

В. И. Юров

ПИТЕР®

ASSEMBLER

ПРАКТИКУМ

2-е издание

- для студентов и специалистов, использующих ассемблер для решения задач прикладного и системного программирования
- полнофункциональные программы для современных операционных платформ
- актуальные задачи прикладного программирования
- алгоритмы «на каждый день»



В. И. Юров

ASSEMBLER

ПРАКТИКУМ

2-е издание

Допущено Министерством образования Российской Федерации
в качестве учебного пособия для студентов высших учебных
заведений, обучающихся по направлению подготовки
дипломированных специалистов
«Информатика и вычислительная техника»



**Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск**

2006

ББК 32.973-018.1я7
УДК 681.3.06(075)
Ю78

Рецензенты:

Туровинин В. В., профессор кафедры военной кибернетики Филиала ВУ ПВО
Тузов В. А., профессор кафедры Технологий программирования
Санкт-Петербургского государственного университета

Юров В. И.

Ю78 Assembler. Практикум. 2-е изд. — СПб.: Питер, 2006. — 399 с.: ил.

ISBN 5-94723-671-0

Цель книги — дополнить учебник «Assembler» того же автора практическим материалом, используя который можно разрабатывать сложные полнофункциональные программы для различных операционных платформ.

Каждая из двенадцати глав практикума посвящена определенной прикладной теме. Исчерпывающе рассмотрены вопросы организации взаимодействия программ на ассемблере с внешним миром. Приведены варианты ассемблированной реализации многих известных и востребованных на практике алгоритмов. Изложение базовых вопросов прикладного программирования сопровождается рассмотрением ряда интересных примеров.

Книга предназначена для студентов и специалистов, применяющих ассемблер для решения задач прикладного и системного программирования.

Допущено Министерством образования Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973-018.1я7
УДК 681.3.06(075)

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Содержание

Благодарности	9
Введение	10
Структура книги	10
От издательства	14
 Глава 1. Программирование целочисленных арифметических операций	 15
Двоичные числа	16
Сложение двоичных чисел	16
Вычитание двоичных чисел	21
Умножение двоичных чисел	24
Деление двоичных чисел	30
Двоично-десятичные числа (BCD-числа)	36
Неупакованные BCD-числа	36
Упакованные BCD-числа	42
Генерация последовательности случайных чисел	44
Конгруэнтный метод генерации последовательности случайных чисел	44
Аддитивный генератор случайных чисел	48
Программа генерации высокослучайных двоичных наборов	50
 Глава 2. Сложные структуры данных	 53
Основные понятия	53
Способы распределения памяти	56
Множество	59
Массив	61
Описание массивов	61
Сортировка массивов	64
Поиск в массивах	81
Структуры	87
Вложенные структуры	87

Содержание

Благодарности	9
Введение	10
Структура книги	10
От издательства	14
 Глава 1. Программирование целочисленных арифметических операций	 15
Двоичные числа	16
Сложение двоичных чисел	16
Вычитание двоичных чисел	21
Умножение двоичных чисел	24
Деление двоичных чисел	30
Двоично-десятичные числа (BCD-числа)	36
Неупакованные BCD-числа	36
Упакованные BCD-числа	42
Генерация последовательности случайных чисел	44
Конгруэнтный метод генерации последовательности случайных чисел	44
Аддитивный генератор случайных чисел	48
Программа генерации высокослучайных двоичных наборов	50
 Глава 2. Сложные структуры данных	 53
Основные понятия	53
Способы распределения памяти	56
Множество	59
Массив	61
Описание массивов	61
Сортировка массивов	64
Поиск в массивах	81
Структуры	87
Вложенные структуры	87

Массивы структур — таблицы	89
Поиск в таблице	89
Список	116
Последовательные списки	116
Связные списки	123
Граф	133
Создание двусвязного списка состояний конечного автомата	137
Создание односвязного списка переходов для состояния конечного автомата	138
Дерево	139
Представление дерева в памяти	141
Построение двоичного дерева	142
Обход узлов дерева	143
Включение узла в упорядоченное бинарное дерево	146
Исключение узла из упорядоченного бинарного дерева	146
Лексикографическое дерево	147
Глава 3. Процедуры в программах на ассемблере	149
Реализация рекурсивных процедур	149
Разработка динамических библиотек (DLL)	156
Шаг 1. Разработка текста DLL-библиотеки	157
Шаг 2. Трансляция и компоновка исходного текста DLL-библиотеки	160
Шаг 3. Создание LIB-файла	161
Шаг 4. Сборка приложения с использованием DLL-библиотеки	161
Шаг 5. Проверка работоспособности приложения с использованием DLL-библиотеки	162
Глава 4. Обработка цепочек элементов	164
Прямой поиск в текстовой строке	165
Поиск с предварительным анализом искомой подстроки	169
Глава 5. Работа с консолью в программах на ассемблере ...	175
Функции BIOS для работы с консолью	175
Функции BIOS для работы с клавиатурой	176
Функции BIOS для работы с экраном	180
Функции MS DOS для работы с консолью	185
Функции MS DOS для ввода данных с клавиатуры	185
Функции MS DOS для вывода данных на экран	188
Работа с консолью в среде Windows	190
Организация низкоуровневого консольного ввода-вывода	190
Глава 6. Преобразование чисел	199
Ввод чисел с консоли	200
Ввод целых десятичных чисел из диапазона 0–99	200

Ввод целых десятичных чисел из диапазона 0–4 294 967 295	201
Ввод целых десятичных чисел из диапазона 0–999 999 999 999 999 999	203
Ввод целых десятичных чисел из диапазона 0–∞	205
Ввод вещественных чисел	206
Вывод чисел на консоль	208
Вывод шестнадцатеричных чисел	208
Вывод целых десятичных чисел из диапазона 0–99	210
Вывод целых десятичных чисел из диапазона 0–∞	211
Вывод целых десятичных чисел из диапазона 0–999 999 999 999 999 999	213
Вывод вещественных чисел	214
 Глава 7. Работа с файлами в программах на ассемблере	218
Работа с файлами в MS DOS (имена 8.3)	219
Создание, открытие, закрытие и удаление файла	219
Чтение, запись, позиционирование в файле	225
Получение и изменение атрибутов файла	233
Работа с дисками, каталогами и организация поиска файлов	235
Работа с файлами в MS DOS (длинные имена)	239
Создание, открытие, закрытие и удаление файла	243
Получение и изменение атрибутов файла	245
Работа с дисками, каталогами и организация поиска файлов	249
Файловый ввод-вывод в Win32	257
Обработка ошибок	260
Создание, открытие, закрытие и удаление файла	260
Чтение, запись, позиционирование в файле	264
Получение и изменение атрибутов файла	269
Работа с дисками, каталогами и организация поиска файлов	272
Файлы, отображаемые в память	279
 Глава 8. Оптимизация программного кода. Профайлер	283
Определение типа процессора	284
Приемы оптимизации	285
Архитектурные особенности процессоров Pentium	286
Кэш	286
Конвейер процессоров семейства P6 (Pentium II/III)	291
Планирование и учет особенностей исполнения команд	293
Циклические конструкции и переходы	293
Выравнивание данных и кода	294
Оптимизация для процессоров семейств P6 и Net Burst (Pentium 4) ...	295
Профайлер	299
 Глава 9. Вычисление CRC	303
CRC-арифметика	306
Прямой алгоритм вычисления CRC	312

Табличные алгоритмы вычисления CRC	316
Основы	316
Прямой табличный алгоритм CRC16	318
Прямой табличный алгоритм CRC32	321
«Зеркальный» табличный алгоритм CRC32	325
 Глава 10. Расширения традиционной архитектуры Intel	329
MMX-технология процессоров Intel	329
MMX-расширение архитектуры процессора Pentium	330
Модель целочисленного MMX-расширения	330
Особенности команд MMX-расширения	332
Система команд	339
Пример применения MMX-технологии	356
Дополнительные целочисленные MMX-команды	367
XMM-расширение архитектуры процессора Pentium	369
Модель XMM-расширения	370
Система команд	372
Описание упакованных и скалярных данных	382
Примеры использования команд XMM-расширения	384
Моделирование команд XMM-расширения	389
Модельно-зависимые регистры	393
Команды RDMSR и WRMSR	394
 Заключение	395
Список литературы	396

Благодарности

Хотелось бы выразить благодарность всем тем людям, которые, возможно и не подозревая об этом, приняли участие в появлении этой книги на свет. Это и мои учителя, Л. К. Ларин, Д. Н. Бибишев, профессора Санкт-Петербургского университета В. А. Тузов и Н. Е. Кири́н, к сожалению, безвременно ушедший от нас. Особая благодарность — моей жене Елене и детям Александру и Юлии за поддержку и внимание в процессе работы над материалом книги и подготовки ее к изданию.

Также выражаю благодарность высокопрофессиональному коллективу компьютерной редакции издательства «Питер», предоставившему возможность издать данную книгу.

Введение

Проявил себя — закрепил...

От Фоменко

Профессия программиста удивительна и уникальна. Давно уже настало время настоящего философского осмысления этой сферы человеческой деятельности, действительно обладающей какими-то особенными, для людей непосвященных чуть ли не магическими, свойствами. Если не брать в рассмотрение коммерческую сторону, то можно сказать, что чужих людей в этой области профессиональной деятельности нет. В чем же ее особенность? Наиболее точно по этому поводу высказался Фредерик Брукс в главе «Пятьдесят лет удивления, восхищения и радости» своей книги «Мифический человеко-месяц, или Как создаются программные системы» [46]: «Немногим Бог дает право зарабатывать на жизнь тем, чем они с радостью занимались бы по собственной воле, по увлечению. Я благодарен судьбе». И далее: «Область связанных с компьютерами знаний претерпела взрыв, как и соответствующая технология. Будучи аспирантом в середине 50-х, я мог прочесть все журналы и труды конференций. Я мог оставаться на современном уровне во всей научной дисциплине. Сегодня мне в моей интеллектуальной жизни приходится с сожалением расставаться с интересами то в одной, то в другой подобласти, поскольку количество документов превысило всякую возможность справиться с ними. Масса интересов, масса замечательных возможностей для учебы, исследований и размышлений. Чудесное затруднение! Не только конца не видно, но и шаг не замедляется. В будущем нас ожидают многие радости». Что еще к этому добавить?

Структура книги

Эту книгу можно рассматривать как своеобразную форму программного продукта. Даже беглое ее пролистывание показывает, как много в ней программного кода. Более того, так как ассемблерный код неэкономичен с точки зрения использования поверхности листа бумаги для его записи, то в тексте книги приведены лишь значимые для каждого конкретного контекста изложения фрагменты программ. Полные тексты этих программ содержатся отдельно, в материалах, прилагаемых к книге. Данные материалы хранятся на сайте издательства «Питер» (<http://www.piter.com>) и доступны по ссылке «Файлы к книгам». Некоторые наиболее объем-

ные по размеру исходного текста программы целиком вынесены в эти материалы без приведения их фрагментов в тексте книги. Для эффективной работы с ними читателю следует внимательно следить за ссылками на них и соответствующими пояснениями. Насколько это возможно, программы были проверены, но было бы опрометчиво утверждать, что вероятность появления ошибок в них равна нулю. Любому программисту, даже имеющему очень скромный опыт практической работы, известно, что вероятность последней ошибки есть всегда. В связи с этим просьба к читателям сообщать о найденных ошибках по указанным ниже адресам электронной почты.

Цель книги — дополнить учебник «Assembler», выпущенный издательством «Питер» [39], практическим материалом, используя который, программирующий на ассемблере сможет разрабатывать сложные полнофункциональные программы для различных операционных платформ. Характер подобранного материала прикладной. Чтобы убедиться в этом, достаточно посмотреть оглавление книги. Книга состоит из 10 разных по объему глав. Ниже приведены краткие сведения о цели и характере содержимого каждой из этих глав.

- Глава 1 «Программирование целочисленных арифметических операций». В этой главе приводятся исчерпывающие сведения об алгоритмах реализации четырех основных арифметических операций над числами различной разрядности. На практике нередко возникают ситуации, когда численные значения данных выходят за пределы максимально представимых диапазонов чисел в компьютере. Тогда нужно использовать алгоритмы для производства вычислений над многобайтовыми (от 1 до ∞) числами. Здесь же приведена реализация этих алгоритмов для двоичных и двоично-десятичных (BCD) чисел. Кроме этого, глава содержит описание алгоритмов генерации псевдослучайных последовательностей, проблема организации которых также возникает достаточно часто.
- Глава 2 «Сложные структуры данных». Содержимое этой главы значительно дополняет и расширяет содержимое одноименной главы 13 учебника. Достаточно перечислить номенклатуру рассмотренных структур данных, названных «сложными», — это множества, массивы, структуры, таблицы, одно- и двусвязные списки, деревья. Для демонстрации работы с этими «сложными структурами данных» подобраны интересные и востребованные на практике алгоритмы. Так, работа с массивами показана на примерах популярных алгоритмов сортировки и поиска, работы с матрицами. Работа со структурами иллюстрируется на примерах организации массивов структур — таблиц. При этом наряду с обычными таблицами рассматривается специальный класс таблиц — таблиц с вычисляемыми входами, или хэш-таблиц. Интересные примеры иллюстрируют выполнение основных операций над элементами одно- и двусвязных списков. Работа с сетью показана на примере организации в программе такой структуры, как конечный автомат. Заканчивается глава рассмотрением элементов компиляции программ. Это логичное и оправданное с практической точки зрения завершение главы о сложных структурах данных. Наверняка каждому из вас приходилось организовывать элементарный языковой интерфейс с пользователем и обрабатывать его ввод. О существующих подходах к практической реализации формальных механизмов распознавания ввода пользователя вы узнаете из главы 2.

