций. Большинство разрабатывающих систем Windows поставляется с диалоговым редактором, наподобие редактора в Resource Workshop фирмы Borland, который вы можете использовать для создания диалоговых окон.

Выходом диалогового редактора является описатель ресурса команд, аналогичный тому, что приведен в строках 18–29 WINAPP.RC, программа 15.6. Мало кто из программистов станет набирать эти запутанные инструкции вручную, поэтому я не стану пояснять их здесь. Когда доходит до диалога, лучше использовать интерактивный редактор, который может создавать необходимые операторы описателя ресурсов.

Как и в случае со всеми ресурсами, диалог снабжен уникальным идентификатором, в данном примере — это символ ID_ABOUT. Подпрограмма HelpAbout в WINAPP.ASM (см. строки 511–534) использует этот идентификатор для выполнения диалога — процесс, который требует нескольких критичных шагов.

Первым из этих шагов является создание *процедурной инстанции*, условно названной thunk (вздох), которая инициализирует регистр сегмента данных, а затем вызывает программный код таким образом, чтобы программа могла найти глобальные данные. Результирующее косвенное обращение к подпрограмме вызывает "вздыхающий звук", который могут слышать только программисты, видящие эльфов.

Процедурная инстанция для диалогового окна — это подпрограмма, обрабатывающая сообщения диалога. Она подобна оконной процедуре (но это не совсем одно и то же). В WinApp подпрограмма DlgProc (строки 415–454) обрабатывает сообщения для диалогового окна about, показанного на рис. 15.4.

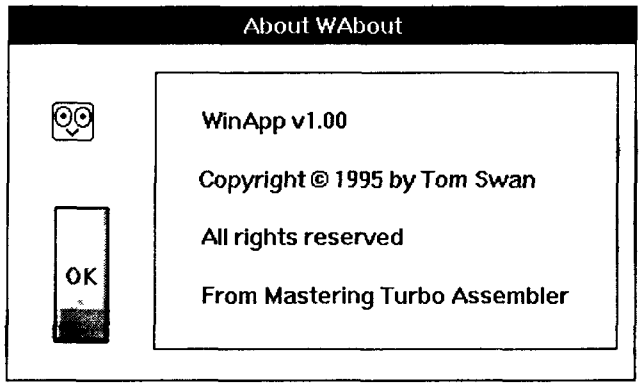


Рис. 15.4. Диалоговое окно About

После создания диалоговой процедурной инстанции подпрограмма HelpAbout вызывает функцию DialogBox Windows (строки 522–528). За исключением идентификатора ресурса ID_ABOUT в строке 524 и имени диалоговой процедуры в строках 515, 516, вы можете использовать код в HelpAbout для активизации большинства диалоговых окон.

После возврата из DialogBox нужно освободить процедурную инстанцию, которую вы создали посредством MakeProcInstance. Строки 530–532 показывают, как правильно удовлетворить это требование.

Диалоговая процедура

Действия диалогового окна в диалоговой процедуре программируются аналогично оконным действиям в оконной процедуре. Как и оконная процедура, диалоговая процедура получает сообщения, предназначенные для диалогового окна.

Однако диалоговая процедура значительно отличается от оконной. Оба вида подпрограмм объявляются с использованием директивы PROC (строка 415), и при этом специфицируется одинаковое число аргументов того же типа (строка 416). Но диалоговая процедура возвращает в ax булевы значения "истинно" или "ложно", а не 32-битовые значения в ax:dx, как это делает оконная процедура. Кроме того, диалоговая процедура не вызывает первоначальный дескриптор сообщения для необработанных сообщений.

Диалоговая процедура должна включать в себя программу по крайней мере для сообщений, показанных в приведенной распечатке (обратитесь к строкам 415–454). Windows пересылает первое сообщение — WM_INITDIALOG прямо перед тем, как диалог становится активным. Вам необходимо воспользоваться этой возможностью для инициализации любых переменных, требующихся диалогу. В данном случае диалог WinApp не имеет переменных и, как показывает строка 429, диалоговая процедура просто осуществляет переход к тому участку подпрограммы, который возвращает значение "истинно". Тем не менее, даже если не выполняется никаких инициализаций, диалоговая процедура должна возвращать "истинно" для сообщения WM_INITDIALOG. Если по какой-то причине процедура не может успешно инициализировать свои переменные, она должна вернуть значение "ложно" для завершения активации диалога.

Второе обязательное сообщение — WM_COMMAND — то же самое, что и для команд меню. В данном случае, однако, сообщение является результатом выбора диалоговых кнопок, команды диалоговой системы меню, если таковые имеются, либо нажатия таких клавиш, как <Enter> или <Esc>.

Как и в случае команд меню, параметр wp идентифицирует то, какая кнопка или команда была выбрана. Одной из этих команд может быть IDOK, указывающая на то, что пользователь настроен закрыть диалог выбором кнопки "OK" либо нажатием <Enter>, если эта кнопка является определенной первоначально.

Другие диалоги могут обладать иными кнопками и командами, но все диалоговые процедуры должны иметь программы для IDOK.

Надлежащим ответом на эту команду в данном случае является вызов функции Windows EndDialog, как показано в строках 441–444. Этой функции передается дескриптор диалогового окна и любые значения, подлежащие возврату процедуре, вызвавшей диалог. Функция DialogBox возвращает это значение в ax (устанавливает его в нуль в приведенной программе в строке 443). Вы можете использовать это значение, если желаете, хотя WinApp его игнорирует.

Замечание

Для добавления команд в диалоговое окно активируйте инструкцию jmp в строке 445 и после нее вставьте свой программный фрагмент.

Резюме

Написание программ для Windows на ассемблере требует значительной подготовки и усилий, но дает ряд преимуществ. Используя язык ассемблера, вы получаете полный контроль над запускающим и завершающим кодом программы и можете наилучшим образом использовать регистры и память. Кроме того, можно избавиться от лишнего багажа, присоединяемого к программам для Windows компиляторами языков высокого уровня.

Эта глава знакомит с техникой программирования для Windows для режима Ideal Turbo Assembler, но она не является всеобъемлющим учебником по программированию для Windows. Чтобы выйти за пределы материала, изложенного в этой главе, вам потребуется справочник по Windows API, а также файлы и утилиты, поставляемые с большинством C, C++ и другими разрабатывающими системами высокого уровня. Turbo Assembler не обеспечивает всех файлов и утилит, необходимых для написания прикладных программ для Windows.

Для упрощения программирования используйте директиву MODEL совместно с опциями WINDOWS и PASCAL. Эти опции автоматически добавляют к подпрограммам начальный и завершающий коды. Объявляйте EXTRN все функции Windows API, вызываемые вашей программой.

Вы можете объявлять инициализированные (DATASEG) и неинициализированные (UDATASEG) переменные. Вы должны резервировать первые 16 байт инициализированных данных для собственного использования Windows.

Оконный класс является структурой, описывающей характеристики окна. Первая инстанция прикладной программы для Windows должна регистрировать оконный класс. Любая программная инстанция должна создать по крайней мере один элемент окна этого класса для использования в качестве основного окна.

Оконный класс, кроме того, специфицирует оконную процедуру, которой Windows передает сообщения. Написание оконных процедур часто называют программированием, управляемым событиями. Типичная оконная процедура включает в себя программирование для многих разнообразных сообщений, генерируемых Windows в ответ на события Программа может также посылать свои собственные сообщения. Необработанные сообщения должны передаваться первоначальному обработчику сообщений, обычно — DefWindowProc.

Ресурсы — это двоичные данные, которые компоновщик присоединяет к исполняемому кодовому файлу программы. Примеры ресурсов в этой главе включают в себя меню, пиктограммы и диалоговые окна. Ресурсы обычно создаются в файле описания ресурсов (либо посредством использования интерактивного редактора). Компилятор ресурсов преобразует описание в двоичный файл, имеющий расширение имени .RES. Компоновщик подсоединяет этот образ к конечной программе. Turbo Assembler не обеспечивает компилятором или редактором ресурсов.

Диалоговая процедура программирует действия для диалогового окна. Как и оконная процедура, диалоговая процедура отвечает на сообщения, пересылаемые диалоговому окну. Диалоговая процедура, однако, возвращает булевы значения "истинно" или "ложно" и не вызывает первоначального обработчика сообщений для необработанных сообщений.

Упражнения

15.1. Функция Ellipse GDI Windows рисует овал или окружность. Приведите объявления, дающие возможность ассемблерной программе вызвать эту функцию.

15.2. Функция GetWindowsDirectory получает путь каталога Windows, обычно C:\WINDOWS для большинства компьютеров. Параметр lpszSysPath адресует буфер цепочки, в котором функция хранит свои результаты. Параметр cbSysPath равен размеру буфера цепочки и должен быть по меньшей мере 144. Покажите как вызвать эту функцию в ассемблерной программе, чтобы получить путь Windows. Покажите, также все необходимые объявления данных. GetWindowsDirectory определяется в C следующим образом:

```
UINT GetWindowsDirectory(LPSTR lpszSysPath, UINT cbSysPath),
```

15.3. Модифицируйте WHello, чтобы его окно раскрывалось на весь экран при первом запуске программы.

15.4. Модифицируйте WHello, чтобы раздавался писк при завершении программы пользователем. Вы можете использовать следующие инструкции — ваша задача определить, где их вставить:

```
push 0
call MessageBeep
```

15.1. Модифицируйте WinApp для сохранения ее дескриптора основного окна в глобальной переменной, названной wMainHnd.

15.2. *Усложненное* Модифицируйте WinApp для отображения ее окна about при первом запуске программы. Подсказка: одним из возможных ответов является использование модификации из упражнения 15.5.

Задания

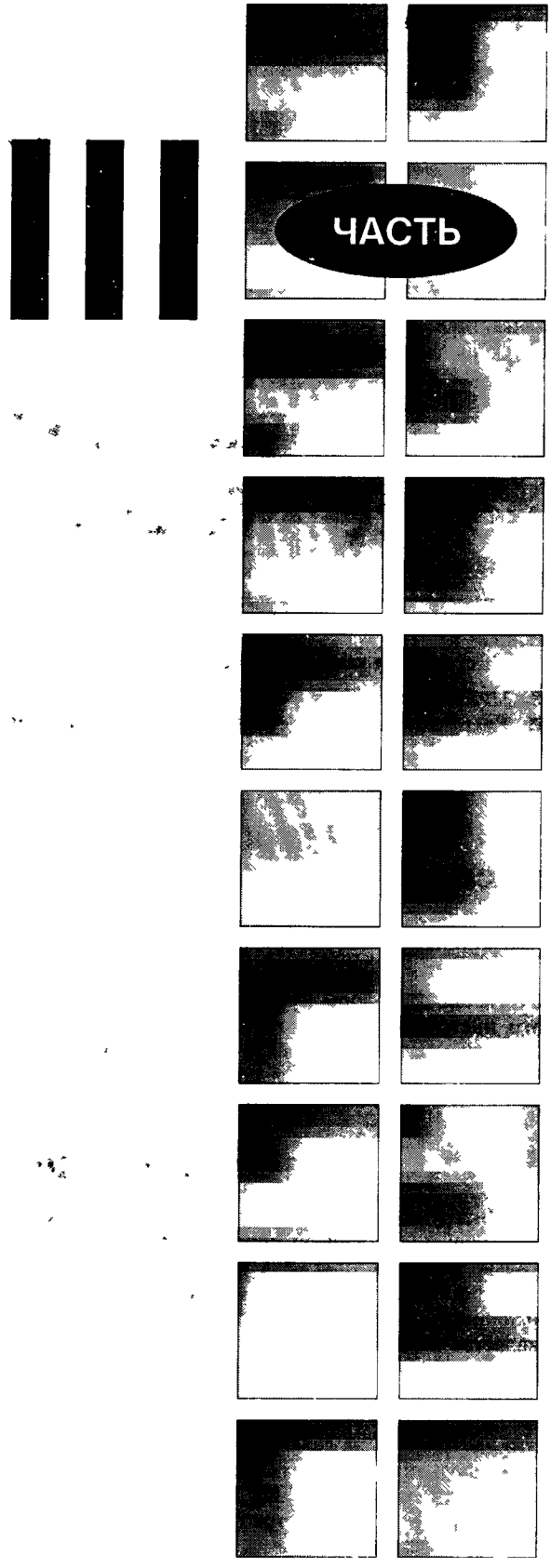
15.1. Демонстрационные программы WHello и WinApp в этой главе содержат несколько дублируемых участков — цикл сообщения AppRun и запускающие инструкции, например, идентичны в обеих программах. Выделите эти участки в модули и ассемблируйте их раздельно, чтобы создать кодовые файлы .OBJ для компоновки с прикладными программами для Windows.

15.2. Установите точку прерывания в TDW для обращения к DefWindowProc в строке 392 в WINAPP.ASM и просмотрите значения сообщений в si, чтобы выяснить характер стандартных сообщений, обрабатываемых Windows. Постарайтесь поставить в соответствие идентификаторы сообщений в WINDOWS.INC Еще лучше, запишите эти сообщения в текстовый файл — вы почти создадите свою собственную утилиту трассировки сообщений!

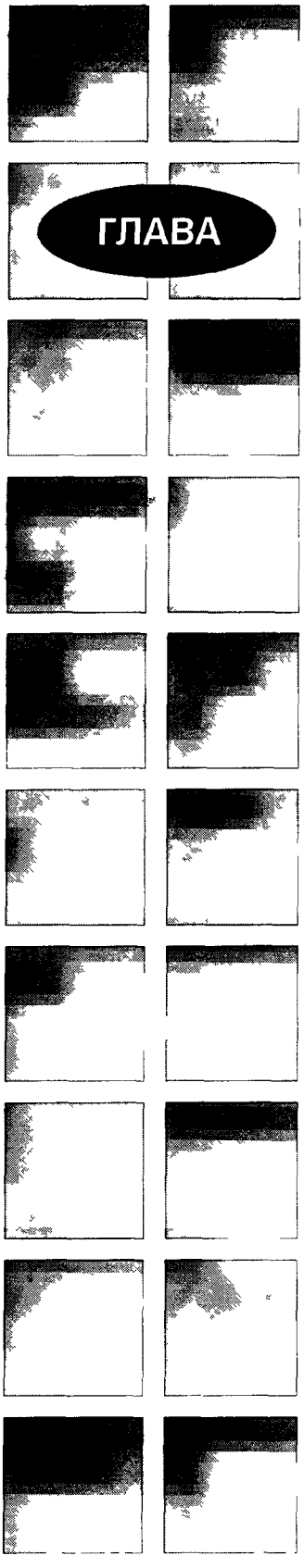
15.3. Напишите собственную программу stub для отображения специального сообщения при попытке пользователей исполнить вашу программу для Windows из приглашения DOS. Ваша stub могла бы отображать замечания по поводу авторских прав, а также давать инструкции относительно того, как запустить Windows и прикладную программу.

15.4. Модифицируйте WinApp для разбора цепочки командной строки опции. Например, для расширения окна программы на весь экран при запуске пользователи могли бы запустить WinApp посредством команды Program Manager *File|Run*, введя winapp -x. Чтобы проверить свой код, напишите программу, отображающую цепочку опций, передаваемых программе.

15.5. Модифицируйте диалоговое окно about WinApp, чтобы оно закрывалось автоматически через заданный промежуток времени (скажем, через три секунды). *Подсказка* используйте функцию Windows GetTickCount в качестве таймера — посмотрите ее спецификацию в справочнике по API. Одним из способов закрытия диалогового окна является пересылка ему сообщения WM_COMMAND с параметром типа слова, равным IDOK, что вызывает выбор кнопки OK диалога.



Справочник



ГЛАВА 16

Справочное руководство по ассемблеру

- О справочнике
- Инструкции защищенного режима
- Обращение к источнику
- Распределение времени для инструкций и двоичное кодирование
- Как пользоваться справочником

О справочнике

В этой главе представлены все мнемонические коды, которые используются в незащищенном режиме процессоров 8086, 8088, 80286, 80386, 80486 и Pentium, приведенные в алфавитном порядке, показываются действующие флаги; приводится синтаксис всех форм инструкций и даются примеры и описания, поясняющие, как эти инструкции работают. Представленный здесь материал суммирует информацию, которая содержится в предшествующих разделах. Прочтите последующие разделы, чтобы научиться пользоваться материалом данной главой, и для ознакомления со значениями различных терминов.

Инструкции защищенного режима

Инструкции защищенного режима процессоров 80286, 80386, 80486 и Pentium не включены в данный справочник. Эти инструкции, как правило, используются только для написания кода операционной системы, что требует искусного соединения процессов, внешне протекающих одновременно, но в действительности выполняющихся по очереди. Основной целью программирования защищенного режима является переключения между такими процессами со скоростью, достаточной для того, чтобы возникала иллюзия одновременного исполнения.

Возможно, кому-то не понравится отсутствие в этом справочнике инструкций защищенного режима, однако после долгих размышлений я решил, что перечислять эти инструкции, не давая одновременно сопутствующего материала, требуемого для написания программного обеспечения многозадачных операционных систем, было бы пустой тратой времени. Для прикладного же программирования инструкции защище нного режима не требуются. Однако эта книга была бы неполной, если бы в

ней не упомянуть инструкцию set защищенного режима (табл. 16.1). Для более подробной информации по использованию этих инструкций и по написанию многозадачных операционных систем обратитесь к соответствующей литературе.

Замечание

Специальные инструкции незащищенного режима 80286, 80386, 80486 и Pentium, такие как bound, enter, leave, и обычные инструкции set раскрываются здесь подробно вместе с описаниями синтаксиса для расширенных 32-битовых регистров. Кроме того, инструкции, ограничивающиеся отдельными процессорами, — помечены.

Обращение к источнику

Для подтверждения форматов инструкций и установок флагов, приведенных в этой главе, были использованы, по крайней мере, пять источников. В случаях, когда любые из этих справочников не во всем совпадали один с другим (что случалось достаточно часто), напечатанная здесь документация подтверждалась экспериментальным путем. Перекрестная проверка выявила удивительное число ошибок в справочниках фирм Intel и Microsoft. Естественно, все эти ошибки здесь исправлены.

Таблица 16.1. **Инструкции защищенного режима**

Мнемоника	Описание
arpl	Adjust RPL Field of Selector (Настройка поля RPL селектора)
clts	Clear Task-Switched Flag in CRO (Очистка флага переключения задач в CRO)
lar	Load Access Rights Byte (Загрузка байта прав доступа)
lgdt	Load Global Descriptor Table Register (Загрузка регистра таблицы глобального описателя)
lidt	Load Interrupt Descriptor Table Register (Загрузка регистра таблицы описателя прерываний)
lldt	Load Local Descriptor Table Register (Загрузка регистра таблицы локального описателя)
lmsw	Load Mashine Status Word (Загрузка слова статуса машины)
lsl	Load Segment Limit (Загрузка ограничителя сегмента)
ltr	Load Task Register (Загрузка регистра задачи)
mov (386)	Move To/From Special Registers* (Перемещение в/из специальных регистров)
sgdt	Store Global Descriptor Table Register (Сохранение регистра таблицы глобального описателя)
sidt	Store Interrupt Descriptor Table Register (Сохранение регистра таблицы описателя прерываний)
sldt	Store Local Descriptor Table (Сохранение таблицы локального описателя)
smsw	Store Mashine Status Word (Сохранение слова статуса машины)
str	Store Task Register (Сохранение регистра задачи)
verr	Verify Segment for Reading (Проверка регистра для чтения)
verw	Verify Register for Writing (Проверка регистра для записи)

* Только для 80386, 80486 и Pentium

Распределение времени для инструкций и двоичное кодирование

Поскольку эта книга является, прежде всего, руководством по программированию прикладных программ на ассемблере, распределение времени для инструкций и двоичное кодирование машинных кодов, генерируемых ассемблером, в ней не приводятся. Если вам они все же потребуются, можно найти эти данные в справочниках Intel, перечисленных в Библиографии.

Значения распределения времени, слепо копируемые многими справочниками, которые, как я подозреваю, в действительности используются очень немногими программистами, опущены по вполне веским причинам. Формулы, применяемые для теоретического расчета распределения времени для конкретных инструкций, на практике оказываются неточными. Такие факторы, как зависящее от конкретного чипа кэширование инструкций, которое предварительно загружает определенную часть машинного кода для более быстрого выполнения, плюс наличие множества сигналов прерываний и состояний ожидания памяти в реальных компьютерных системах, с большой долей вероятности превращают в мартышкин труд даже наиболее тщательно сконструированные формулы для расчета распределения времени. Секундомер и хороший профайлер принесут вам больше пользы, чем часы, проведенные за расчетом распределения времени для инструкций, выполняющих циклические операции. Ваши программы будут выполняться с наибольшей скоростью, если вы последуете нескольким советам при выборе между различными форматами инструкций:

- Инструкции, которые обращаются к аккумулятору — al, ax или eax (относится только к процессорам 80386 и более поздним) — выполняются быстрее других. (Эти инструкции могут также занимать меньшее число байтов машинного кода.) По этой причине любые подобные инструкции приводятся первыми в разделах *синтаксис/пример* данной главы. Например, посмотрите первые две строки синтаксиса для adc и первые две строки синтаксических форм для процессоров 80386/486.
- Инструкции, которые используют только регистры для всех операндов, выполняются быстрее тех, что обращаются к данным, хранимым в памяти. В особенности это справедливо для инструкций 8086, обращающихся к данным, размещенных по нечетным адресам, поскольку 8086 может загружать данные из четных адресов несколько быстрее, чем из нечетных. Другими словами, если у вас имеется возможность выбора между регистровыми переменными и переменными памяти, используйте регистровые — ваша программа сможет выполняться быстрее.
- Арифметические инструкции imul, mul и div – удручающе медленные. Всегда используйте сдвиг и другие логические операции для умножения и деления с основанием 2 либо, если возможно, — математический сопроцессор.

Форматы двоичного машинного кода для инструкций не приведены. По сути, сложные битовые форматы и двоичные коды операций для отдельных инструкций редко упоминаются где-либо в этой книге. В конечном счете, одной из причин использования ассемблера является желание избежать необходимости заботиться о подобных подробностях. В тех редких случаях, когда вам потребуется знать каждый бит, сгенерированный для конкретной инструкции, вы достаточно легко сможете написать тестовую программу и просмотреть ассемблерный код с помощью Turbo Debugger.

Это коротко о том, чего здесь нет. Теперь рассмотрим то, что содержит справочник.

Как пользоваться справочником

Справочник описывает в отдельности все мнемоники, за исключением обычных инструкций jump (перехода) и set (установки) (только для 80386/486). Для большей наглядности описания оформлены в виде таблиц. Несколько мнемоник, генерирующих одинаковые машинные коды, такие как cmps, cmpsb, cmpsw и cmpsd, приводятся вместе, но только тогда, когда это не нарушает алфавитного порядка справочника. Например, sal и shl приводятся отдельно, несмотря на то, что эти две мнемоники представляют идентичные инструкции.

Данные для каждой мнемоники разделены на части, каждая из которых имеет свое назначение.

- *Заголовок* — приводит мнемонику, имя, процессор, для которого действительна мнемоника, и показывает влияние на установку флагов.
- *Назначение* — дает краткое описание инструкции.
- *Синтаксис/Пример* — показывает различные формы, которые может принимать инструкция, и перечисляет допустимые используемые регистры и операнды памяти. Этот раздел приводит также типичные примеры программ для каждой формы инструкции. Если инструкция допустима для нескольких процессоров, каждая синтаксическая форма для 80286, 80386 и более поздних процессоров приводится отдельно.
- *Образец кода* — помещает инструкцию в краткий пример программы, давая практический пример того, как данная инструкция может быть использована в типичной программе.

- *Описание* — досконально поясняет действие инструкции. Кроме того, здесь описывается любое необычное использование флагов и назначение регистров.
- *Смотри также* — осуществляет ссылку к другим инструкциям, каким-либо образом имеющим отношение к данной.

Еще немного о заголовках

В качестве примера заголовка справочника на рис. 16.1 приведен заголовок для инструкции and. Мнемоника and приведена строчными буквами и показывает вам в точности, как следует писать инструкцию в программе. Имя инструкции печатается прямо напротив мнемоники. Под этими двумя символами находится список процессоров и флагов. Процессор 80186, не использующийся в компьютерах, совместимых с IBM PC, здесь не приводится. Функционально эквивалентные процессоры 8086 и 8088 приводятся вместе как 8086/88. Колонки, помеченные 80386/486, относятся к 80386, 80486, Pentium и (скрестите пальцы) к будущим совместимым процессорам. Это пересмотренное издание дополнено шестью новыми инструкциями — bswap, cmpxchg, invd, invlpg, wbinvd и xadd — требующими процессор 80486 или более поздний. Звездочка под номером процессора показывает, какой процессор поддерживает данную инструкцию. В настоящем примере заголовок показывает, что and доступна для всех четырех процессоров.

Флаги перечисляются справа от номеров процессоров (см. рис. 16.1). Под каждым флагом приведены один или больше символов, показывающих, как данная инструкция воздействует на биты флага. Цифра 0 или 1 показывает, что инструкция переустанавливает или устанавливает флаг в это значение Прочерк (-) показывает, что инструкция не изменяет установки этого флага. Звездочка говорит о том, что значение флага является объектом изменения в соответствии с правилами, приведенными в табл. 16.2. Когда применяются другие правила и условия или когда приводится более одного символа (см. sal, например), установки флагов обсуждаются в *описании* инструкции.

and **Логическое И**

Процессор·	8086/88	80286	80386/486	Флаги·	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		0	-	-	-	*	*	u	*	0

Рис. 16.1. Пример заголовка справочника

Еще немного о разделах Синтаксис/Пример

В табл. 16.3 приводятся символы, используемые в разделах *Синтаксис/Пример*. Совместно с этой таблицей справочные материалы по синтаксису точно указывают вам, какие формы каждой инструкции допустимы. Например, одной из строк синтаксиса и примера для shl является следующая:

```
shl regW, memW, cl        shl [aword + bx], cl
```

Таблица 16.2 **Стандартное использование флагов**

Флаг	Имя	Установка в 1, если..., иначе переустановка в 0
of	Overflow	Положительное значение слишком велико или (переполнение) отрицательное — слишком мало
sf	Sign (знак)	Старший бит значения = 1
zf	Zero (нуль)	Все биты результата = 0
af	Auxiliary (дополнительный)	Имел место перенос или заимствование до 4 младших битов al
pf	Parity (паритет)	Восемь младших битов результата содержат четное число единиц (четное равенство)
cf	Carry (перенос)	Имел место перенос или заимствование всех битов результата

Таблица 16.3 **Символы, используемые в Справочнике**

Символы	Значение
|	ИЛИ
&	И
[]	Объекты в скобках являются необязательными
farTarget	Адресация ссылки в чужой сегмент
nearTarget	Адресация ссылки внутри текущего сегмента

Таблица 16.3. **Символы, используемые в Справочнике**

Символы	Значение
shortTarget	Адресация ссылки в пределах от —128 до 127 байт
imm6	6-битовое значение (только для инструкции esc)
immB	Любое непосредственное 8-битовое значение
immW	Любое непосредственное 16-битовое значение
immDW	Любое непосредственное 32-битовое значение
memB	Любая ссылка к 8-битовому байту памяти
memW	Любая ссылка к 16-битовому слову памяти
memDW	Любая ссылка к двойному слову памяти
memFW	Любая ссылка к 48-битовому слову памяти
memQW	Любая ссылка к 64-битовому четверному слову памяти
memALL	Любая B, W, DW, FW или QW ссылка к памяти
regB	Любой основной 8-битовый регистр байта
regW	Любой основной регистр 16-битового слова
regDW	Любой основной регистр 32-битового двойного слова процессора 80386/486
no operands	Не требует никаких операндов

Табл. 16.3 показывает, что эта форма shl требует двух операндов: регистра основного назначения для слова (16-битового) или ссылки к слову памяти и регистра cl. Пример справа от синтаксиса показывает, как такая форма инструкции могла бы выглядеть в программе. Помните, что пример показывает только одну из многих возможных комбинаций регистров и ссылок к памяти.

Замечание

Если не указывается иначе, ссылки к памяти включают в себя все режимы адресации, описанные в главе 5.

Еще немного о примерах и образцах

Все примеры и разделы с образцами кода были ассемблированы и проверены. Вы можете быть уверены, что код, приведенный здесь, до последней запятой представляет инструкции, в том виде как они могли бы выглядеть в программе.

Замечание

Если вы запускаете любой из примеров, будьте внимательны с инструкциями, производящими операции чтения и записи в аппаратные порты. Из-за системной зависимости таких инструкций, как in, out и outs, примеры этих мнемоник могут ассемблироваться, но, возможно, они не выполняют никакой полезной функции. Они могут привести даже к разрушению системы. Такие примеры помечены особым комментарием, предупреждающим вас о том, чтобы не запускать код.

aaa **Подгонка к ASCII после сложения**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	u	u	*	u	*

Назначение Подгоняет численную сумму двух распакованных BCD-цифр к распакованному BCD-формату, который легко преобразуется в ASCII

Синтаксис/Пример `aaa no operands` `aaa`

Образец кода

```
mov ah, 07              ; Первое число = 07
mov al, 08              ; Второе число = 08
add al, al              ; Сумма в al = 0Fh (15 дес.)
sub ah, ah              ; Очистка до 00
aaa                     ; Подгонка: ah = 01, al = 05
or ax, 3030h            ; Преобразование цифр в ASCII
```

Описание После сложения двух распакованных BCD-чисел и сохранения 8-битового результата в al aaa преобразует al обратно в распакованный формат BCD. Если предыдущая инструкция add сгенерировала перенос или если al больше 9, тогда ah увеличивается и оба — и cf, и af

устанавливаются в 1; в противном случае af и cf устанавливаются в 0. Четыре старших бита (верхняя половина) al всегда обнуляются. Как показывает пример, после выполнения aaa вы можете посредством инструкции OR ah и al с 030h преобразовать BCD-результат в ASCII.

См. также aad, aam, aas, daa, das

aad — **Подгонка к ASCII перед делением**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	*	*	u	*	u

Назначение Преобразование двух распакованных BCD-чисел в ax в двоичное значение

Синтаксис/Пример aad no operands — aad

Образец кода

```
mov ah, 7              ; Установка ah в 7ASCII
mov al, 6              ; Установка al в 6ASCII
and ax,0F0Fh           ; Преобразование ASCII в BCD (ax = 0706h)
aad                    ; Преобразование в двоичное (ax = 004Ch)
```

Описание Присваивается два распакованных значения BCD — ah (старшая значащая цифра) и al (младшая значащая цифра), затем выполняется aad для преобразования чисел в 16-битовое двоичное значение в ax. Несмотря на имя инструкции, aad может быть использована в любое время — она не должна предшествовать делению. Наибольшее возможное значение, которое может преобразовывать aad — 0909, равное шестнадцатеричному 063h либо десятичному 99. Следовательно, после использования aad применительно к распакованным BCD значениям в диапазоне от 0000 до 0909 регистр ah всегда равен 0.

См. также aaa, aam, aas, daa, das

aam — **Подгонка к ASCII после умножения**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	*	*	u	*	u

Назначение Преобразует 16-битовые двоичные значения от 0 до 99 десятичных в ax в распакованные BCD-числа, которые легко преобразуются в ASCII

Синтаксис/Пример aam no operands — aam

Образец кода

```
mov ax, 04Ch           ; Установка ax в 76 дес.
aam                    ; Преобразование в BCD
                       ;  (ax = 0706h)
or ax, 3030h           ; Преобразование ax в
                       ;  ASCII (ax = 3736h)
```

Описание aam используется для преобразования значения в ax, меньшего либо равного шестнадцатеричному 063h, из двоичного в распакованный формат BCD со старшей значащей цифрой, помещаемой в ah, и с младшей значащей цифрой в al. Эта операция является обратной aad. Несмотря на имя, инструкции не обязательно должно предшествовать умножение.

См. также aaa, aad, aas, daa, das

aas — **Подгонка к ASCII после вычитания**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	u	u	*	u	*

Назначение Подгоняет численную разность двух BCD распакованных чисел к распакованному формату BCD, который легко преобразуется в ASCII

Синтаксис/Пример aas no operands — aas

Образец кода

```
mov ah, 01             ; Установка ah в BCD 01
mov al, 04             ; Установка al в BCD 04
mov bl, 07             ; Установка bl в BCD 07
sub al, bl             ; al <- al - bl (14-7)
aas                    ; Подгонка к BCD (ax =0007)
or ax, 3030h           ; Преобразование в ASCII
                       ;  (AX = 3037H)
```

Описание Проводит вычитание двух BCD-чисел, помещая результат в al, и выполняет aas для преобразования численной разности в формат BCD, который может быть затем преобразован в ASCII. Если предшествующая операция sub требует заимствования, тогда aas также вычитает 1 из ah и устанавливает af и cf в 1; в противном случае ah остается неизменным и два флага устанавливаются в 0. В примере из 0104 (десятичное значение 14 в распакованном формате BCD) вычитается 07 (в bl). Ответ помещается в формате BCD в ax — 0007.

См. также aaa, aad, aam, daa, das

adc — Сложение с переносом

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение Суммирует байты, слова и двойные слова (только для 80386/486) плюс текущее значение (1 или 0) флага переноса

Синтаксис/Пример

```
adc al, immB                          adc al, 2
adc ax, immW                          adc ax, 1024
adc regB:memB, immB                   adc bl, 2
adc regW:memW, immW                   adc [word bx], 1024
adc regW:memB, immB                   adc cx, 2
adc regB:memB, regB                   adc [byte bx], dl
adc regW:memW, regW                   adc dx, bx
adc regB, regB:memB                   adc bl, bh
adc regW, regW:memW                   adc dx, [word bx]
```

только для 80386/486

```
adc eax, immDW                        adc eax, 65537
adc regDW:memDW, immDW                adc edx, 65537
adc regDW:memDW, immB                 adc [dword bx], 2
adc regDW:memDW, regDW                adc edx, ecx
adc regDW, regDW:memDW                adc ecx, [dword bx]
```

Образец кода

```
DATASEG
var dd 01FFFEh              ; 131070 десятичное
CODESEG
mov ax, 5                   ; Значение для сложения
mov bx, offset var          ; Адресация var
add [word bx], ax           ; Сложение младшего слова
adc [word bx + 2], 0        ; Сложение с переносом (var = 131075)
```

Описание При сложении значений нескольких байтов или нескольких слов adc следует использовать после начального add для младших значений, чтобы был возможен перенос в старшие байты или слова. Пример демонстрирует, как работает эта инструкция, прибавляя 5 к значению двойного слова, хранящегося в метке var. Инструкция adc прибавляет возможный перенос, образующийся при начальном сложении младшего слова с непосредственным значением 5.

См. также add, sbb, sub

add — Сложение без переноса

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение Сложение двух байтов, слов или двойных слов (только для 80386/486)

Синтаксис/Пример

```
add al, immB                          add al, 2
add ax, immW                          add ax, 1024
add regB:memB, immB                   add bl, 2
add regW:memW, immW                   add [word bx], 1024
add regW:memB, immB                   add cx, 2
add regB:memB, regB                   add [byte bx], dl
add regW:memW, regW                   add dx, bx
add regB, regB:memB                   add bl, bh
add regW, regW:memW                   add dx, [word bx]
```

только для 80386/486

```
add eax, immDW                        add eax, 65537
add regDW:memDW, immDW                add edx, 65537
add regDW:memDW, immB                 add [dword bx], 2
```

```
add regDW:memDW, regDW                    add edx, ecx
add regDW, regDW:memDW                    add ecx, [dword bx]
```

Образец кода

```
DATASEG
var dd 01FFFEh          ; 131070 десятичное
CODESEG
mov ax, [word var]      ; Загрузка ax:dx значе-
mov dx, [word var+2]    ; нием двойного слова
add ax, [word var]      ; Сложение младшего слова
adc dx,[word var+2]     ; Сложение старшего слова+cf
mov [word var], ax      ; Сохранение ax·dx для
mov [word var + 2], dx  ; значения двойного слова
```

Описание

add используется для сложения значений любых двух байтов, слов или двойных слов (только для 80386), хранящихся в регистровых или переменных памяти. (Оба операнда не могут быть ссылками к памяти.) Слагаемое двух операндов хранится в первом из операндов. При сложении значений, состоящих из нескольких байтов, вслед за add используйте adc, осуществляя сложение с возможным переносом. В примере используется add совместно с adc для сложения значения двойного слова с самим собой.

См. также adc, sbb, sub, xadd

and — Логическое И

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		0	-	-	-	*	*	u	*	0

Назначение Осуществляет логическое И для значений двух байтов, слов или двойных слов (только для 80386/486)

Синтаксис/Пример

```
and al, immB                              and al, 0Fh
and ax, immW                              and ax, 0FF00h
and regB:memB, immB                       and bl,01h
and regW:memW, immW                       and [word bx], 0000h
and regW:memB, immB                       and cx, 0080h
and regB:memB, regB                       and [byte bx], dl
and regW:memW, regW                       and dx, cx
and regB, regB:memB                       and bl, bh
and regW, regW:memW                       and dx, [word bx]
```

только для 80386/486

```
and eax, immDW                            and eax, 0FF000000h
and regDW:memDW, immDW                    and edx, 0FFFF0000h
and regDW:memDW, immB                     and [dword bx], 01h
and regDW:memDW, regDW                    and edx, ecx
and regDW, regDW:memDW                    and ecx, [dword bx]
```

Образец кода

```
and dl, 0Fh             ; Установка старших 4 MSD в 0
```

Описание *Используется для выполнения логического И битов любых двух байтов, слов или двойных слов* (только для 80386/486), чьи значения хранятся в регистровых либо переменных памяти. (Оба операнда не могут быть ссылками к памяти.) Соответствующие биты первого операнда устанавливаются в 1 только в том случае, если в обоих операндах они равны 1. Пример использует инструкцию and, чтобы установить первые 4 бита в байтовом регистре в 0.

См. также or, xor, test

bound — Проверка принадлежности индекса множества указанному диапазону

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
		*	*		-	-	-	-	-	-	-	-	-

Назначение Проверяет принадлежность индекса множества указанному диапазону.

Синтаксис/Пример

```
bound regW, memDW                         bound si, [word bx]
```

только для 80386/486

```
bound regDW, memQW                        bound esi, [dword bx]
```

Образец кода

```
DATASEG
LowBound DW 199
CODESEG
P286
mov si, 105             ; Загрузка si значением индекса
bound si, [LowBound]    ; Проверяет, находится ли индекс в заданных границах
```

Описание Присваивает значение индекса первому операнду и адрес диапазона значений индексов — второму операнду. Эта конструкция должна содержать два слова (или два двойных слова для 80386/486) с размещением сперва младшего значения (по младшему адресу). Если значение первого операнда не находится в численном диапазоне этих двух значений, функцией Print Screen осуществляется прерывание 5; поэтому необходимо выявить и предотвратить операции Print Screen перед использованием bound.

См. также iret

bsf — Побитовый просмотр вперед

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	*	-	-	-

Назначение Просмотр битов в порядке от LSD к MSD (от младшего бита к старшему)

Синтаксис/Пример
```
bsf regW, regW:memW                    bsf cx, dx
bsf regDW, regDW:memDW                 bsf ecx, [dword var]
```

Образец кода
```
P386
mov dx, 0800h          ; Установка 11-го бита в 1
bsf cx, dx             ; Просмотр (cx = 000Bh)
jz short @@10          ; Пропуск сдвига, если все биты = 0
shr dx, cl             ; Сдвиг dx посредством cl (dx =0001)
@@10:
```

Описание Первый операнд bsf содержит результат просмотра справа налево второго операнда (начиная с бита 0). Если все биты являются 0, тогда zf устанавливается в 1 и первый операнд остается неизменным. Если единичный бит обнаруживается, тогда zf устанавливается в 0 и первый операнд устанавливается в номер бита. Пример использует это значение для сдвига бита в dx на позицию младшего бита LSD.

См. также bsr

bsr — Просмотр битов назад

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	*	-	-	-

Назначение Просмотр битов в порядке от MSD к LSD (от старшего бита к младшему)

Синтаксис/Пример
```
bsr regW, regW:memW                    bsr cx, dx
bsr regDW, regDW:memDW                 bsr ecx, [dword var]
```

Образец кода
```
P386
mov dx, 0040h          ; Установка 6-го бита в 1
bsr cx, dx             ; Просмотр (cx = 000Bh)
jz short @@10          ; Пропуск сдвига, если все биты = 0
shr dx, cl             ; Сдвиг dx посредством cl (dx =0001)
@@10:
```

Описание Первый операнд bsr получает результат просмотра второго операнда слева направо (начиная от старшего бита MSD). Если все биты являются 0, тогда zf устанавливается в 1 и первый операнд остается неизменным. Если единичный бит обнаруживается, тогда zf устанавливается в 0 и первый операнд устанавливается в номер бита. Пример использует это значение для сдвига бита в dx на позицию младшего бита LSD.

См. также bsf

bswap — Обмен байтов

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Обменивает байты в 32-битовом регистре для преобразования между форматами с малым и большим окончанием

Синтаксис/Пример
```
bswap regDW                            bswap eax
```

Образец кода
```
P486
mov eax, 0ABCD1234h    ; Присвоение контрольного значения eax
bswap eax              ; Обмен байтов (eax = 3412CDAB)
bswap eax              ; Обмен байтов (eax = ABCD1234)
```

Описание Эта инструкция используется на процессорах 80486 и более поздних для преобразования между формами с малым и большим окончанием. Процессоры Intel хранят данные в форме с малым окончанием (младшие значения в нижних адресах). Процессоры Motorola хранят данные в форме с большим окончанием (младшие значения в верхних адресах). Можно использовать bswap для конвертирования файлов данных для компьютерных систем, основывающихся на этих процессорах, таких как PC и Macintosh. Как демонстрирует образец кода, инструкция действует как переключатель. Можно использовать bswap для конвертирования данных без необходимости определять текущий формат данных.

Не забудьте специфицировать 32-битовый регистр для этой инструкции. Если вы специфицируете 16-битовый регистр, результаты непредсказуемы.

См. также xchg

bt[rel] — Проверка бита

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	*

Назначение Копирует бит во флаг переноса

Синтаксис/Пример

```
bt regW:memW, immB                    bt ax, 14
bt regW:memW, regW                    bt [word var], cx
bt regDW:memDW, immB                  bt eax, 8
bt regDW:memDW, regDW                 bt [dword var], ecx
```

Образец кода

```
P386
mov dx, 0200h           ; Присвоение dx контрольного значения
bt dx, 9                ; Копирование 9-го бита в cf и дополнение
jc @@10                 ; Проверка cf
call procedure          ; Вызов процедуры, если 9-й бит равен 0
@@10:
```

Описание Первым операндом bt должен быть регистр слова или двойного слова либо ссылка к памяти. Вторым операндом может быть регистр слова (0–15), либо двойного слова (0–31), либо непосредственное значение. При исполнении bt копирует бит из позиции первого операнда, указанной вторым операндом, в cf. Можно использовать jc или jnc для проверки, равен ли этот бит 1 или 0.

См. также btc, btr, bts, test

btc — Проверка бита и дополнение

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	*

Назначение Копирует бит во флаг переноса, а затем дополняет бит в первоначальном значении

Синтаксис/Пример

```
btc regW: memW, immB                  btc ax, 14
btc regW:memW, regW                   btc [word var], cx
btc regDW:memDW, immB                 btc eax, 8
btc regDW:memDW, regDW                btc [dword var], ecx
```

Образец кода

```
P386
mov dx, 0200h           ; Присвоение dx контрольного значения
btc dx, 9               ; Копирование 9-го бита в cf и дополнение
jc @@10                 ; Проверка cf
call procedure          ; Вызов процедуры, если 9-й бит равен 0
@@10:
```

Описание Операнды и действия идентичны bt, но после копирования указанного бита в cf этот бит дополняется (переключается) в первоначальном значении. В примере эти действия приводят к тому, что dx остается равным 0. Несмотря на это, нулевой флаг *не* устанавливается.

См. также bt, btr, bts, test

btr — Проверка бита и сброс

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	*

Назначение Копирует бит во флаг переноса, а затем сбрасывает бит в первоначальном значении

Синтаксис/Пример

```
btr regW: memW, immB                    btr [word var], 5
btr regW:memW, regW                     btr dx, cx
btr regDW:memDW, immB                   btr [dword var], 6
btr regDW:memDW, regDW                  btr edx, ecx
```

Образец кода

```
P386
mov dx, 0ABCDh          ; Присваивает dx проверяемое значение
mov cx, 15              ; Присваивает cx номер бита
btr dx, cx              ; Копирование бита в cf и сброс
```

Описание

Операнды и действие идентичны bt, но после копирования указанного бита в cf этот бит сбрасывается в первоначальном значении в 0. В примере это приводит к изменению dx на 02BCDh.

См. также

bt, btc, bts, test

bts — Проверка бита и установка

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	*

Назначение

Копирует бит во флаг переноса, а затем устанавливает бит в первоначальном значении

Синтаксис/Пример

```
bts regW: memW, immB                    bts dx, 4
bts regW:memW, regW                     bts [word var], cx
bts regDW:memDW, immB                   bts eax, 3
bts regDW:memDW, regDW                  bts [dword var], edx
```

Образец кода

```
P386
mov dx, 0ABCDh          ; Присваивает dx проверяемое значение
mov cx, 14              ; Присваивает cx номер бита
btr dx, cx              ; Копирование бита в cf и установка
```

Описание

Операнды и действие идентичны bt, но после копирования указанного бита в cf этот бит устанавливается в первоначальном значении в 1. В примере это приводит к изменению dx на 0EBCDh.

См. также

bt, btc, btr, test

call — Вызов процедуры

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение

Вызов подпрограммы процедуры

Синтаксис/Пример

```
call nearTarget                         call Here
call farTarget                          call far ptr There
call regW                               call bx
call memW                               call [word bx]
call memDW                              call [dword bx]
```

только для 80386/486

```
call regDW                              call eax
call memFW                              call [fword si]
```

Образец кода

```
call Times2             ; Вызов подпрограммы
jmp Exit                ; Выход из программы
PROC Times2             ; Подпрограмма
add ax, ax              ; Сложение двойного слова в
adc dx, dx              ;   ax dx с самим собой
ret                     ; Возврат из подпрограммы
ENDP
```

Описание

Инструкция call выталкивает в стек адрес следующей инструкции, а затем осуществляет переход по целевому адресу программы, приводя к тому, что инструкции, которые содержатся в подпрограмме процедуры, начинают выполняться. Обычно подпрограмма завершается инструкцией ret, извлекающей из стека адрес возврата и продолжающей программу инструкцией, которая следует за вызывающей инструкцией call. В большинстве программ целью вызова является метка, помечающая первую инструкцию подпрограммы. Но целью может быть также и ссылка к памяти либо 16-битовый регистр, содержащий адрес подпрограммы. Пример вызывает маленькую подпрограмму Times2, которая суммирует значение, хранящееся в ax:dx, с самим собой. Инструкция ret приводит к тому, что программа продолжается с jmp Exit.

См. также

ret

cbw **Преобразование байта в слово**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Распространяет байт, имеющий знак, на слово, имеющее знак

Синтаксис/Пример `cbw no operands` `cbw`

Образец кода
```
mov al, -1              ; Устанавливает al в -1
cbw                     ; Распространяет al на ax (ax = -1)
```

Описание cbw используется для распространения 8-битового, имеющего знак, значения, содержащегося в al, на 16-битовое значение такой же величины, содержащееся в ax. Инструкция работает, копируя MSD (старший значащий бит) al во все биты ah, устанавливая ah в 0FFh, если al был отрицательным (MSD=1), либо устанавливая ah в 00h, если al был положительным (MSD=0).

См. также cdq, cwd, cwde

cdq **Преобразование двойного слова в четверное**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Распространяет двойное слово, имеющее знак, на четверное слово, имеющее знак

Синтаксис/Пример `cdq no operands` `cdq`

Образец кода
```
P386
mov eax, -1             ; Устанавливает eax в -1
cdq                     ; Распространяет eax на eax:edx (eax:edx = -1)
```

Описание cdq используется для распространения 32-битового, имеющего знак, значения, содержащегося в eax, на 64-битовое значение такой же величины, содержащееся в регистровой паре eax:edx. Инструкция работает, копируя MSD (старший значащий бит) eax во все биты edx, устанавливая edx в 0FFFFFFFFh, если eax был отрицательным (MSD=1), либо устанавливая edx в 0, если edx был положительным (MSD=0).

См. также cbw, cwd, cwde

clc **Очистка флага переноса**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	0

Назначение Устанавливает флаг переноса в 0

Синтаксис/Пример `clc no operands` `clc`

Образец кода
```
PROC AnyProc
; Код процедуры
@@ErrExit:
 stc                    ; Установка переноса (ошибка)
 ret                    ; Возврат в точку вызова
@@NoErrExit:
 clc                    ; Очистка переноса (нет ошибки)
 ret                    ; Возврат в точку вызова
ENDP AnyProc
```

Описание Выполнение clc сбрасывает флаг переноса в 0. Как показывает пример кода, эта инструкция часто используется при передаче флага ошибки назад из подпрограммы, очищая cf, если не было обнаружено никакой ошибки.

См. также cmc, stc

cld **Очистка флага направления**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	0	-	-	-	-	-	-	-

Назначение Сброс флага направления в 0

Синтаксис/Пример `cld no operands` `cld`

Образец кода

```
DATASEG
s1 db Copy me              ; Исходная цепочка
s2 db 80 dup (?)           ; Целевая цепочка
CODESEG                    ; Замечание: предположим, что es = ds
mov cx, 4                  ; Присвоение cx значения счетчика
mov si, offset s1          ; Адресация источника посредством ds:si
mov di, offset s2          ; Адресация приемника посредством es:di
cld                        ; Автоматическое увеличение si и di
rep movsb                  ; Копирование 4 символов из источника в приемник
```

Описание cld используется для сброса флага назначения в 0. Следует всегда выполнять cld перед повторением операции над цепочками — это приводит к автоматическому увеличению либо si, либо di, либо обоих из них при df=0. Пример использует cld для повторяющейся цепочечной инструкции movsb, копирующей 4 символа из цепочки s1 в цепочку s2.

См. также std

cli — Очистка флага прерывания

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	0	-	-	-	-	-	-

Назначение Сбрасывает флаг разрешения прерываний в 0

Синтаксис/Пример cli no operands — cli

Образец кода

```
sti                        ; Разрешает прерывания
hlt                        ; Ожидание прерывания
cli                        ; Запрещает прерывания
```

Описание Выполнение cli запрещает распознавание внешних маскируемых прерываний. Для уверенности в правильной работе PC прерывания не должны отключаться надолго. Пример предлагает один из возможных способов синхронизации программы с внешним событием, реализуя паузу посредством hlt до момента осуществления прерывания, а затем немедленно запрещая прерывания.

См. также sti

cmc — Дополнение флага переноса

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	*

Назначение Дополняет (переключает) флаг переноса

Синтаксис/Пример cmc no operands — cmc

Образец кода

```
PROC AnyProc
; Код процедуры
@@Exit:
 cmc                       ; Дополнение флага ошибки
 ret                       ; Возврат в точку вызова
ENDP
```

Описание cmc используется для дополнения флага переноса, меняя cf на 0, если он был 1, или на 1, если он был 0. Одним из вариантов является использование в процедуре, возвращающей cf в качестве флага ошибки; но поскольку другие операции оставляют cf в противоположном состоянии, перед возвратом флаг переноса должен быть переключен.

См. также clc, stc

cmp — Сравнение

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение Сравнивает два операнда

Синтаксис/Пример

```
cmp al, immB                    cmp al, 2
cmp ax, immW                    cmp ax, 1024
cmp regB|memB, immB             cmp bl, 2
cmp regW|memW, immW             cmp [word bx], 1024
cmp regW|memW, immB             cmp cx, 2
cmp regW|memB, regB             cmp [byte bx], dl
cmp regW|memW, regW             cmp dx, bx
```

```
cmp regB, regB|memB                    cmp bl, bh
cmp regW, regW|memW                    cmp dx, [word bx]
```

только для 80386/486

```
cmp eax, immDW                         cmp eax, 65537
cmp regDW|memDW, immDW                 cmp [dword si], 99123
cmp regDW|memDW, immB                  cmp [dword bx], 2
cmp regDW|memDW, regDW                 cmp edx, ecx
cmp regDW, regDW|memDW                 cmp ecx, [dword bx]
```

Образец кода

```
cmp ax, cx              ; Сравнение ax и cx
je @@10                 ; Переход, если ax = cx
inc ax                  ; Увеличение ax, если ax <> cx
@@10
```

Описание

cmp используется для сравнения значений любых двух байтов, слов или двойных слов (только для 80386). Оба операнда одновременно не могут быть ссылками к памяти. Обычно вслед за cmp следует инструкция перехода, осуществляя соответствующее действие, основанное на результатах сравнения. В примере cmp используется для проверки того, содержат ли регистры ax и cx одинаковые значения. Если нет — ax увеличивается. Инструкция cmp работает, вычитая второй операнд из первого, отбрасывая результат, но сохраняя флаги, которые могут быть затем проверены. Следовательно, если использовать cmp для проверки того, чем именно одно значение отличается от другого, следует перечислить операнды в том порядке, в каком требуется для вашего выражения. Например, если вы хотите узнать — ax < bx? — воспользуйтесь cmp ax, bx, за которой должна следовать инструкция jl.

См. также cmps, cmpxchg, sub

cmps cmpsb cmpsd cmpsw — Сравнение цепочек

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение Сравнение цепочек значений

Синтаксис/Пример

```
cmps [es ]memB, memB                   cmps [byte dest],[byte source]
cmps [es ]memW, memW                   cmps [word es.si], [word di]
cmpsb no operands                      cmpsb
cmpsw no operands                      cmpsw
```

только для 80386/486

```
cmps [es ]memDW, memDW                 cmps [dword dest], [dword source]
cmpsd no operands                      cmpsd
```

Образец кода

```
DATASEG
s1 db Woe is me
s2 db Woe is you
CODESEG
ASSUME es  DGROUP       ; Сообщает TASM, куда указывает es
mov si, offset s1       ; Адресация исходной цепочки
mov di, offset s2       , Адресация цепочки назначения
mov cx, 10              , Присвоение cx счетчика
cld                     ; Автоувеличение si, di
repe cmps [s1], [s2]    ; Нахождение первого несовпадения
repe cmpsb              , Замечание: то же, что и предшествующая строка
```

Описание

Инструкции сравнения цепочек сравнивают два значения в памяти. Предваряя эти инструкции префиксами repe или repne и устанавливая значение счетчика в cx, можно построить инструкции, сравнивающие последовательности значений. Первый операнд является *источником* и должен адресоваться посредством ds:si, если только замещающий сегмент не используется, как в [es:метка]. Второй операнд является *назначением* и должен адресоваться посредством es:di. Инструкции выполняют вычитание *[источник]* — *[назначение]*, отбрасывая результат и сохраняя флаги — аналогично работе cmp. Кроме того, если df=0, si и di увеличиваются на число байтов, подлежащих сравнению. Если df=1, индексные регистры уменьшаются.

Следует использовать cmps, если вы хотите, чтобы Turbo Assembler проверил, что операнды адресуются посредством ds:si или es:si и es:di, а также тогда, когда необходимо использовать es для замещения операнда-источника. Либо используйте остальные три мнемоники, если не хотите специфицировать операнды полностью: cmpsb — для сравнения байтов, cmpsw — для сравнения слов, cmpsd (только для 80386) — для сравнения

двойных слов. Не имеет значения, какую форму инструкции вы используете — по-прежнему только от вас зависит загрузка si и di верными адресами. (Например, последние две строки в примере, отыскивающие первый несовпадающий символ в двух цепочках, формируют одинаковый код.)

См. также `ins, insb, insd, insw, lods, lodsb, lodsd, lodsw, movs, movsb, movsd, movsw, outs, outsb, outsd, outsw, scas, scasb, scasd, scasw, stos, stosb, stosd, stosw`

cmpxchg — Сравнение и обмен

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		*	-	-	-	*	*	*	*	*

Назначение Сравнивает и осуществляет обмен данных между аккумулятором и назначением, которое может быть регистром или ссылкой к памяти

Синтаксис/Пример

```
cmpxchg regB ; memB, regB                    cmpxchg bh, ah
cmpxchg regW ; memW, regW                    cmpxchg bx, ah
cmpxchg regDW ; memDW, regDW                 cmpxchg ebx, eax
```

Образец кода

```
mov    ebx, 12345678h      ; Присвоение проверяемого
                           ;  значения ebx
mov    eax, 87654321h      ; Присвоение проверяемого
                           ;  значения eax
cmpxchg ebx, eax           ; Помещение ebx в eax
cmpxchg ebx, eax           ; Помещение eax в ebx
```

Описание Эта инструкция, состоящая из двух частей и доступная только на процессорах 80486 и более поздних, начинается с выполнения сравнения значений аккумулятора и назначения другого регистра или назначения, хранящегося в ячейке памяти. Если аккумулятор (eax в образце кода) отличается по значению от назначения (ebx), значение назначения загружается в аккумулятор. Если значения аккумулятора и назначения равны, назначение также загружается в аккумулятор, однако эффект в этом случае равен нулю, так как все же происходит перезагрузка одинаковых значений.

Флаги устанавливаются, как и для инструкции cmp. Флаг zf устанавливается в 1, если значения источника и назначения равны; он устанавливается в 0, если эти два значения первоначально не равны. Другими словами, если zf равен 0, значение в аккумуляторе меняется на значение назначения.

См. также `cmp, xchg`

cwd — Преобразование слова в двойное слово

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Расширяет слово, обладающее знаком, до двойного слова, обладающего знаком

Синтаксис/Пример

```
cwd no operands                              cwd
```

Образец кода

```
mov ax, -1                 ; Установка ax в -1
cwd                        ; Распространение ax на ax:dx
                           ; (ax:dx = -1)
```

Описание cwd используется для распространения 16 битового, обладающего знаком, значения, хранящегося в ax, на 32-битовое значение такой же величины, хранящееся в регистровой паре ax:dx. Инструкция работает, копируя MSD (старший бит) ax во все биты в dx, устанавливая dx в 0FFFFH, если ax было отрицательным (MSD=1), либо устанавливая dx в 0, если ax было положительным (MSD=0).