- Глава 3 «Процедуры в программах на ассемблере». Также достаточно интересная глава, которая является существенным дополнением главы 15 «Модульное программирование» учебника. Большое внимание уделено в ней реализации рекурсивных процедур в программах на ассемблере. Реализация рекурсии в любом языке — предмет дискуссии, причем от полного неприятия до слепого поклонения. Мы не стали принимать участие в этой дискуссии, а просто показали технологию разработки рекурсивных программ на языке низкого уровня. Попутно обсуждению подвергаются проблемы передачи параметров и сохранения локальных параметров процедуры. В несколько более скромном объеме приведены сведения об организации вложенных процедур. В этой главе также содержится очень важный материал для программирующих под Windows — о разработке и об организации работы с DLL-библиотеками в программах на ассемблере.
- Глава 4 «Обработка цепочек элементов» содержит пример реализации некоторых полезных алгоритмов поиска подстроки в текстовой строке. Материал этой главы представляет собой существенное дополнение (а где-то предлагает и альтернативные решения) главы 12 «Цепочечные команды» учебника.
- Глава 5 «Работа с консолью в программах на ассемблере» в полном объеме рассматривает проблему ввода информации с клавиатуры и вывода информации на экран компьютера. Для этого приведено описание соответствующих средств BIOS, операционных платформ MS DOS и Windows.
- Глава 6 «Преобразование чисел» предлагает набор алгоритмов для преобразования представлений чисел между различными системами счисления. Этот вид преобразования данных также часто встречается на практике. Из-за того, что в ассемблере нет для этого соответствующих средств, каждый из программистов решает эту задачу по-своему, исходя из своего опыта и знаний.
- Глава 7 «Работа с файлами в программах на ассемблере» содержит систематизированные сведения по работе с файлами из программ на ассемблере. За неимением соответствующих средств языковой поддержки, со стороны программистов на практике также наблюдается свободное творчество. Приведенные сведения относятся к уровню практической реализации задач в MS DOS и Windows с учетом возможности использования как длинных, так и коротких имен.
- Как оценить эффективность кода, который вы пишете? В этом вам поможет материал главы 8 «Оптимизация программного кода. Профайлер». В ней содержится описание принципов оптимизации кода программы на ассемблере (и не только), а также описание двух макрокоманд, использующих средства процессора Pentium, которые помогут вам решить эту задачу.
- Глава 9 «Вычисление CRC» представляет варианты решения одной интересной практической задачи, которой на практике можно найти достаточно много применений. Обладая этим инструментом, можно проводить быстрые оценочные проверки целостности данных, которыми манипулирует ваша программа. Суть этих алгоритмов с первого взгляда не очень очевидна, поэтому данная глава содержит подробное их объяснение.

■ В главе 10 «Расширение традиционной архитектуры Intel» приводятся сведения о порядке использования команд MMX- и XMM-расширений микропроцессора Intel. Более того, предлагается альтернативный подход к решению проблемы поддержки работы с этими командами в программах на ассемблере. Механизм решения этой проблемы будет полезен и в других случаях, когда требуется обеспечить поддержку новых команд микропроцессора, работая со старыми версиями компилятора (и не только ассемблера), процесс обновления которых по объективным причинам значительно более инертен, чем процесс обновления системы команд микропроцессора.

Необходимо подчеркнуть тот факт, что программы книги реализованы с помощью двух версий ассемблера — 16- и 32-разрядной. Выбор операционной платформы и средств реализации задач книги производился исходя из их конкретной постановки. Главный критерий здесь — подчеркнуть особенности реализации алгоритма. Если для этого достаточно платформы MS DOS, то задача реализовывалась с использованием средств этой ОС. При необходимости вы достаточно легко сможете доработать свою программу так, чтобы она функционировала в среде Windows. Для этого книга содержит достаточно много практических примеров. Большое количество задач реализовано непосредственно для функционирования в среде Windows. Сделано это в основном на примерах консольных приложений.

Материал книги не является автономным. В книге содержится много примеров, построенных на алгоритмах, описанных в общедоступной литературе. Это сделано намеренно. Во-первых, таким образом книгу можно интегрировать в качестве дополнительного учебного пособия в различные алгоритмические дисциплины. Во-вторых, экономится место для подробного пояснения сути алгоритмов. Для пояснения наиболее сложных алгоритмов в некоторых примерах книги введен псевдоязык, синтаксис которого рассмотрен в главе 2. Ради экономии места такой вид пояснения используется для текстов программ в файлах, прилагаемых к книге. Поэтому имейте в виду, что исходные тексты программ в материалах, прилагаемых к книге, немного отличаются от соответствующих текстов программ в книге — прежде всего тем, что первые снабжены более подробными комментариями и псевдокодом. Для общеизвестных алгоритмов приведены ссылки на источники, где с ними можно познакомиться более подробно. Более того, система нумерации программ сделана так, чтобы указать читателю ссылку на источники, которые подойдут для более глубокого изучения соответствующих алгоритмов. Следует отметить, что псевдоязык использован с двоякой целью. Первую из целей мы уже отметили, что касается второй, то она заключается в том, чтобы подготовить читателя к изучению теории компиляции. Так, пояснение примеров программ на ассемблере производится не только с помощью обычных комментариев, но и с добавлением фрагментов программ на псевдокоде. Такое смешанное пояснение ассемблерной реализации алгоритма не только дополнительно иллюстрирует сам алгоритм, но и показывает суть третьего и четвертого этапов компиляции программы — генерации и оптимизации кода, так как при этом можно видеть, какие ассемблерные (машинные) конструкции могут соответствовать конструкциям языка высокого уровня. Для улучшения понимания реализации алгоритмов на ассемблере в некоторых из них отсутствует какая-либо оптимизация, вплоть до того, что при-

сутствуют фрагменты кода, которые можно удалить. Это сделано исходя из двух соображений. Во-первых, с тем чтобы сохранить замысел автора алгоритма. Читатель волен сам решать, что и в каком объеме может быть улучшено и усовершенствовано в программе для решения конкретной практической задачи. Во-вторых, оптимизированный текст труднее для понимания. Более того, как будет показано в главе 8, оптимизация имеет смысл в случае, когда ясна программно-аппаратная среда, в которой будет работать программа.

Следует обратить особое внимание, что книга не имеет дискеты или компакт-диска с текстами программ. Все они содержатся на сайте издательства «Питер» в разделе «Файлы к книгам».

Завершает книгу достаточно подробный список литературы, которая была использована в процессе подготовки книги и которую можно рекомендовать как основу для дальнейшего изучения затронутых в ней вопросов.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты автора v_yugov@mail.ru или компьютерной редакции издательства «Питер» comp@piter.com.

Мы будем рады узнать ваше мнение! Все пожелания читателей будут учтены, при необходимости дан ответ. Кроме того, обнаруженные в программах ошибки будут исправлены, и соответствующие изменения будут оперативно внесены в содержимое материалов, прилагаемых к книге, которые можно найти на web-сайте издательства <http://www.piter.com>.

Подробную информацию о наших книгах вы найдете на web-сайте издательства <http://www.piter.com>.

Глава 1

Программирование целочисленных арифметических операций

Всякое математическое доказательство, за которым мы можем следить, вырази́мо конечным числом символов. Эти символы, правда, могут быть связаны с понятием бесконечности, но связь эта такова, что ее можно установить за конечное число шагов. Так, когда в случае математической индукции мы доказываем теорему, зависящую от параметра n , мы доказываем ее сначала для $n = 0$ и затем устанавливаем, что случай, когда параметр имеет значение $n + 1$, вытекает из случая, когда параметр имеет значение n . Тем самым мы убеждаемся в правильности теоремы для всех положительных значений параметра n . Более того, число правил действия в нашем дедуктивном механизме должно быть конечным, даже если оно кажется неограниченным из-за ссылки на понятие бесконечности. Ведь и само понятие бесконечности вырази́мо в конечных терминах.

Н. Винер, «Кибернетика, или Управление и связь в животном и машине»

В главе 8 «Арифметические команды» учебника было приведено достаточно подробное (с соответствующими рассуждениями и примерами) описание возможностей процессора по выполнению арифметических операций над целочисленными данными. Также было отмечено, что при выполнении этих операций возникают характерные ситуации, обнаружение и обработка которых возлагается на программиста. Именно этому вопросу мы и уделим внимание в данном материале. Вначале вспомним основные моменты.

Процессор Intel имеет средства для обработки целочисленных арифметических данных двух форматов: двоичного и двоично-десятичного (BCD). Данные в двоичном формате рассматриваются как числа со знаком и без знака. При этом необходимо заметить, что не существует отдельного формата для чисел со знаком и без знака. Один и тот же двоичный код может рассматриваться как значение со знаком и без знака. Все зависит от того, как трактуется старший бит операнда. Для чисел без знака он является старшим значащим битом операнда. Для чисел со знаком смысл старшего значащего бита операнда меняется: его нулевое значение соответствует положительному числу, единичное — отрицательному. Остальные

разряды операнда — значащие. Но здесь есть один нюанс, смысл которого в том, что остальные разряды не являются модулем числа. Для положительного числа они действительно являются абсолютной величиной числа, а вот для отрицательного числа они представляют собой так называемый дополнительный код числа. Идея дополнительного кода состоит в том, что процессору совсем необязательно иметь отдельные устройства для сложения и вычитания. Достаточно только одного — сумматора.

Есть всего лишь две команды в системе команд процессора, которые воспринимают старший бит операнда как знаковый, — это команды IMUL и IDIV. В остальных случаях забота о правильной трактовке старшего бита ложится на программное обеспечение. Программирующий на ассемблере должен сам предусматривать особенности обработки знаковых битов.

Другая характерная ситуация при выполнении арифметических действий — переполнение и антипереполнение. Их причина — ограниченность разрядной сетки операнда. При выполнении операции сложения или умножения возможен выход результата за пределы разрядной сетки. Если результат больше максимально представимого значения для операнда данной размерности, то говорят о ситуации *переполнения*. Иначе, если результат меньше минимально представимого числа, то говорят о ситуации *антипереполнения* (потери точности). При этом результат также верен, но при его соответствующей трактовке. Все эти рассуждения приведены в главе 8 «Арифметические команды» учебника, и повторять мы их не будем. Сосредоточимся на практическом аспекте этого вопроса. Ситуация переполнения может иметь место при вычислениях, в которых заранее не известен размер операндов.

Двоичные числа

Прежде чем программировать,
запишите программу в псевдокодах.

Д. Ван Тассел

Сложение двоичных чисел

Сложение чисел размером 1 байт без учета знака

```

:-----+
:| Процедура: add_unsign. Сложение чисел размером 1 байт без учета знака |
:-----+
:| Вход: summand_1 и summand_2 – слагаемые. |
:-----+
:| Выход: sum_b или sum_w – значение суммы с учетом переполнения. |
:-----+
.data
summand_1      db      ?
summand_2      db      ?
sum_w          label word
sum_b          db      0
carry          db      0
.code
add_unsign     proc

```

```

        mov     al, summand_2
        add     al, summand_1
        mov     sum_b, al
        jnc     end_p          ; проверка на переполнение
        adc     carry, 0
end_p:   ret
add_unsign endp

```

Программа учитывает возможное переполнение результата. Сложение двоичных чисел большей размерности (2/4 байта) выполняется аналогично. Для этого необходимо заменить директивы DB на DW/DD и регистр AL на AX/EAX.

Сложение чисел размером N байтов без учета знака

```

;-----+
;| Процедура: add_unsign_N. Сложение чисел размером N байтов без учета знака |
;-----+
;| Вход: summand_1 и summand_2 – слагаемые. N – длина в байтах.              |
;-----+
;| Выход: summand_1 или carry+summand_1 – сумма с учетом переполнения.       |
;-----+
.data
summand_1    db      ?          ; первое слагаемое
N = $ - summand_1 ; длина в байтах summand_1 и summand_2
carry        db      0          ; перенос сложения последних байтов
summand_2    db      ?          ; второе слагаемое
.code
add_unsign_N proc
        mov     cl, N
        xor     si, si
cyc1:      mov     al, summand_2[si]
        adc     summand_1[si], al
        inc     si
        loop    cyc1
        jnc     end_p          ; проверка на переполнение
        adc     carry, 0
end_p:     ret
add_unsign_N endp

```

Программа учитывает возможное переполнение результата. Сегмент данных может быть задан, например, так:

```

.data
summand_1    db      0, 34, 56, 78 ; первое слагаемое
N = $ - summand_1 ; длина в байтах summand_1 и summand_2
carry        db      0          ; перенос сложения последних байтов
summand_2    db      0, 43, 65, 230 ; второе слагаемое

```

Далее при рассмотрении программы деления многобайтовых двоичных чисел нам понадобится макрокоманда сложения без учета знака чисел размером N байтов (порядок следования байтов не соответствует порядку следования байтов в процессорах Intel, то есть старший байт находится по младшему адресу). Приведем ее.