См. также `cbw, cdq, cwde`

cwde — Преобразование слова в расширенное двойное слово

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Расширяет слово, обладающее знаком, до расширенного двойного слова, обладающего знаком

Синтаксис/Пример

```
cwde no operands                             cwde
```

Образец кода

```
mov ax, -1              ; Установка ax в -1
cwde                    ; Распространение ax на eax
                        ; (eax = -1)
```

Описание

cwde используется для распространения 16-битового, обладающего знаком, значения, хранящегося в ax, на 32-битовое значение такой же величины, хранящееся в eax. Инструкция работает, копируя старший бит (MSD) ax во все биты в старшем слове eax, устанавливая старшее слово в 0FFFFH, если ax было отрицательным (MSD=1), либо устанавливая dx в 0, если ax было положительным (MSD=0).

См. также cbw, cdq, cwd

daa — Десятичная подгонка после суммирования

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	*	*	*	*	*

Назначение Подгоняет численную сумму двух упакованных BCD чисел к упакованному формату BCD

Синтаксис/Пример

```
daa no operands                                    daa
```

Образец кода

```
mov al, 053h            ; Упаковка 5 и 3 в al
mov bl, 018h            ; Упаковка 1 и 8 в bl
add al, bl              ; al <- al + bl (al = 06Bh)
daa                     ; Подгонка результата (al=071h)
```

Описание После суммирования двух упакованных 8-битовых байтов и помещения результата в al исполните daa для преобразования двоичной суммы обратно в упакованный формат. Если оба, и af, и cf, равны 1, тогда сумма была больше десятичного 99. (Можно использовать эту информацию для генерации переноса при выполнении сложения чисел, содержащих несколько цифр.) Если af=1, а cf=0, значит, сумма двух младших цифр была меньше 9 и перенос автоматически учитывается в старшей цифре результата. (Обычно можно не обращать на это внимания.) Если оба, и af, и cf, равны 0, значит, никакой перенос не производился (и daa не изменяет значения в al).

См. также aaa, aad, aam, aas, das

das — Десятичная подгонка после вычитания

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	-

Назначение Подгоняет численную сумму двух упакованных BCD чисел к упакованному формату BCD

Синтаксис/Пример

```
das no operands                                    das
```

Образец кода

```
mov al, 007h            ; Упаковка 0 и 7 в al
mov bl, 014h            ; Упаковка 1 и 4 в bl
add al, bl              ; al <- al - bl (al = 0F3h)
das                     ; Подгонка результата (al=093h)
```

Описание После суммирования двух упакованных 8-битовых байтов поместите результат в al и исполните das для преобразования двоичной суммы обратно в упакованный формат. Если оба, и af, и cf, равны 0, значит, во время вычитания потребовалось заимствование. Если cf=0, а af=1, значит, потребовалось заимствование для двух младших цифр и результат подгоняется соответствующим образом. (Обычно на это можно не обращать внимания.) Если cf=1, значит, результат является отрицательным десятичным дополнением и вы можете вычесть 100 из результата, содержащегося в al, чтобы определить абсолютное значение. Другими словами, если cf=1, а al=93h, как в примере, верное значение равняется -7 или (93–100).

См. также aaa, aad, aam, aas, daa

dec — Уменьшение

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	-

Назначение Вычитание 1 из регистра или переменной

Синтаксис/Пример

```
dec regB¦memB                                      dec cl
dec regW¦memW                                      dec [word var]
```

только для 80386/486

```
dec regDW:memDW                                    dec edx
```

Образец кода

```
mov cx, 100              ; Присвоение cx счетчика
@@10:
call AnyProc             ; Вызов процедуры
dec cx                   ; Вычитание 1 из счетчика
jnz @@10                 ; Переход, если cx > 0
```

Описание

dec используется для уменьшения значения байта, слова или двойного слова (только для 80386) в регистре или в памяти на 1. Это аналогично вычитанию 1 из значений, не имеющих знака, с помощью sub, но работает быстрее. Пример демонстрирует один из способов организации цикла, вызывая AnyProc (не показана) 100 раз и выполняя переход jnz до тех пор, пока cx не станет равным 0.

См. также inc

div — Деление без знака

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	u	u	u	u	u

Назначение Деление двух значений без знаков

Синтаксис/Пример

```
div regB:memB                                      div dl
div regW:memW                                      div [word var]
```

только для 80386/486

```
div regDW:memDW                                    div [dword bx]
```

Образец кода

```
DATASEG
var dd 01FFFEh ; 131070 десятичное
CODESEG
mov ax, [word var]       ; Загрузка младшего слова в ax
mov dx, [word var +2 ]   ; Загрузка старшего слова в dx
mov bx, 1024             ; Загрузка делителя в bx
div bx                   ; ax <- 131070/1024 (ax = 127)
```

Описание

div используется для деления целых значений. Операнд относится к делителю. Делимые регистры определяются размером делителя. Байтовые делители делят содержимое ax, помещая частное в al, а остаток — в ah. Делители типа слова делят содержимое dx:ax (младшее слово в ax), помещая частное в ax, а остаток — в dx. Делители типа двойного слова (только для 80386) делят содержимое edx:eax (младшее двойное слово в eax), помещая частное в eax, а остаток — в edx.

Если результат деления больше максимального значения, которое может поместить регистр, назначенный для частного, или если делитель равен 0 — тогда генерируется прерывание 0. Если не предприняты шаги для предотвращения этого прерывания, DOS остановит программу и отобразит сообщение об ошибке деления. Это усложняется еще тем фактом, что для процессоров 8086/88 адрес возврата прерывания указывает на инструкцию, следующую за div, а для процессоров 80286 и 80386 – на инструкцию div, вызвавшую сбой.

См. также idiv

enter — Процедура ввода

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
		*	*		-	-	-	-	-	-	-	-	-

Назначение Создает границы стека для локальных переменных процедуры

Синтаксис/Пример

```
enter immW,0                                       enter 2,0
enter immW,1                                       enter 8,1
enter immW,immB                                    enter 0,3
```

Образец кода

```
Proc AnyProc
enter 8,0                ; Резервирование 8 байт для локальных переменных
; Код процедуры
leave                    ; Возврат зарезервированной области стека
ret                      ; Возврат в точку вызова
ENDP AnyProc
```

Описание Большей частью используемая языками высокого уровня enter подготавливает bp, вычитая из bp то число байтов, которое указано первым операндом, и резервирует в стеке пространство для переменных, которое может затем адресоваться посредством ss:bp. Второй операнд равен уровню секционирования и может быть непосредственным значением 0 или 1 для более быстрого выполнения либо более высоким непосредственным значением. Уровень используется такими языками, как Pascal, которые допускают секционирование реальных процедур, обеспечивая метод доступа внутренних процедур к локальным переменным, объявленным на внешних уровнях. Пример показывает, как использовать enter для резервирования 8 байт стека для локальных переменных. Для восстановления этого пространства непосредственно перед ret необходимо выполнить leave.

См. также leave, ret

esc — Выход

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Передает инструкции сопроцессору

Синтаксис/Пример
```
esc imm6, regB:regW                    esc 5, ax
esc imm6, memAll                       esc 5, [var]
```

Образец кода
```
fld st(0)              , Выталкивание операнда
wait                   , Ожидание, требуемое для
esc 8, ax              , 8087
```

Описание Можно использовать esc для передачи инструкций сопроцесоору. Первый операнд представляет действующий код инструкции. Второй операнд специфицирует значение приемника или источника для инструкции сопроцессора. Поскольку Turbo Assembler распознает мнемоники инструкций математического сопроцессора, esc редко используется на практике. Если вы все же собираетесь применить esc, помните, что для 8087 требуется инструкция wait перед каждой инструкцией математического сопроцессора. Turbo Assembler автоматически вставляет wait, при необходимости — еще одна причина использовать вместо esc мнемоники сопроцессора.

См. также wait

hlt — Остановка

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Останавливает программу до прерывания или перезагрузки

Синтаксис/Пример
```
hlt no operands                        hlt
```

Образец кода
```
cli                    , Отключает (запрещает) маскируемые прерывания
hlt                    ; Пауза, пока не возникает NMI (немаскируемое прерыва-
ние)
                       , либо перезагрузка
sti                    , Включает маскируемые прерывания
```

Описание hlt выполняется для осуществления паузы до тех пор, пока не будет распознан следующий сигнал прерывания либо получен сигнал перезагрузки. Если маскируемые прерывания отключены, hlt приостанавливает программу до сигнала перезагрузки либо до распознавания немаскируемого прерывания.

См. также cli, sti

idiv — Целочисленное деление со знаком

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		u	-	-	-	u	u	u	u	u

Назначение Деление двух значений со знаком

Синтаксис/Пример
```
idiv regB:memB                         idiv dl
idiv regW:memW                         idiv [word var]
```

только для 80386/486
```
idiv regDW:memDW                       idiv [dword bx]
```

Образец кода
```
mov ax, 100            ; Присвоение делимого ax
mov bl, -3             ; Присвоение делителя bl
```

```
idiv bl                    ; al <- ax/bl (остаток в ah)
neg al                     ; Нахождение абсолютного значения al
```

Описание

idiv используется для деления целых значений со знаком. Операнд относится к делителю. Регистр делимого определяется размером делителя. Делители типа байта делят содержимое ax, помещая частное в al, а остаток — в ah. Делители типа слова делят содержимое dx:ax (младшее слово в ax), помещая частное в ax, а остаток — в dx. Делители типа двойного слова (только для 80386/486) делят содержимое edx:eax (младшее слово в eax), помещая частное в eax, а остаток — в edx. Остаток всегда имеет тот же знак, что и делимое.

Пример делит десятичное 100 на -3, помещая частное в al (0DFh), а остаток — в ah (01). Помните, что отрицательные значения, такие как 0DFh, выражаются в форме значений в дополнительном коде. Чтобы вычислить абсолютное значение (3 в данном случае), используйте neg, как в примере.

Если результат деления больше максимального значения, которое может поместиться в регистре, назначенном для частного, либо если делитель равен 0 — тогда генерируется прерывание 0. Если не предприняты шаги для предотвращения этого прерывания, DOS остановит программу и отобразит сообщение об ошибке деления. Это усложняется еще больше тем фактом, что для процессоров 8086/88 адрес возврата прерывания указывает на инструкцию, следующую за div, а для процессоров 80286 и 80386 — на инструкцию div, вызвавшую сбой.

См. также div

imul — Умножение целых чисел со знаком

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	u	u	u	u	*

Назначение Умножает два значения, обладающие знаками

Синтаксис/Пример

```
imul regB|memB                          imul [byte bx]
imul regW|memW                          imul cx
```

только для 80286, 80386/486

```
imul regW, immB                         imul cx, 9
imul regW, immW                         imul bx, 451
imul regW, regW|memW, immB                 imul cx, [word bx], 3
imul regW, regW|memW, immW                 imul ax, bx, 300
```

только для 80386/486

```
imul regDW|memDW                        imul [dword bx]
imul regDW, immB                        imul ebx, 10
imul regDW, immDW                       imul eax, 32769
imul regW, regW|memW                    imul bx, cx
imul regDW, regDW|memDW                 imul ecx, [dword bx]
imul regDW, regDW|memDW, immB           imul eax, edx, 12
imul regDW, regDW|memDW, immDW          imul eax, [dword bx], 35790
```

Образец кода

```
mov al, 4                  ; Умножаемое
mov bl, -2                 ; Множитель
imul bl                    ; ax <- al*bl
                           ; (ax = 0FFF8h, cf = of = 0)
mov al, 127                ; Умножаемое
mov bl, -128               ; Множитель
imul bl                    ; ax <- al*bl
                           ; (ax = 0C080h, cf = of = 1)
```

Описание

В зависимости от процессора imul обладает тремя основными форматами, включающими от одного до трех операндов. Некоторые формы требуют полностью определенных регистров. Более простая форма умножает байтовый регистр или переменную на al, помещая результат в ax. Аналогичная форма умножает регистр слова или переменную на ax, помещая результат в dx:ax (младшее слово в ax). Только на процессорах 80386/486 imul может умножать eax на регистр либо переменную двойного слова, помещая результат в edx:eax. Во всех этих формах, если после умножения оба — и cf, и of — равны 0, тогда старшая часть результата является просто знаковым расширением младшей части. Другими словами, как показывает первая часть примера, умножение 4*(-2) устанавливает ax в 0FFF8h. Поскольку cf и of равны 0, ah (0FFh) расширяет знак 8-битового ответа в al (0F8h), создавая полное 16-битовое значение. Когда оба — и cf, и of — устанавливаются в 1, как во второй части примера, тогда результат занимает всю ширину

регистра назначения — в данном случае ax, который равен значению в дополнительном коде 0C080h, или -16256 десятичному — результату 127*(-128).

Процессоры 80286 и 80386/486 расширяют эти основные формы инструкциями imul с несколькими операндами. В формате с двумя операндами первый операнд является умножаемым, второй — непосредственным множителем. Результат замещает регистр, определенный в качестве умножаемого. В формате с тремя операндами первый операнд специфицирует регистр назначения для результата, второй содержит умножаемое, а третий является непосредственным множителем типа байта или слова. 80386/486 еще больше расширяет эти формы, допуская различные комбинации регистров двойных слов, ссылок к памяти и непосредственных значений. При всех этих отличиях, если cf и of равны 0 после выполнения imul, то результат точно соответствует специфицированному регистру назначения (всегда первый операнд); в противном случае результат слишком велик для этого регистра.

См. также mul

in — Ввод из порта

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Вводит значения из портов

Синтаксис/Пример

```
in al, immB                          in al, 14h
in al, dx                            in al, dx
in ax, immB                          in ax, 01Fh
in ax, dx                            in ax, dx
```

только для 80386

```
in eax, immB                         in eax, 0Fh
in eax, dx                           in eax, dx
```

Образец кода

```
Ctrl8259 EQU 021h          ; 8259 маскирует порт
in al, Ctrl8259            ; Считывание подключенной маски 8259
and al, EnableIRQ          ; Очистка маскируемого бита
out Ctrl8259               ; Запись новой маски 8259
```

Описание Инструкция in считывает значение аппаратного порта в al, ax или eax (только для 80386/486). Как показывает пример, in часто используется совместно с out и логическими инструкциями, такими как and и or, для проверки и изменения битовых переключателей различных адресов портов компьютера. Простейшая форма in считывает значение в al из непосредственного адреса порта в диапазоне 0–255. Чтобы получить доступ к более высоким адресам портов, необходимо специфицировать адрес в регистре dx.

См. также ins, out

inc — Увеличение

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	-

Назначение Прибавляет 1 к регистру или переменной

Синтаксис/Пример

```
inc regB|memB                        inc [byte bx]
inc regW|memW                        inc dx
```

только для 80386/486

```
inc regDW|memDW                      inc ecx
```

Образец кода

```
mov dx, 0                  ; Инициализация dx
@@10:
callAnyProc                ; Вызов процедуры
inc dx                     ; dx <- dx + 1
cmp dx, 1000               ; dx = 1000?
jne @@10                   ; Переход, если dx <> 1000
```

Описание inc используется для увеличения значения регистра или переменной памяти типа байта, слова или двойного слова (только для 80386/486) на 1. Это аналогично прибавлению 1 к значениям без знака, но работает быстрее. Пример использует inc для создания простого цикла, применяя dx в качестве управляющей переменной для вызова процедуры AnyProc (не показана) 1000 раз. (Могут быть и более эффективные способы организации подобных циклов.)

См. также dec

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Осуществляет ввод из порта в последовательность байтов, слов или двойных слов в памяти

Синтаксис/Пример

```
ins di ! memB, dx                    rep ins [var], dx
ins di ! memW, dx                    rep ins [word var], dx
insb no operands                     rep insb
insw no operands                     rep insw
```

только для 80386/486

```
ins regDW!memDW, dx                  rep ins [dword var], dx
insd no operands                     rep insd
```

Образец кода

```
; ! ЗАМЕЧАНИЕ: Не запускайте этот пример!
mov cx, 100              ; Число слов, подлежащих считыванию
mov dx, 049h             , Специфицирует адрес порта
mov di, offset s1        ; Адрес назначения
cld                      ; Автоувеличение di
rep insw                 ; Загрузка цепочки из порта
```

Описание Как и в цепочечных инструкциях, назначения регистров для ins и сходных с ней форм insb, insd (только для 80386/486) и insw фиксированы, даже если вы полностью специфицируете метки адресов. Регистром назначения всегда является es:di, и сегмент не может быть перезагружен. Номер порта должен помещаться в dx. (Не забывайте делать это и для сходных мнемоник!) Если df=0, тогда ins увеличивает di; если df=1, тогда ins уменьшает di. Обычно можно перед ins набирать rep, повторяя инструкцию указанное число раз в cx, как показано в примере.

См. также cmpsb, cmpsd, cmpsw lods, lodsb, lodsd, lodsw, movs, movsb, movsd, movsw, outs, outsb, outsd, outsw, scas, scasb, scasd, scasw, stos, stosb, stosd, stosw

int Вызов подпрограммы обслуживания прерывания

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	0	0	-	-	-	-	-

Назначение Вызов по номеру подпрограммы обслуживания прерываний

Синтаксис/Пример

```
int 3                                int 3
int immB                             int 21h
```

Образец кода

```
DATASEG
message db Mastering Turbo Assembler, $
CODESEG
mov dx, offset message     ; Адресация цепочки сообщения
mov ah, 9                  ; Номер функции DOS
int 21h                    ; Вызов обработчика функции DOS
```

Описание Хотя имеется две формы int, в программе они проявляются одинаково. Первая форма — специальный однобайтовый код (0CCh), который отладчики обычно используют для замещения инструкций в указанных точках специальным прерыванием. Этот код вы можете вставить сами, чтобы заставить большинство отладчиков (Turbo Debugger в том числе) делать останов в различных местах. Вторая форма специфицирует байтовое значение в качестве номера прерывания, который может лежать в пределах от 0 до 255, представляя один из 256 4-байтовых адресов указателей векторов, хранимых в памяти, начиная с адреса 0000:0000. Выполнение int запускает подпрограмму обслуживания прерывания, хранимую по адресу вектора, соответствующего номеру прерывания. Непосредственно перед этим процессор выталкивает в стек флаги и адрес возврата, которые восстанавливаются в подпрограмме обслуживания прерывания при исполнении iret. Кроме того, флаги if и tf устанавливаются в 0. (Эти два флага восстанавливаются инструкцией iret, и, поскольку флаги изменяются только для подпрограммы обслуживания прерывания, некоторые справочники по 8086 неверно указывают на то, что if и tf не изменяются инструкцией int.)

См. также into, iret

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	*	*	-	-	-	-	-

Назначение Генерирует прерывание 4, если of = 1

Синтаксис/Пример
```
into no operands                                    into
```

Образец кода
```
P386
imul ecx, [dword bx]        ; ecx <- ecx * [bx]
into                        ; Прерывание при переполнении
```

Описание Инсталировав подпрограмму обслуживания прерывания для прерывания 4, можно использовать into, чтобы вызвать выполнение этого кода, если флаг переполнения устанавливается предыдущей операцией. Инструкция into ведет себя аналогично int, выталкивая флаги и адрес возврата в стек, переустанавливая tf, if и осуществляя переход по вектору прерывания 4. Код прерывания может обработать причину прерывания и выполнить iret, продолжив выполнение программы. Пример демонстрирует, как можно использовать into, чтобы обнаружить переполнение при исполнении инструкции imul для процессора 80386.

См. также int, iret

invd — Разрешение кэширования

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Организует поток внутренней инструкции кэширования для 80486, а также осуществляет специальный цикл для обмена с шиной, который разработчики аппаратного обеспечения могут использовать в качестве команды для организации любых потоков кэширования — внешних для процессора

Синтаксис/Пример
```
invd no operands                                    invd
```

Образец кода
```
P486
invd                        ; Организует поток кэширования
                            ; и осуществляет цикл шинного потока
```

Описание Используйте эту инструкцию только для процессоров 80486. Она не требует никаких операндов и не влияет ни на какие флаги. Intel утверждает, что invd является "зависимой от исполнения", имея в виду, что последующие процессоры могут выполнять эту инструкцию иначе. Видимо, нет веских причин использовать эту инструкцию в программах прикладного уровня.

См. также invlpg, wbinvd

invlpg — Разрешение входа TLB

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Разрешает вход в TLB — translation lookaside buffer (буфер трансляция-хранение)

Синтаксис/Пример
```
invlpg memAll                                       invlpg table
```

Образец кода (нет: см. Описание)

Описание Эта инструкция допускается только для процессоров 80486, и Intel отмечает, что она может как реализоваться, так и нет — на последующих процессорах. Инструкция используется для разрешения входа в TLB, который осуществляет трансляцию линейных и физических адресов. Ее не следует использовать в прикладных программах.

См. также invd, wbinvd

iret iretd — Возврат из прерывания

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	*	*	*	*	*	*	*	*

Назначение Осуществляет возврат из подпрограммы обслуживания прерывания

Синтаксис/Пример
```
iret no operands                                    iret
```

только для 80386/486

```
iretd no operands                                    iretd
```

Образец кода

```
PROC MyISR
push ax                    ; Сохранение любого изменяемого регистра
sti                        ; Разрешение маскируемых прерываний
; Код процедуры
iret                       ; Возврат из прерывания
ENDP
```

Описание

iret выполняется в качестве последней инструкции в подпрограмме обслуживания прерывания (ISR). Инструкция выталкивает из стека адрес возврата cs:ip и флаги, продолжая программу с точки прерывания. Используется одна и та же инструкция iret, независимо от того, было прерывание сгенерировано внешне или внутренне вследствие нарушения условий неверного деления или инструкциями int и into. В системах, базирующихся на 80386/486, iretd может использоваться для возврата 32-битового сегмента, выталкивая из стека полный расширенный указатель инструкции eip.

См. также

iny, into

j-condition — Условный переход

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение

Осуществляет переход к другой ячейке при установке и/или переустановке определенного флага

Синтаксис/Пример

```
условие shortTarget                                  ige @@30
```

Образец кода

```
cmp ax, 1024               ; Сравнение ax и 1024
jb @@20                    ; Переход, если ax < 1024
```

Описание

Все условные переходы действуют аналогично и поэтому перечислены вместе, для того чтобы их было легче просматривать. Кроме того, хотя некоторые из мнемоник представляют те же самые инструкции (например, je и jz), их мнемоники приведены отдельно. Как показывает табл. 16.4, некоторые установки флагов контролируют, был ли осуществлен переход. Целевым адресом условного перехода является смещение со знаком в пределах от –128 до +127 байт от адреса *следующей* инструкции. Только в системах 80386 смещение может находиться в пределах от –32768 до +32767 байт.

Пример демонстрирует, как использовать условный переход вслед за cmp, проверяющим значение регистра. Сравнивая ax с 1024 и вслед за этим выполняя jb, пример осуществляет переход к целевому адресу, если значение ax *ниже* 1024. Условия, использующие слова "выше" и "ниже" ("above" и "below" соответственно), относятся к сравнению без знака, а использующие слова "больше" и "меньше" ("greater" и "less") — к сравнению со знаками.

См. также

jmp

Таблица 16.4. **Обзор условных переходов**

Инструкция	Переход если...	Флаги
ja	Выше	(cf=0) & (zf=0)
jae	Выше или равно	(cf=0)
jb	Ниже	(cf=1)
jbe	Ниже или равно	(cf=1) | (zf=1)
jc	Перенос	(cf=1)
jcxz	cx равен 0	-
jecxz	ecx равен 0	- (только для 80386/486)
je	Равно	(zf=1)
jg	Больше	(sf=of) & (zf=0)
jge	Больше или равно	(sf=of)
jl	Меньше	(sf<>of)

Окончание таблицы 16.4

Инструкция	Переход если...	Флаги
jle	Меньше или равно	(sf<>of) | (zf=1)
jo	Переполнение	(of=1)
jp	Паритет	(pf=1)
jpe	Паритет четный	(pf=1)
jpo	Паритет нечетный	(pf=0)
js	Знак	(sf=1)
jz	Нуль	(zf=1)
jna	Не выше	(cf=1) | (zf=1)
jnae	Не выше или равно	(cf=0)
jnb	Не ниже	(cf=0)
jnbe	Не ниже или равно	(cf=0) & (zf=0)
jnc	Не перенос	(cf=0)
jne	Не равно	(zf=0)
jng	Не больше	(sf<>of) | (zf=1)
jnge	Не больше или равно	(sf<>of)
jnl	Не меньше	(sf=of)
jnle	Не меньше или равно	(sf=of) & (zf=0)
jno	Не переполнение	(of=0)
jnp	Не паритет	(pf=0)
jns	Не знак	(sf=0)
jnz	Не нуль	(zf=0)

jmp **Безусловный переход**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Переход в новое место

Синтаксис/Пример

```
jmp shortTarget                    jmp short @@10
jmp nearTarget                     jmp CloseBy
jmp farTarget                      jmp far OverThere
jmp regW;memW                      jmp bx
jmp memDW                          jmp [dword bx]
```

только для 80386/486

```
jmp regDW                          jmp ecx
```

Образец кода

```
or bx, bx                ; bx = 0?
jnz Continue             ; Переход, если bx<>0
jmp Exit                 ; Иначе переход к выходу
Continue
```

Описание Инструкция jmp приводит к тому, что программа продолжается с адреса, указанного в качестве смещения относительно инструкции, *следующей* за jmp. В ассемблерных программах Turbo Assembler вычисляет смещение относительно метки, указанной в качестве операнда, автоматически применяя наиболее эффективную из возможных форм инструкции. Очень редко имеются веские причины для ручного вычисления смещения.

При переходе по более высокому адресу следует использовать оператор SHORT, как в jmp SHORT Nearby, либо Turbo Assembler вставит лишние инструкции nop на тот случай, если потом окажется, что адрес находится далее чем в 128 байтах.

Вместо точно определенной метки можно указать целевой адрес в регистре либо ссылку к памяти. Процессоры 80386/486 допускают использование расширенных регистров для помещения в них 32-битовых смещений адресов. Это мощное средство особенно полезно при создании "таблиц перехода", содержащих списки ячеек, которым передается управление при определенных условиях.

См. также j-condition

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Копирует sf, zf, saf и cf в регистр ah

Синтаксис/Пример
```
lahf no operands                    lahf
```

Образец кода
```
lahf                    ; Загружает флаги в ah
test ah, 0Dh            ; Проверяет sf, zf, cf
jnz @@10                ; Переход, если любой из этих флагов =1
```

Описание lahf выполняется для загрузки пяти флагов sf(7), zf(6), af(4), pf(2) и cf(0) в младшие четыре бита регистра ah. Номера битов показаны в скобках. После выполнения lahf остальные биты в ah являются неопределенными и также могут измениться.

См. также sahf

lds — Загрузка указателя и ds

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Загружает указатель из памяти в регистр и в ds

Синтаксис/Пример
```
lds regW, memDW                     lds si, [bp + 4]
```
только для 80386/486
```
lds regDW, memFW                    lds edi, [bx]
```

Образец кода
```
push cs                 , Выталкивание сегмента
mov ax, offset var      , Загрузка смещения
push ax                 ; Выталкивание смещения
;
;
push bp                 ; Сохранение bp
mov bp, sp              ; Адресация стека посредством bp
lds si, [bp + 2]        ; Загрузка указателя в ds si
```

Описание lds используется для загрузки 16-битового регистра общего пользования (обычно si) и сегментного регистра ds 32-битовым указателем, хранящимся в памяти. Операндом memDW может быть любой из обычных режимов адресации, за исключением непосредственного адреса, который не допускается. Процессоры 80386/486 загружают 48-битовый указатель в расширенный 32-битовый регистр и в ds. Пример демонстрирует, как извлечь указатель, быть может, переданный подпрограмме посредством адреса в стеке. Первая часть примера выталкивает в стек значения сегмента cs и смещения переменной (не показана); вторая часть использует lds совместно с bp для загрузки ds:si значением указателя.

См. также lea, les, lfs, lgs, lss

lea — Загрузка эффективного адреса

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Загружает в регистр смещение адреса ссылки к памяти

Синтаксис/Пример
```
lea regW, memW                      lea bx, [bp + 2]
```
только для 80386/486
```
lea regW:regDW, memW:memDW                lea edi, [dword bp + 2]
```

Образец кода
```
DATASEG
array db 80 dup (0)
CODESEG
lea bx, [array + si]    ; Используйте это...
mov bx, offset array    ; ...вместо этих
add bx, si              ; двух строк

lea bx, [array+bp+si]   ; Используйте это...
mov bx, offset array    ; ...вместо этих
add bx, bp              ; трех строк
add bx, si
```

Описание lea применяется для загрузки смещения адреса, называемого также эффективным адресом, в регистр типа слова или двойного слова в системах 80386/486. При многократном использовании сложной ссылки к памяти или при необходимости загрузки регистра, обычно bx, адресом элемента таблицы (возможно, для применения совместно с инструкцией xlat) можно использовать lea для вычисления смещения адреса. Пример демонстрирует, как с помощью lea можно выполнить работу двух или трех инструкций. Первая строка кода выполняет ту же задачу, что и две строки под ней; четвертая кодовая строка выполняет работу последних трех строк.

См. также lds, les, lfs, lgs, lss

leave — Процедура leave

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
		*	*		-	-	-	-	-	-	-	-	-

Назначение Удаляет из стека область, выделенную для локальной переменной инструкцией enter

Синтаксис/Пример
```
leave no operands                    leave
```

Образец кода
```
PROC AnyProc
enter 6,0                ; Резервирование 6 байт для локальных переменных
; Код процедуры
leave                    ; Возвращение зарезервированной
                         ; области стека
ret                      ; Возврат в точку вызова
ENDP AnyProc
```

Описание Инструкция leave используется непосредственно перед инструкцией ret для возвращения области стека, ранее зарезервированной инструкцией enter в начале процедуры. Обычно компиляторы языков высокого уровня используют leave и enter при выполнении функций и процедур, но, конечно, вы можете использовать эти инструкции и в чисто ассемблерных программах. Leave выполняет два шага: mov sp, bp и pop bp, восстанавливая таким образом указатель стека и bp, которые были вытолкнуты в стек инструкцией enter.

См. также enter, ret

les — Загрузка указателя и es

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Загружает указатель из памяти в регистр и в es

Синтаксис/Пример
```
les regW, memDW                      les di, [bp +4]
```
только для 80386/486
```
les regDW, memFW                     les esi, [bx]
```

Образец кода
```
push ds                  ; Выталкивание сегмента
mov ax, offset var       ; Загрузка смещения
push ax                  ; Выталкивание смещения
;
;
push bp                  ; Сохранение bp
mov bp, sp               ; Адресация стека посредством bp
les di, [bp + 2]         ; Загрузка указателя в es:di
```

Описание les используется для загрузки 16-битового регистра общего пользования (обычно di) и сегментного регистра es 32-битовым указателем, хранящимся в памяти. Операндом memDW может быть любой из обычных режимов адресации, за исключением непосредственного адреса, который не допускается. Процессоры 80386/486 могут загружать 48-битовый указатель в расширенный 32-битовый регистр и в es. Пример демонстрирует, как установить es:di для указания на переменную, быть может, переданную подпрограмме посредством адреса в стеке. Первая часть примера выталкивает в стек значения сегмента ds и смещения переменной (не показана); вторая часть использует les совместно с bp для загрузки es:di значением указателя.

См. также lds, lea, lfs, lgs, lss

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Загружает указатель из памяти в регистр и в fs (lfs) или в gs (lgs)

Синтаксис/Пример

```
lfs regW, memDW                    lfs di, [bp + 4]
lfs regDW, memFW                   lfs,esi, [bx]
lgs regW, memDW                    lgs di, [bp + 4]
lgs regDW, memFW                   lgs esi, [bx]
```

Образец кода

```
push cs                  ; Выталкивание сегмента
push 0                   ; Выталкивание верхнего смещения
push offset var          ; Выталкивание нижнего смещения
:
:
push bp                  ; Сохранение bp
mov bp, sp               ; Адресация стека посредством bp
lgs edi, [bp + 2]        ; Загрузка указателя в gs:edi
```

Описание lfs и lgs используются для загрузки 16- или 32-битового указателя смещения плюс 16-битового значения сегмента адреса в любой 16- или 32-битовый регистр и в один из сегментных регистров fs или gs, доступных только в системах 80386/486. За исключением возможности загружать 48-битовые указатели, эти две инструкции аналогичны lds и les и, как правило, используются в процедурах для получения доступа к переменным, переданным подпрограммам по адресу в стеке.

См. также lds, lea, les, lss

lock — Запирание шины

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Назначает сигнал запирания шины для следующей инструкции

Синтаксис/Пример

```
lock no operands                   lock xchg [semaphore], al
```

Образец кода

```
; Замечание: не запускайте это!
mov dl, 1                ; Устанавливает dl в 1
@@10:
Lock xchg [semaphore], dl ; Обмен dl и памяти
or dl, dl                ; dl = 0?
jz @@10                  ; Переход до тех пор, пока dl не станет <> 0
```

Описание lock (замок) используется в качестве префикса инструкций, которые обращаются к памяти, разделяемой более, чем одним процессором. (PC обладают единственным процессором, поэтому lock редко используется в программном обеспечении PC.) Обычно lock служит префиксом xchg в системах 8086/88, movs, ins и outs — в системах 80286/386/486, adc, add, and, bt, btc, btr, bts, dec inc, neg, not, or, sbb, sub и xor — в системах 80386/486, когда один из операндов является ссылкой к памяти. Нет необходимости предварять xchg префиксом lock в системах 80286/386/486, которые делают это автоматически.

Гипотетический пример показывает типичное использование lock — установку флага, названного *semaphore* при подготовке для исключительного использования устройства или, быть может, ячеек памяти. Lock перед xchg предотвращает обращение двух процессоров к одному и тому же байту; таким образом, если dl равен 0, программа может спокойно выполняться, пока второй процессор, который выполняет аналогичную или даже ту же самую подпрограмму, выполнит паузу до тех пор, пока первый процессор снова не установит *semaphore* в ноль.

См. также xchg

lods lodsb lodsd lodsw — Загрузка цепочки значений

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Загрузка цепочки значений в аккумулятор

Синтаксис/Пример

```
lods [es:]memB                     lods [byte source]
lods [es:]memW                     lods [word es:si]
```

```
lodsb no operands                                    lodsb
lodsw no operands                                    lodsw
```

только для 80386/486

```
lods [es:]memDW                                      lods [dword source]
lodsd no operands                                    lodsd
```

Образец кода

```
mov si, offset string     ; Адресация цепочки посредством ds:si
mov cs, MaxCount          ; Максимальное число циклов, подлежащих выполнению
cld                       ; Автоувеличение si
@@10:
lodsb                     ; al <- [ds:si]; si <- si + 1
call Subroutine           ; Вызов процедуры
loop @@10                 ; Цикл до тех пор, пока cx не
                          ;  станет = 0
```

Описание

Операндом lods всегда является ds:si или, при переназначении сегмента, es:si. Даже если оператор обращается к метке по имени, все равно необходимо инициализировать si для адресации этой переменной — все, что может сделать Turbo Assembler, это проверить, действительно ли переменная, которую вы определили, находится в ожидаемом сегменте. В большинстве случаев вы будете пользоваться сходными мнемониками lodsb, lodsd (только для 80386/486) и lodsw для загрузки байтов, слов и двойных слов в al, ax и eax. При каждом выполнении lods, если df=0, si увеличивается; если df=1, si уменьшается.

Чаще всего инструкция используется в цикле, просматривающем цепочку значений, как это показано в примере. Регистр si инициализируется для адресации переменной, cx присваивает максимальное число циклов, подлежащих выполнению, а флаг df очищается, чтобы lods увеличивала si. Затем цикл загружает байты из ds:si в al, вызывая подпрограмму (не показана) до тех пор, пока cx не станет равным 0.

Можно предварять lods префиксами повторения, такими как repe, но в этом нет особого смысла, так как результатом будет загрузка в аккумулятор единственного значения — задача, которую проще выполнить с помощью других инструкций.

См. также cmpsb, cmpsd, cmpsw, ins, insb, insd, insw, movs, movsb, movsd, movsw, outs, outsb, outsd, outsw, scas, scasb, scasd, scasw, stos, stosb, stosd, stosw

loop — Цикл по cx

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Уменьшает cx, а затем осуществляет переход, если cx не равен 0

Синтаксис/Пример

```
loop shortTarget                                     loop StartLoop
```

Образец кода

```
jcxz @@20                 ; Пропуск цикла, если cx = 0
@@10:
call Subroutine           ; Вызов процедуры
loop @@10                 ; cx <- cx -1; переход, если cx <> 0
@@20:
```

Описание

Эта инструкция весьма удобна для создания циклов, повторяющихся заданное регистром cx число раз. При каждом выполнении цикла cx уменьшается на 1. Если при этом cx остается не равным 0, тогда осуществляется переход по целевому адресу цикла, который должен быть не выше 126 байт и не ниже 127 байт относительно loop. Поскольку loop уменьшает cx *перед* проверкой равенства cx=0, то если cx=0 в начале повторяющегося участка, этот участок выполнится 65536 раз. Чтобы избежать этого, следует предварить повторяющийся участок инструкцией jcxz, как это сделано в примере.

См. также jcxz, loope, loopz, loopne, loopnz

loope loopz — Цикл по cx, при выполнении условия равенства

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Уменьшает cx, а затем осуществляет условный переход, если cx не равен 0

Синтаксис/Пример

```
loope shortTarget                                    loope @@20
loopz shortTarget                                    loopz StartLoop
```

Образец кода

```
DATASEG
array    db     ABCDEFG, 0
arraySize = $-array
```

```
CODESEG
mov cx, arraySize          ; Присвоение размера множества cx
mov si, offset array       ; Адресация множества посредством ds:si
cld                        ; Автоувеличение si
@@10:
lods [byte array]          ; al<-[ds:si]; si<-si+1
cmp al, 32                 ; al=32?
loope @@10                 ; Переход, пока да и cx<>0
je AllBlank                ; Переход, если цепочка = все пробелы
dec si                     ; si адресует первый символ, отличный от пробела
```

Описание loope или loopz, обе представляющие одну и ту же инструкцию, используются для уменьшения cx и осуществления перехода к целевому адресу, если при этом cx остается не равным 0 и zf=1. Как и в случае loop, целевой адрес должен быть не далее чем на 126 байт позади и не более 127 байт впереди loope. Пример показывает, как использовать loope для просмотра байтового множества. Длина множества присваивается cx, адрес множества — si. Затем цикл, содержащий три инструкции, загружает байты множества в al, осуществляя переход к метке @@10: в соответствии с инструкцией loope, если cx не равен 0 и если предшествующая cmp обнаруживает 32 — значение пробела в коде ASCII. По завершении цикла je определяет, все ли символы в цепочке являются пробелами. Если нет, si уменьшается, указывая таким образом на первый символ, отличный от пробела.

См. также loop, loopne, loopnz

loopne loopnz — Цикл по cx, если условие равенства ложно

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Уменьшает cx, затем осуществляет условный переход, если cx не равно 0

Синтаксис/Пример
```
loopne shortTarget                    loopne @@Begin
loopnz shortTarget                    loopnz @@110
```

Образец кода
```
mov cx, arraySize          ; Присвоение размера множества cx
mov si, offset array + arraySize -1 ; Адресация конца множества
std                        ; Автоуменьшение si
@@10:
lods [byte array]          ; al<-[ds:si]; si<-si-1
cmp al, .                  ; al = .?
loopne @@10                ; Переход, пока нет и cx <> 0
jne Exit                   ; Переход, если ни одной . не найдено
inc si                     ; si адресует .
```

Описание Эти две мнемоники представляют одну и ту же инструкцию и действуют почти так же, как loope и loopz, за исключением того, что переход по целевому адресу осуществляется только тогда, когда после уменьшения cx он еще не равен 0 и zf=0. Пример использует loopne для обнаружения точки в цепочке имени файла, начиная просмотр с конца цепочки и осуществляя переход к Exit (не показан), если ни одной точки не обнаружено, или уменьшая si на символ точки, если он обнаружен.

См. также loop, loope, loopz

lss — Загрузка указателя и ss

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Загружает указатель из памяти в регистр и в ss

Синтаксис/Пример
```
lss regW, memDW                       lss si, [bp + 2]
lss reg DW, memFW                     lss edi, [bx]
```

Образец кода
```
mov [oldss], ss            ; Сохранение старого стекового сегмента
mov [oldsp], sp            ; и старого указателя стека
lss sp, [newstack]         ; Загрузка ss:sp новыми значениями
;
;
mov sp, [oldsp]            ; Восстановление sp
                           ; (прерывания отключены)
mov ss, [oldss]            ; Восстановление ss
```

Описание В системах 80386/486 lss используется для загрузки значений 16- или 32-битового указателя *смещения плюс 16-битового адреса сегмента в любой 16- или 32-битовый регистр* и в регистр стекового сегмента. Обычно значение смещения загружают в sp, но нет никаких ограничений в использовании lss для загрузки других регистров. Одним из способов применения lss является подхватывание адреса альтернативного стека, как демонстрирует пример.

См. также lds, lea, les, lfs

mov — *Перемещение данных*

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Перемещает данные между регистрами либо между регистрами и памятью

Синтаксис/Пример

```
mov al, memB                          mov al, [abyte]
mov ax, memW                          mov sx, [aword]
mov memB, al                          mov [abyte], al
mov memW, ax                          mov [aword], ax
mov regB:memB, regB:immB              mov dl, cl
mov regW:memW, regW:immW              mov [aword], 1024
mov regB, memB                        mov dl, [abyte]
mov regW, memW                        mov dx, [aword]
```

только для 80386/486

```
mov eax, memDW                        mov eax, [adword]
mov memDW,eax                         mov [adword], eax
mov regDW:memDW, regDW:immDW          mov edx, 99999
mov regDW, memDW                      mov edx, [adword]
```

Образец кода

```
DATASEG
var db 10 dup (0)          ; 10-байтовая переменная
CODESEG
mov bx, 0                  ; Инициализация bx в 0
mov cx, 10                 ; Инициализация cx в 10
@@10:
mov [byte var+bx], cl      ; Копирование cl в память
inc bx                     ; Увеличение указателя
loop @@10                  ; Цикл по cx
```

Описание Инструкция mov, вероятно, наиболее часто из всех инструкций используется в программах для 8086. Различные формы mov позволяют переносить байты, слова и двойные слова (только для 80386/486) между регистрами либо между регистрами и памятью, применяя все обычные режимы адресации памяти.

Имеется несколько ограничений в использовании mov, которые не бросаются в глаза в перечне возможного синтаксиса. Одним из них является направление mov справа налево, т.е. перенос значения второго операнда к первому. Значение второго операнда никогда не изменяется. Когда оба операнда являются регистрами, только один из этих операндов может быть сегментным регистром; следовательно, можно писать mov es,ax и mov [aword],ds, *но нельзя* писать mov ds,es. Когда одним операндом является сегментный регистр, для следующей инструкции прерывания отключаются, позволяя mov в sp следовать за mov в ss и устраняя опасность того, что сигнал прерывания поступит прежде, чем произойдет полная инициализация указателя стека ss:sp. Еще одним ограничением является то, что оба операнда не могут быть ссылками к памяти — все перемещения в памяти и из нее должны происходить через регистр. (См. movs для ознакомления с инструкцией, которая осуществляет перемещения между ячейками памяти.)

Когда одним из регистровых операндов является al, ax или eax (только для 80386/486), Turbo Assembler генерирует более быструю форму mov. Если аккумулятор свободен, следует использовать его в инструкциях mov для ускорения выполнения программы.

Пример показывает, как mov применяется для инициализации регистров, использованных здесь для хранения в переменной значений 10, 9, ..., 1. Другая инструкция mov копирует значение cl в память, применяя режим базовой адресации посредством bx.

См. также movs, lods, stos

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Перемещает цепочки значений непосредственно между двумя ячейками памяти

Синтаксис/Пример

```
movs memB, [es:]memB                    mos [var1], [var2]
movs memW, [es:]memW                    movs [var3], [es:;si]
movsb no operands                       movsb
movsw no operands                       movsw
```

только для 80386/486

```
movs memDW, [es:]memDW                  movs [edi], es:var4]
movsd no operands                       movsd
```

Образец кода

```
mov ax,@data            ; Инициализация ds по адресу
mov ds,ax               ; сегмента данных
mov es,ax               ; Устанавливает es = ds
ASSUME es:DGROUP        ; "Говорит" tasm, куда указывает es
mov si, offset string   ; Адресует исходную строку
mov di, offsey strcopy  ; Адресует назначение
mov cx, strlen          ; Присваивает счетчик cx
jcxz Exit               ; Не копирует, если cx = 0
cld                     ; Автоувеличение si, di
rep movsb               ; Копирование строки в strcopy
```

Описание Инструкция movs и сходные с ней формы movsb, movsd (только для 80386/486) и movsw перемещают одно значение в памяти непосредственно в другую ячейку памяти. Первым операндом всегда должен быть es:di, адресующий назначение перемещения. Вторым операндом должен быть ds:si или, при переуказании сегмента, es:si, адресующий источник перемещения. Расширенные 32-битовые регистры edi и esi могут использоваться в программах для 80386/486. Выполнение movs копирует один байт из исходной ячейки в ячейку назначения. После этого, если df=0, оба — и si, и di (или esi и edi), дополняются числом байтов, подлежащих перемещению. Назначения этих регистров фиксированы — даже если, как в некоторых примерах, вы специфицируете полностью определенные метки, которые Turbo Assembler проверит, чтобы убедиться, что переменные находятся в соответствующих сегментах. По-прежнему на вас лежит забота по загрузке di и si смещениями адресов переменных. Упрощенные формы movs не требуют никаких операндов. Различные мнемоники ничем не отличаются по своему действию.

Обычно movs предваряется префиксом rep, повторяющим инструкцию указанное в cx число раз. Как показывает пример, это дает возможность создавать мощные инструкции для перемещения блоков памяти с одного места на другое — в данном случае копируя string в strcopy. В качестве дополнительного напоминания инструкции, инициализирующие сегментные регистры, также показаны в примере. Эффективное использование movs (как и других цепочечных инструкций) требует тщательного планирования и управления сегментными регистрами.

См. также cmpsb, cmpsd, cmpsw, ins, insb, insd, insw, lods, lodsb, lodsd, lodsw, outs, outsb, outsd, outsw, rep scas, scasb, scasd, scasw,stos, stosb, stosd, stosw

movsx Перемещение и расширение значений со знаком

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Перемещает значения со знаком из меньших регистров и ячеек памяти в большие, расширяя знаковый бит

Синтаксис/Пример

```
movsx regW, regB:memB                   movsx dx, al
movsx regDW, regB:memB                  movsx eax, [abyte]
movsx regDW, regW:memW                  movsx edx, dx
```

Образец кода

```
mov al, -1              ; al = -1
mov dx, 0               ; dx = 00000h
movsx dx, al            ; dx = 0FFFFh

mov [abyte], -1         ; [abyte] = -1
mov eax, 0              ; eax = 000000000h
movsx eax, [abyte]      ; eax = 0FFFFFFFFh

mov ax, -1              ; ax =  -1
mov edx, 0              ; edx = 000000000h
movsx edx, ax           ; edx = 0FFFFFFFFh
```

Описание

В системах 80386/486 movsx используется для копирования имеющих знак значений, преобразуя их в значения с большим числом битов. Например, можно использовать movsx для загрузки регистра слова, такого как ax, значением байта из памяти, заставляя процессор автоматически инициализировать ah и расширяя знак скопированного значения. Назначением (первым операндом) должен быть регистр. Источником (вторым операндом) может быть регистр или ссылка к памяти. Пример демонстрирует, как использовать *movsx для переноса значений между несхожими регистрами.*

См. также

mov, movs, movzx

movzx — **Перемещение с расширением значения**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение

Перемещает значения без знака из меньших регистров и ячеек памяти в большие, обнуляя старшие биты

Синтаксис/Пример

```
movzx regW, regB¦memB                    movzx bx, [abyte]
movzx regDW, regB¦memB                   movzx edx, dl
movzx regDW, regW¦memW                   movzx edx, [aword]
```

Образец кода

```
mov al, -1              ; al = 1
mov dx, -1              ; dx = 0FFFFh
movzx dx, al            ; dx = 00001h

mov [abyte], 1          ; [abyte] = 1
mov eax, -1             ; eax = 0FFFFFFFFh
movzx eax, [abyte]      ; eax = 000000001h

mov ax, 1               ; ax = 1
mov edx, -1             ; edx = 0FFFFFFFFh
movzx edx, ax           ; edx = 000000001h
```

Описание

В системах 80386/486 movzx используется для копирования не обладающих знаком значений с меньшим числом битов — в большие регистры, аналогично тому, как используется movsx. Например, movzx может загрузить расширенный 32-битовый регистр, такой как ecx, значением слова из памяти, заставив процессор автоматически обнулить верхние 16 бит ecx. Назначением (первым операндом) movzx должен быть регистр. Источником (вторым операндом) может быть регистр или ссылка к памяти. Пример демонстрирует, *как использовать movzx для переноса значений между несхожими регистрами.*

См. также

mov, movs, movsx

mul — **Перемножение без знака**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	u	u	u	u	*

Назначение

Перемножает два значения, не обладающих знаками

Синтаксис/Пример

```
mul regB¦memB                            mul bl
mul regW¦memW                            mul [aword]
```

только для 80386/486

```
mul regDW¦memDW                          mul ebx
```

Образец кода

```
DATASEG
multiplicand dw 1024
multiplier  dw 32
answer      dw 0
CODESEG
mov ax, [multiplicand]    ; Загрузка умножаемого в ax
mul [multiplier]          ; dx:ax <- ax * multiplier
jc Exit                   ; Переход, если результат занимает
                          ;  больше 16 бит
mov [answer], ax          ; Иначе, сохранить ответ (answer)
```

Описание

Перемножение без знака в 8086 в целом сходно с перемножением со знаком (см. imul). Единственным операндом mul, представляющим множитель, должен быть регистр общего назначения либо ссылка к памяти. Размер множителя определяет размещение умножаемого и произведения. Если множителем является байт, тогда умножаемым — al, а произведение помещается в ax. Если множителем является слово, тогда умножаемое находится в ax, а произведение помещается в dx:ax, размещая в ax младшую часть резуль-

тата. Если множителем является двойное слово (только для 80386/486), тогда умножаемое — eax, а произведение помещается в edx:eax, размещая младшие 32 бита в eax. Переполнение регистра назначения невозможно.

После выполнения mul флаги of и cf могут использоваться для определения размера результата. Оба флага устанавливаются в 1, если произведение занимает больше битов, чем специфицированный источник; в противном случае — оба флага устанавливаются в 0. Если cf=1 после mul bx, тогда результат занимает все 32 бита двойного регистра dx:ax. Как показывает пример, можно проверить cf (или of) после mul, чтобы определить, больше ли результат по размеру, чем исходные операнды.

См. также imul

neg — Двоичное дополнение отрицание

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение Осуществляет инвертирование знака (формирует двоичное дополнение) значения

Синтаксис/Пример

```
neg regB¦memB                    neg [abyte]
neg regW¦memW                    neg ax
```

только для 80386/486

```
neg regDW¦memDW                  neg edx
```

Образец кода

```
mov ax, 6              ; Присвоение значений
mov dx, 8              ; ax и dx
sub ax, dx             ; ax <- ax-dx (ax = 0FFFEh)
jae @@10               ; Переход, если ax >= 0
neg ax                 ; Нахождение абсолютного значения (ax = 0002)
mov dl, -              ; Отображение знака минус
mov ah, 2              ; посредством функции DOS 2
int 21h                ; Вызов DOS
@@10:                  ; Продолжать отсюда
```

Описание neg применяется для формирования двоичного дополнения (в дополнительном коде) значения регистра или ячейки памяти. Когда исходное значение является отрицательным числом в форме двоичного дополнения, neg находит абсолютное значение этого числа. Инструкция действует, вычитая исходное значение из 0. Это действие эквивалентно изменению всех битов с 0 на 1, с 1 на 0 и прибавлению затем 1. Как показывает пример, если результат вычитания отрицателен — знак минуса может быть передан в стандартный выходной файл DOS, а результат в ax обращен данной командой в абсолютное значение, отображая, таким образом, полное отрицательное десятичное число.

См. также нет

nop — Нет операции

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Занимает один байт машинного кода, но не производит никакого действия

Синтаксис/Пример

```
nop no operands                  nop
```

Образец кода

```
jmp @@20               ; Переход к метке
nop                    ; Вставляется Turbo Assembler
;
;
@@20                   ;...если эта метка в пределах 128 байт
```

Описание Turbo Assembler вставляет инструкцию nop, чтобы зарезервировать байты в тех случаях, когда точный размер инструкции определяется кодом программы позднее. Например, переход jmp к метке, находящейся впереди, предположительно занимает три байта в длину. Но, если назначение jmp окажется в пределах 128 байт, ассемблер заменит jmp более эффективной 2-байтовой формой, оставляя ненужный третий байт равным nop. (Можно избежать этой ситуации, предваряя целевой адрес находящейся впереди метки оператором SHORT.) Еще одно использование nop проявляется при отладке. Если вы

хотите удалить инструкцию, то вместо того, чтобы останавливать отладчик, загружать свой редактор, производить изменение и реассемблировать программу — просто наберите несколько байтов nop (90h) перед этой инструкцией. Затем можно запустить программу и просмотреть влияние отсутствия этой инструкции — весьма полезная методика отладки.

Инструкция nop является идентичной инструкциям xchg ax,ax и xchg eax,eax (только для 80386/486), которые ассемблируются в машинный код как nop.

См. также xchg

not — Логическое отрицание

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Переключает все биты значения из 1 в 0 и из 0 в 1

Синтаксис/Пример

```
not regB|memB                          not dh
not regW|memB                          not dx
```

только для 80386/486

```
not regDW|memDW                        not [dword var]
```

Образец кода

```
DATASEG
false EQU 0                ; Значение, представляющее "ложно"
true EQU -1                ; Значение, представляющее "истинно"
flag db true               ; Инициализация флага на "истинно"
CODESEG
cmp [flag], false          ; Флаг = "ложно"?
je @@10                    ; Переход, если флаг = "ложно"
call Subroutine            ; Иначе, вызов подпрограммы
@@10:
not [flag]                 ; Переключение значения флага
```

Описание not используется для переключения всех битов значения из 1 в 0 и из 0 в 1. Это часто бывает полезно для переключения значения флага между "истинно" и "ложно", как в данном примере. (Подпрограмма, к которой происходит обращение, не показана.)

См. также neg

or — Логическое ИЛИ

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		0	-	-	-	*	*	u	*	0

Назначение Производит логическую операцию ИЛИ со значениями двух байтов, слов или двойных слов (только для 80386/486)

Синтаксис/Пример

```
or al, immB                            or al, 80h
or ax, immW                            or ax, 01h
or regB|memB, immB                     or bl, 0AAh
or regW|memW, immW                     or [word bx], 0800h
or regW|memW, immB                     or cx, 03h
or regB|memB, regB                     or [byte bx], dl
or regW|memW, regW                     or dx, dx
or regB, regB|memB                     or bl, bh
or regW, regW|memW                     or dx, [word bx]
```

только для 80386/486

```
or eax, immDW                          or eax, 080000000h
or regDW|memDW, immDW                  or edx, 0FFFF0000h
or regDW|memDW, immB                   or [dword bx], 01h
or regDW, regDW|memDW                  or ecx, [dword bx]
```

Образец кода

```
mov ax, 01234h             ; ax = 01234h
and ax, 000FFh             ; ax = 00034h
or ax, 08000h              ; ax = 08043h

or dx, dx                  ; dx = 0?
jz Target                  ; Переход, если dx = 0
                           ; Продолжать, если dx <> 0
```

Описание or используется для выполнения логического ИЛИ над битами значений любых двух байтов, слов или двойных слов (только для 80386/486), хранимых в регистрах или пере-

менных памяти. (Оба операнда не могут быть ссылками к памяти.) Соответствующий бит в первом операнде устанавливается в 1 только в том случае, если соответствующий бит в любом из двух операндов равен 1. Первая часть примера использует or для установки старшего типа ax в 1, после того как ah обнуляется предшествующей and посредством маски 000ffh.

Другим типичным использованием or является проверка равенства значения 0, как демонстрирует вторая часть примера. Произведение операции ИЛИ над одним значением устанавливает флаг нуля в 1 только в том случае, если все биты значения равны 0. Обратите внимание, что это также устанавливает в 0 и cf, и of — факт, который может оказаться полезным при определенных обстоятельствах.

См. также and, xor

out — Вывод в порт

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Вывод значения в порт

Синтаксис/Пример

```
out immB, al                                    out 14h, al
out dx, al                                      out dx, al
out immB, ax                                    out 01Fh, ax
out dx, ax                                      out dx, ax
```

только для 80386/486

```
out immB, eax                                   out 0Fh, eax
out dx, eax                                     out dx, eax
```

Образец кода

```
Ctrl8259 EQU 021h          ; 8259 маскирует порт
in al, Ctrl8259            ; Считывание подключенной маски 8259
and al, EnableIRQ          ; Очистка замаскированного бита
out Ctrl8259               ; Запись новой маски 8259
```

Описание Инструкция out записывает значение из al, ax или eax (только для 80386/486) в аппаратный порт. Как показывает пример, out часто используется совместно с инструкциями in и логическими инструкциями, такими как and и or, для проверки и изменения бита, осуществляющего переключение на различные адреса портов компьютера. (Это тот же пример, что показан для in.) Простейшая форма out записывает байт из al в непосредственный адрес порта в диапазоне 0–255. Для доступа к более высоким адресам портов следует специфицировать этот адрес в регистре dx.