Сложение без учета знака чисел размером N байтов (макрокоманда)

```

.data
;summand_1    db      ?          ; первое слагаемое
;N = $ - summand_1 ; длина в байтах summand_1 и summand_2
;carry        db      0          ; перенос сложения последних байтов
;summand_2    db      ?          ; второе слагаемое
.code

```

```

add_unsign_N macro carry, summand_1, summand_2, N
:-----+
: | Макрокоманда: add_unsign_N carry, summand_1, summand_2, N
: | Сложение без учета знака чисел размером N байтов.
:-----+
: | Вход: summand_1 и summand_2 – слагаемые. N – длина в байтах.
:-----+
: | Выход: summand_1 или carry + summand_1 – сумма с учетом переполнения.
:-----+
: | Порядок следования байтов – старший байт по младшему адресу (не Intel).
:-----+
        local    cyc1, end_p
        mov     cl, N
        mov     si, N - 1
cyc1:    mov     al, summand_2[si]
        adc     summand_1[si], al
        dec     si
        loop    cyc1
        jnc     end_p
        adc     carry, 0
end_p:   nop
        endm

```

Сложение чисел размером 1 байт с учетом знака

```

:-----+
: | Процедура: add_sign. Сложение чисел размером 1 байт с учетом знака
:-----+
: | Вход: summand_1 и summand_2 – слагаемые.
:-----+
: | Выход: sum_b или sum_w – сумма в зависимости от наличия расширения знака.
:-----+
.data
sum_w      label word
summand_1  db      ?
carry      db      0           ; знаковое расширение
summand_2  db      ?
.code
add_sign   proc
        mov     al, summand_2
        add     summand_1, al
        jc      @@cf1_of1
        jo      @@cf0_of1
:-----+ cf=0 и of=0 -> результат верный и cf=1 of=0 -> результат верный
r true:    jmp     end_p           ; результат -> summand_1
@@cf1_of1: jno    @@cf1_of0
:-----+ cf=1 of=1 -> результат неверный
        mov     carry, 0ffh      ; расширение знака = 1. результат -> sum_w
        jmp     end_p
:-----+ cf=1 и of=0 -> результат верный
@@cf1_of0: jmp     r true           ; результат -> summand_1
:-----+ cf=0 и of=1 -> результат неверный
@@cf0_of1: mov     carry, 0       ; расширение знака = 0. результат -> sum_w
        jmp     end_p
end_p:    ret
add_sign  endp

```

Программа учитывает возможное переполнение результата и перенос в старшие разряды. Для этого отслеживаются условия, задаваемые флагами, и выполняются действия:

- CF = OF = 0 — результат правильный и является положительным числом;
- CF = 1, OF = 0 — результат правильный и является отрицательным числом;

- $CF = OF = 1$ — результат неправильный и является положительным числом, хотя верный результат должен быть отрицательным (для корректировки необходимо увеличить размер результата в два раза и заполнить это расширение нулевым значением);
- $CF = 0, OF = 1$ — результат неправильный и является отрицательным числом, хотя правильный результат должен быть положительным (для корректировки необходимо увеличить размер результата в два раза и произвести расширение знака).

Сложение чисел большей размерности (2/4 байта) со знаком выполняется аналогично, для этого необходимо внести изменения в соответствующие фрагменты программы. В частности, необходимо заменить директивы DB на DW/DD и регистр AL на AX/EAX.

Сложение с учетом знака чисел размером N байтов

```

:-----+
: | Процедура: add_sign N. Сложение с учетом знака чисел размером N байтов. |
:-----+
: | Вход: summand_1 и summand_2 – слагаемые. N – длина в байтах. |
:-----+
: | Выход: summand_1 или carry + summand_1 – сумма с учетом переполнения. |
:-----+
.data
summand_1 db ? ; первое слагаемое
N = $ - summand_1 ; длина в байтах summand_1 и summand_2
carry db 0 ; расширение знака
summand_2 db ? ; второе слагаемое
.code
add_sign_N proc
mov cx, N
mov si, 0 - 1
cycl: inc si
mov al, summand_2[si]
adc summand_1[si], al
loop cycl
jc @@cf1_of1
jo @@cf0_of1
;----- cf=0 и of=0 -> результат верный и cf=1, of=0 -> результат верный
r_true: jmp end_p ; результат -> summand_1
@@cf1_of1: jno @@cf1_of0
;----- cf=1 и of=1 -> результат неверный
mov carry, 0ffh ; расширение знака = 1. результат -> sum_w
jmp end_p
;----- cf=1 и of=0 -> результат верный
@@cf1_of0: jmp r_true ; результат -> summand_1
;----- cf=0 и of=1 -> результат неверный
@@cf0_of1: mov carry, 0 ; расширение знака = 0. результат -> sum_w
end_p: ret
add_sign_N endp

```

Сегмент данных может быть задан, например, так:

```

.data
summand_1 db 32, 126, -120 ; первое слагаемое
N = $ - summand_1 ; длина в байтах summand_1 и summand_2
carry db 0 ; расширение знака
summand_2 db 126, 125, -120 ; второе слагаемое

```

Программа учитывает возможное переполнение результата. Обратите внимание на порядок задания значений слагаемого. Если слагаемое положительное, то

проблем нет. Отрицательное слагаемое размером N байтов для своего задания требует некоторых допущений. Старший байт отрицательного слагаемого задается со знаком и в процессе трансляции будет преобразован в значение, являющееся двоичным дополнением исходного значения. Остальные байты в своем исходном виде должны быть частью дополнения исходного значения. Поэтому для работы с числами со знаком удобно иметь программу, которая бы выполняла вычисление значения модуля отрицательного числа и, наоборот, по значению модуля вычисляла его дополнение.

Вычисление двоичного дополнения числа размером N байтов

```

+-----+
+| Процедура: calc_complement. Вычисление дополнения числа размером N байтов. |
+-----+
+| Вход: bx — адрес операнда в памяти. |
+|      cx — длина операнда. |
+-----+
+| Выход: bx — адрес результата в памяти. |
+-----+
+| Порядок следования байтов — младший байт по младшему адресу. |
+-----+
.code
calc_complement proc
    xor     si, si
    neg     byte ptr [bx]      ; дополнение первого байта
    cmp     byte ptr [bx], 0    ; нулевой операнд — особый случай
    jne     short $+3
    stc
    dec     cx                  ; установить cf, так как есть перенос
    jcxz    @m1
@@cyc1:   inc     si
    not     byte ptr [bx][si]
    adc     byte ptr [bx][si], 0
    loop    @cyc1
@m1:      ret
calc_complement endp

```

Для значений размерностью 1/2/4 байта дополнение можно получать с помощью одной команды NEG:

```
neg     operand
```

Для значений в N байтов необходимо реализовывать алгоритм. Дополнение первого байта требуется вычислять с учетом того, что он может быть нулевым. Попытка получить его дополнение с помощью команды NEG обречена на провал. Флаг CF в этом случае также должен устанавливаться программно. Подумайте, почему?

Вычисление модуля числа размером N байтов

```

+-----+
+| Процедура: calc_abs. Вычисление модуля числа размером N байтов. |
+-----+
+| Вход: bx — адрес операнда в памяти. |
+|      cx — длина операнда. |
+-----+
+| Выход: bx — адрес результата в памяти. |
+-----+
+| Порядок следования байтов — младший байт по младшему адресу. |
+-----+

```



```
.code
calc_abs      proc
:-----  определим знак операнда
mov       si, cx
dec       si
test      byte ptr [bx][si], 80h ;проверяем знак операнда
jz        @exit                ;число положительное
call      calc_complement
@@exit:      ret
calc_abs     endp
```

Вычитание двоичных чисел

Вычитание чисел размером 1 байт без учета знака

```
+-----+
:| Процедура: sub_unsign. Вычитание чисел размером 1 байт без учета знака. |
+-----+
:| Вход: minuend и deduction – уменьшаемое и вычитаемое. |
+-----+
:| Выход: minuend – результат вычитания. |
+-----+
.data
minuend      db      ?           ; уменьшаемое
deduction    db      ?           ; вычитаемое
.code
sub_unsign   proc
mov          al, deduction
sub          minuend, al
:-----  оцениваем результат на случай уменьшаемое < вычитаемого
jnc         end_p                ;нет заема
:-----  обрабатываем ситуацию заема из старшего разряда –
:         получаем модуль (если нужно)
neg         minuend
end_p:       ret
sub_unsign   endp
```

Программа учитывает возможное соотношение: уменьшаемое < вычитаемого. Вычитание чисел большей размерности (2/4 байта) выполняется аналогично. При этом необходимо заменить директивы DB на DW/DD и регистр AL на AX/EAX.

Вычитание чисел размером N байтов без учета знака

```
+-----+
:| Процедура: sub_unsign_N. Вычитание чисел размером N байтов без учета знака. |
+-----+
:| Вход: minuend и deduction – уменьшаемое и вычитаемое. N – длина в байтах. |
+-----+
:| Выход: minuend – значение разности. |
+-----+
.data
minuend      db      ?           ; уменьшаемое
N = $ - minuend
deduction    db      ?           ; длина в байтах minuend и deduction
:         ; вычитаемое
.code
sub_unsign_N proc
mov          cx, N
xor          si, si
cyl:         mov     al, deduction[si]
sbb          minuend[si], al
jnc          @@m1
neg          minuend[si]
@@m1:        inc     si
loop         cyl
```

```

                ret
sub_unsign_N   endp

```

Программа учитывает возможный заем из старших разрядов. Длина уменьшаемого должна быть не меньше длины вычитаемого, недостающие разряды вычитаемого должны быть нулевыми. В любом случае, результат — абсолютное значение.

Сегмент данных может быть задан, например, так:

```

.data
N               equ      5                :длина в байтах minuend и deduction
Minuend         db       30, 43, 65, 230, 250 : уменьшаемое
Deduction       db       45, 34, 65, 78, 250 : вычитаемое

```

Вычитание чисел размером 1 байт с учетом знака

```

;+-----+
;| Процедура: sub_sign. Вычитание чисел размером 1 байт с учетом знака. |
;+-----+
;| Вход: minuend и deduction — уменьшаемое и вычитаемое. |
;+-----+
;| Выход: minuend — значение разности. |
;+-----+
.data
N               equ      2                : длина в байтах результата в ситуации
                                           : расширения знака для расчета его модуля
minuend         db       ?                : уменьшаемое
carry           db       0                : расширение знака
deduction       db       ?                : вычитаемое
.code
sub_sign        proc
                mov     al, deduction
                sub     minuend, al
                ;----- оцениваем результат
                jnc     no_carry           :нет заема
                ;----- обрабатываем ситуацию заема из старшего разряда —
                ;         получаем модуль (если нужно)
                neg     minuend
no_carry:       jmp     end_p
                jns     no_sign
                ;----- обрабатываем ситуацию получения отрицательного результата —
                ;         получаем модуль (если нужно)
                neg     minuend
no_sign:       jmp     end_p
                jno     no_overflow
                ;----- обрабатываем ситуацию переполнения —
                ;         получаем модуль (если нужно).
                ;         расширяем результат знаком — получаем модуль (если нужно)
                mov     carry, 0ffh
                call    calc_abs
no_overflow:
end_p:         ret
sub_sign       endp

```

Программа учитывает возможный заем из старших разрядов. Вычитание чисел большей размерности (2/4 байта) выполняется аналогично. Необходимо заменить директивы DB на DW/DD и регистр AL на AX/EAX. Подробности зависимости состояния флагов от результата см. в главе 8 «Арифметические команды» учебника.

Вычитание чисел размером N байтов с учетом знака

```

;+-----+
;| Процедура: sub_sign_N. Вычитание чисел размером N байтов с учетом знака. |
;+-----+

```

```

;| Вход: minuend и deduction – уменьшаемое и вычитаемое, N – длина в байтах. |
;+-----+
;| Выход: minuend – значение разности. |
;+-----+
.data
minuend      db      ?           ; уменьшаемое
              len_minuend = $ - minuend ; длина в байтах уменьшаемого и вычитаемого
carry        db      0           ; расширение знака
deduction    db      ?           ; вычитаемое
.code
sub_sign_N   proc
              mov     cx, len_minuend
              mov     si, 0
@@m1:        mov     al, deduction[si]
              sbb     minuend[si], al
              inc     si
              loop    @@m1
;----- оцениваем результат
              jnc     no_carry      ; нет заема
;----- обрабатываем ситуацию заема из старшего разряда –
;          получаем модуль (если нужно)
              N = len_minuend + 1
              mov     carry, 0ffh
              call    calc_abs
              jmp     end_p
no_carry:     jns     no_sign
;----- обрабатываем ситуацию получения отрицательного результата –
;          получаем модуль (если нужно)
              N = len_minuend
              call    calc_abs
              jmp     end_p
no_sign:     jno     no_overflow
;----- обрабатываем ситуацию получения отрицательного результата –
;          получаем модуль (если нужно).
;          расширить результат знаком – получаем модуль (если нужно)
              N = len_minuend + 1
              mov     carry, 0ffh
              call    calc_abs
no_overflow:
end_p:        ret
sub_sign_N   endp

```

Описанная процедура вычисляет модуль разности и учитывает возможный заем из старших разрядов. Если вычисления модуля разности не требуется, то прокомментируйте строки, содержащие команду **CALL calc_abs**. Подробности зависимости состояния флагов от результата см. в главе 8 «Арифметические команды» учебника.

Сегмент данных может быть задан, например, так:

```

.data
minuend      db      25h, 0f4h, 0eh ; уменьшаемое
              len_minuend = $ - minuend ; длина в байтах уменьшаемого и вычитаемого
carry        db      0           ; расширение знака
deduction    db      5h, 0f4h, 0fh ; вычитаемое

```

Далее при рассмотрении программы деления многобайтовых двоичных чисел нам понадобится макрокоманда вычитания с учетом знака чисел размером N байтов (порядок следования байтов не соответствует порядку следования байтов на процессорах Intel, то есть старший байт находится по младшему адресу). Приведем ее.

Вычитание с учетом знака чисел размером N байтов (макрокоманда)

```

:-----+
:| Макрокоманда: sub_sign_N minuend, deduction, N. |
:| Вычитание с учетом знака чисел размером N байтов. |
:-----+
:| Вход: minuend и deduction – уменьшаемое и вычитаемое. N – длина в байтах. |
:-----+
:| Выход: minuend – значение разности. |
:-----+
:| Порядок следования байтов – старший байт по младшему адресу (не Intel). |
:-----+
sub_sign_N      macro    minuend, deduction, N
                local    cycl, m1
                push     si
                mov      cx, N
                mov      si, N - 1
cycl:           mov      al, deduction[si]
                sbb      minuend[si], al

                jnc      m1
                neg      minuend[si]
m1:             dec      si
                loop     cycl
                pop      si
endm

```

Умножение двоичных чисел

В отличие от сложения и вычитания операция умножения реализуется двумя типами команд — учитывающими и не учитывающими знаки операндов.

Умножение чисел размером 1 байт без учета знака

```

:-----+
:| Программа: mul_unsign.asm. Умножение чисел размером 1 байт без учета знака. |
:-----+
:| Вход: multiplier1 и multiplier2 – множители размером 1 байт. |
:-----+
:| Выход: product – значение произведения. |
:-----+
.data
product      label word
product_1     label byte
multiplier1   db      ?           ; множитель 1
                                     ; (младшая часть произведения)
product_h     db      0           ; старшая часть произведения
multiplier2   db      ?           ; множитель 2
.code
mul_unsign    proc
                mov     al, multiplier1
                mul      multiplier2
                ;----- оцениваем результат
                jnc      no_carry    ; нет переполнения – на no_carry
                ;----- обрабатываем ситуацию переполнения
                mov      product_h, ah ; старшая часть результата
no_carry:      mov      product_1, al ; младшая часть результата
                ret
mul_unsign    endp
main:         ; ...
                call    mul_unsign
                ; ...
end main

```

Здесь все достаточно просто и реализуется средствами самого процессора. Проблема состоит лишь в правильном определении размера результата. Произведение чисел большей размерности (2/4 байта) выполняется аналогично. Необходимо заменить директивы DB на DW/DD, регистр AL на AX/EAX, регистр AH на DX/EDX.

Умножение чисел размером N и M байтов без учета знака

Для умножения чисел размером N и M байтов существует несколько стандартных алгоритмов, описанных в литературе [5, 8]. В этом разделе мы рассмотрим только один из них. По сути это закодированный на языке ассемблера процессоров Intel алгоритм умножения неотрицательных целых чисел, предложенный Кнуттом [5].

Умножение N-байтового числа на число размером M байтов

```

ПРОГРАММА mul_unsign_NM
// mul_unsign_NM – программа на псевдоязыке умножения N-байтового числа
// на число размером M байтов
// (порядок – старший байт по младшему адресу (не Intel)).
// Вход: U и V – множители размерностью N и M байтов соответственно.
// b = 256 – размерность машинного слова.
// Выход: W – произведение размерностью N + M байтов.
ПЕРЕМЕННЫЕ
INT_BYTE u[n]; v[n]; w[n+m]; k=0;
INT_WORD b=256; temp_word
НАЧ ПРОГ
ДЛЯ j:=M-1 ДО 0                                // J изменяется в диапазоне M-1..0
  НАЧ БЛОК 1
  // проверка на равенство нулю очередного элемента множителя (не обязательно)
  ЕСЛИ v[j]=0 ТО ПЕРЕЙТИ_НА m6
  k:=0; i:=n-1                                    // i изменяется в диапазоне N-1..0
  ДЛЯ i:=N-1 ДО 0
    НАЧ БЛОК 2
    // перемножаем очередные элементы множителей
    temp_word:=u[i]*v[j]+w[i+j+1]*k
    w[i+j+1]:=temp_word MOD b                      // остаток от деления temp_word\b -> w[i+j+1]
    k:=temp_word\b                                // целая часть частного temp_word\b -> k
  КОН БЛОК 2
  w[j]:=k
  m6:
  КОН БЛОК 1
КОН ПРОГ
+-----+
+| Программа: mul_unsign_NM.asm.                  |
+| Умножение N-байтового числа на число размером M байтов. |
+-----+
+| Вход: U - un-1...u1u0 – множитель_1 размерностью N байтов. |
+| V - vm-1...v1v0 – множитель_2 размерностью M байтов. |
+-----+
+| Выход: W – значение произведения (длина N + M). |
+-----+
+| Порядок – старший байт по младшему адресу (не Intel). |
+-----+
.data
U      db      ?                ; U
      I = $ - U                ; I = N
V      db      ?                ; V
      J = $ - V                ; J = M
      len_product = $ - U
W      db      len_product dup (0) ; W
K      db      0                ; перенос 0 ≤ k < 255
b      dw      100h              ; размер машинного слова
.code

```

```

mul_unsign_NM proc
push    dx
:m1:
        mov     bx, j - 1
        ;----- Для j:=M-1 ДО 0 //J изменяется в диапазоне M-1..0
        mov     cx, j
m2:
        ;НАЧ_БЛОК_1
        push    cx                ; вложенные циклы
        cmp     v[bx], 0          ; ЕСЛИ v[j]==0
        je      m6                ; ТО ПЕРЕЙТИ_НА m6
:m3
        mov     si, i - 1         ; i=0..n-1 ; k:=0; i:=n-1
                                   ; // i изменяется в диапазоне N-1..0
        mov     cx, i
        mov     k, 0              ; Для i:=N-1 ДО 0 НАЧ_БЛОК_2
        ;----- // перемножаем очередные элементы множителей
m4:
        mov     al, u[si]         ; temp_word:=u[i]*v[j]+w[i+j+1]+k
        mul     byte ptr v[bx]
        movzx   dx, w[bx+si+1]
        add     ax, dx
        mov     dl, k
        add     ax, dx            ; t=ax - временная переменная
        ;----- w[i+j+1]:=temp_word MOD b
        ; //остаток от деления temp_word\b -> w[i+j+1]
        ; k:=temp_word\b
        ; //целая часть частного temp_word\b -> k
        xor     dx, dx
        div     b
        mov     k, al
        mov     w[bx+si+1], dl
:m5
        dec     si
        loop    m4                ; КОН БЛОК_2
        mov     al, k              ; w[j]:=k
m6:
        dec     bx
        pop     cx
        loop    m2                ; КОН БЛОК_1
        ret                     ; КОН ПРОГ
        pop     dx
mul_unsign_NM endp
main:
        ...
        call    mul_unsign_NM
        ....
end main

```

В отличие от обычного умножения «в столбик» в данном алгоритме сложение частичных произведений выполняется параллельно умножению. Программа производит умножение значений в следующем порядке — старший байт по младшему адресу. Это неестественно для процессоров Intel, поэтому программу необходимо соответствующим образом изменить. Текст измененной процедуры `mul_unsign_NM_I` приведен среди файлов, прилагаемых к книге.

Процедуру умножения чисел без учета знака `mul_unsign_NM` удобно представить в виде макрокоманды `mul_unsign_NM_r u, i, v, j, w`. Это без излишних усложнений сделает ее вызов более универсальным. При последующем рассмотрении программы деления многобайтовых двоичных чисел она будет использована нами с большой пользой. Текст макрокоманды приведен среди файлов, прилагаемых к книге. Там же имеется вариант этой макрокоманды `mul_unsign_NM` на случай естественного для процессоров Intel расположения операндов — младший байт по младшему адресу.

Умножение чисел размером 1 байт с учетом знака

```

+-----+
+| Программа: mul_sign.asm. Умножение чисел размером 1 байт с учетом знака. |
+-----+
+| Вход: multiplier1 и multiplier2 – множители размерностью 1 байт со знаком. |
+-----+
+| Выход: product – значение произведения. |
+-----+
.data
product      label word
product_l    label byte
multiplier1  db      ?           ; множитель 1
                                ; (младшая часть произведения)
product_h    db      0           ; старшая часть произведения
multiplier2  db      ?           ; множитель 2
.code
mul_sign     proc
mov          al, multiplier1
imul         multiplier2
;----- оцениваем результат
jnc          no_carry            ; нет переполнения – на no_carry
;----- обрабатываем ситуацию переполнения
mov          product_h, ah       ; старшая часть результата,
                                ; знак результата – старший бит product_h
no_carry:    mov          product_l, al ; младшая часть результата.
                                ; product_h – расширение знака
ret
mul_sign     endp
main:
;...
call        mul_sign
;...

end main

```

Аналогично умножению без знака здесь также все достаточно просто и реализуется средствами самого процессора. Проблема та же — правильное определение размера результата. Произведение чисел большей размерности (2/4 байта) выполняется аналогично. Необходимо заменить директивы DB на DW/DD, регистр AL на AX/EAX, регистр AH на DX/EDX. Более того, в отличие от команды MUL команда IMUL допускает более гибкое расположение операндов.

Умножение чисел размером N и M байтов с учетом знака

Система команд процессора содержит два типа команд умножения — с учетом знаков операнда (IMUL) и без него (MUL). При умножении операндов размером 1/2/4 байта учет знака производится автоматически — по состоянию старших (знаковых) битов. Если умножаются числа размером в большее количество байтов, то для получения правильного результата необходимо учитывать знаковые разряды только старших байтов. В основе программы, реализующей алгоритм умножения чисел размером N и M байтов с учетом знака, лежит рассмотренная выше процедура умножения чисел произвольной размерности без учета знака.