См. также in, outs

outs outsb outsd outsw — Вывод цепочки значений в порт

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
		*	*		-	-	-	-	-	-	-	-	-

Назначение Выводит последовательность байтов, слов или двойных слов из памяти в порты

Синтаксис/Пример

```
outs dx, [es.]si:memB                           rep outs dx, [var]
outsdx, [es:]si:memW                            rep outs dx, [word var]
outsb no operands                               rep outsb
outsw no operands                               rep outsw
```

только для 80386/486

```
outs dx, regDW:memDW                            rep outs dx, [dword var]
utsd no operands                                rep outsd
```

Образец кода

```
; Замечание: Не запускайте это пример!
DATASEG
string db A string is a wonderful thing
slen = $-srtring
CODESEG
mov si, offset string      ; Адресует цепочку посредством ds:si
mov dx, <port number>      ; Присвоение номера порта dx
mov cx, slen               ; Присвоение длины цепочки cx
cld                        ; Автоувеличение si
rep outsb                  ; Пересылка цепочки в выходной порт
```

Описание Как и в случае всех цепочечных инструкций, назначения регистров outs (и ее кратких форм outsb, outsd, только для 80386/486, и outsw) фиксированы, даже если вы точно оп-

ределите метки адресов. Исходным регистром является ds:si, если только не будет использоваться переадресация через es:, как в [byte es:si] или [word es:var]. Номер порта должен помещаться в dx. (Не забывайте делать это и для кратких мнемоник.) Если df=0, тогда outs увеличивает si; если df=1, тогда outs уменьшает si на число байтов, которые должны быть пересланы в выходной порт с каждым выполнением outs. Обычно outs предваряется префиксом rep, повторяющим инструкцию определенное в cx число раз, как проиллюстрировано в примере.

См. также cmpsb, cmpsd, cmpsw, ins, insb, insd, insw, lods, lodsb, lodsd, lodsw, movs, movsb, movsd, movsw, scas, scasb, scasd, scasw, stos, stosb, stosd, stosw

pop — **Выталкивание из стека**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Выталкивает слово или двойное слово (только для 80386) из стека

Синтаксис/Пример

```
pop regW                                    pop ax
pop memW                                    pop [word var]
pop es ¦ ds ¦ ss                            pop es
```

только для 80386/486

```
pop regDW                                   pop ecx
pop memDW                                   pop [dword var]
pop fs ¦ gs                                 pop gs
```

Образец кода

```
push ax                 ; Сохранение ax в стеке
push bx                 ; Сохранение bx в стеке
;
;       Различные инструкции
;
pop bx                  ; Восстановление сохраненного значения bx
pop ax                  ; Восстановление сохраненного значения ax
push cs                 ; Выталкивание cs в стек
pop es                  ; Выталкивает es, устанавливая es = cs
```

Описание pop выполняется для удаления одного слова или двойного слова из области стека, адресуемой посредством ss:sp (или ss:esp в 80386/486). После копирования стекового значения в указанный регистр, sp (или esp) увеличивается на число перенесенных байтов. После этого следующим объектом перезаписи для pop становится значение, расположенное выше нового указателя стека. Наиболее распространенным использованием pop (см. первую часть примера) является восстановление значения регистра, ранее помещенного в стек посредством push. Другое использование pop — загрузка сегментного регистра, как во второй части примера, в которой es устанавливается равным cs. (Выталкивание в регистр cs является запретной операцией.) При выталкивании значения в сегментный регистр прерывания временно отключаются до *следующей* инструкции, давая, таким образом, возможность pop sp следовать за pop ss и устраняя опасность возникновения прерывания до того, как указатель стека будет полностью инициализирован.

Часто забывают о возможности выталкивать значения слов и двойных слов (для 80386/486) в ячейки памяти, используя все режимы адресации памяти. Таким образом, инструкции вида pop [aword+bx+si] и pop [aword+si] совершенно допустимы, хотя и являются несколько необычными командами.

См. также popa, popad, popf, popfd, push, pusha, pushad, pushf, pushfd

popa — **Выталкивание всех регистров основного назначения**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
		*	*		-	-	-	-	-	-	-	-	-

Назначение Выталкивает регистры di, si, bp, sp (отбрасывается), bx, dx, cx и ax из стека

Синтаксис/Пример

```
popa no operands                                    popa
```

Образец кода

```
PROC AnyProc
pusha                   ; Сохранение всех регистров основного назначения
;
;  Код процедуры
;
```

```
popa                    ; Восстановление всех регистров
                        ;   основного назначения
ret                     ; Возврат в точку вызова
ENDP
```

Описание popa используется в системах 80286 и 80386/486 для выталкивания из стека 16-битовых регистров di, si, bp, sp, bx, dx, cx и ax в указанном порядке. Хотя сохраненное значение sp и удаляется из стека, это значение не помещается в sp. Обычно popa используется после предшествующего выполнения pusha, вытолкнувшей в стек значения всех этих регистров (в обратном порядке). Инструкция использует 16 байт стекового пространства.

См. также pop, popad, popf, popfd, push, pusha, pushad, pushf, pushfd

popad — Выталкивание всех 32-битовых регистров основного назначения

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Выталкивает регистры edi, esi, ebp, esp (отбрасывается), ebx, edx, ecx и eax из стека

Синтаксис/Пример popad no operands — popad

Образец кода

```
pushad                  ; Сохранение всех 32-битовых регистров основного назна-
чения
;
; Другой код
;
popad                   ; Восстановление сохраненных регистров
```

Описание popad используется в системах 80386/486 для выталкивания из стека 32-битовых регистров edi, esi, ebp, esp, ebx, edx, ecx и eax в указанном порядке. Хотя сохраненное значение esp и удаляется из стека, это значение не помещается в esp. Обычно popad используется после предшествующего выполнения pushad, вытолкнувшей в стек значения этих же самых регистров (в обратном порядке). Инструкция использует 32 байта стекового пространства.

См. также pop, popa, popf, popfd, push, pusha, pushad, pushf, pushfd

popf — Выталкивание флагов

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	*	*	*	*	*	*	*	*

Назначение Удаляет из стека все флаги

Синтаксис/Пример popf no operands — popf

Образец кода

```
xor ax, ax              ; Устанавливает ax = 0000
push ax                 ; Выталкивание ax в стек
popf                    ; Выталкивает стек во флаги, устанавливая, таким образом,
                        ;  все флаги в 0
```

Описание Выполнение popf удаляет из стека верхнее слово и помещает биты этого слова во флаги 8086. Обычно это выполняется после предшествующего выполнения pushf, вытолкнувшей в стек значения флагов, возможно, для сохранения результата сравнения либо иной операции. Другим использованием popf является удаление флагов из стека в подпрограмме обслуживания прерывания. Можно также присвоить различные значения битов регистру слова, вытолкнуть этот регистр в стек, а затем выполнить popf для переноса битов во флаги, устанавливая их в новые значения.

См. также pop, popa, popad, popfd, push, pusha, pushad, pushf, pushfd

popfd — Выталкивание расширенных флагов

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
					*	*	*	*	*	*	*	*	*

Назначение Выталкивает из стека расширенные флаги 80386 за исключением vm и rf

Синтаксис/Пример popfd no operands — popfd

Образец кода

```
pushfd                  ; Сохранение расширенных флагов
;
; Другой код
;
popfd                   ; Восстановление расширенных флагов
```

Описание popfd выполняется для выталкивания из стека двух верхних слов и помещения соответствующих битов этих слов в регистр расширенных флагов. Обычно это выполняется после предшествующего выполнения pushfd, вытолкнувшей в стек значения расширенных флагов, возможно, для сохранения результата сравнения или другой операции. (См. popf о других возможных использованиях.) Флаги 80386/486 vm (виртуальный флаг 8086, бит 17) и rf (итоговый флаг, бит 16) посредством popfd не изменяются.

См. также pop, popa, popad, popf, push, pusha, pushad, pushf, pushfd

push **Выталкивание в стек**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Переносит значения в стек

Синтаксис/Пример

```
push regW                                   push ax
push memW                                   push [word bx]
push cs¦es¦ds¦ss                            push cs
```

только для 80286, 80386/486

```
push ¦mmB                                   push 0Fh
push ¦mmW                                   push 256
```

только для 80386/486

```
push regDW                                  push ecx
push memDW                                  push [dword bx]
push ¦mmDW                                  push 99999
push fs ¦ gs                                push gs
```

Образец кода

```
push ax                     ; Сохранение ax в стеке
push bx                     ; Сохранение bx в стеке
push cx                     ; Сохранение cx в стеке
;
; Другой код, изменяющий ax, bx, cx
;
pop cx                      ; Восстановление начального cx
pop bx                      ; Восстановление начального bx
pop ax                      ; Восстановление начального ax
P386
push 99999                  ; Turbo Assembler ошибочно отвергает это
;
db 066h, 068h               ; Но можно закодировать эту инструкцию
dd 99999                    ; данными двумя строками
```

Описание push используется для переноса слова или двойного слова (только для 80386) в стек. Выполнение push сначала уменьшает указатель стека на 2 (или на 4 в случае выталкивания двойного слова 80386). Затем значение указанного операнда копируется в область, адресуемую посредством ss:sp. Обратите внимание, что это вызывает увеличение стека в сторону нижних адресов памяти. Наиболее привычным использованием push является сохранение значения регистра в стеке, как демонстрирует первая часть примера. Позднее может быть использована pop для восстановления сохраненных значений.

Разрешается выполнять push, но не pop, со значением регистра кодового сегмента cs. Можно также выталкивать значения из памяти, используя все обычные режимы адресации. Таким образом, инструкции вида push [bx] и push [value + si] являются разрешенными, но часто пренебрегаемыми формами инструкций. Кроме того, процессоры 80286 и 80386/486 позволяют выталкивание непосредственных значений, например push 0. Недочет в Turbo Assembler 1.0 не допускает выталкивание 32-битовых непосредственных значений посредством инструкций вида push 99999, которые вызывают ошибку "constant too large" (постоянная слишком велика). Для того чтобы обойти эту, очевидно, временную проблему, используйте команды db и dd из второй части примера, чтобы вставить машинный код этой инструкции непосредственно в свою программу.

См. также pop, popa, popad, popf, popfd, pusha, pushad, pushf, pushfd

pusha **Выталкивание всех регистров основного назначения стек**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
		*	*		-	-	-	-	-	-	-	-	-

Назначение Переносит регистры ax, cx, dx, bx, sp, bp, si и di в стек

Синтаксис/Пример pusha no operands — pusha

Образец кода

```
PROC AnyProc
pusha                    ; Сохранение регистров основного назначения
;
; другой код
;
popa                     ; Восстановление регистров
ret                      ; Возврат в точку вызова
ENDP
```

Описание pusha используется для выталкивания в стек регистров ax, cx, dx, bx, bp, si и di в указанном порядке. Значением, выталкиваемым для sp, является значение sp, *предшествующее* выполнению pusha. (Это значение позднее отбрасывается командой popa, не оказывая никакого влияния на sp.) Обычно за pusha следует popa для восстановления сохраненных регистров, чаще всего в подпрограммах или в программе обслуживания прерываний.

См. также pop, popa, popad, popf, popfd, push, pushad, pushf, pushfd

pushad — Выталкивание всех 32-битовых регистров основного назначения в стек

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Переносит регистры eax, ecx, edx, ebx, esp, ebp, esi и edi в стек

Синтаксис/Пример pushad no operands — pushad

Образец кода

```
PROC AnyProc
pushad                   ; Сохранение 32-битовых регистров основного назначения
;
; другой код
;
popad                    ; Восстановление 32-битовых регистров
ret                      ; Возврат в точку вызова
ENDP
```

Описание pushad используется для выталкивания в стек регистров eax, ecx, edx, ebx, ebp, esi и edi в указанном порядке. Значением, выталкиваемым для esp, является значение esp, *предшествующее* выполнению pushad. (Это значение позднее отбрасывается popad, не оказывая никакого влияния на esp.) Обычно за pushad следует popad для восстановления сохраненных регистров, чаще всего в подпрограммах или в программе обслуживания прерываний.

См. также pop, popa, popad, popf, popfd, push, pusha, pushf, pushfd

pushf — Выталкивание флагов в стек

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Переносит флаги в стек

Синтаксис/Пример pushf no operands — pushf

Образец кода

```
or ax, ax                ; Проверка ax = 0?
pushf                    ; Сохранение результата сравнения
;
; другой код, который может модифицировать флаги
;
popf                     ; Восстановление результата "or"
jz Exit                  ; Переход, если ax был 0
```

Описание pushf выполняется для переноса 16-битового регистра флагов 8086 в стек. Выталкиваются все биты флага, в том числе и неиспользуемые. Это слово можно вытолкнуть в регистр основного назначения либо использовать popf для восстановления сохраненных битов флага, возможно для восстановления результата предыдущего сравнения.

См. также pop, popa, popad, popf, popfd, push, pusha, pushad, pushfd

pushfd — Выталкивание расширенного регистра флагов в стек

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Переносит расширенный регистр флагов 80386 в стек

Синтаксис/Пример

```
pushfd no operands                    pushfd
```

Образец кода

```
P386
pushfd                  ; Выталкивание расширенного регистра флагов
pop eax                 ; Копирование флагов в eax
```

Описание

pushfd выполняется для переноса 32-битового расширенного регистра флагов 80386/486 в стек. Выталкиваются все биты флага, в том числе и неиспользуемые. Это двойное слово можно вытолкнуть в регистр основного назначения либо использовать popfd для восстановления сохраненных битов флага, возможно, для восстановления результата предыдущего сравнения.

См. также

pop, popa, popad, popf, popfd, push, pusha, pushad, pushf

rcl — Вращение влево через перенос

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	-	-	-	-	*

Назначение

Вращает биты влево через флаг переноса

Синтаксис/Пример

```
rcl regB|memB, 1                      rcl al, 1
rcl regB|memB, cl                     rcl [abyte], cl
rcl regW|memW, 1                      rcl [aword], 1
rcl regW|memW, cl                     rcl bx, cl
```

только для 80286, 80386/486

```
rcl regB|memB, immB                   rcl dl, 4
rcl regW|memW, immB                   rcl [aword], 4
```

только для 80386/486

```
rcl regDW|memDW, 1                    rcl eax, 1
rcl regDW|memDW, cl                   rcl [dword bx], cl
rcl regDW|memDW, immB                 rcl ecx, 4
```

Образец кода

```
mov cl, 4               ; Присвоение счетчика вращения cl
rcl ax, cl              ; Вращает ax влево в соответствии со счетчиком в cl
```

Описание

rcl используется для циклической перестановки (вращения) битов в регистрах слова, байта, двойного слова (только для 80386/486) и в значениях памяти влево (к старшему биту — MSD), включая в первоначальное значение флаг переноса. Другими словами, старый MSD сдвигается в cf, который сдвигается к новому LSD (младшему биту), остальные биты сдвигаются на одну позицию влево. Повторение этого действия со временем восстановило бы первоначальное значение и cf.

Для всех процессоров второй операнд специфицирует число перестановок битов, подлежащих выполнению. В процессорах 8086 и 8088, если второй операнд является численным, то он должен быть равен 1. Чтобы осуществить вращение более, чем на один бит, необходимо присвоить счетчик поворота cl и указать этот регистр в качестве второго операнда. Процессоры 80286 и 80386/486 позволяют использовать в качестве второго операнда любые непосредственные значения, как, например, rcl cx,4 — для поворота cx на 4 бита влево. 80386/486 еще более расширяет эти формы, давая возможность осуществлять вращение 32-битовых расширенных регистров.

Когда вторым операндом является непосредственное значение 1, после выполнения rcl флаг of равен исключающему ИЛИ над cf и старшим битом (MSD) вновь повернутого значения. Таким образом, если флаг после rcl reg|mem,1 of=0, тогда старших два бита первоначального значения были либо 11, либо 00. Одним из способов использования этого факта является остановка вращения, как только бит 1 появляется в значении MSD. Например, если первоначальным значением в ax является 01000000b, выполнение rcl ax,1 приводит к тому, что cf=0 и ax=10000000b, что устанавливает of в 1, — условие, которое можно проверить посредством jo или jno. Во всех остальных случаях, когда второй операнд rcl не является непосредственным значением 1, флаг of не определен. Кроме того, если счетчик вращения равен 0, of и cf остаются неизменными — особенность, имеющая небольшое практическое значение.

См. также

rcr, rol, ror, sal, sar, shl, shr

rcr — Вращение вправо через перенос

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	-	-	-	-	*

Назначение

Вращает биты вправо через флаг переноса

Синтаксис/Пример

```
rcr regB|memB, 1                    rcr al, 1
rcr regB|memB, cl                   rcr [abyte], cl
rcr regW|memW, 1                    rcr [aword], 1
rcr regW|memW, cl                   rcr bx, cl
```

только для 80286, 80386/486

```
rcr regB|memB, immB                 rcr dl, 4
rcr regW|memW, immB                 rcr [aword], 4
```

только для 80386/486

```
rcr regDW|memDW, 1                  rcr eax, 1
rcr regDW|memDW, cl                 rcr [dword bx], cl
rcr regDW| memDW, immB              rcr ecx, 4
```

Образец кода

```
mov cl, 2               ; Присвоение счетчика вращения cl
rcr ax, cl              ; Вращает ax вправо в соответствии со счетчиком в cl
```

Описание

rcr используется для циклической перестановки (вращения) битов в регистрах слова, байта, двойного слова (только для 80386/486) и в значениях памяти вправо (к младшему разряду LSD), включая в первоначальное значение и флаг переноса. Другими словами, старый LSD сдвигается в cf, который сдвигается к новому MSD, остальные биты сдвигаются на одну позицию вправо. Повторение этого действия со временем восстановило бы первоначальное значение и cf.

Для всех процессоров второй операнд указывает число перестановок битов, подлежащих выполнению. В процессорах 8086 и 8088, если второй операнд является численным, то он должен быть равен 1. Чтобы осуществить вращение более, чем на один бит, необходимо присвоить счетчик поворота cl и специфицировать этот регистр в качестве второго операнда. Процессоры 80286 и 80386/486 позволяют использовать в качестве второго операнда любые непосредственные значения, как, например, в rcr dx, 3 для поворота dx на 3 бита вправо. 80386/486 еще более расширяет эти формы, давая возможность осуществлять вращение 32-битовых расширенных регистров.

Когда вторым операндом является непосредственное значение 1, после выполнения rcr флаг of равен исключающему ИЛИ над cf и MSD вновь повернутого значения. Таким образом, если после rcr reg | mem, 1 флаг of=1, тогда cf и первоначальный MSD были отличными; в противном случае они оба были равны либо 1, либо 0. of=1 указывает на изменение знака первоначального значения в результате вращения. Во всех остальных случаях, когда второй операнд rcr не является непосредственным значением 1, флаг of не определен. Кроме того, если счетчик вращения равен 0, of и cf остаются неизменными — особенность, имеющая небольшое практическое значение.

См. также rcl, rol, ror, sal, sar, shl, shr

rep repe repz — Повтор и повтор, пока выполненяется условие равенства

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Повторяет цепочечную инструкцию при выполнении условия

Синтаксис/Пример

```
rep movs|movsb|movsw                rep movs[byte di],[byte es si]
rep stos|stosb|stosw                rep stosw
repe mps|cmpsb|cmpsw                repe cmps [word str1],[word str2]
repz cmps|cmpsb|cmpsw               repz cmpsb
repe scas|scasb|scasw               repe scasw
repz scas|scasb|scasw               repz scas [byte es.var]
```

только для 80286, 80386/486

```
rep ins|insb|insw                   rep insb
rep outs|outsb|outsw                rep outs dx, [word es:si]
```

только для 80386/486

```
rep movs|movsd                      rep movs [dword edi],[dword esi]
rep stos|stosd                      rep stosd
rep ins|insd                        rep ins [dword var], dx
rep outs|outsd                      rep outs dx, [dword si]
repe cmps|cmpsd                     repe cmpsd
repz cmps|cmpsd                     repz cmps [dword str1],[dword str2]
repe scas|scasd                     repe scasd
repz scas|scasd                     repz scas [dword es;esi]
```

Образец кода

```
UDATASEG
```

```
string db 80 dup (?)        ; Неинициализированная переменная
strlen = $ - string         ; Длина цепочки
CODESEG
mov ax, @data               ; Инициализация сегментного
mov es, ax                  ;  регистра es
ASSUME es:DGROUP            ; Сообщает tasm, куда указывает es
mov di, offset string       ; Адресация цепочки посредством es:di
mov cx, strlen              ; Присвоение длины цепочки cx
cld                         ; Автоувеличение di
mov al,                     ; Присвоение значения ASCII al
rep stosb                   ; Заполнение цепочки пробелами
```

Описание

Три мнемоники (rep, repe и repz) представляют один и тот же префикс инструкции, который может быть присоединен к любой цепочечной инструкции, как показано в примерах и образце кода. Хотя эти мнемоники и являются идентичными, их действия отличны в зависимости от инструкции, которую они предваряют. rep следует использовать перед movs, stos, ins, outs и перед краткими мнемониками этих инструкций, repe и repz — перед cmps и scas и их краткими эквивалентами.

Префикс rep повторяет выполнение цепочечной инструкции cx раз; repe и repz также повторяют выполнение цепочечной инструкции в соответствии со значением, cx, но заканчивают повторение, если после очередной итерации zf=0. Таким образом, можно использовать эти два префикса для повторения цепочечного сравнения или просмотра определенное число раз либо до тех пор, пока цепочечная инструкция не обнаружит определенное значение. Инструкция lods (и ее краткие мнемоники) также может повторяться, но в этом никогда нет смысла. (Результатом повторения lods является загрузка аккумулятора одним значением после того, как все повторения закончатся — нет никакого способа использовать промежуточные загружаемые значения.)

Просмотрите различные цепочечные инструкции в данной главе для более подробного ознакомления с деталями и операндами, которые можно использовать с такими инструкциями, как cmps (для краткости они здесь не повторяются). И еще, хотя префиксы повтора, приведенные здесь, не изменяют никаких флагов, помните, что цепочечные инструкции, следующие за префиксами, могут изменять установку флагов.

См. также cmps, ins, movs, outs, repne, repnz, scas, stos

repne repnz — Повтор, пока не выполненяется условие равенства

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение *Повторяет инструкцию сравнения или просмотра цепочек при выполнении условия*

Синтаксис/Пример

```
repne cmps|cmpsb|cmpsw                  repne cmps [word str1],[word str2]
repnz cmps|cmpsb|cmpsw                  repnz cmpsb
repne scas|scasb|scasw                  repne scasw
repnz scas|scasb|scasw                  repnz scas [byte es:var]
```

только для 80386/486

```
repne cmps|cmpsd                        repne cmpsd
repnz cmps|cmpsd                        repnz cmps [dword str1], [dword str2]
repne scas|scasd                        repne scasd
repnz scas|scasd                        repnz scas [dword es:esi]
```

Образец кода

```
DATASEG
string db Thisstringhasn tanyblanks
strlen = $ - string
CODESEG
mov ax, @data               ; Инициализация сегментного
mov es, ax                  ; регистра es
ASSUME es:DGROUP            ; Сообщает tasm, куда указывает es
mov di, offset string+strlen-1  ; Адресация конца цепочки
mov cx, strlen              ; Присвоение длины цепочки cx
std                         ; Автоуменьшение di
mov al,                     ; Искомое значение
repne scasb                 ; Просмотр на наличие пробелов
jcxz Exit                   ; Exit, если пробелов не найдено es:di адресует последний
                            ;  символ, отличный от пробела
```

Описание

Префиксы repne и repnz, оба представляющие один и тот же машинный код, повторяют цепочечную инструкцию cmps или scas (а также их краткие мнемоники) cx раз, но заканчивают повторение раньше, если очередная итерация устанавливает zf=1. Для более под-

робного ознакомления просмотрите те заметки относительно repe и repz, которые действуют аналогично, но распознают противоположное значение флага zf. Пример демонстрирует, как использовать repne для просмотра цепочки от конца к началу, оставляя di адресующим последний символ в цепочке, отличный от пробела или, если ни одного пробела не было обнаружено, осуществляя переход к метке Exit (не показана).

См. также cmps, ins, movs, outs, rep, repe, repz, scas, stos

repz **Повтор, пока равенство равно нулю**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

См. rep repe

ret retf retn **Возврат дальний и ближний**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Осуществляет возврат из процедуры (подпрограммы)

Синтаксис/Пример

```
ret no operands                    ret
retn no operands                   retn
retf no operands                   retf
ret immW                           ret 8
retn immW                          retn 16
retf immW                          retf 4
```

Образец кода

```
PROC AnyProc
;
; Код процедуры
;
ret                    ; Возврат в точку вызова
ENDP AnyProc
```

Описание Эти три мнемоники ret типично используются в качестве завершающей инструкции, активированной оператором call. И ret, и retn, являющиеся синонимами для одной и той же инструкции, выталкивают в регистр ip 16-битовый адрес возврата, продолжая программу с инструкции, следующей за call, которая ранее вытолкнула этот адрес в стек перед активизацией процедуры. Инструкция retf выталкивает из стека два слова, присваивая первое слово cs, а второе — ip. Таким образом, программа продолжается в другом кодовом сегменте; retf следует применять, только если было произведено *дальнее* обращение к подпрограмме, обычно с применением инструкции call FAR AnyProc.

При использовании упрощенных моделей памяти (как в большинстве примеров программ этой книги), вероятно, лучше использовать только ret. Это дает возможность Turbo Assembler решать, ассемблировать код для retf или для retn, а также использовать соответствующую инструкцию call. Можно принудительно выбрать ближние и дальние обращения и возвраты, но помните, что применение retf тогда, когда следовало бы использовать retn, несомненно, рано или поздно приведет к разрушению программы.

Можно вслед за любой из этих трех мнемоник помещать значение без знака, которое будет прибавлено к указателю стека *после* того, как вытолкнется адрес возврата. Языки высокого уровня, такие как Pascal, используют эту форму ret для завершения процедур и функций, которым передаются параметры через стек. Подгонка указателя стека посредством ret позволяет процедуре самой удалять стековые параметры, вместо того чтобы предоставлять это вызывающему коду. Поскольку значение, прибавляемое к ret, является непосредственным (фиксированным), этот метод не очень полезен в таких языках, как C, которые позволяют передавать функциям переменное число параметров.

См. также call

rol **Циклический сдвиг влево**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	-	-	-	-	*

Назначение Циклически сдвигает биты значения влево

Синтаксис/Пример

```
rol regB|memB, 1                   rol al, 1
```

```
rol regB|memB, cl                    rol [abyte], cl
rol regW|memW, 1                     rol [aword], 1
rol regW|memW, cl                    rol bx, cl
```

только для 80286, 80386/486

```
rol regB|memB, immB                  rol dl, 4
rol regW|memW, immB                  rol [aword], 4
```

только для 80386/486

```
rol regDW|memDW, 1                   rol eax, 1
rol regDW|memDW, cl                  rol [dword bx], cl
rol regDW| memDW, immB               rol ecx, 4
```

Образец кода

```
mov cl, 5                ; Загружает счетчик в cl
rol [aword], cl          ; Поворачивает слово влево 5 раз
```

Описание

rol используется для циклической перестановки битов в регистрах слова, байта, двойного слова (только для 80386/486) и в значениях памяти влево (к старшему разряду — MSD). Старое значение MSD сдвигается в LSD, остальные биты сдвигаются на одну позицию влево. Кроме того, старый MSD копируется в cf. Повторение этого действия со временем восстановило бы первоначальное значение, но не обязательно восстановит cf. (Это почти идентично тому, как действует rcl, за исключением того, что cf не участвует в сдвиге в качестве дополнительного бита.)

Для всех процессоров второй операнд специфицирует число перестановок битов, подлежащих выполнению. В процессорах 8086 и 8088, если второй операнд является численным, он должен быть равен 1. Чтобы осуществить вращение более, чем на один бит, необходимо присвоить счетчик поворота cl и указать этот регистр в качестве второго операнда. Процессоры 80286 и 80386/486 позволяют использовать в качестве второго операнда любые непосредственные значения, как, например, в команде rol dx,2 для поворота dx на 2 бита влево. 80386/486 еще более расширяет эти формы, давая возможность осуществлять вращение 32-битовых расширенных регистров.

Когда вторым операндом является непосредственное значение 1, после выполнения rol флаг of равен исключающему ИЛИ над cf и MSD вновь повернутого значения. (Для ознакомления с более подробным описанием этого флага см. rcl). Во всех остальных случаях, когда второй операнд rol не является непосредственным значением 1, флаг of не определен. Кроме того, если счетчик вращения равен 0, of и cf остаются неизменными — особенность, имеющая небольшое практическое значение. Гораздо полезнее может оказаться то, что после каждого rol cf равен LSD только что повернутого значения.

См. также rcl, rcr, ror, sal, sar, shl, shr

ror — Циклический сдвиг вправо

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	-	-	-	-	*

Назначение Циклически сдвигает биты значения вправо

Синтаксис/Пример

```
ror regB|memB, 1                     ror al, 1
ror regB|memB, cl                    ror [abyte], cl
ror regW|memW, 1                     ror [aword], 1
ror regW|memW, cl                    ror bx, cl
```

только для 80286, 80386/486

```
ror regB|memB, immB                  ror dl, 4
ror regW|memW, immB                  ror [aword], 4
```

только для 80386/486

```
ror regDW|memDW, 1                   ror eax, 1
ror regDW|memDW, cl                  ror [dword bx], cl
ror regDW| memDW, immB               ror ecx, 4
```

Образец кода

```
mov cl, 8                ; Загрузка счетчика в cl
ror ax, cl               ; Поворачивает ax вправо 8 раз (Замечание: это равносильно
                         ; xchg ah, al!)
```

Описание

ror используется для циклической перестановки битов в регистрах слова, байта, двойного слова (только для 80386/486) и в значениях памяти вправо (к младшему разряду — LSD). Старое значение LSD сдвигается в старший значащий разряд — MSD, остальные биты сдвигаются на одну позицию вправо. Кроме того, старый LSD копируется в cf. Повторение этого действия со временем восстановило бы первоначальное значение, но

не обязательно восстановило бы cf. (Это почти идентично действию rcr, за исключением того, что cf не участвует во вращении в качестве дополнительного бита.)