Умножение N-байтового числа на число размером M байтов с учетом знака

```

// mul_sign_NM – программа на псевдоязыке умножения N-байтового числа
// на число размером M байтов
// (порядок – старший байт по младшему адресу (не Intel))
// Вход: U и V – множители со знаком размерностью N и M байтов соответственно;

```

```

// b = 256 – размерность машинного слова.
// Выход: W – модуль (дополнение) произведения размерностью N + M байтов.
ПЕРЕМЕННЫЕ
INT_BYTE u[n];           //множитель 1 размерностью N байтов
INT_BYTE v[n];           //:множитель 2 размерностью M байтов
INT_BYTE w[n+m]; k=0;    //перенос 0 ≤ k < 255
INT_BYTE sign=0;         //информация о знаке
INT_WORD b=256; temp_word //b – размер машинного слова
НАЧ ПРОГ
// определим знак результата
ЕСЛИ БИТ_7_БАЙТА((u[0] AND 80h) XOR v[0])=1 TO sign:=1 //результат
// будет отрицательным
// получим модули сомножителей:
u:=|u|
v:=|v|
w:=mul_unsign_NM() //в этой точке – модуль результата
// восстанавливаем знак результата
ЕСЛИ sign=0 TO ПЕРЕЙТИ_НА @@m
// для отрицательного результата вычислить дополнение значения w длиной i+j
w:=calc_complement_r() // в этой точке – двоичное дополнение результата
@@m:
КОН ПРОГ
+-----+
| Программа: mul_sign_NM.asm. |
| Умножение N-байтового числа на число размером M байтов. |
+-----+
| Вход: U - un-1...u0 – множитель 1 размерностью N байтов. |
| V - vm-1...v0 – множитель 2 размерностью M байтов. |
+-----+
| Выход: W – значение произведения (длина N + M). |
+-----+
| Порядок – старший байт по младшему адресу (не Intel). |
| Включить описание процедур calc_complement_r, calc_abs_r, mul_unsign_NM. |
| Помните, что задание отрицательных многобайтных значений в |
| сегменте данных должно производиться в дополнительном коде |
| (и в порядке старший байт по младшему адресу)! |
+-----+
.data
U db ? : множитель 1 размерностью N байтов
I = $ - U
V db ? : множитель 2 размерностью M байтов
J = $ - V
len_product = $ - U
W db len_product dup (0) : результат длиной N + M байтов
k db 0 : перенос 0 ≤ k < 255
b dw 100h : размер машинного слова
sign db 0 : информация о знаке
.code
mul_sign_NM proc
:-----определим знак результата
xor ax, ax : ЕСЛИ БИТ_7_БАЙТА((u[0] AND 80h)
: XOR v[0])=1 TO sign:=1

mov al, u
and al, 80h
xor al, v
bt ax, 7
jnc $ + 7
mov sign, 1 : результат будет отрицательным
lea bx, v : модули сомножителей: u:=|u|; v:=|v|
mov cx, j
call calc_abs_r
lea bx, u
mov cx, i
call calc_abs_r

```



```

:----- теперь умножаем
call    mul_unsign_NM    ; w:=mul_unsign_NM()
:----- в этой точке — модуль результата
:       восстанавливаем знак результата
xor     si, si
cmp     sign, 0          ; ЕСЛИ sign==0 ТО ПЕРЕЙТИ_НА @@m
je      @@m
:----- для отрицательного результата вычислить
:       дополнение значения w длиной i + j
mov     cx, i + j        ; w:=calc_complement_r(); w[0]:=0-w[0]
lea     bx, w
call    calc_complement_r
:----- в этой точке — двоичное дополнение результата
@@m:
mul_sign_NM    ret          ;КОН_ПРОГ
main:
        :...
        call    mul_sign_NM
        :...

end main

```

Процедура `mul_sign_NM` выполняет умножение с учетом знака исходных значений в следующем порядке байтов — старший байт по младшему адресу. Если вы используете отрицательные значения операндов, то для правильной работы тестовой программы в сегменте данных их необходимо задать в дополнительном коде.

В данной программе появились две новые процедуры — `calc_complement_r` и `calc_abs_r`, вычисляющие соответственно дополнение и модуль числа размером N байтов. Подобные процедуры уже были разработаны и введены нами для значений, порядок следования байтов которых характерен для процессоров Intel. Чтобы различать эти две пары процедур, процедуры для вычисления дополнения и модуля числа размером N байтов с порядком следования байтов, отличным от Intel, мы назвали реверсивными.

Вычисление дополнения числа размером N байтов (реверсивное)

```

:-----+-----
:| Процедура: calc_complement_r. Вычисление дополнения числа длиной N байтов. |
:-----+-----
:| Вход: BX — адрес операнда в памяти. CX — длина операнда.                |
:-----+-----
:| Выход: BX — адрес дополнения операнда в памяти.                          |
:-----+-----
:| Порядок — старший байт по младшему адресу (не Intel).                    |
:-----+-----
calc_complement_r proc
    dec     cx
    mov     si, cx
    neg     byte ptr [bx][si]    ; дополнение первого байта
    cmp     byte ptr [bx][si], 0 ; операнд = 0 — особый случай
    jne     short $+3
    stc
    jcxz    @@exit_cycl          ; установить cf, так как перенос
@@cycl:
    dec     si
    not     byte ptr [bx][si]
    adc     byte ptr [bx][si], 0
    loop    @@cycl
@@exit_cycl:
    ret
calc_complement_r endp

```

Для значений размерностью 1/2/4 байта дополнение можно получать с помощью одной команды **NEG**:

```
neg    operand
```

Дополнение значений N байтов вычисляет алгоритм, реализованный в процедуре `calc_complement_r`. Обратите внимание, что первый байт может быть нулевым, поэтому алгоритм учитывает это обстоятельство. Попытка получить его дополнение с помощью команды **NEG** обречена на провал. Флаг **CF** в этом случае также должен устанавливаться программно. Подумайте, почему?

Вычисление модуля числа размером N байтов (реверсивное)

```
+-----+
:| Процедура: calc_abs_r. Вычисления модуля числа размером N байтов. |
+-----+
:| Вход: BX — адрес операнда в памяти; CX — длина операнда.         |
+-----+
:| Выход: BX — адрес модуля операнда в памяти.                       |
+-----+
:| Порядок — старший байт по младшему адресу (не Intel!).           |
+-----+
.code
calc_abs_r    proc
               test    byte ptr [bx], 80h : проверяем знак операнда
               jz      @@exit              : число положительное
               call    calc_complement_r
@@exit:       ret
calc_abs_r    endp
```

Для вычислений над операндами, порядок следования байтов которых характерен для процессоров **Intel**, нам придется разработать еще один вариант процедуры умножения значений размерностью в произвольное количество байтов. Псевдокод и соответствующая ему программа на ассемблере `mul_sign_NM_I` приведены среди файлов, прилагаемых к книге.

Деление двоичных чисел

Аналогично операции умножения деление реализуется двумя типами команд — учитывающими и не учитывающими знаки операндов. Эти команды накладывают ощутимые ограничения на размерность (а соответственно и на диапазон значений) операндов. Поэтому мы рассмотрим вначале пример стандартного применения команд, после чего займемся реализацией алгоритмов деления чисел произвольной размерности.

```
+-----+
:| Программа: div_unsign.asm. Деление без учета знака                |
:| значения размером 2 байта на значение размером 1 байт.          |
+-----+
:| Вход: u — делимое; v — делитель.                                   |
+-----+
:| Выход: w — частное, r — остаток.                                    |
+-----+
.data
u          dw    ?           : делимое
v          db    ?           : делитель
w          db    0
r          db    0
.code
div_unsign proc
```

```

        mov     ax, u
        div     v
:----- сформировать результат
        mov     r, ah      : остаток
        mov     w, al      : частное
        ret
div_unsign
main:
        :...
        call    div_unsign
        :...
end main

```

Деление чисел большей размерности (4/8 байтов) выполняется аналогично. Необходимо заменить директивы DB на DW/DD, регистр AX на EAX/EDX:EAX, регистр AL на AX/EAX, регистр AH на DX/EDX.

Деление с учетом знака значения размером 2 байта на значение размером 1 байт

```

+-----+
: Программа: div_sign.asm. Деление с учетом знака
: значения размером 2 байта на значение размером 1 байт.
+-----+
: Вход: u – делимое; v – делитель.
+-----+
: Выход: w – частное, r – остаток.
+-----+
.data
u      dw      ?      : делимое
v      db      ?      : делитель
w      db      0
r      db      0
.code
div_sign proc
        mov     ax, u
        idiv    v
:----- сформировать результат
        mov     r, ah      : остаток
        mov     w, al      : частное
:----- если нужно получить модуль – уберите знаки комментария
:        mov     cx, 1      : длина операнда
:        lea     bx, w
:        call    calc_abs
:        или
:        neg     w
        ret
div_sign endp
main:
        :...
        call    div_sign
        :...
end main

```

Деление чисел большей размерности (4/8 байтов) выполняется аналогично. Необходимо заменить директивы DB на DW/DD, регистр AX на EAX/EDX:EAX, регистр AL на AX/EAX, регистр AH на DX/EDX.

Деление N-байтового беззнакового целого на число размером 1 байт

```

+-----+
: Программа: div_unsign N_1.asm. Деление без учета знака
: значения размером N байтов на значение размером 1 байт.
+-----+

```

```

:| Вход: u – делимое; v – делитель. |
:-----+-----|
:| Выход: w – частное, r – остаток. |
:-----+-----|
:| Порядок следования байтов – старший байт по младшему адресу (не Intel). |
:-----+-----|
.data
u          db      ?           : делимое
N = $ - u          : длина в байтах значения u
v          db      ?           : делитель
w          db      N dup (0)
r          dw      0           : остаток
.code
div_unsign_N_1 proc
mov        r, 0
xor        si, si             ; j = 0
mov        cx, N
xor        dx, dx
xor        bx, bx
@@m1:     mov        ax, 256     : размер машинного слова
mul        r                  : результат в dx:ax
mov        bl, u[si]
add        ax, bx
div        v
;----- сформировать результат
mov        w[si], ax          : частное
shr        ax, B
mov        r, ax              : остаток в r
inc        si
loop       @@m1
ret
div_unsign_N_1 endp
main:
;...
call       div_unsign_N_1
;...
end main

```

Сегмент данных может быть задан, например, так:

```

.data
u          db      5, 6, 7     : делимое
N = $ - u          : длина в байтах значения u
v          db      15          : делитель
w          db      N dup (0)
r          dw      0           : остаток

```

В программе `div_unsign_N_1.asm` порядок следования байтов делимого неестествен для процессора Intel. Поэтому среди файлов, прилагаемых к книге, приведен вариант программы `div_unsign_N_1_I.asm`, в котором эта проблема устранена.

Далее при рассмотрении программы деления многобайтовых двоичных чисел нам понадобится макрокоманда `div_unsign_N` деления N -байтового беззнакового целого на число размером 1 байт (порядок следования байтов не характерен для процессоров Intel, то есть старший байт находится по младшему адресу). Текст макрокоманды приведен среди файлов, прилагаемых к книге.

Деление (N+M)-разрядного беззнакового целого на число размером M байтов

```

// div_unsign_NM – программа на псевдоязыке деления
// (N+M)-разрядного беззнакового целого на число размером N байтов
// (порядок – старший байт по младшему адресу (не Intel))

```

```

// Вход: U и V —  $u = u_{m-1} \dots u_1 u_0$  — делимое;  $v = v_{n-1} \dots v_1 v_0$  — делитель, m — длина делимого
// n — длина делителя; b=256 — основание системы счисления.
// Выход:  $q = q_{m-1} \dots q_1 q_0$  — частное,  $r = r_{n-1} \dots r_1 r_0$  — остаток.
// Ограничения:  $v_{n-1} \neq 0$  OR 0ffh; N>1.
ПЕРЕМЕННЫЕ
INT_BYTE u[n+m]: // делимое размерностью n+m байтов (нумерация от 0 до n+m-1) —
// дополнительный байт для нормализации
// байты делимого размещаются, начиная с n+m-1 (старший байт делимого)
// и заканчивая 0 байтом (младший байт делимого)
INT_BYTE v[n]: // делитель размерностью n байтов (нумерация от 0 до n-1)
INT_BYTE w[m+1]: // для промежуточных вычислений
INT_BYTE q[m]: // частное
INT_BYTE r[n]: // остаток
INT_WORD qq=0: // частичное частное
INT_WORD rr=0: // частичный остаток
INT_BYTE temp_r[n]:
INT_BYTE borrow=0: // факт заема на шаге D4
INT_BYTE k=0: // перенос  $0 \leq k < 255$ 
INT_BYTE sign=0: // информация о знаке
INT_WORD b=256; temp // b — размер машинного слова
INT_BYTE d=0, carry=0, j, mm
BOOL tf=TRUE
НАЧ ПРОГ
// шаг 1 — нормализация:
d:=b/(v[n-1]+1)
u[n+m..0]:=u[n+m-1..0]*d
v[n-1..0]:=v[n-1..0]*d
// шаг 2 — начальная установка j:
mm:=m-n; j:=mm
@@m7: // шаг 3 — вычислить частичное частное qq:
qq:=(u[j+n]*b+u[j+n-1]) / v[n-1]
rr:=(u[j+n]*b+u[j+n-1]) MOD v[n-1]
ДЕЛАТЬ ПОКА tf
НАЧ БЛОК 1
ЕСЛИ (qq=b OR qq*v[n-2] > b*rr+ u[j+n-1]) TO
НАЧ БЛОК 2
qq:=qq-1
rr:=rr+v[n-1]
ЕСЛИ (rr≥b) TO tf:=FALSE
КОН БЛОК 2
ИНАЧЕ tf:=FALSE
КОН БЛОК 1
// шаг 4 — умножить и вычесть:
u[j+n..j]:=u[j+n..j]-qq*v[n-1..0]
ЕСЛИ (u[j+n..j]<0) TO // запоминаем факт заема, получаем дополнение
НАЧ БЛОК 3
borrow:=1
u[j+n..j]:=calc_complement_r(u[j+n..j])
КОН БЛОК 3
// шаг 5 — проверка остатка:
q[j]:=qq
ЕСЛИ (borrow<>1) TO
НАЧ БЛОК 4
// шаг 6 — компенсирующее сложение:
q[j]:=q[j]-1
u[j+n..j]:=u[j+n..j]+v[n-1..0]
КОН БЛОК 4
// шаг 7 — цикл по j:
j:=j-1
ЕСЛИ (j≥0) TO ПЕРЕЙТИ НА @@m7
// шаг 8 — денормализация:
// вычислим остаток:
r[n-1..0]:=u[n-1..0]/d
// q[m..0] — частное, r[n-1..0] — остаток
КОН ПРОГ