Для всех процессоров второй операнд специфицирует число перестановок битов, подлежащих выполнению. В процессорах 8086 и 8088, если второй операнд является численным, он должен быть равен 1. Чтобы осуществить вращение более, чем на один бит, необходимо присвоить счетчик поворота cl и указать этот регистр в качестве второго операнда. Процессоры 80286 и 80386/486 позволяют использовать в качестве второго операнда любые непосредственные значения, как, например, в команде ror dx,4 для сдвига dx на 4 бита вправо. 80386/486 еще более расширяет эти формы, давая возможность осуществлять вращение 32-битовых расширенных регистров.

Когда вторым операндом является непосредственное значение 1, после выполнения ror флаг of равен исключающему ИЛИ над cf и MSD вновь повернутого значения. Таким образом, если после ror reg | mem, 1 флаг of=1, значит, первоначальные LSD и MSD были различны; в противном случае эти два крайние бита значения были оба равны 1 или 0. Во всех остальных случаях, когда второй операнд ror не является непосредственным значением 1, флаг of не определен. Кроме того, если счетчик вращения равен 0, of и cf остаются неизменными — особенность, имеющая небольшое практическое значение. Полезнее может остаться факт, что после каждого ror cf равен MSD только что повернутого значения.

См. также rcl, rcr, rol, sal, sar, shl, shr

sahf — Сохранение регистра ah во флагах

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	*	*	*	*	*

Назначение Копирует биты 7, 6, 4, 2 и 0 из ah в помеченные флаги

Синтаксис/Пример

```
sahf no operands                                    sahf
```

Образец кода

```
xor ah, ah              ; Обнуление ah
sahf                    ; Обнуление sf, zf, af, pf, cf
```

Описание sahf выполняется для сохранения битов из ah в 5 флагах. Влияние оказывается на следующие флаги (номера битов указаны в скобках): sf(7), zf(6), af(4), pf(2) и cf(0). На другие флаги воздействие не оказывается. Эта инструкция иногда используется совместно с математическим сопроцессором.

См. также lahf

sal — Арифметический сдвиг влево

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	*	*	u	*	*

Назначение Сдвигает биты влево

Синтаксис/Пример

```
sal regB|memB, 1                                    sal [abyte], 1
sal regB|memB, cl                                   sal ax, cl
sal regW|memW, 1                                    sal dx, 1
sal regW|memW, cl                                   sal [aword + bx], cl
```

только для 80286, 80386/486

```
sal regB|memB, immB                                 sal cx, 8
sal regW|memW, immB                                 sal [word + 4], 4
```

только для 80386/486

```
sal regDW|memDW, 1                                  sal edx, 1
sal regDW|memDW, cl                                 sal [dword es:di], cl
sal regDW| memDW, immB                              sal [dword bx], 4
```

Образец кода

```
DATASEG
value dd 12345678       ; Значение двойного слова,
CODESEG                 ;  подлежащее умножению на 2
shl [word value], 1     ; Сдвиг младшего слова
rcl [word value + 2], 1 ; Сдвиг старшего слова
jc Exit                 ; Переход при переполнении
```

Описание Мнемоники sal и shl — синонимы для одной и той же инструкции и генерируют идентичный машинный код. Обычно sal используется для умножения значений без знака на

2, а shl — для простого сдвига битов на одну позицию влево. Применение sal обеспечивает лучшую "читабельность" программы, указывая математический сдвиг, но можно использовать обе мнемоники.

Выполнение sal или shl сдвигает старший бит (MSD) значения во флаг переноса. Младший бит (LSD) становится равным 0. Повторение этого действия со временем устанавливает все биты в указанном регистре или ячейке памяти в 0.

Когда вторым операндом является непосредственное значение 1, после сдвига of=1 только в том случае, если новый cf не равен новому MSD. Если of=0, то биты нового cf и нового MSD различаются. Можно использовать этот факт для выявления сдвига нуля в позицию MSD первоначально ненулевого значения. Когда второй операнд не является непосредственным значением 1 — of не определен.

Пример показывает, как использовать сдвиг и вращение слова (см. rcl) для умножения значения двойного слова на 2. Начальный shl сдвигает младшее слово, копируя MSD в cf. Затем rcl поворачивает старшее слово, сдвигая cf в новый LSD (старшего слова). Последующие инструкции rcl могли быть подключены для сдвига даже больших многобайтовых значений. Если после завершающего rcl флаг переноса равен 1, значит, происходит переполнение.

См. также rcl, rcr, rol, ror, sar, shl, shr

sar — Арифметический сдвиг вправо

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	*	*	u	*	*

Назначение Сдвигает биты вправо

Синтаксис/Пример

```
sar regB¦memB, 1                          sar bl, 1
sar regB¦memB, cl                         sar ch, cl
sar regW¦memW, 1                          sar [aword], 1
sar regW¦memW, cl                         sar [word bx], cl
```

только для 80286, 80386/486

```
sar regB¦memB, ¦mmB                       sar [byte bp + 2], 4
sar regW¦memW, ¦mmB                       sar dx, 4
```

только для 80386/486

```
sar regDW¦memDW, 1                        sar [dword bp - 8], 1
sar regDW¦memDW, cl                       sar eax, cl
sar regDW¦ memDW, ¦mmB                    sar edx, 16
```

Образец кода

```
DATASEG
value dw 08000h             ; -32768
CODESEG
mov cl, 4                   ; Присвоение счетчика сдвига cl
sar [value], cl             ; Value = -2048 (-32768/16)
```

Описание В отличие от sal и shl, которые являются синонимами, sar не синоним для shr. Это часто сбивает с толку, но имеются веские причины для этого очевидного противоречия. Инструкция sar сдвигает значение регистра или ячейки памяти вправо, копируя младший бит (LSD) старого значения в cf, но, в отличие от shr, sar *не изменяет бит старого MSD*. При этом действии начальный знак сдвинутого значения остается неизменным; следовательно, можно использовать sar для выполнения деления на 2 целых чисел со знаком, в то время как shr может осуществлять деление только целых чисел без знака. Пример демонстрирует, как это происходит, используя sar для деления –32768 на 16 или на 2.

Когда вторым операндом sar является непосредственное значение 1, of устанавливается в 0. Когда второй операнд не является непосредственным значением 1, влияние на of не определено.

При использовании sar для деления отрицательных значений, представленных в форме двоичного дополнения, на 2, имейте в виду, что –1 (0FFFFh, например), деленное на 2, равно –1, а не 0. Некоторые справочники ссылаются на этот эффект "усечения в сторону отрицательной неопределенности", предполагая, что sar не во всех случаях дает тот же ответ, что и idiv на 2, которая дает 0 для –1/2 (т.е. "усекая в сторону нуля").

См. также rcl, rcr, rol, ror, sal, shl, shr

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение

Вычитает целые числа, принимая во внимание возможное заимствование от предыдущей sub или sbb

Синтаксис/Пример

```
sbb al, immB                          sbb al, 8
sbb ax, immW                          sbb ax, 256
sbb regB|memB, immB                   sbb [byte bx], 4
sbb regW|memW, immW                   sbb [word si], 600
sbb regW|memW, immB                   sbb dx, 3
sbb regB|memB, regB                   sbb ah, al
sbb regW|memW, regW                   sbb dx, ax
sbb regB, regB|memB                   sbb cl, [byte bp + 4]
sbb regW, regW|memW                   sbb ax, bx
```

только для 80386/486

```
sbb eax, immDW                        sbb eax, 35
sbb regDW|memDW, immB                 sbb ecx, 4
sbb regDW|memDW, immDW                sbb [dword bx], 18
sbb regDW|memDW, regDW                sbb [dword bx + si], eax
sbb regDW, regDW|memDW                sbb edx, [dword bp + 6]
```

Образец кода

```
DATASEG
v1 dd 87654321          ; Значение двойного слова
v2 dd 12345678          ; Значение, вычитаемое из v1
CODESEG
mov ax, [word v2]       ; Получение младшего слова v2
mov dx, [word v2 + 2]   , Получение старшего слова v2
sub [word v1], ax       , Вычитание младших слов
sbb [word v1 + 2], dx   , Вычитание старших слов с заимствованием
```

Описание

После выполнения sub или sbb над значениями нескольких байтов, слов или двойных слов (только для 80386/486) используется sbb для вычитания старших частей значений, принимая во внимание возможное заимствование. Если вы не вычитаете таким способом значения, состоящие из нескольких частей, всегда используйте вместо этого sub, которая не принимает во внимание заимствование.

Обычно sbb используется, как в примере, для вычитания двух больших целых чисел, в данном случае — двух двойных слов, обозначенных v1 и v2. Сначала программа загружает ax и dx значением v2. Затем sub вычитает младшие слова и sbb завершает вычитание, вычитая старшие слова и учитывая возможное заимствование при выполнении sub. (Замечание: в системах 80386/486 значения двойных слов могут вычитаться непосредственно.)

См. также

sub

scas scasb scasd scasw — Просмотр цепочки

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение

Просматривает цепочку для отыскания особых значений

Синтаксис/Пример

```
scas memB                             scas [byte di]
scas memW                             scas [word string]
scasb no operands                     scasb
scasw no operands                     scasw
```

только для 80386/486

```
scas memDW                            scas [dword string]
scasd no operands                     scasd
```

Образец кода

```
DATASEG
string db 2Bh or not 2Bh  ; Цепочка
strlen = $ - string       ; Длина цепочки
CODESEG
mov ax, @data             ; Инициализация es для
mov es, ax                ; адресации сегмента данных
ASSUME es:DGROUP          ; Сообщает tasm, куда указывает es
mov di, offset string     ; Адресация цепочки посредством es:di
mov cx, strlen            ; Присвоение cx длины цепочки
```

```
mov al,                    ; Присвоение al искомого значения
repne scasb                ; Просмотр на предмет первого пробела
;-----теперь di адресует "o" в "or"
```

Описание

Как и в случае всех цепочечных инструкций, назначения регистров фиксированы для scas и ее кратких эквивалентных форм scasb, scasd (только для 80386/486) и scasw. Инструкция вычитает значение байта, слова или двойного слова, адресуемое посредством es:di, и обычно используется с префиксами повтора repe и repne для просмотра переменных на предмет особых значений. Как и в случае cmp, результат вычитания отбрасывается — сохраняются только флаги. Значения байтов вычитаются из al, значения слов — из ax. В системах 80386/486 значения двойных слов, адресуемых либо посредством es:di, либо посредством es:edi, вычитаются из eax. Переполнение сегмента не допускается; следовательно, значения цепочек должны храниться в сегменте, адресуемом посредством es. После выполнения scas, если df=0, то di (или edi) увеличивается в размере в зависимости от специфицированного операнда — на 1 для байтов, на 2 для слов и на 4 для двойных слов. Если df=1, тогда di (или edi) уменьшается в зависимости от размера операнда.

Пример использует scasb для просмотра цепочки символов, отыскивая первый символ пробела. После выполнения этого кода, если cx равен 0, значит, ни одного пробела не было найдено. Для поиска первого символа, *не* соответствующего значению в al, нужно было бы использовать префикс повтора repe вместо repne.

См. также

cmpsb, cmpsd, cmpsw, ins, insb, insd, insw, lods, lodsb, lodsd, lodsw, movs, movsb, movsd, movsw, outs, outsb, outsd, outsw, stos, stosb, stosd, stosw

set-условие — Установка байта по условию

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение

Сохраняет значение байта в регистре или в памяти, если специфицированное условие истинно (сохраняемый байт = 1) либо ложно (сохраняемый байт = 0).

Синтаксис/Пример

```
setусловие regB|memB                    setae al
```

Образец кода

```
DATASEG
bits db 01101001b
bytes db 8 dup (?)
CODESEG
P386
mov bx, offset bytes       ; Адресация байтов посредством ds:bx
mov ah, [bits]             ; Загрузка упакованных битов в ah
mov cx, 8                  ; Присвоение cx счетчика цикла
@@10:
shl ah, 1                  ; Сдвиг одного бита в cf
setc [byte bx]             ; Установка или переустановка распакованного байта
loop @@10                  ; cx<-cx-1; переход, если cx<>0
```

Описание

В системах 80386/486 вслед за инструкцией cmp можно помещать любую из инструкций set-условие, приведенных в табл. 16.5. Можно также использовать эти инструкции после test либо любого другого кода, воздействующего на различные флаги. Если условие, указанное в центральной колонке таблицы, совпадает с установками флагов, приведенных справа, тогда значение целевого байта регистра или памяти устанавливается в 1, показывая "истинно"; в противном случае целевой байт устанавливается в 0. Эти условия являются отражением тех, которые поддерживаются инструкциями условного перехода (см. j-условие в данной главе), за исключением jcxz и jecxz.

Пример демонстрирует, как использовать setc для распаковки битов в байте. При каждом прохождении цикла, если инструкция shr сдвигает 1 в cf, то setc устанавливает байт [bx] в 1; в противном случае setc переустанавливает байт в 0. После завершения этого цикла неинициализированная байтовая переменная содержит восемь значений: 00 01 01 00 00 01.

См. также

j-условие

Таблица 16.5. **Краткий перечень инструкций set-условие**

Инструкция	Установка байта в 1, если... иначе, установка байта в 0	Флаги
seta	above (выше)	(cf = 0) & (zf = 0)
setae	above or equal (выше или равно)	(cf = 0)
setb	below (ниже)	(cf = 1)
setbe	below or equal (ниже или равно)	(cf = 1) | (zf = 1)
setc	carry (перенос)	(cf = 1)
sete	equal (равно)	(zf = 1)
setg	greater (больше)	(sf = of) & (zf = 0)
setge	greater or equal (больше или равно)	(sf = of)
setl	less (меньше)	(sf <> of)
setle	less or equal (меньше или равно)	(sf<>of) | (zf = 1)
seto	overflow (переполнение)	(of = 1)
setp	parity (паритет)	(pf = 1)
setpe	parity even (паритет четный)	(pf = 1)
setpo	parity odd (паритет нечетный)	(pf = 0)
sets	sign (знак)	(sf = 1)
setz	zero (ноль)	(zf = 1)
setna	not above (не выше)	(cf = 1) | (zf = 1)
setnae	not above or equal (не выше или равно)	(cf = 1)
setnb	not bellow (не ниже)	(cf = 0)
setnbe	not bellow or equal (не ниже или равно)	(cf = 0) & (zf = 1)
setnc	not carry (нет переноса)	(cf = 0)
setne	not equal (не равно)	(zf = 0)
setng	not greater (не больше)	(sf <> of) | (zf=1)
setnge	not greater or equal (не больше или равно)	(sf <> of)
setnl	not less (не меньше)	(sf = of)
setnle	not less or equal (не меньше или равно)	(sf = of) & (zf = 0)
setno	not overflow (не переполнение)	(of = 0)
setnp	not parity (не паритет)	(pf = 0)
setns	not sign (не знак)	(sf = 0)
setnz	not zero (не нуль)	(zf = 0)

shl **Сдвиг влево**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	*	*	u	*	*

Назначение Сдвигает биты влево

Синтаксис/Пример

```
shl regB|memB, 1                    shl [abyte], 1
shl regB|memB, cl                   shl ax, cl
shl regW|memW, 1                    shl dx, 1
shl regW|memW, cl                   shl [aword + bx], cl
```

только для 80286, 80386/486

```
shl regB|memB, immB                 shl cx, 8
shl regW|memW, immB                 shl [word + 4], 4
```

только для 80386/486

```
shl regDW|memDW, 1                  shl edx, 1
shl regDW|memDW, cl                 shl [dword es:di], cl
shl regDW| memDW, immB              shl [dword bx], 4
```

Образец кода

```
mov cl, 4                ; Присвоение cl счетчика сдвига
```

```
shl ax, cl                    ; Умножение ax на 16 (2^4)
```

Описание Инструкции shl и sal генерируют идентичный машинный код. См. те замечания по sal для ознакомления с тем, как действует shl и на какие флаги она оказывает воздействие.

См. также rcl, rcr, rol, ror, sal, sar, shr

shld — Сдвиг влево значений двойной точности

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		u	-	-	-	*	*	u	*	*

Назначение Сдвигает биты кратных значений влево

Синтаксис/Пример

```
shld regW:memW, regW, immB          shld ax, bx, 1
shld regDW:memDW, regDW, immB       shld [bx], eax, 2
shld regW:memW, regW, cl            shld bx, cx, cl
shld regDW:memDW, regDW, cl         shld [edi], edx, cl
```

Образец кода

```
DATASEG
v1 dd 00012345h          ; Первые 4 из 8 слов
v2 dd 6789ABCDh          ; Вторые 4 из 8 слов
CODESEG
P386
mov cl, 8                ; Присвоение счетчика сдвига cl
mov eax, [v2]            ; Загрузка вторых 4 слов в eax
shld [v1], eax, cl       ; Сдвиг eax в [v1] cl раз
shl [v2], cl             ; Завершение 64-битового сдвига согласно cl
; v1 = 01234567          ; Значения после предыдущего
; v2 = 89ABCD00          ;  кода
```

Описание В системах 80386/486 shld используется для сдвига значений двойной точности влево. Первый операнд специфицирует назначение и может быть словом или двойным словом регистра либо ссылкой к памяти. Второй операнд специфицирует источник битов, сдвигаемых в первый операнд. Это значение должно быть регистром слова или двойного слова. Третий операнд специфицирует число сдвигов, подлежащих выполнению и может быть непосредственным значением от 0 до 31 либо значением в регистре cl. Значения, большие 31, усекаются по модулю 32.

Пример показывает типичное использование shld. Два значения двойных слов, v1 и v2, образуют в памяти 64- битовую переменную. Только четыре инструкции требуются для сдвига этой переменной влево на любое число битов (вплоть до 32) — в данном примере на 8. Прежде всего в cl загружается счетчик сдвига. Затем вторая часть значения загружается в eax. Инструкция shld сдвигает биты из eax в значение двойного слова [v1], которое также сдвигается влево равное число раз. Инструкция shl завершает сдвиг, сдвигая [v2] в соответствии с этим же счетчиком, хранящимся в cl. Эффективным результатом является умножение за очень короткий промежуток времени полного 64-битового значения двойной точности на 2^8 (256 десятичное).

См. также shrd

shr — Сдвиг вправо

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*u	-	-	-	*	*	u	*	*

Назначение Сдвигает биты вправо

Синтаксис/Пример

```
shr regB:memB, 1                    shr [abyte], 1
shr regB:memB, cl                   shr ax, cl
shr regW:memW, 1                    shr dx, 1
shr regW:memW, cl                   shr [aword + bx], cl
```

только для 80286, 80386/486

```
shr regB:memB, immB                 shr cx, 8
shr regW:memW, immB                 shr [word + 4], 4
```

только для 80386/486

```
shr regDW:memDW, 1                  shr edx, 1
shr regDW:memDW, cl                 shr [dword es:di], cl
shr regDW: memDW, immB              shr [dword bx], 4
```

Образец кода

```
mov ax, 10500            ; Присвоение значения ax
mov cl, 3                ; Присвоение счетчика сдвига cl
```

```
shr ax, cl                ; Деление 10500 на 8 (ax=1312)
```

Описание

Выполнение shr сдвигает значение младшего разряда (LSD) старого значения байта, слова или двойного слова (только для 80386/486) во флаг переноса. Нулевой бит сдвигается в новый старший разряд (MSD). Повторение этого действия со временем установит все биты в специфицированном регистре или ячейке памяти в 0. Имейте в виду, что shr и sar не являются синонимами, несмотря на тот факт, что сходные инструкции sal и shl — синонимы (см. эти инструкции для более подробного ознакомления).

Когда второй операнд shr — непосредственно 1, of устанавливается в MSD первоначального значения. Когда второй операнд не является непосредственным значением 1 — эффект не определен.

Пример демонстрирует обычное использование shr для деления значения на 8. Сперва значение загружается в ax, а счетчик сдвига присваивается cl. Затем shr сдвигает ax вправо число раз, определенное в cl. Результат равен числу 10500, деленному на 2^3 или 1312, отбрасывая остаток.

См. также rcl, rcr, rol, ror, sal, sar, shl

shrd — Сдвиг вправо значений двойной точности

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		u	-	-	-	*	*	u	*	*

Назначение Сдвигает биты кратных значений вправо

Синтаксис/Пример

```
shrd regW|memW, regW, immB                    shrd [bx], ax, 3
shrd regDW|memDW, regDW, immB               shrd [edi], edx, 4
shrd regW|memW, regW, cl                    shrd ax, bx, cl
shrd regDW|memDW, regDW, cl                   shrd eax, ebx, cl
```

Образец кода

```
shrd edx, ecx, 4          , Сдвиг битов в четырех
shrd ecx, eax, 4          ; регистрах общего
shrd ebx, eax, 4          ; назначения 4 раза
shr  eax, 4
```

Описание

В системах 80386/486 shrd используется для сдвига значений двойной точности вправо. Первый операнд специфицирует назначение и может быть словом или двойным словом регистра либо ссылкой к памяти. Второй операнд специфицирует источник битов, сдвигаемых в первый операнд. Это значение должно быть регистром слова или двойного слова. Третий операнд специфицирует число сдвигов, подлежащих выполнению, и может быть непосредственным значением от 0 до 31 либо значением в регистре cl. Значения, большие 31, усекаются по модулю 32.

Можно использовать shrd для деления значений повышенной точности на 2. Пример демонстрирует эту идею, осуществляя сдвиг 4 раза вправо 128- битового) значения, содержащегося в регистрах eax, ebx, ecx и edx, вместе со старшей частью значения, содержащегося в eax. Это делит значение повышенной точности на 2^4, или 16. Для более подробного ознакомления прочтите заметки, касающиеся shld, которая действует идентично shrd, за исключением направления сдвига.

См. также shld

stc — Установка флага переноса

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	1

Назначение Устанавливает флаг переноса в 1

Синтаксис/Пример

```
stc no operands                                    stc
```

Образец кода

```
PROC AnyProc
; Код процедуры
@@ErrExit:
stc                       ; Установка переноса (ошибка)
ret                       ; Возврат в точку вызова
@@NoErrExit:
clc                       ; Очистка переноса (нет ошибки)
ret                       ; Возврат в точку вызова
END AnyProc
```

Описание Выполнение stc устанавливает флаг переноса в 1. Как демонстрирует образец кода, инструкция часто используется для передачи флага ошибки обратно из подпрограммы, устанавливая cf, если была обнаружена ошибка. (Этот же пример приведен для clc.)

См. также clc, cmc

std **Установка флага направления**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	1	-	-	-	-	-	-	-

Назначение Устанавливает флаг направления в 1

Синтаксис/Пример
```
std no operands                                    std
```

Образец кода
```
DATASEG
string db 10 dup (?)
strlen = $ - string
CODESEG
mov ax, @data                ; Инициализация es для
mov es, ax                   , адресации сегмента данных
ASSUME es DGROUP             ; Сообщает tasm, куда указывает es
mov cx, strlen               ; Присвоение cx длины цепочки
mov di, offset string+strlen-1  , di адресует конец цепочки
std                          , Автоуменьшение di
@@10:
mov al, cl                   ; Присвоение al следующего значения
stos [string]                ; Сохранение al в цепочке
loop @@10                    , cx<-cx-1; переход, если cx <> 0
```

Описание std используется для установки флага направления в 1. Всегда выполняйте std (или сходную с ней инструкцию cld) перед повторением цепочечной операции, автоматически уменьшающей либо оба si и di, либо один из них, если df=1. Пример демонстрирует, как использовать std после первой инициализации cx длиной 10-байтовой цепочечной переменной и адресации цепочки посредством es:di. Цикл, состоящий из трех инструкций, присваивает успешные значения al, которые stos сохраняет в цепочке, автоматически уменьшая di, поскольку df=1. Результатом является установка цепочки в десять значений 1, 2,..., 10.

См. также cld

sti **Установка флага разрешения прерываний**

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	1	-	-	-	-	-	-

Назначение Устанавливает флаг разрешения прерываний в 1

Синтаксис/Пример
```
sti no operands                                    sti
```

Образец кода
```
cli                          ; Запрещает маскируемые прерывания
;
; Код, запускаемый при отключении маскируемых прерываний
;
sti                          ; Разрешение маскируемых прерываний снова
```

Описание Выполнение sti устанавливает флаг разрешения прерываний if в 1, давая возможность процессору распознавать маскируемые прерывания. Инструкция обычно используется в качестве одной из первых команд в программе обслуживания прерывания, которая запускается с if=0. Установка if в 1 посредством sti позволяет распознавать прерывания во время выполнения ISR (Interrupt Service Routine — программа обслуживания прерывания).

См. также cli

stos stosb stosd stosw ***Сохранение цепочки***

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Накапливает цепочку значений в память

Синтаксис/Пример
```
stos memB                                          stos [byte destination]
stos memW                                          stos [word destination]
stosb no operands                                  stosb
```

```
stosw no operands                    stosw
```

только для 80386/486

```
stos memDW                           stos [dword destination]
stosd no operands                    stosd
```

Образец кода

```
DATASEG
buffer db 512 dup (0ffh)
CODESEG
mov ax, @data            ; Инициализация es для
mov es, ax               ;  адресации сегмента данных
ASSUME es DGROUP         , Сообщает tasm, куда указывает es
mov di, offset buffer    , Адресация буфера посредством es di
mov cx, 512/2            , Присвоение размера буфера/2 cx
xor ax, ax               ; Установка ax в 0000
cld                      , Автоувеличение di
rep stosw                , Заполнение буфера словами 00
```

Описание

stos или эквивалентные краткие мнемоники stosb, stosd (только для 80386/486) и stosw используются для сохранения цепочек значений в памяти. Как и в случае всех цепочечных инструкций, назначения регистров фиксированы, даже если в качестве операнда stos специфицирована точная адресная метка. (Краткие мнемоники не требуют операндов.) Инструкция сохраняет значение al, ax или eax (только для 80386/486) в ячейке, адресуемой посредством es:di. Размер сохраняемого значения зависит от размера специфицированного операнда, если только вы не выберете краткую мнемонику: stosb — для хранения байтов, stosw — для хранения слов и stosd — для хранения двойных слов 80386/486. После исполнения инструкции, если df = 0, di увеличивается на размер сохраненного значения — на 1 для байтов, на 2 для слов или на 4 для двойных слов. Если df=1, то di уменьшается на эти значения. Обычно stos используется совместно с префиксом повтора rep и счетчиком в cx для сохранения значений в нескольких ячейках. Как демонстрирует пример, это дает возможность быстро и просто заполнить блоки памяти значениями, в данном случае инициализируя 512-байтовый буфер нулями. Поскольку размер буфера (512) делится на два, вместо stosb используется stosw, повторяясь 256 раз вместо 512.

См. также

cmpsb, cmpsd, cmpsw, ins, insb, insd, insw, lods, lodsb, lodsd, lodsw, movs, movsb, movsd, movsw, outs, outsb, outsd, outsw, scas, scasb, scasd, scasw

sub — Вычитание

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		*	-	-	-	*	*	*	*	*

Назначение

Вычитает целые числа

Синтаксис/Пример

```
sub al, immB                         sub al, 3
sub ax, immW                         sub ax, 1000
sub regB|memB, immB                  sub dl, 5
sub regW|memW, immW                  sub [word bx], 256
sub regW|memW, immB                  sub bx, 8
sub regB|memB, regB                  sub [byte es di], dl
sub regW|memW, regW                  sub cx, cx
sub regB, regB|memB                  sub ah, al
sub regW, regW|memW                  sub dx, [word bp + 4]
```

только для 80386/486

```
sub eax, immDW                       sub eax, 164532
sub regDW|memDW, immB                sub [dword bp - 8], 2
sub regDW|memDW, immDW               sub edx, 99999
sub regDW|memDW, regDW               sub [dword array+bx+di], edx
sub regDW, regDW|memDW               sub edi, ecx
```

Образец кода

```
DATASEG
v1 dd 155612             ; Значение двойного слова
v2 dd 35996              , Значение, вычитаемое из v1
CODESEG
mov ax, [word v2]        ; Загрузка младшего слова v2 в ax
sub [word v1], ax        , Вычитание младших слов
mov ax, [word v2 + 2]    , Загрузка старшего слова v2 в ax
sbb [word v1 + 2], ax    ; Вычитание старших слов
```

Описание

sub используется для вычитания двух байтов, слов или двойных слов (только для 80386/486), обладающих знаком или не обладающих им. Второй операнд вычитается из первого, с замещением результата первоначального значения первого операнда. Инст-

рукция sub обычно вычитает два значения непосредственно либо начинает собой последовательность вычитаний значений повышенной точности, вычитающую значения, превышающие максимальный размер регистра.