```

```

+-----+
: | Программа: div_unsign_NM.asm.
: | Деление (N+M)-байтового беззнакового целого на число размером N байтов.
+-----+
: | Порядок – старший байт по младшему адресу (не Intel).
: | Необходимо включить описание процедур и макрокоманд:
: | calc_complement_r, sub_sign_N, add_unsign_N, div_unsign_N, mul_unsign_NM.
+-----+
.data
u0      db      0          : дополнительный старший байт делимого
                        : для нормализации
u       db      ?          : делимое
m = $ - u          : длина в байтах значения u
v0      db      0          : для компенсирующего сложения  $0v_{n-1}...v_1v_0$ 
v       db      ?          : делитель
n = $ - v          : длина в байтах значения v
mm = m - n
w       db      m + 1 dup (0) : для промежуточных вычислений
q       db      mm dup (0)   : частное
qq      dw      0            : частичное частное :qq db 0
rr      dw      0            : частичный остаток
r       db      n dup (0)    : остаток
d       db      0
temp    dw      0
temp_r  db      n dup (0)
borrow  db      0            : факт заема на шаге D4
k       db      0            : перенос  $0 \leq k < 255$ 
b       dw      100h         : размер машинного слова
carry   db      0
.code
div_unsign_NM proc
:----- // шаг D1 (1) – нормализация
xor     ax, ax              : d:=b/(v[n-1]+1)
mov     di, v
inc     di                  :  $v_{n-1}+1$ 
mov     ax, b
div     di
mov     d, al               : d=b/(v1+1)
mul_unsign_NM u, m, d, 1, w :  $u[n+m-1..0] := u[n+m-1..0]*d$ 
cld
push    ds
pop     es
lea     si, w
lea     di, u0
mov     cx, m + 1
rep     movsb
mul_unsign_NM v, n, d, 1, w :  $v[n-1..0] := v[n-1..0]*d$ 
cld
push    ds
pop     es
lea     si, w + 1
lea     di, v
mov     cx, n
rep     movsb
:----- // шаг D2 – начальная установка j: mm:=m-n; j:=mm
mov     si, 0               : n=0 (? n=n+m)
@@m7:   :----- // шаг 3 – вычислить частичное частное qq
@@m1:   xor     ax, ax        : qq:=(u[j+n]*b+u[j+n-1]) / v[n-1]
        mov     al, u0[si]    : rr:=(u[j+n]*b+u[j+n-1]) MOD v[n-1]
        mul     b
        shl     eax, 16
        shrd    eax, edx, 16 : результат умножения в eax
        xor     edx, edx
        mov     di, u0[si+1]

```

```

add     eax, edx
shld    edx, eax, 16    ; восстановили пару dx:ax для деления
xor     bx, bx
mov     bl, v           ; v->bx
div     bx
mov     qq, ax
mov     rr, dx

@@m2:   ; проверим
@@m4:   mov     ax, qq    ; выполнение неравенства
        cmp     ax, b    ; qq<=b
        je      @m9      ; на qq=qq-1
;----- qq*vn-2>b*rr+uj+n-2
        mul     v + 1    ; qq*vn-2
        mov     temp, ax ; temp=vn-2*qq
        xor     ax, ax
        mov     ax, b
        mul     rr
        xor     dx, dx
        mov     dl, u0[si+2]
        add     ax, dx
        cmp     temp, ax ; qq*vn-2 > b*rr+uj+n-2
        jna     @m3
@@m9:   dec     qq
        xor     ax, ax
        mov     al, v
        add     rr, ax
        jmp     @m4
@@m3:   ;----- D4 //шаг 4 – умножить и вычесть
        mul_unsign_NM v, n, qq, 1, w : u[j+n...j]:=u[j+n...j]-qq*v[n-1..0]
        mov     bx, si
        push    si
        sub_sign N u0[bx], w, <n+1> ; v<->w
        jnc     @m5      ; переход, если нет заема
                        ; (результат положительный)
        mov     borrow, 1 ; запоминаем факт заема, получаем дополнение
        pop     si
        lea     bx, u0[si]
        mov     cx, n + 1
        call    calc_complement_r
;----- D5 //шаг 5 – проверка остатка
@@m5:   mov     ax, qq    ; q[j]:=qq
        mov     q[si], al
        cmp     borrow, 1 ; был заем на шаге 4 ? (ЕСЛИ borrow<=1 ТО)
        jne     @m6
;----- D6 – компенсирующее сложение
        mov     borrow, 0 ; аннулируем факт заема
        dec     q[si]     ; q[j]:= q[j]-1
        mov     bx, si
        push    si       ; u[j+n...j]:=u[j+n...j]+v[n-1..0]
        add_unsign N carry, u0[bx], v0, <n+1> ;перенос не нужен
;----- D7 //шаг 7 – цикл по j: j:=j-1
@@m6:   pop     si
        inc     si
        cmp     si, mm    ; ЕСЛИ j≥0 ТО ПЕРЕЙТИ_НА @@m7
        jle     @m7
;----- D8 – шаг В – денормализация
        mov     bx, si    ; // вычислим остаток:
        div_unsign_N u0[bx], N, d, r, temp_r : r[n-1..0]:=u[n-1..0]/d
        ret     ; // q[m..0] – частное, r[n-1..0] – остаток
div_unsign_NM
main:   ;...
        call    div_unsign_NM
        ;...
end main

```

Значения делимого и делителей в сегменте данных могут быть заданы, например, так:

```
.data
u0      db      0          : дополнительный байт для нормализации
u       db      1fh, 0c3h, 12h, 0ffh : делимое — 1fc312ffh
m = $ - u      : длина в байтах значения u
v0      db      0          : для сложения 0vn-1...v1v0
v       db      3fh, 50h    : делитель — 3f50h
n = $ - v      : длина в байтах значения v
mm = m - n
:...
```

Программа не работает, если первый байт делителя равен 0ffh. Сам алгоритм не изменяется, а проблема устраняется просто — необходимо лишь в нужных местах поменять разрядность операндов.

Порядок следования байтов делимого неестествен для процессора Intel. Программу деления многобайтовых двоичных чисел с порядком следования байтов «младший байт по младшему адресу» остается вам в качестве упражнения.

Двоично-десятичные числа (BCD-числа)

Работай постоянно, не почитай работу для себя бедствием или бременем и не желай себе за это похвалы и участия. Общее благо — вот чего ты должен желать.

Марк Аврелий

Понятие о BCD-числах и элементарных действиях с ними приведены в главе 8 «Арифметические команды» учебника. В отличие от действий с двоичными числами работа с BCD-числами в процессоре реализована косвенно. В его системе команд нет инструкций, которые непосредственно выполняют основные арифметические действия над BCD-числами в том виде, как это делается для двоичных операндов. Тем более нет средств, которые каким-то образом учитывали бы знак. Все это должен делать сам программист. Ниже приведены макрокоманды, которые выполняют базовые арифметические операции над BCD-числами различной разрядности.

Неупакованные BCD-числа

Перед тем как приступать к работе с BCD-числами, необходимо договориться о том, как они будут представляться в оперативной памяти — старший байт BCD-числа по старшему адресу или старший байт BCD-числа по младшему адресу. Изменить фрагмент программы для поддержки того или иного способа представления чисел не представляет особого труда. Поэтому для однозначности рассуждений выберем естественный для процессоров Intel порядок следования байтов — младший байт по младшему адресу.

Сложение неупакованных BCD-чисел (макрокоманда)

```
+-----+
| Макрокоманда: add bcd summand_1. Сложение неупакованных BCD-чисел |
| размером len_1 и Ten_2 байтов и помещение результата в sum.       |
+-----+
```



```

:| Вход: summand_1 и summand_2 – адреса младших байтов слагаемых;
:| len_1 и len_2 – длины слагаемых в байтах.
:-----
:| Выход: sum – адрес младшего байта поля суммы.
:| Желательно, чтобы это поле имело длину на единицу больше, чем длина
:| самого длинного слагаемого.
:-----
:| Порядок следования байтов – младший байт по младшему адресу (Intel).
:-----
add_bcd macro summand_1, len_1, summand_2, len_2, sum
    local m1, m2, m3
    push si
    push bx
    mov ax, len_1
    cmp ax, len_2
    jna m2
    mov cx, len_1 ; длина большего для сложения (см. ниже)
    push cx
    mov cx, len_2 ; длина меньшего для сложения (см. ниже)
    push cx
    mov cx, ax
    lea bx, summand_1 ; адрес большего источника для сложения
    lea si, summand_2 ; адрес меньшего источника для movsb
    jmp m2
m2: mov cx, len_2 ; длина большего для сложения (см. ниже)
    push cx
    mov cx, len_1 ; длина меньшего для сложения (см. ниже)
    push cx
    mov cx, len_2
    lea bx, summand_2 ; адрес большего источника для сложения
    lea si, summand_1 ; адрес меньшего источника для movsb
m3: ; ----- заполняем sum нулями – длина определена выше
    cld
    xor al, al
    lea di, sum
    rep stosb
    ; ----- пересылка меньшего (по длине) BCD-числа в sum
    cld
    push ds
    pop es
    lea di, sum ; адрес источника см. выше
    pop cx ; длина была определена выше
    ; и соответствует меньшему BCD-числу
    rep movsb
    pop cx ; цикл по большому
    xor si, si
m1: mov al, [bx][si]
    adc al, sum[si]
    aaa
    mov sum[si], al
    inc si
    loop m1
    adc sum[si], 0
    pop bx
    pop si
endm

```

Макрокоманда обрабатывает байты, исходя из предположения, что младший байт слагаемых находится по младшему адресу. Слагаемые необязательно имеют одинаковую длину, но сумма должна иметь размер поля на единицу больше, чем длина самого длинного слагаемого. В начале процесса сложения в поле результата помещается меньшее (по длине) слагаемое.

```

+-----+
| Вход: u – адрес первого сомножителя.
|       i – длина u.
|       v – адрес второго сомножителя.
|       j – длина v.
|       w – адрес области длиной i+j байтов для помещения результата.
|       b=256 – размерность машинного слова.
+-----+
| Выход: w – произведение размерностью i+j байтов.
+-----+
| Порядок следования байтов – младший байт по младшему адресу (Intel).
+-----+
.data
k      db      0          ; перенос 0 ≤ k < 255
b      dw      10         ; основание системы счисления
.code
mul_bcd macro u, i, v, j, w
    local m2, m4, m6
    push si
;----- очистим w
    cld
    push ds
    pop es
    xor ax, ax
    lea di, w
    mov cx, i + j
    rep stosb
:m1
    xor bx, bx          ; j=0..m-1
    mov cx, j
:m2:
    push cx              ; вложенные циклы
    cmp v[bx], 0
    je m6
:m3
    xor si, si          ; i=0..n-1
    mov cx, i
    mov k, 0
    mov al, u[si]
    mul v[bx]
    xor dx, dx
    mov d1, w[bx+si]
    add ax, dx
    xor dx, dx
    mov d1, k
    add ax, dx          ; t=(ax) – временная переменная
;----- корректируем результат – (ah)=цифра переноса, (al)=результат
    aam
    mov k, ah
    mov w[bx+si], al
:m5
    inc si
    loop m4
    mov al, k
    mov w[bx+si], al
:m6:
    inc bx
    pop cx
    loop m2
    pop si
endm

```

Нам понадобится и другой вариант этой команды — `mul_bcd_r`, который обрабатывает операнды с порядком следования байтов — старший байт по младшему адресу. Он приведен среди файлов, прилагаемых к книге.