См. также sbb

test — Проверка битов

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		0	-	-	-	*	*	u	*	0

Назначение Сравнивает значения, выполняя логическое И

Синтаксис/Пример

```
test al, immB                          test al, 00001000b
test ax, immW                          test ax, 000Fh
test regB|memB, immB                   test [byte bx], 080h
test regW|memW, immW                   test dx, 01000h
test regB|memB, regB                   test ah, cl
test regW|memW, regW                   test ax, cx
```

только для 80386/486

```
test eax, immDW                        test eax, 02h
test regDW|memDW, immDW                test [dword bx+di], 04000000h
test regDW|memDW, regDW                test edx, ebx
```

Образец кода

```
mov ax, -1              ; Загрузка проверяемого значения в ax
test ax, 08000h         ; MSD = 1?
jz @@10                 ; Переход, если MSD <> 1
neg ax                  ; Иначе, нахождение абсолютного значения
@@10:
```

Описание Инструкция test во всех отношениях идентична and, за исключением того, что результат операции логического И отбрасывается — сохраняются только флаги, которые могут проверяться инструкциями условного перехода. Наиболее привычным использованием test является определение равенства одного или более битов единице в значениях байта, слова или двойного слова (только для 80386/486). Чтобы продемонстрировать это, образец кода загружает проверяемое значение в ax, затем применяет test с непосредственным значением 08000h – другими словами, сравнивает двоичное значение с битом 1 в позиции старшего разряда (MSD). Если это значение И ax равно 0, устанавливая zf в 1, значит, MSD ax равен 0; в противном случае MSD ax равен 1. Инструкция jz выявляет это условие, выполняя neg для нахождения абсолютного значения ax только в том случае, если значение является отрицательным числом (MSD=1).

См. также and

wait — Ожидание до готовности

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Выполняет ожидание, пока контакт процессора "ЗАНЯТ" не будет активирован

Синтаксис/Пример

```
wait no operands                                wait
```

Образец кода

```
cstat dw 0              ; Слово состояния сопроцессора
CODESEG
; Turbo Assembler помещает здесь wait автоматически
fstsw [cstat]           ; Сохранение состояния в cstat
wait                    ; Ожидание до окончания
mov ax, [cstat]         ; Загрузка состояния в ax
```

Описание Выполнение wait останавливает процессор до тех пор, пока контакт процессора "ЗАНЯТ" не будет активирован (не будет установлено верхнее состояние), указывая на то, что присоединенное устройство не занято. Эта инструкция обычно используется для синхронизации кода с сопроцессором, давая возможность программе продолжаться только после завершения вычислений. Математические сопроцессоры 80287, 80387 и встроенный 80486 автоматически синхронизируются с основным процессором и не требуют выполнения wait. Математический сопроцессор 8087 требует wait перед каждой инструкцией сопроцессора. Turbo Assembler автоматически вставляет wait, требуемые 8087. Как демонстрирует пример, можно также вставлять wait *за* инструкциями сопроцессора, когда инструкции и сопроцессора, и программы обращаются к одним и тем же ячейкам памяти. Поскольку сопроцессор запускается независимо от основного процес-

сора, если бы wait не была использована перед mov ax, программа могла бы попытаться считать значение cstat прежде, чем сопроцессор завершил сохранение там значения. Хотя Turbo Assembler и помещает wait перед инструкцией fstw, для предотвращения всякой возможности конфликта следует добавлять wait и после нее.

См. также esc

wbinvd — Обратная запись и разрешение кэширования

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		-	-	-	-	-	-	-	-	-

Назначение Организует поток внутренней инструкции кэширования для 80486, а также осуществляет специальный цикл для обмена с шиной, который разработчики аппаратного обеспечения могут использовать в качестве команды для записи обратно в память и организации любых потоков кэширования, являющихся внешними для процессора.

Синтаксис/Пример

```
wbinvd no operands                    wbinvd
```

Образец кода

```
P486
wbinvd                  ; Организует поток кэширования и осуществляет цикл обратной
                        , шинной записи
db 0fh                  , Используется с Turbo
db 09h                  , Assembler 4 0, не распознающим wbinvd
```

Описание Применяйте эту инструкцию только для процессоров 80486. Она не требует никаких операндов и не влияет ни на какие флаги. Intel утверждает, что invd является "зависимой от исполнения", имея в виду, что последующие процессоры могут выполнять эту инструкцию иначе. Как и в случае invd, мало веских причин, если они имеются вообще, использовать эту инструкцию в программах прикладного уровня.

Замечание: ошибка в Turbo Assembler 4.0 мешает ассемблированию wbinvd. Turbo Debugger 4 0, однако, распознает ее. Чтобы вставить эту инструкцию в программу, можно определить байты 0fh и 09h, как показано в последних двух строках примера.

См. также invd, invlpg

xadd — Обмен и сложение

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
			*		*	-	-	-	*	*	*	*	*

Назначение Суммирует и осуществляет обмен двух значений

Синтаксис/Пример

```
xadd regB:memB, regB                  xadd ah, bh
xadd regW:memW, regW                  xadd ax, bx
xadd regDW:memDW, regDW               xadd eax, ebx
```

Образец кода

```
P486
mov  eax, 012340000h      , Присвоение проверяемого значения eax
mov  ebx, 00000ABCDh      , Присвоение проверяемого значения ebx
xadd eax, ebx             , eax=1234ABCDh, ebx=12340000h
```

Описание Эта инструкция копирует значение из регистра или ячейки памяти назначения (eax в образце кода) в регистр источника (ebx), а также помещает сумму первоначальных двух операндов в назначение (eax). Флаги устанавливаются, как для инструкции add.

См. также add, xchg

xchg — Обмен

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение Обменивает значения регистра со значениями других регистров и ячеек памяти

Синтаксис/Пример

```
xchg ax, regW                         xchg ax, cx
xchg regW, ax                         xchg bx, ax
xchg regB, regB:memB                  xchg dh, dl
xchg regB:memB, regB                  xchg [byte bp + 4], ah
xchg regW, regW:memW                  xchg ax, bx
xchg regW:memW, regW                  xchg [word bx], dx
```

только для 80386/486

```
xchg eax, regDW                          xchg eax, ebx
xchg regDW, eax                          xchg ecx, eax
xchg regDW, regDW:memDW                  xchg edx, [dword bx+di]
xchg regDW:memDW, regDW                  xchg [dword var], eax
```

Образец кода

```
DATASEG
array db 80 dup (?)          ; Адресуется [bx]
arraysize = $ - array        ; Размер множества
newbx dw offset array        ; Область для помещения bx
newcx dw arraysize           ; Область для помещения cx
CODESEG
PROC Outer
call Inner                   ; Инициализация/резервирование bx, cx
;
; Другой код в процедуре
;
PROC Inner
xchg bx, [newbx]             ; Инициализация/восстановление bx
xchg cx, [newcx]             ; Инициализация/восстановление cx
ret                          ; Возврат в точку вызова
ENDP Inner
ENDP Outer
```

Описание

xchg используется для обмена значений двух байтов, слов или двойных слов (только для 80386/486), которые могут находиться в регистрах или ячейках памяти. Два операнда, один из которых должен быть регистром, обмениваются значениями без необходимости использования промежуточного значения в стеке либо в другом регистре. Пример использует xchg для инициализации и сохранения значений двух регистров — bx и cx. Обращение call к Inner в начале процедуры Outer осуществляет обмен с двумя переменными типа слова. По завершении процедуры код обращается к Inner, снова выполняя две инструкции xchg перед возвратом в точку вызова Outer, восстанавливая bx и cx в их первоначальных значениях, и сохраняя при этом *текущие* значения регистров. Позднее, когда процедура снова вызовется, bx и cx будут загружены значениями, которыми они обладали в конце предыдущего прохождения процедуры, давая возможность коду выполняться с того места, где он был прерван.

Тривиальное отступление: особая форма xchg ax,ax генерирует тот же машинный код, что и инструкция nop — единственный байт 90h.

См. также

bswap, cmpxchg, lock, nop, xadd

xlat xlatb — Перевод байта из таблицы в al

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		-	-	-	-	-	-	-	-	-

Назначение

Переустановка значения байта из таблицы в al

Синтаксис/Пример

```
xlat memB                                xlat [es:table]
xlatb no operands                        xlatb
```

Образец кода

```
DATASEG
table db 120, 202, 100, 64, 98, 250, 14, 8
CODESEG
mov al, 3                    , Загрузка индекса в al
mov bx, offset table         ; Адресация таблицы посредством ds:bx
xlatb                        ; Установка al в 64
```

Описание

Инструкция xlat устанавливает значение al в связанное значение из таблицы байтов. Таблица должна адресоваться посредством ds:bx или, в случае переадресации на es:, посредством es:bx. Значение в al представляет индекс в таблице с первым байтом таблицы, обладающим индексом 0. Выполнение xlat загружает байт из ds:bx+al (или из es:bx+al) в al. Образец кода демонстрирует, как это происходит; al загружается четвертым байтом (индекс = 3) из небольшой таблицы, устанавливая al в 64.

Простая инструкция xlat не требует операндов. Если подключить операнд, она может ссылаться на bx, как в xlat [bx] и xlat [es:bx], либо на таблицу по имени, как в xlat [table] и xlat [es:table]. Не имеет значения, какую форму вы выбрали, однако вы по-прежнему должны загрузить bx смещением адреса таблицы — инструкция не сделает это за вас. Краткая форма xlatb, действующая идентично xlat, не может использоваться с операндами.

Некоторые справочники предлагают применять lea для загрузки bx эффективным адресом элемента таблицы. Например, использовать инструкцию lea bx, [matrix+si] для инициализации bx смещением адреса строки двумерной матрицы, индексируемой посредством si, а затем воспользоваться xlatb для перевода индекса колонки в al в один из байтов этой строки. Другим типичным применением xlat является перевод символов ASCII в альтернативные символы, чтобы дать возможность перепрограммировать клавиатуру либо преобразовывать значения в различные таблицы кодировки.

См. также lea

xor — Исключающее ИЛИ

Процессор:	8086/88	80286	80386/486	Флаги:	of	df	if	tf	sf	zf	af	pf	cf
	*	*	*		0	-	-	-	*	*	u	*	0

Назначение Осуществляет операцию логического исключающего ИЛИ со значениями двух байтов, слов или двойных слов (только для 80386/486)

Синтаксис/Пример

```
xor al, immB                          xor al, 0FFh
xor ax, immW                          xor ax, 08000h
xor regB:memB, immB                   xor [byte bx], 01h
xor regW:memW, immW                   xor cx, 0400h
xor regW:memW, immB                   xor [word bp +2], 10h
xor regB:memB, regB                   xor ah, cl
xor regW:memW, regW                   xor dx, cx
xor regB, regB:memB                   xor ah, [byte bx]
xor regW, regW:memW                   xor dx, dx
```

только для 80386/486

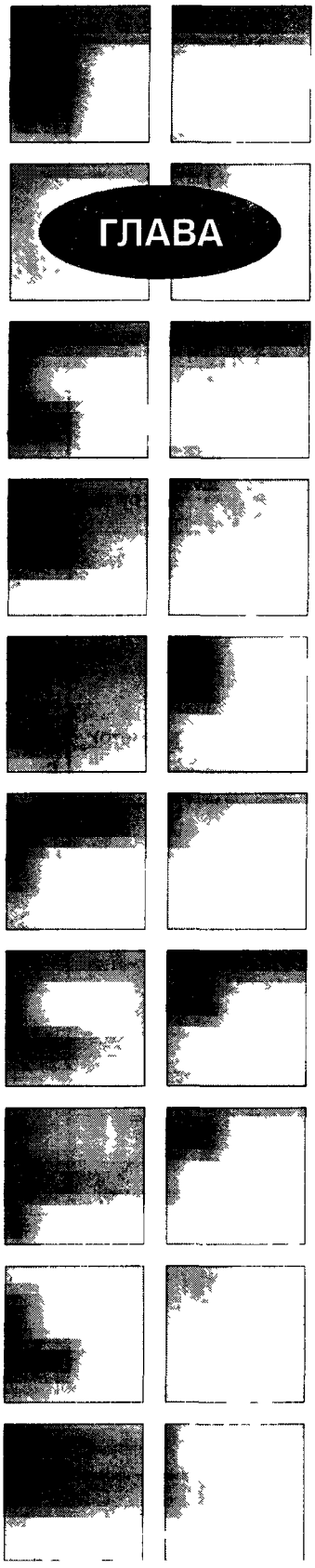
```
xor eax, immDW                        xor eax, 004000000h
xor regDW:mem DW, immDW               xor edx, 0FFFFFFFFh
xor regDW:memDW, immB                 xor [dword bx], 01h
xor regDW:memDW, regDW                xor eax, eax
xor regDW, regDW:memDW                xor edx, [dword bx + si]
```

Образец кода

```
xor ax, ax              ; Установка ax в 0000
xor bx, 0FFFFh          , Логическое инвертирование всех битов bx
```

Описание xor используется для выполнения логического исключающего ИЛИ со значениями двух байтов, слов или двойных слов (только для 80386). Результат этой операции замещает значение первого операнда. На второй операнд часто ссылаются, как на *маску*. Типичным использованием xor является переключение битов с изменением единицы на нули и нулей на единицы. Кроме того, благодаря правилам исключающего ИЛИ, поскольку 1 может получиться только тогда, когда два соответствующих первоначальных бита различны, выполнение xor регистра с самим собой устанавливает его в нуль — обычная техника 8086, сохраняющая один байт. (Инструкция mov ax, 0 занимает три байта, xor ax, ax — два.)

См. также and, or, not



17 ГЛАВА

Справочник по Turbo Assembler

- **Символы**
- **Операторы**
- **Эквиваленты режимов**
- **Директивы**

Символы

В табл. 17.1 описываются различные выделенные курсивом символы, используемые в данной главе, а в табл. 17.2 дается подробное описание символов *предупреждающего класса*. Заранее определенные символы Turbo Assembler приведены в табл. 17.3.

Таблица 17.1 **Символы и значения**

Символ	Значение
|	Или
[]	Не обязательно*
...	Последовательности цифр или продолжение
[]...	Повторение необязательного элемента
:.=	Эквивалентный
align (в линию)	byte|word|dword|para|page байт|слово|дв.слово|параграф|страница
argument (аргумент)	Макропараметры
boundary(граница)	Кратная 2 (т.е. 2, 4, 8)
class (класс)	Текст, представляющий классификацию сегмента
codesym (код симв)	Символ кода (например, метка)
columns (колонки)	Число колонок
combine (составной)	at expr|common|memory|private|public|stack в expr|общий|память|личный|общедоступный|стек

Окончание таблицы 17.1

Символ	Значение
condituon (условие)	Выражение, принимающее значение истинно (<>0) или ложно (=0)
count (счетчик)	1, 2, ..., 65535
datasym (данн.симв.)	Символ данных (например, метка)
definition (определение)	Элемент директивы, определенный в синтаксисе директивы
distance (дистанция)	near|far ближняя|дальняя
dx	db|dd|df|dp|dq|dt|dw
entry point (точка входа)	Кодовая метка, определяющая начало программы
expr (выраж.)	Цифровое выражение
fieldname (имя поля)	Имя поля записи
fields (поля)	Любые участки данных, созданные db, dw и т.п.
filename (имя файла)	Имя файла (с информацией о пути и дисководе)
groupname (имя группы)	Имя составной сегментной группы
language (язык)	basic|c|fortran|pascal|prolog
macroname (макроимя)	Имя определенного макроса
memorymodel (модель памяти)	tiny|small|medium|compact|large|huge|tpascal
memref (ссылка к памяти)	Ссылка к памяти
name (имя)	Идентификатор типа MyData или YourCode
parameter (параметр)	Замещаемый шаблон параметра
prefix (префикс)	Двусимвольный префикс локальной метки, обычно @@
recordname (имя записи)	Имя записи типа данных
register (регистр)	ax|bx|cx|di|ds|dx|es|si| только для 80386/486:eax|ebx|ecx|edi|edx|esi
rows (строки)	Число строк
segexpr (сегм.выраж.)	SEG expr (см. operator SEG)
size (размер)	Целая константа
statements (утверждения)	Ассемблерные инструкции или директивы
structname (структ.имя)	Имя структуры
text (текст)	Любая последовательность символов
type (тип)	near|far|proc|byte|word|dataptr|dword|fword| pword|qword|tbyte|structname
use (использование)	use16|use32
version_id (идент.версии)	M400—(MASM 4.0)|M500—(MASM 5.0)|M510—(MASM 5.1)| M520-(MASM 5.2 aka Quick ASM)|T100-(TASM 1.0)| T101 — (TASM1.01)| T200—(TASM2.0)|T250—(TASM 2.5)|T300— (TASM3.0)| T310—(TASM 3.1)|T320—(TASM 3.2)|T400-(TASM 4.0)
warnclass (предупр.класс)	aln|ass|brk|icg|lco|opi|opp|ops|ovf|pdc|pqk|pro| res|tpi (см. табл. 17.2)
width (ширина)	Целая константа

* Не путайте квадратные скобки [], в которые заключаются необязательные опции, с квадратными скобками, используемыми в косвенных ссылках к памяти, например mov ax, [count]

Таблица 17.2. **Символы для предупреждающего класса**

Символ	Значение
aln	segment alignment (выравнивание сегмента)
ass	assumes 16-bit segment (предполагает 16-битовый сегмент)
brk	brackets needed (требуются скобки)
icg	inefficient code generation (генерация недействующего кода)
gtp	global and symbol type mismatch (несоответствие глобального и символьного типов)
int	interrupt 3 (int 3) generation (генерация прерывания 3 (int 3))
lco	location counter overflow (переполнение счетчика положения)
mcp	MASM compatibility pass (MASM-совместимый)
opi	open IF conditional (открыт условный IF (ЕСЛИ))
opp	open procedure (открыта процедура)
ops	open segment (открыт сегмент)
ovf	arithmetic overflow (арифметическое переполнение)
pdc	pass-dependent construction (конструкция, зависимая от передачи)
pck	assuming constant for [const] (предполагается тип constant для [const])
pro	protected-mode memory-write needs cs: override (защищенный режим записи в память требует отмены cs:)
res	reserved word (зарезервированное слово)
tpi	Turbo Pascal illegal construction (запрещенная конструкция Turbo Pascal)
uni	uninitialized segment warning off (предупреждение об неинициализированном сегменте отключено)

Таблица 17.3. **Заранее определенные символы Turbo Assembler**

Символ	Тип	Описание
@32Bit	Численный	Сегментная модель; 0 = 16 бит, 1 = 32 бит
@Code	Псевдоним	Имя кодового сегмента
@CodeSize	Численный (байт)	0 = маленькая, компактная модель; 1 = остальные модели
@Cpu	Численный (слово)	Разрешает инструкции процессора; номера битов: (1 — включено, 0 — выключено) 0 — 8086, 2 — 80186; 3 — 80386, 4 — 80486, 5 — 80586; 6 — не используется, 7 — защищенный режим; 8 — 8087; 9 — не используется; 10 — 80287; 11 — 80387; 12, 13, 14, 15 — не используются
@CurSeg	Псевдоним	Имя текущего сегмента
@Data	Псевдоним	Имя ближнего сегмента данных
@DataSize	Численный (байт)	Модель данных памяти: 0 = tiny, small, medium; 1 = compact; 2 = huge
@FarData	Псевдоним	Имя дальнего сегмента данных
@FarData?	Псевдоним	Имя дальнего неинициализированного сегмента данных
@FileName	Псевдоним	Ассемблерное имя файла в виде приравниваемого символа
@Interfase	Численный	Язык или оперативная система; номера битов: 0 — никакого языка; 1 — C; 2 — SysCal; 3 — StdCall; 4 — Pascal; 5 — FORTRAN; 6 — Basic; 7 — Prolog; 8 — C++. Бит 7 представляет DOS/Windows (0) или Windows NT (1), если бит какого-либо другого языка также включен
@Object	Псевдоним	Имя текущего объекта
@Stack	Псевдоним	Сегмент или группа, предполагаемые для ss
@Startup	Псевдоним	Ближняя метка, помечающая начало программы, использующей директиву STARTUPCODE
@Table_<object-name>	Псевдоним	Таблица типа данных, содержащая таблицу объектного метода
@Tableaddr_<object-name>	Псевдоним	Метка адреса таблицы объектных виртуальных методов
@WordSize	Численный (байт)	Размер адреса сегмента: 2=16 бит; 4=32 бита

Символ	Тип	Описание
??Date	Цепочка	Сегодняшняя дата
??FileName	Цепочка	Имя ассемблерного файла в виде цепочки символов
??Time	Цепочка	Текущее время в формате код-страны DOS
??Version	Численный (слово)	Номер версии Turbo Assembler: верхний байт = основным цифрам версии, нижний байт = дополнительным цифрам версии

Замечание: Обозначение типа *псевдоним* представляет другие символы. Например, @Code представляет имя текущего кодового сегмента — в любом месте, где можно использовать имя сегмента, можно вместо него применить псевдоним. Численные типы представляют целочисленные значения — их следует использовать везде, где нужно было применить и другое численное значение. Цепочки представляют непрерывные цепочки символов, которые можно вставлять в директивы db для создания переменных, содержащих имя файла, дату и время.

Операторы

Для большей наглядности операторы напечатаны в табл. 17.4 прописными буквами. В программах можно писать операторы как прописными, так и строчными буквами. Табл. 17.5 приводит возможные значения SYMTYPE, который является эквивалентом оператора .TYPE (с начальной точкой) MASM. Оператор TYPE (без начальной точки) возвращает значения в соответствии с табл. 17.6. Если это значение положительно — TYPE представляет размер *expr* в байтах, если отрицательно — указатель NEAR (ближний) или FAR (дальний). Turbo Assembler требует, чтобы перед использованием TYPE был активирован режим MASM. Несмотря на сходство, TYPE не является эквивалентом оператора .TYPE MASM.

Таблица 17.4. **Операторы Turbo Assembler**

Символ	Синтаксис	Описание
()	*(expr)*	Представляет выражение
*	*expr * expr*	Умножение
+	*expr +expr*	Сложение
+	*+expr*	Одиночный плюс
-	*expr - expr*	Вычитание
-	*-expr*	Одиночный минус
.	*memref.field*	Разделитель членов структуры
/	*expr/expr*	Деление
:	*segname:groupname:expr*	Отказ от сегмента
?	*dx?*	Неинициализированные данные
[]	*[memref]*	Косвенная ссылка
AND	*expr* AND *expr*	Логическое И
BYTE	BYTE [PTR] *expr*	8-битовый байт данных
CODEPTR	CODEPTR	Размер адреса процедуры (2 — малые модели; 4 — большие модели)
DATAPTR	DATAPTR	Размер адреса данных (2 — малые модели; 4 — большие модели)
DUP	*count* DUP *(expr[,expr]...)*	Дубликат
DWORD	DWORD [PTR] *expr*	32-битовые данные двойного слова
EQ	*expr* EQ *expr*	Равенство
FAR	FAR [PTR] *expr*	Адрес дальнего кода
FWORD	FWORD [PTR] *expr*	48-битовый указатель дальних данных
GE	*expr* GE *expr*	Больше или равно
GT	*expr* GT *expr*	Больше
HIGH	HIGH *expr*	Верхний порядок
LARGE	LARGE *expr*	Принудительно делает смещение размером 32 бита

Окончание таблицы 17.4

Символ	Синтаксис	Описание
LE	*expr* LE *expr*	Меньше или равно
LENGTH	LENGTH *datasym*	Длина (счетчик элементов)
LOW	LOW *expr*	Нижний порядок
LT	*expr* LT *expr*	Меньше
MASK	MASK *recordname : fieldname*	Битовая маска
MOD	*expr* MOD *expr*	Модуль (остаток деления)
NE	*expr* NE *expr*	Не равно
NEAR	NEAR *expr*	Адрес ближнего кода
NOT	NOT *expr*	Логическое отрицание
OFFSET	OFFSET *expr*	Смещение адреса
OR	*expr* OR *expr*	Логическое ИЛИ
PROC	PROC *codesym*	Кодовая процедура
PTR	*type* PTR *expr*	Указатель
PWORD	PWORD [PTR] *expr*	Указатель дальних данных
QWORD	QWORD [PTR] *expr*	64-битовые данные четверного слова
SEG	SEG *expr*	Адрес сегмента
SHL	*expr* SHL *count*	Сдвиг бита влево
SHORT	SHORT *expr*	Короткий адрес кода
SHR	SHR *expr*	Сдвиг бита вправо
SIZE	SIZE *datasym*	Размер в байтах
SMALL	SMALL *expr*	16-битовое маленькое смещение
SYMTYPE	SYMTYPE *expr*	Тип символа (табл. 17.5)
TBYTE	TBYTE [PTR] *expr*	80-битовые десятибайтовые данные
THIS	THIS *type*	Присвоение текущего адреса
TYPE	TYPE *expr*	Размер символа (табл 17.6)
UNKNOWN	UNKNOWN	Удаление набранной информации
WIDTH	WIDTH *record : field*	Ширина в битах поля записи
WORD	WORD [PTR] *expr*	16-битовое слово данных
XOR	*expr* XOR *expr*	Логическое исключающее ИЛИ

Таблица 17.5. **Возможные значения SYMTYPE**

Номер бита	Если бит = 1, тогда символ...
0	принадлежит кодовому сегменту
1	принадлежит сегменту данных
2	константой (т.е. имеет место равенство)
3	непосредственная ссылка к памяти
4	регистр
5	определен
6	(неиспользуемый бит)
7	внешний для модуля

Замечание: если биты 2 и 3 равны 0, то символ является косвенной ссылкой к памяти типа [bx+si]. Если все биты равны 0, то символ не определен. (Это условие вызывает ошибку ассемблирования при использовании SYMTYPE в директивах размещения данных, таких как db и dw.)

Таблица 17.6. **Возможные значения TYPE**

Значение	Представляемый тип	Значение	Представляемый тип
0	CONSTANT	8	QWORD
1	BYTE	10	TBYTE
2	WORD	0FFFFh	NEAR
4	DWORD	0FFFEh	FAR
6	FWORD или PWORD	n	Число байтов в структуре, таблице или группе

Эквиваленты режимов

Эквиваленты режимов MASM и Ideal представлены в табл. 17.7.

Таблица 17.7. **Эквиваленты режимов MASM и Ideal**

Режим MASM	Режим Ideal	Режим MASM	Режим Ideal	Режим MASM	Режим Ideal
.186	P186	.CODE	CODESEG	.FARDATA?	UFARDATA
.286	P286	.COMMENT	(нет)	.LALL	%MACS
.286C	P286N	.CONST	CONST	.LFCOND	%CONDS
.286P	P286N	.CREF	%CREF	.LIST	%LIST
.287	P287	.DATA	DATASEG	.MODEL	MODEL
.386	P386	.DATA?	UDATASEG	%OUT	DISPLAY
.386C	P386N	.ERR	ERR	PAGE	%PAGESIZE
.386P	P386N	.ERR1	ERRIF1	.RADIX	RADIX
.387	P387	.ERR2	ERRIF2	.SALL	%NOMACS
.486	P486	.ERRB	ERRIFB	SEQ	(нет)*
.486C	P486N	.ERRDEF	ERRIFDEF	.SFCOND	%NOCONDS
.486P	P486N	.ERRDIF	ERRIFDIF	.STACK	STACK
.487	P487	ERRDIFI	ERRIFDIFI	SUBTTL	%SUBTTL
.586	P586	.ERRE	ERRIFE	.TFCOND	(нет)
.586C	P586N	.ERRIDN	ERRIFDN	TITLE	%TITLE
.586P	P586N	.ERRIDNI	ERRIFDNI	.TYPE	SYMTYPE**
.587	P587	.ERRNB	ERRIFNB	.XALL	(нет)
.8086	P8086	.ERRNDEF	ERRIFNDEF	.XCREF	%NOCREF
.8087	P8087	.ERRNZ	ERRIF	.XLIST	%NOLIST
.ALPHA	DOSSEG	.FARDATA	FARDATA		

*Turbo Assembler обычно собирает сегменты в том порядке, в каком они перечисляются во время ассемблирования. При использовании ранних версий TASM следует применять директиву DOSSEG (.ALPHA) для подбора сегментов в алфавитном порядке В TASM 4 0 DOSSEG является устаревшим. Нет никакого эквивалента в режиме Ideal для директивы MASM SEQ.

**SYMTYPE — оператор, все остальные инструкции в данной таблице — директивы.

Директивы

Большинство операторов и директив в табл. 17.8a–17.8f напечатаны прописными буквами для большей наглядности. В программах операторы и директивы можно писать как прописными, так и строчными буквами. В этих таблицах приведены только директивы режима Ideal. Эквивалентные директивы режима MASM представлены в табл. 17.7.

Таблица 17.8а. **Директивы Turbo Assembler 1.0**

Директива
Имя
Синтаксис

:
Определение метки ближнего кода
codesym:

=
Определение численного равенства
name = expr

ALIGN
Выравнивание счетчика размещения
ALIGN boundary

ARG
Определение процедурных аргументов
ARG arglist [=name] [ВОЗВРАЩАЕТ *arglist*]
arglist ::=definition [,definition]...
definition ::=name:typedef
typedef::=type|PTR [type]|[distance[PTR[type]]]

ASSUME
Установка первоначального сегментного регистра
ASSUME segreg:segmentname[,segreg:segmentname]...
ASSUME segreg:NOTHING [,segreg:NOTHING]...
ASSUME NOTHING

%BIN
Установка ширины поля листинга объектного кода
%BIN size

CODESEG
Начало нового кодового сегмента
CODESEG [name]

COMM
Определение общих переменных
COMM dtfinition [,definition]...
definition::=[distance]name:type[:count]

%CONDS
Список всех условных утверждений
%CONDS

CONST
Начало сегмента постоянных данных
CONST

%CREF
Список перекрестных ссылок
%CREF

%CREFALL
Список всех символов перекрестных ссылок
%CREFALL

%CREFREF
Список только ссылаемых символов в перекрестных ссылках
%CREFREF

%CREFUREF
Список только символов, на которые нельзя сослаться, в перекрестных ссылках
%CREFUREF

%CTLS
Список управления листингом
%CTLS

DATASEG
Начало нового сегмента данных
DATASEG

DB
Определение байта
[name]DB expr [,expr]...