Деление N-байтового беззнакового целого BCD-числа на BCD-число размером 1 байт (макрокоманда)

```

+-----+
| Макрокоманда: div_bcd_1_r. Деление N-байтового беззнакового целого |
| BCD-числа на BCD-число размером 1 байт.                             |
+-----+
| Вход: u – делимое; N – длина делимого, v – делитель.               |
+-----+
| Выход: w – частное, r – остаток.                                     |
+-----+
| Порядок следования байтов – старший байт по младшему адресу (не Intel) (!) |
+-----+
div_bcd_1_r    macro    u, N, v, w, r
                local   ml
                mov     r, 0
                lea     si, u                ; j=0
                xor     di, di              ; j=0
                mov     cx, N
                xor     dx, dx
                xor     bx, bx
ml:            mov     ah, r
                mov     al, [si]
                aad
                div     v
                ;----- сформировать результат
                mov     w[di], al          ; частное
                mov     r, ah              ; остаток
                inc     si
                inc     di
                loop    ml
                ;----- если нужно – получим модуль (уберите знаки комментария)
                mov     cx, N              ; длина операнда
                lea     bx, w
                call    calc_abs_r
endm

```

Деление неупакованных BCD-чисел

```

+-----+
| Программа: div_bcd_NM_r.asm.                                         |
| Деление неупакованных BCD-чисел u и v размером n + m и n байтов.   |
+-----+
| Вход: u=u1u2...um+n – делимое, v=v1v2...vn – делитель.         |
| b=256 – размер машинного слова.                                     |
+-----+
| Выход: q=q1q2...qm – частное, r=r1r2...rn – остаток.           |
+-----+
| Порядок следования байтов – старший байт по младшему адресу (не Intel) (!) |
| Нужно вставить описание макрокоманд calc_complement_r, mul_bcd_r, |
| sub_bcd_r, add_bcd_r, div_bcd_1_r.                                   |
+-----+
.data
u0          db      0                ; дополнительный байт для нормализации
u           db      1, 0, 3, 5, 9, 4, 6 ; делимое
            m = $ - u                ; длина в байтах значения u
v0          db      0                ; для сложения 0v1v2...vn
v           db      5, 9, 1          ; делитель
            n = $ - v                ; длина в байтах значения v
            nm = m - n
w           db      m+1 dup (0)       ; для промежуточных вычислений
q           db      nm dup (0)       ; частное
qq          db      0
r           db      n dup (0)        ; остаток
b           dw      10                ; основание системы счисления

```

```

d          db      0
temp       dw      0
temp_r     db      n dup (0)
borrow     db      0          ; факт заема на шаге D4
k          db      0          ; перенос 0 ≤ k < 255
carry      db      0

.code
div_bcd_NM_r proc
;----- D1 - нормализация
xor        ax, ax
mov        dl, v
inc        dl          ; v1+1
mov        ax, b
div        dl          ; d=b/(v1+1)
mov        mul_bcd_r u, m, d, 1, w
cld
push       ds
pop        es
lea        si, w
lea        di, u0
mov        cx, m + 1
rep        movsb
mul_bcd_r v, n, d, 1, w
cld
push       ds
pop        es
lea        si, w + 1
lea        di, v
mov        cx, n
rep        movsb

:D2:
mov        si, 0          ; n=0

:D3:
@@m7:      mov        al, u0[si]
cmp        v, al
jne        @@m1
mov        qq, 9          ; qq=b-1
jmp        @@m2

@@m1:      xor        ax, ax
mov        al, u0[si]
mul        b
xor        dx, dx
mov        dl, u0[si+1]
add        ax, dx
mov        bl, v          ; v->bx
div        bl
mov        qq, al

@@m2:      ;----- проверим выполнение неравенства
@@m4:      mul        v+1
mov        temp, ax        ; temp=v2*qq
xor        ax, ax
mov        al, u0[si]
mul        b
xor        edx, edx
mov        dl, u0[si+1]
add        dx, ax
mov        al, qq
mul        v          ; eax=qq*v1
sub        dx, ax
xchg       dx, ax
mul        b
xor        bx, bx
mov        bl, u0[si+2]
add        ax, bx
cmp        temp, ax

```

```

        jna     @m3
        dec     qq
        mov     al, qq
        jmp     @m4
@m3:    :----- D4
        mul_bcd_r v, n, qq, 1, w
        mov     bx, si
        push    si
        sub_bcd_r u0[bx], <n+1>, w, <n+1>, u0[bx] ; v<->w
        jnc     @m5 ; переход, если нет заема
                          ; (результат положительный)
        mov     borrow, 1 ; запоминаем факт заема, получаем дополнение
        pop     si
@m5:    :----- D5
        mov     al, qq
        mov     q[si], al
        cmp     borrow, 1 ; был заем на шаге D4 ?
        jne     @m6
        :----- D6 — компенсирующее сложение
        mov     borrow, 0 ; сбросим факт заема
        dec     q[si]
        mov     bx, si
        push    si
        add_bcd_r u0[bx], <n+1>, v0, <n+1>, u0 ; перенос не нужен
@m6:    :----- D7
        pop     si
        inc     si
        cmp     si, mm
        jle     @m7
        :----- D8 — денормализация
        mov     bx, si
        div_bcd_l_r u0[bx], N, d, r, temp_r
        ret
div_bcd_NM_r endp
main:
        :...
        call    div_bcd_NM_r
        :...

end main

```

Упакованные BCD-числа

В отличие от неупакованных BCD-чисел, разработчики команд процессора Intel весьма сдержанно отнеслись к проблеме обработки упакованных BCD-чисел. Существуют только две команды — DAA и DAS, которые поддерживают процесс сложения и вычитания упакованных BCD-чисел. Умножение и деление этих чисел не поддерживается вовсе. По этой причине при необходимости выполнения арифметических вычислений с упакованными BCD-числами есть смысл предварительно преобразовывать их в неупакованное представление, выполнять необходимые действия, результат которых конвертировать (если нужно) обратно в упакованный вид. Так как действия по преобразованию не являются основными в процессе вычислений, то желательно, чтобы они были максимально быстрыми. Можно предложить много вариантов подобного преобразования. Рассмотрим один из них.

Преобразование упакованного BCD-числа размером N байтов в неупакованное BCD-число (макрокоманда)

```

+-----+
: | Макрокоманда: BCD_PACK_TO_UNPACK. Преобразование упакованного BCD-числа |
: | размером N байтов в неупакованное BCD-число размером N*2 байтов.       |
+-----+

```

```

;+-----+
;| Порядок следования байтов – младший байт по младшему адресу (Intel). |
;+-----+

```

```
BCD_PACK_TO_UNPACK macro PACK, N, UNPACK
```

```

    Local    cyc1
    push     ds
    pop      es
    mov      ecx, N
    cld
                                ; порядок обработки BCD-цифр –
                                ; начиная с младшей

    lea      edi, UNPACK
    lea      esi, PACK
    xor      ax, ax
    lodsb
                                ; загрузить очередные 2 упакованные
                                ; BCD-цифры из PACK в al

    ror      ax, 4
    ror      ah, 4
    stosw
    loop     cyc1

```

```
endm
```

Преобразование неупакованного BCD-числа размером N байтов в упакованное BCD-число (макрокоманда)

```

;+-----+
;| Макрокоманда: BCD_UNPACK_TO_PACK. Преобразование неупакованного BCD-числа |
;| размером N байтов в упакованное BCD-число. |
;+-----+

```

```

;| Порядок следования байтов – младший байт по младшему адресу (Intel). |
;+-----+

```

```
BCD_UNPACK_TO_PACK macro UNPACK, N, PACK
```

```

    Local    cyc1
    push     ds
    pop      es
    mov      ecx, N
    ;----- определяем N/2 (размерность PACK) – если нечетное.
    ; округляем в большую сторону
    shr      ecx, 1
    bt       ecx, 0
    jc       $ + 4
    setc     bl
    inc      ecx
                                ; добавляем 1 для округления вверх
                                ; предыдущие три команды можно
                                ; заменить одной: adc ecx, 0
                                ; теперь в ecx правильное значение
                                ; счетчика цикла в соответствии
                                ; с размерностью UNPACK
                                ; порядок обработки BCD-цифр –
                                ; начиная с младшей

    cld

    lea      edi, PACK
    lea      esi, UNPACK
    xor      ax, ax
    lodsw
                                ; загрузить очередные 2 неупакованные
                                ; BCD-цифры из UNPACK в ax

    rol      ah, 4
    rol      ax, 4
    stosb
    loop     cyc1
    test     bl, bl
    jnz      $+7
    and      byte ptr [edi-1], 0f0h

```

```
endm
```

Генерация последовательности случайных чисел

Знание некоторых принципов нередко возмещает незнание некоторых фактов.

К. Гельвецкий

Важный класс вычислительных алгоритмов, востребованных на практике, — алгоритмы генерации последовательности случайных величин. По определению, последовательностью случайных чисел является последовательно формируемый массив значений, в котором каждый очередной элемент получен вне всякой связи с другими его элементами и появление этого элемента возможно с вероятностью, подчиненной некоторому закону распределения. Если появление любого числа в этой последовательности равновероятно, то говорят о равномерном законе распределения случайных чисел.

На практике в большинстве случаев применяются программные методы генерации случайных чисел. На самом деле случайные последовательности, получаемые по некоторому алгоритму, являются псевдослучайными. Это происходит из-за того, что связь между значениями в последовательности, образуемой программным путем, обычно все-таки существует. Данное обстоятельство приводит к тому, что на каком-то этапе генерируемая последовательность чисел начинает повторяться — «входит в период». Рассмотрим несколько наиболее известных и пригодных для практического использования программных методов генерации псевдослучайных величин (все же, понимая смысл выражения, будем по привычке и для удобства называть их случайными).

Конгруэнтный метод генерации последовательности случайных чисел

Конгруэнтный метод генерации последовательности случайных чисел получил широкое распространение. Он описан во многих источниках, но конкретных рекомендаций по его использованию на платформе Intel автор не встретил. Попытаемся устранить этот недостаток.

В основе этого метода генерации последовательности случайных чисел лежит понятие конгруэнтности. По определению, два числа A и B конгруэнтны (сравнимы) по модулю M в случае, если существует число K , при котором $A - B = K \cdot M$, то есть если разность $A - B$ делится на M , и числа A и B дают одинаковые остатки от деления на M . Например, числа 85 и 5 конгруэнтны по модулю 10, так как при делении на 10 дают остаток 5 (при $K = 1$). В соответствии с этим методом каждое число в результирующей последовательности получается исходя из следующего соотношения:

$$X_{n+1} = (a \cdot X_n + c) \bmod m, \text{ где } n \geq 0.$$

При задании начального значения X_0 , констант a и c , данное соотношение однозначно определяет последовательность целых чисел X_n , составленную из остатков от деления на m предыдущих членов последовательности, в соответствии с соотношением выше. Величина этих чисел не будет превышать значение m . Если каж-

дое число этой последовательности разделить на m , то получится последовательность случайных чисел из интервала $0...1$. Но не спешите подставлять в показанное соотношение какие-либо значения. Основная трудность при использовании этого метода — подбор компонентов формулы. В зависимости от значения c различают два вида конгруэнтного метода — мультипликативный ($c = 0$) и смешанный ($c \neq 0$).

Для простоты изложения будем генерировать небольшие по величине случайные числа. Это дает возможность легко отслеживать особенности работы рассматриваемых алгоритмов с использованием стандартной возможности перенаправления ввода-вывода. Для этого необходимо, чтобы текст программы содержал строки:

```
mov     dl, ah
mov     ah, 02
int     21h
```

Запуск программы производится командной строкой вида
`prog.exe > p.txt`

При этом создается файл `p.txt`, в который и выводятся результаты работы программы. Если отказаться от этой возможности, то вывод будет производиться на экран, что не очень удобно для последующего анализа получающейся последовательности случайных чисел. Более подробно о возможностях работы с файлами и экраном читайте материал в главах 5 и 7, посвященных работе с файлами и консолю из программ на языке ассемблера.

Большинство представленных ниже программ функционируют в бесконечном цикле, с тем чтобы можно было изучать периодичность последовательности, создаваемой по тому или иному методу генерации случайных чисел. Поэтому для окончания работы программы ее необходимо завершить принудительно (при работе в Windows для этого можно просто закрыть окно DOS). В результате работы программы будет создан файл `p.txt`, который можно открыть в любом редакторе, допускающем просмотр файлов в шестнадцатеричном виде (например во встроенном редакторе файлового менеджера Windows Commander).

Для увеличения диапазона необходимо внести соответствующие, не принципиальные с точки зрения алгоритма, изменения в тексты программ.

Мультипликативный конгруэнтный метод генерации последовательности случайных чисел

Мультипликативный конгруэнтный метод задает последовательность неотрицательных целых чисел X_i ($X_i < m$), получаемых по формуле:

$$X_{n+1} = (a \cdot X_n) \bmod m.$$

На значения накладываются ограничения:

- X_0 — нечетно;
- $a = 5^{2p+1}$ ($p = 0, 1, 2, \dots$) или $a = 2^m + 3$ ($m = 3, 4, 5, \dots$) — обе эти записи означают, что младшая цифра a при представлении a в восьмеричной системе счисления должна быть равна 3 или 5 (проще говоря, остаток от деления $a/8$ должен быть равен 3 или 5);
- $m = 2^l$ ($l > 4$).

При соблюдении этих ограничений длина периода будет равна $m/4$.


```

+-----+
| Программа: rand_mult_cong_1.asm. Генератор линейной (мультипликативной) |
| конгруэнтной последовательности случайных чисел (с=0). |
+-----+
| Вход:  $X_0$ , a, m. |
+-----+
| Выход: d1 – значение очередного случайного числа. |
+-----+
.data
m          db      128
a          db      11
x          db      3          ; начальное значение
.code
cyc1:      :----- первое число в последовательности x=3
            mov     al, x          ; вычисляем очередное случайное число
            :----- X=(a*X) mod m
            mul     a          ; a*x в ah:al
            div     m          ; в ah случайное число
            mov     x, ah
            :----- вывод в файл – командная строка rand_mult_cong.exe > p.txt
            mov     d1, ah
            mov     ah, 02
            int     21h
            jmp     cyc1
end_cyc1:   :...