Директива
Имя
Синтаксис

DD
Определение двойного слова
[name]DD [type PTR] expr [,expr]...

%DEPTH
Установка уровня глубины вложенности листинга
%DEPTH size

DF
Определение указателя дальнего слова
[name]DF [type PTR] expr [,expr]...

DISPLAY
Отображение текста, взятого в кавычки, во время ассемблирования программ
DISPLAY "text"

DOSSEG
Включение стандартного порядка сегментов (устарело в TASM 4.0)
DOSSEG

DP
Определение 48-битового дальнего указателя
[name] DP [type PTR] expr[,expr]...

DQ
Определение четверного слова
[name] DQ expr[,expr]...

DT
Определение 10-байтовой переменной
[name] DT expr[,expr]

DW
Определение слова
[name] DW [type PTR] expr[,expr]...

ELSE
Начало альтернативного блока условного перехода
IF condition
statements
ELSE
statements
ENDIF

EMUL
Эмулирование инструкций сопроцессора
EMUL

END
Конец исходного текста
END [entry point]

ENDIF
Конец блока условного перехода
If condition
statements
ENDIF

ENDP
Конец процедуры
ENDP [name]

ENDS
Конец сегмента или структуры
ENDS [name]

EQU
Приравнивание символа *(name)* значению *(expr)*
name EQU expr

ERR
Вызов сообщения об ошибке
ERR

Директива
Имя
Синтаксис

ERRIF
Вызов ошибки, если выражение истинно
ERRIF condition

ERRIF1
Вызов ошибки при первом проходе транслятора
ERRIF1

ERRIF2
Вызов ошибки при втором проходе транслятора
ERRIF2

ERRIFB
Вызов ошибки, если аргумент пуст
ERRIFB argument

ERRIFDEF
Вызов ошибки, если символ является определенным
ERRIFDEF name

ERRIFDIF
Вызов ошибки, если аргументы различны
ERRIFDIF argument1, argument2

ERRIFDIFI
Вызов ошибки, если аргументы различны, без учета того, набирались они строчными буквами или прописными
ERRIFDIFI argument1, argument2

ERRIFE
Вызов ошибки, если выражение ложно
ERRIFE expr

ERRIFIDN
Вызов ошибки, если аргументы идентичны
ERRIFIDN argument1, argument2

ERRIFIDNI
Вызов ошибки, если аргументы идентичны, без учета того, набирались они строчными буквами или прописными
ERRIFIDNI argument1, argument2

ERRIFNB
Вызов ошибки, если аргумент не пуст
ERRIFNB argument

ERRIFNDEF
Вызов ошибки, если символ является неопределенным
ERRIFNDEF name

EVEN
Выравнивание кода по четному адресу
EVEN

EVENDATA
Выравнивание данных по четному адресу
EVENDATA

EXITM
Выход из макроса
EXITM

EXTRN
Определение внешнего символа
EXTRN definition [,definition]

FARDATA
Начало дальнего сегмента данных
FARDATA [name]

GLOBAL
Определение глобального символа
GLOBAL definition [definition]

Директива
Имя
Синтаксис

GROUP
Определение сегментной группы
GROUP name segmentname [,segmentname]

IDEAL
Переключение в режим Ideal
IDEAL

IF
Ассемблирование, если условие истинно
IF condition

IF1
Ассемблирование при первом проходе транслятора
IF1

IF2
Ассемблирование при втором проходе транслятора
IF2

IFB
Ассемблирование, если аргумент пуст
IFB argument

IFDEF
Ассемблирование, если символ определен
IFDEF name

IFDIF
Ассемблирование, если аргументы различны
IFDIF argument1, argument2

IFDIFI
Ассемблирование, если аргументы различны, без учета того, набирались они строчными или прописными символами
IFDIFI argument1, argument2

IFE
Ассемблирование, если выражение равно 0 (ложно)
IFE expr

IFIDN
Ассемблирование, если аргументы идентичны
IFIDN argument1, argument2

IFIDNI
Ассемблирование, если аргументы идентичны, без учета того, набирались они строчными или прописными символами
IFIDNI argument1, argument2

IFNB
Ассемблирование, если аргумент не пуст
IFNB argument

IFNDEF
Ассемблирование, если имя не определено
IFNDEF name

%INCL
Список подключенных файлов
%INCL

INCLUDE
Подключение отдельных файлов
INCLUDE "filenamt"

INCLUDELIB
Подключение библиотечных файлов во время компоновки
INCLUDELIB "filename"

IRP
Вставка повторяющегося параметра
IRP parameter, <text[,text] >
statements
ENDM

Директива
Имя
Синтаксис

IRPC
Вставка повторяющегося параметра для символов
IRPC parameter, text
statements
ENDM

JUMPS
Разрешение подгонки при условном переходе
JUMPS

LABEL
Определение символа, имеющего тип
LABEL name type

%LINUM
Установка числа строк поля листинга
%LINUM size

%LIST
Листинг включен
%LIST

LOCAL
Определение локального символа в макросе
LOCAL name[,name]

LOCAL
Определение локального символа в процедуре
LOCAL definition [,definition] [=name]
definition = name type[count]

LOCALS
Разрешение локальных меток
LOCALS [prefix]

MACRO
Начало определения макроса
MACRO name [parameter [,parameter]]

%MACS
Список макрорасширения
%MACS

MASM
Разрешение MASM-совместимого ассемблирования
MASM

MASM51
Разрешение усовершенствований MASM версии 5 1
MASM51

MODEL
Выбор модели памяти
MODEL memorymodel [, language]

MULTERRS
Разрешение нескольких ошибок на строку
MULTERRS

NAME
Изменение имени процедуры
NAME filename

%NEWPAGE
Начало новой страницы листинга
%NEWPAGE

%NOCONDS
Список не ложных условных утверждений
%NOCONDS

%NOCREF
Список неперекрестных ссылок
%NOCREF [name [,name]]

Директива
Имя
Синтаксис

%NOCTLS
Список управления, не относящегося к листингам
%NOCTLS

NOEMUL
Отключение эмуляции сопроцессора
NOEMUL

%NOINCL
Список неподключенных файлов
%NOINCL

NOJUMPS
Отключение настройки при условном переходе
NOJUMPS

%NOLIST
Запрещение листинга
%NOLIST

NOLOCALS
Запрещение локальных меток
NOLOCALS

%NOMACS
Список только тех макроутверждений, которые генерируют код
%NOMACS

NOMASM51
Запрещение усовершенствования MASM версии 5 1
NOMASM51

NOMULTERRS
Запрещение нескольких ошибок на строку
NOMULTERRS

%NOSYMS
Список таблицы, не содержащей символов
%NOSYMS

%NOTRUNC
Поля в листинге шириной в слово
%NOTRUNC

NOWARN
Запрещение предупреждающих сообщений
NOWARN [warnclass]

ORG
Установка счетчика положения в начальное значение
ORG

P186
Разрешение инструкций 80186
P186

P286
Разрешение всех инструкций 80286
P286

P286N
Разрешение инструкций незащищенного режима 80286
P286N

P287
Разрешение инструкции сопроцессора 80287
P287

P386
Разрешение всех инструкций 80386
P386

P386N
Разрешение инструкций незащищенного режима 80386
P386N

Директива *Имя* *Синтаксис*
P387 Разрешение инструкций сопроцессора 80387 *P387*
P8086 Разрешение инструкций только 8086/88 *P8086*
P8087 Разрешение инструкций сопроцессора 8087 *P8087*
%PAGESIZE Установка высоты и ширины страницы листинга *%PAGESIZE [rows] [columns]*
%PCNT Установка ширины поля segment offset листинга *%PCNT*
PNO87 Запрещение инструкции сопроцессора *PNO87*
%POPLCTL Выталкивание управления листингом из ассемблерного стека *%POPLCTL*
PROC Определение новой процедуры *PROC name [distance] [USES reglist] arglist] [=name]* *[RETURNS arglist]* *reglist = register [register]* *definition = name typedef* *typedef = type | PTR [type] | [distance [PTR [type]]]*
PUBLIC Определение символа общего пользования *PUBLIC name [,name]*
PURGE Удаление макроопределения *PURGE macroname [,macroname]*
%PUSHLCTL Выталкивание управления листингом в ассемблерный стек *%PUSHLCTL*
QUIRKS Разрешение ухищрений MASM *QUIRKS*
RADIX Установка первоначального источника *RADIX expr*
RECORD Определение записи поля битов *RECORD name definition [,definition]* *definition = fieldname width[=expr]*
REPT Повтор утверждений *REPT expr* *statements* *ENDM*

Директива *Имя* *Синтаксис*
SEGMENT Определение сегмента *SEGMENT name [align] [combine] [use] ['class']*
STACK Начало нового стекового сегмента *STACK [size]*
STRUC Определение структуры *STRUCT name* *fields* *ENDS [name]*
SUBTTL Объявление субтитула листинга *SUBTTL "text"*
%SYMS Разрешение символьной таблицы листинга *%SYMS*
%TABSIZE Установка ширины колонки таблицы листинга *%TABSIZE width*
%TEXT Установка ширины исходного поля листинга *%TEXT width*
%TITLE Установка заглавия листинга *%TITLE "text"*
%TRUNC Выделение длинных полей в листингах *%TRUNC*
UDATASEG Начало нового сегмента неинициализированных данных *UDATASEG*
UFARDATA Начало нового сегмента дальних неинициализированных данных *UFARDATA*
UNION Определение группы *UNION name* *fields*
USES Автоматическое выталкивание и удаление регистров (только в языковых моделях) *USES register [,register]*
WARN Разрешение предупреждающих сообщений *WARN [warnclass]*

Таблица 17.8b. **Директивы Turbo Assembler 2.0**

Директива
Имя
Синтаксис

P486
Разрешение инструкций 80486
P486

P486N
Разрешение инструкций незащищенного режима 80486
P486N

P487
Разрешение инструкций сопроцессора 80487
P487

P586
Разрешение инструкций 80586 (Pentium)
P586

P586N
Разрешение инструкций незащищенного режима 80586
P586N

Директива
Имя
Синтаксис

P587
Разрешение инструкций сопроцессора 80587
P587

PUBLICDLL
Определение точек входа динамически присоединяемой библиотеки
PUBLICDLL [language] symbol [,[language] symbol]...

RETCODE
Генерация зависимого от модели ret
RETCODE

STARTUPCODE
Вставка зависящего от модели инициализирующего кода
STARTUPCODE

Таблица 17.8c. **Директивы Turbo Assembler 2.5**

Директива
Имя
Синтаксис

ENTERD
То же, что и enter, но использует ebp и esp
ENTERD

ENTERW
То же, что и enter, но использует bp и sp
ENTERW

Директива
Имя
Синтаксис

LEAVED
То же, что и leave, но использует ebp и esp
LEAVED

LEAVEW
То же, что и leave, но использует bp и sp
LEAVEW

Таблица 17.8d. **Директивы Turbo Assembler 3.0**

Директива
Имя
Синтаксис

CATSTR
Объявление макроса текстовой цепочки
name CATSTR string [,string]...

ENUM
Создание пронумерованного типа данных
name ENUM [name [name [,name]]
[{name [,name]...}]

EXITCODE
Вставка зависящего от модели кода прерывания
EXITCODE expr

FASTIMUL
Генерация эффективных инструкций imul или shift/add
FASTIMUL register, register | memref, value

FLIPFLAG
Генерация эффективной инструкции XOR
FLIPFLAG register, memref

GETFIELD
Получение значения из поля записи
GETFIELD name register, memref|register

GOTO
Начало макрорасширения с метки
GOTO label

Директива
Имя
Синтаксис

INSTR
Поиск подцепочки в макросе цепочки
name INSTR [start_expr,] string1, string2

LARGESTACK
Переназначение размера стека MODEL равным 32 битам
LARGESTACK

MASKFLAG
Генерация эффективной побитовой инструкции AND
MASKFLAG register, memref

SETFIELD
Установка значения в поле записи
SETFIELD name memref|register, register

SETFLAG
Генерация эффективной побитовой инструкции OR
SETFLAG register, memref

SIZESTR
Возвращение длины цепочки макроса
name SIZESTR string

SMALLSTACK
Переназначение размера стека MODEL равным 16 бит
SMALLSTACK

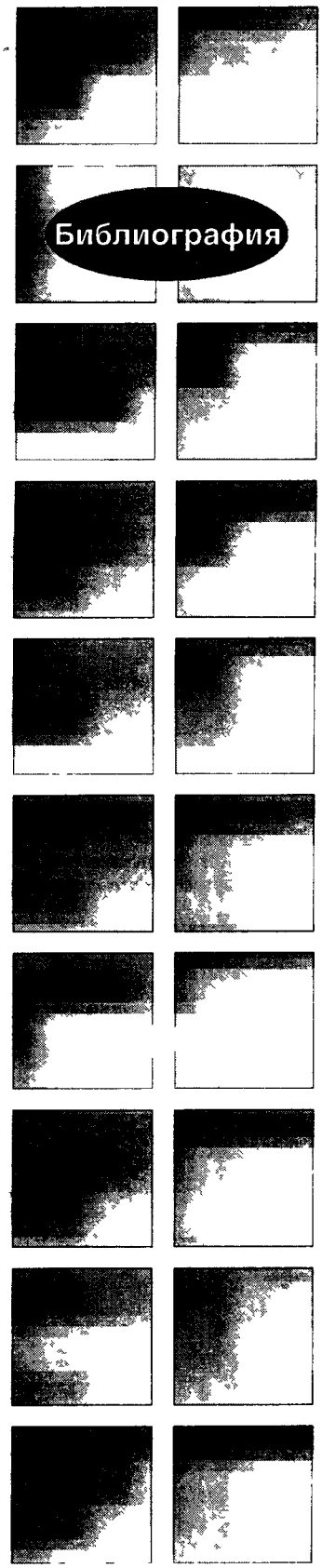
Директива *Имя* *Синтаксис*	*Директива* *Имя* *Синтаксис*
SUBSTR Определение текстового макроса в качестве подцепочки *name SUBSTR string position_expr [size_expr]*	**TESTFLAG** Генерация эффективной инструкции TEST *TESTFLAG register memref*
TABLE Объявление таблицы объектных методов Создается TASM см главу 14	**TYPEDEF** Создание имени типа данных *TYPEDEF name type*
TBLINIT Инициализация указателя TBM *TBLINIT ds bx*	**VERSION** Выбор режима совместимости с ассемблером *VERSION version_id*
TBLINST Создание инстанции TBM для объекта *TBLINST*	**WHILE** Повторение тела макроса *WHILE expr* *macro_body* *ENDM*
TBLPTR Размещение объектной виртуальной таблицы *TBLPTR*	

Таблица 17 8e **Директивы Turbo Assembler 3.1**

Директива *Имя* *Синтаксис*	*Директива* *Имя* *Синтаксис*
POPSTATE Выталкивание состояния Turbo Assembler из внутреннего стека *POPSTATE*	**PUSHSTATE** Выталкивание состояния Turbo Assembler во внутренний 16-уровневый стек *PUSHSTATE*

Таблица 17 8f **Директивы Turbo Assembler 3.2**

Директива *Имя* *Синтаксис*	*Директива* *Имя* *Синтаксис*
IRETW Вытаскивание флагов в виде СЛОВА из 32-битового возврата isr *IRETW*	**PROCTYPE** Создание процедурного (определяемого пользователем) типа данных *PROCTYPE name [description]* *(см PROC)*
POPAW Вытаскивание всех регистров слова из 32-битового кода *POPAW*	**PUSHAW** Выталкивание всех регистров слова в 32-битовый код *PUSHAW*
POPFW Вытаскивание флагов в качестве СЛОВА из 32-битового кода *POPFW*	**PUSHFW** Выталкивание флагов в качестве слова в 32-битовый код *PUSHFW*
PROCDESC Объявление прототипа процедуры *PROCDESC name [description]* *(см PROC)*	



Библиография

Borland International Turbo Assembler 4.0, Turbo Debugger 4.0, Turbo Pascal, Turbo C++, Borland C++. Scotts Valley, CA.

Программы в этой книге были проверены с самыми последними доступными версиями. Некоторые версии Pascal, C и C++ поставляются вместе с Turbo Assembler. Версию 4.0 Turbo Assembler можно приобрести отдельно в фирме Borland.

Brief *Borland International (Краткий справочник)*. CA.

Я использовал "Краткий справочник" при написании всех программ этой книги, так же, как и текста глав. На рынке имеется множество хороших изданий по программированию, но вы не ошибетесь, если выберете этот справочник.

Campbell, Joe *C Programmers Guide to Serial Communications (Руководство для программистов на C по последовательным сетям)*, Second Edition. Indianapolis, IN: Sams Publishing, 1994.

Каждый программист, планирующий писать программное обеспечение для сетей на *любом* языке, должен прочесть эту превосходную книгу. Замечание: требуется компилятор C — автор использует Aztec C, но и предпочитаемый вами компилятор, вероятно, будет работать, если вы несколько измените ваши программы.

Duncan, Ray *Advanced MS-DOS (Расширенная MS DOS)*. Redmond, WA: Microsoft Press, 1986.

Это одна из лучших книг по программированию в MS DOS. Она содержит много примеров на ассемблере и хорошо организованные справочные данные по функциям MS DOS и IBM PC BIOS, а также включает в себя хорошую главу, поясняющую, как писать инсталируемые драйверы устройств.

Intel Corporation *iAPX 86/88, 186/188 User's Manual — Programmer's Reference (Руководство пользователя — справочник программиста iAPX 86/88)* Santa Clara, CA,1986.

Серьезные программисты на ассемблере должны приобрести этот и другие технические справочники Intel, производителя 8086, 80286, 80386, 80486 и других процессоров. Несмотря на некоторые ошибки, справочники приводят все данные по форматам битов машинных кодов и временному разделению инструкций — данные, которые вам потребуются для углубленной работы с ассемблером. Полезные листинги псевдокода описывают, как действуют отдельные инструкции. Возможно, вы не обнаружите этих справочников в книжных магазинах; для получения более подробной информации пишите по адресу: Intel Literature Sales, P.O. Box 58130, Santa Clara, CA 95052-8130. В США и Канаде вы можете заказать эти книги, позвонив по телефону (800) 548-4725.

Intel Corporation *80286 and 80287 Programmer's Reference Manual (Справочное руководство программистов 80286 и 80287).* Santa Clara, CA, 1987.

Intel Corporation *80386 Programmer's Reference Manual (Справочное руководство программистов 80386).* Santa Clara, CA, 1987.

Intel Corporation *i486 Microprocessor Programmer's Reference Manual (Справочное руководство программистов микропроцессора i486).* Santa Clara, CA, 1990.

Jensen, K and Wirth, N *Pascal User Manual and Report (Руководство пользователя и сообщения по Pascal),* 2nd ed. New York: Springer-Verlag, 1974.

Эта книга явилась началом снежной лавины по Pascal. Поскольку в настоящее время все испытывают определенную недостаточность данных, этот справочник полезен, прежде всего, в качестве общего руководства при создании переносимых программ на стандартном Pascal. Особенно это будет полезно, если вы планируете оптимизировать ваши программы с помощью ассемблера, используя методы, обсужденные в главе 12. Имейте в виду: некоторые процедуры, такие как get и put, не поддерживаются Turbo Pascal.

Kernighan, B., and Ritchie, D. *The C Programming Language (Язык программирования C),* 2nd ed. Englewood Cliffs: Prentice-Hall, 1988.

Каждый начинающий программист на C должен прочесть этот учебник от корки до корки. Как и книга Йенсена и Вирта, Керниган и Ричи (популярное второе название книги) особенно полезна в качестве руководства для создания переносимых программ на стандартном C, которые вы планируете оптимизировать с помощью ассемблера, используя методы, обсужденные в главе 13.

Microsoft Corporation *Microsoft Macro Assembler 5.1 Reference (Справочник по Microsoft Macro Assembler 5.1).* Redmond, WA, 1987.

Если вы располагаете Turbo Assembler, вам не нужно приобретать Microsoft Macro Assembler. Но если вы не против заплатить за два ассемблера, то спешу сообщить, что справочники по MASM хорошо написаны и будут полезным дополнением к вашей библиотеке по программированию. Замечание: MASM не поддерживает режим Ideal Turbo Assembler.

Strauss, Edmund *80386 Technical Reference (Технический справочник по 80386)* New York, NY: Brady, 1987.

Будучи общим руководством по 80386, эта книга дублирует значительную часть материалов "Справочного руководства программистов 80386" Intel. Но даже при этом вы найдете там некоторую полезную информацию по использованию инструкций защищенного режима.

Swan, Tom *Mastering Turbo Pascal 6 0 (Освоение Turbo Pascal 6.0),* Fourth Edition. Indianapolis, IN: Howard W. Sams, 1988.

В главе 14 настоящей книги приведена более подробная информация по Pascal для добавления inline ассемблера к Turbo Pascal. Замечание: эта глава была написана до появления Turbo Assembler.

Tanenbaum, Andrew S. *Operating Systems: Design and Implementation (Операционные системы: разработка и воплощение)* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Несомненно, это одна из лучших книг (быть может, *лучшая*), когда-либо написанных по многозадачным, многопользовательским операционным системам. Ее изложение — сжатое и точное, но одновременно и высокотехничное. Хотя ее содержание ориентировано на программистов C и содержит очень мало ассемблерных кодов, понимание книги облегчит начало программирования в защищенном режиме 80386 многозадачных операционных систем

The Waite Group *The Waite Group's MS-DOS Papers* Indianapolis, IN: Howard W. Sams, 1988.

Множество ассемблерных примеров и интересные выдержки из нескольких различных авторов делают эту книгу заслуживающей прочтения. Она содержит полезные советы, касающиеся программирования IBM PC и ассемблера.

Оглавление

Предисловие 5
Благодарность 6

ЧАСТЬ I. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА 7

Глава 1. Введение 8
Изучение языка ассемблера 8
Преимущества языка ассемблера 10
Недостатки языка ассемблера 11
Требования к аппаратному обеспечению 11
Требования к программному обеспечению 12
Пользователям Microsoft Windows 12
Как пользоваться этой книгой 12
Как подготовить ваши диски 14
Использование гибких дисков 14
Ввод исходных текстов программ 15
Резюме 15
Упражнения 16

Глава 2. Первые шаги 17
Основные части программы на ассемблере 17
Ассемблирование программы 24
Понимание объектного кода 25
Опции командной строки 25
Обработка ошибок 26
Введение в Turbo Debugger 28
Написание COM- и EXE-программ 31
Распечатка листингов программ 34
Резюме 36
Упражнения 36
Задания 36

Глава 3. Двоичные биты 38
Все о памяти 38
Сколько битов содержится в байте? 39
Двоичная арифметика и логика 39
Счет в двоичной системе 40
Двоичные и шестнадцатеричные числа 41
Запись значения в дополнительном коде 43
Логические операторы 45
Сдвиги и циклические сдвиги 48
Резюме 49
Упражнения 49
Задания 50

Глава 4. Программирование на языке ассемблера 51
Сегментация памяти 51
Регистры процессора 8086 53
Основные группы команд процессора 55
Команды передачи данных 55
Арифметические команды 62
Логические команды 69
Команды управления потоком 76
Команды управления процессором 86
Команды для работы со строками 90
Резюме 95
Упражнения 96
Задания 96

Глава 5. Простые структуры данных 97
Адресация данных в памяти 97
Способы адресации памяти 98
Выражения и операторы 103
Простые переменные 103
Инициализированные и неинициализированные данные 105
Строковые переменные 106
Локальные метки 107
Пакет для работы с ASCIIZ-строками 108
Программирование по частям 115
Пакет для ввода-вывода строк 118
Компоновка модулей в программу 121
Изучение модуля STRINGS 124
Резюме 130
Упражнения 131
Задания 132

Глава 6. Сложные структуры данных 133
Структуры 133
Еще о числовых переменных 138
Массивы в языке ассемблера 139
Объединения и записи 144
Эффективные логические операции 149
Использование заранее указываемых макроопределений 152
Выполнение VERSION 153
Преобразования чисел и строк 153
Программирование преобразователя между различными системами счисления 163
Резюме 165
Упражнения 166
Задания 166

Глава 7. Ввод и вывод 168
Стандартный ввод и вывод 168
Написание DOS-фильтров 175
Печать текста 180
Использование видеопамяти 182
Модуль для работы с видеопамятью 186
Модуль управления клавиатурой 197
Резюме 201
Упражнения 202
Задания 202

Глава 8. Макросы и условное ассемблирование 204
Что такое макрос 204
Преимущества и недостатки макросов 205
Создание макросов 205
Удаление макросов 205
Подстановка параметров 206
Макросы и переменные 210
Макросы и код 212
Условная компиляция 214
Запуск библиотеки макросов DOS 221
Резюме 228
Упражнения 229
Задания 229

Глава 9. Работа с дисковыми файлами 230
Понятие о дескрипторах файлов 230
Общие представления о дисках и файлах 231
Максимальное количество файлов 231
Открытие и закрытие файлов 231
Заполнение файловых буферов 232
Закрытие файлов 233
Обработка дисковых ошибок 233
Создание новых файлов 236
Чтение командной строки DOS 236
Чтение и запись текстовых файлов 242
Чтение и запись файлов данных 246
Чтение каталога диска 247
Резюме 250
Упражнения 251
Задания 251

Глава 10. Обработка прерываний 252
Мы прерываем эту программу... 252
Написание процедур обработки прерываний 253
Маскируемые и немаскируемые прерывания 253
Векторы прерываний и микросхема 8259 254
Почему команда hlt не останавливает программу 256
Обработка прерываний 257
Внедрение в прерывание часов PC 259
Команда окончания прерывания 261
Прерывания и стеки 262
Использование команд int и into 264
Отлавливание прерывания ошибки деления 264
Исправление ошибки деления 265
Установка обработчика ошибки деления 265
Установка кода TSR в памяти 269
Прерывания, управляющие последовательным обменом 270
Отладка с прерываниями 280
Пошаговое выполнение 280
Резюме 284
Упражнения 285
Задания 285

Глава 11. Дополнительные темы 286
Углубление ваших знаний ассемблера 286
Двоично-кодированные десятичные числа (BCD) 287
Математическая библиотека для работы с BCD 290
Отдельные дополнительные директивы 294
Обработка таблиц 297
Жесткий способ определения сегментов 300
Директива SEGMENT 300
Директива ASSUME 303
Директива GROUP 304
Использование сегментов в программах 305
Дальние сегменты данных 308
Программирование 80286 и старших моделей процессоров 310
Программирование процессора 80386 315
Директива VERSION 319
Перечисляемые типы данных 320
Становимся SMART (ушлыми) 321
Некоторые дополнительные команды 323
Резюме 328
Упражнения 329
Задания 330

ЧАСТЬ II. **ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ** 331

Глава 12. Использование языка ассемблера при программировании на Pascal 332
Возможность усовершенствования 333
Выявление критичных участков программы (критичного кода) 333
Преобразование программы на Pascal к ассемблерному коду 334
Внешние процедуры и функции 335
Модель памяти Pascal 336
Вызов процедур Pascal 338
Демонстрационный пример 339
Адресация данных кодового сегмента 342
Адресация констант с определенным типом 343
Вызов функций Pascal 343
Адресация переменных Pascal 344
Вызов внешних функций 344
Передача параметров 345
Использование модели памяти TPASCAL 347
Использование директивы ARG 348
Написание внешних строчных функций 349
Проблема быстродействия 351
Резюме 355
Упражнения 356
Задания 357

Глава 13. Использование языка ассемблера при программировании на C и C++ 358
Применение C и C++ совместно с языком ассемблера 359
Определение критичного кода 359
Использование регистров 359
Ассемблерные операторы Inline 360
Разделение данных 363
Внешние программы на ассемблере 367
Вызов ассемблерных функций из C 372
Ассемблирование и компоновка внешних модулей 373
Отладка многоязыковых программ 374
Вызов функций C из языка ассемблера 376
Результаты выполнения функции 377
Локальные переменные 377
Вызов функций C++ из ассемблера 379
Передача аргументов функции 382
Использование классов C++ с языком ассемблера 386
Резюме 391
Упражнения 391
Задания 392

Глава 14. Программирование с объектами 393
Объектно-ориентированное программирование (ООП) посредством TASM 393
Для чего используется ООП 393
Основы объектов TASM 396
Полиморфизм 421
Другие узкие места и "завлекалочки" ООП 433
Резюме 437
Упражнения 437
Задания 438

Глава 15. Программирование для Windows 439
Введение в программирование для Windows с применением TASM 439
Развитие прикладных программ для Windows с использованием TASM 455
Резюме 469
Упражнения 469
Задания 470

ЧАСТЬ III. **СПРАВОЧНИК** 471

Глава 16. Справочное руководство по ассемблеру 472
О справочнике 472
Инструкции защищенного режима 472
Обращение к источнику 473
Распределение времени для инструкций и двоичное кодирование 474
Как пользоваться справочником 474

Глава 17. Справочник по Turbo Assembler 529
Символы 529
Операторы 532
Эквиваленты режимов 534
Директивы 534

Библиография 541