```

Если m является степенью 2, как в данном случае, то вместо команды DIV можно использовать две команды сдвига.

```

+-----+
| Программа: rand_mult_cong_1.asm. Генератор линейной (мультипликативной) |
| конгруэнтной последовательности случайных чисел (с=0). |
+-----+
| Вход:  $X_0$ , a, m – в соответствии с указанными ограничениями. |
+-----+
| Выход: d1 – значение очередного случайного числа. |
+-----+
cyc1:      :----- вычисляем очередное случайное число X=(a*X) mod m
            mov     al, x
            mul     a          ; a*x в ah:al
            shr     ax, cl
            xor     al, al
            rol     ax, cl          ; в al случайное число
            :----- вывод в файл – командная строка rand_mult_cong.exe > p.txt
            :...

```

Полный пример этой программы (rand_mult_cong_2.asm) вы найдете среди файлов, прилагаемых к книге.

Используя эти программы, можно получить последовательность случайных чисел, содержащую 32 значения — это ее период. Чтобы увеличить период, необходимо каким-либо способом сгенерировать значения a или x , удовлетворяющие приведенным выше ограничениям. Так, значение a можно вычислить, используя фрагмент:

```

.data
divider    db      8
.code
:----- вычисляем a исходя из соотношения: a mod 8 = 5
:
: одним из способов получить значение a (m ≥ a)
: удовлетворяем условию a mod 8 = 5
m2:        mov     al, a
            xor     ah, ah
            div     divider

```

```

cmp     ah, 5           : остаток равен 5?
je      m1
cmp     ah, 3           : остаток равен 3?
je      m1
inc     a
jmp     m2
m1:
; ...
; теперь а найдено до конца

```

Изменить (увеличить) период можно, корректируя значение m , для чего требуется исправить соответствующие команды в программах `rand_mult_cong_1.asm` и `rand_mult_cong_2.asm`, ориентированные на определенную разрядность регистров. Существует другая возможность увеличения периода — использование *смешанного конгруэнтного метода* генерации последовательности случайных чисел.

Смешанный конгруэнтный метод генерации последовательности случайных чисел

Соотношение смешанного конгруэнтного метода выглядит так:

$$X_{n+1} = (a \cdot X_n + c) \bmod m,$$

где $n \geq 0$.

При правильном подборе начальных значений элементов, кроме увеличения периода последовательности случайных чисел, уменьшается корреляция (зависимость) получаемых случайных чисел.

На значения накладываются ограничения:

- $X_0 \geq 0$;
- $a = 2^l + 1$, где $l \geq 2$;
- $c > 0$ взаимно просто с m (это выполнимо, если c — нечетно, а $m = 2^p$, где $p \geq 2$);
- $m = 2^p$ ($p \geq 2$) и m кратно 4.

```

+-----+
:| Программа: rand_mix_cong_1.asm. Генератор линейной (смешанной) |
:| конгруэнтной последовательности случайных чисел (c>0).         |
+-----+
:| Вход: X0, a, c, m – в соответствии с указанными ограничениями. |
+-----+
:| Выход: dl – значение очередного случайного числа.               |
+-----+
.data
m      db 128           ; 128=27
a      db 9
x      db 3             ; начальное значение
c      dw 3
.code
mov     cl, 7           ; значение степени m = 27 в cl
;----- вычисляем очередное случайное число X=(a*X) mod m.
; первое число в последовательности x=3
cyc1:  mov     al, x
mul     a               ; a*x в ah:al
add     ax, c
shrd    ax, ax, cl
xor     al, al
rol     ax, cl          ; в al случайное число
;----- вывод в файл – командная строка rand_mult_cong.exe > p.txt
; ...
end_cyc1:
; ...

```

Величина периода случайной последовательности, получаемой с помощью данной программы, составляет 128. Сегмент кода программы `rand_mix_cong_1.asm` можно оформить в виде процедуры. Начальное значение X_0 можно выбирать двумя способами: задавать константой в программе или генерировать случайным образом. В последнем случае достаточно опереться на такты системного таймера, как в следующей макрокоманде:

```

+-----+
+| Макрокоманда: rCMOS. Чтение значений CMOS. |
+-----+
+| Вход: al — адрес ячейки, значение которой читаем. |
+-----+
+| Выход: al — прочтенное значение. |
+-----+
rCMOS      macro
            out      70h, al
            xor      al, al
            in       al, 71h          ; вводим в регистр AL из порта
                                     ; значение ячейки CMOS
endm

```

Применение — получить значение секунд из CMOS для `x_start`:

```

.code
            mov      al, 0h
            rCMOS
            mov      x, al           ; x = x_start
            ...

```

Таким способом можно получить начальное значение из диапазона 0–59. Для получения большего по величине начального значения подходит величина размером 32 бита из области данных BIOS по адресу 0040:006с. Здесь содержится счетчик прерываний от таймера. Извлечь это значение можно при помощи следующего программного фрагмента:

```

push      ds
push      word ptr 40h
pop       ds
mov       eax, dword ptr ds:006ch
pop       ds

```

Заменяя команду `MOV` командой `MOV AX, word ptr ds:006ch` или `MOV AL, byte ptr ds:006ch`, можно использовать младшие 8 или 16 битов значения из этой области BIOS. Команда `MOV AL, byte ptr ds:006ch` позволяет случайным образом загрузить в регистр AL значение из диапазона 0–0ffh.

При работе под Windows в качестве начального можно использовать значения из счетчика меток реального времени TSC [39].

Попытки создать программный датчик случайных чисел без опоры на какие-либо теоретические выкладки обречены на провал. Рассмотрим еще несколько способов генерации случайных чисел.

Аддитивный генератор случайных чисел

Генератор, формирующий очередное случайное число в соответствии с отношением

$$X_{n+1} = (X_n + X_{n-k}) \bmod m,$$

называется аддитивным.

В трехтомнике Кнута [5] обсуждаются подобные генераторы и рекомендован следующий вариант этой формулы:

$$X_{n+1} = (X_{n-24} + X_{n-55}) \bmod m.$$

Здесь $n \geq 55$, $m = 2^l$, $X_0, \dots, X_{54}$ — произвольные числа, среди которых есть и нечетные. При этих условиях длина периода последовательности равна $2^l - 1$ ($2^{55} - 1$).

Для генерации первых 55 случайных чисел можно использовать один из рассмотренных выше генераторов. Возьмем датчик линейной (смешанной) конгруэнтной последовательности случайных чисел ($c > 0$).

```

+-----+
| Программа: rand_add.asm. Аддитивный генератор случайных чисел. |
+-----+
| Вход: X0, a, c, m. |
| случайная последовательность длиной 55 значений, получаемая с помощью |
| программы генерации высокослучайных двоичных наборов (rand_mix_cong_1.asm). |
| N = 700 — количество элементов в последовательности + 1. |
| L = 7 — значение степени m = 2L |
+-----+
| Выход: dl — значение очередного случайного числа. |
+-----+

.data
    N = 700                ; число элементов в последовательности + 1
    L = 7                  ; значение степени m=27
m      db      128         ; 128 = 27
mm     dw      256         ; 256 = 28
a      db      9
c      dw      3
;
x      db      3, N dup (0ffh) ; массив значений x (начальное значение
                                ; равно 3) длиной N + 1

.code
;----- далее фрагменты из rand_mix_cong_1.asm
xor     si, si
mov     ecx, 54            ; счетчик цикла для формирования
                          ; начальной последовательности
;----- вычисляем очередное случайное число X=(a*X) mod m
cyc1:   mov     al, x
mul     a                  ; a*x в ah:al
add     ax, c
shrd    ax, ax, L          ; L — значение степени m=27
xor     al, al
rol     ax, L              ; в al случайное число
;----- вывод в массив x и файл — командная строка
; rand_mult_cong.exe > p.txt
inc     si
mov     x[si], al
mov     dl, al
mov     ah, 02
int     21h
loop    cyc1
;----- далее продолжаем формирование случайной последовательности.
; но уже аддитивным методом
cyc12:  mov     ecx, N - 55
inc     si
xor     dx, dx
mov     al, x[si-24]
mov     dl, x[si-55]
add     ax, dx
xor     dx, dx
div     mm
mov     x[si], dl

```

```

mov     ah,02
int     21h
loop    cyc12
exit:
....

```

Судя по результатам, этот метод достаточно хорош. В частности, он неплох тем, что позволяет задавать длину последовательности без оглядки на значение m .

Следующий рассматриваемый нами датчик можно использовать в программах на языке ассемблера для получения случайной последовательности 0 и 1.

Программа генерации высокослучайных двоичных наборов

Для процесса генерации требуются два значения одинаковой размерности — Y и C . Значение Y будет первым в последовательности случайных чисел. Значение C играет роль маски, в соответствии с которой впоследствии будет производиться операция «исключающее ИЛИ». Генерация очередного случайного числа происходит в два этапа. На первом этапе значение Y сдвигается влево на один разряд. При этом нас интересует содержимое выдвинутого бита. Его значением устанавливается флаг CF . На втором этапе, если $CF = 1$, корректируем значение $Y = Y \text{ XOR } C$ и сохраняем Y ; в противном случае сохраняем сдвинутое значение в качестве очередного числа последовательности.

```

+-----+
+| Программа: rand_bin_1.asm. Генерация высокослучайных двоичных наборов |
+| (сокращенный вариант). |
+-----+
+| Вход: у, с – в соответствии с указанными ограничениями. |
+-----+
+| Выход: dl – значение очередного случайного числа. |
+-----+
.data
Y      db      35h          : 0bh
C      db      33h          : 03h
.code
cyc1:  shl      Y, 1
        jnc     m1
        mov     al, Y
        xor     al, C
        mov     Y, al
        ;----- вывод на экран (в файл – командная строка
        :      rand_bin.exe > p.txt)
m1:    ....
        jmp     cyc1
end_cyc1:
....

```

Содержимое файла, в который перенаправлен вывод программы, показывает, что период последовательности достаточно удовлетворительный (при удачном подборе Y и C). Для того чтобы получить случайную последовательность 0 и 1, необходимо на каждой итерации выделять младший или старший бит очередного значения. Доработаем программу `rand_bin_1.asm`, чтобы продемонстрировать этот прием.

```

+-----+
+| Программа: rand_bin_2.asm. Генерация высокослучайных двоичных наборов |
+| (полный вариант). |
+-----+

```

```

:| Вход: у, с — в соответствии с указанными выше ограничениями.
:-----+
:| Выход: dl — значение очередного случайного числа.
:-----+
.data
Y      db      35h          ; 0bh
C      db      33h          ; 03h
.code
;----- получаем очередное значение Y
push    ds
push    word ptr 40h
pop     ds
mov     eax, dword ptr ds:006ch
pop     ds
mov     Y, al
;----- формируем случайные 8-битовые наборы в регистре DL
xor     dl, dl
mov     ecx, 8
cycl:   shl     Y, 1
jnc     m1
mov     al, Y
xor     al, C
mov     Y, al
jmp     $+5          ; на shr
m1:     mov     al, Y
shr     al, 1
rcr     dl, 1
loop    cycl
;----- вывод на экран (в файл — командная строка
:      rand_bin.exe > p.txt) очередного значения
:      ....
exit:   ....

```

Этот датчик хорош, когда его используют для получения одиночных случайных значений, а не всей последовательности сразу. Поэтому в этой программе отсутствует цикл, характерный для предыдущих программных датчиков, и ее удобно оформить в виде процедуры или макрокоманды.

В заключение данной темы — высказывание Джорджа Марсальи, которое приводит Кнут [5]: «Генератор случайных чисел во многом подобен сексу: когда он хорош — это прекрасно, когда он плох, все равно приятно». Возможно, экспериментируя с примерами данного раздела, вы испытали подобное чувство.

Оценка качества последовательности производится по определенным критериям, подробное рассмотрение которых не является предметом данной книги, но все же требует упоминания. В большинстве случаев для быстрой оценки качества работы генератора случайной последовательности применимы следующие критерии:

- **Длина периода и длина аперииодичности.** Под длиной периода понимается длина максимальной, не содержащей самой себя, числовой цепочки в последовательности случайных чисел. Длина аперииодичности — длина подинтервала последовательности случайных чисел, в пределах которого не встречаются одинаковые значения.
- **Равномерность последовательности псевдослучайных чисел.** Этот критерий означает вероятность попадания значений, генерируемых датчиком случайных чисел, в каждый из m подинтервалов, на которые можно разбить весь интервал значений, генерируемых данным датчиком случайных чисел.

- *Стохастичность последовательности псевдослучайных чисел.* Этот критерий означает определение вероятности появления j единиц (нулей) в определенных n разрядах генерируемых датчиком случайных чисел.
- *Независимость элементов псевдослучайной последовательности.* Критерий определяет степень корреляции (зависимости) двух случайных величин в сгенерированной последовательности. При проведении оценки на основе данного критерия можно рассматривать любые, а не только рядом стоящие случайные величины последовательности.

Более подробную информацию о подходах к оценке случайных последовательностей можно получить в литературе [5].

Глава 2

Сложные структуры данных

Сутью искусства программирования обычно считается умение составлять операции. Но не менее важно умение составлять данные.

Н. Вирт

Основные понятия

Процесс разработки программы на ассемблере традиционно осложняется тем, что в этом языке ограничены средства описания данных, привычные для языков программирования высокого уровня. В главе 13 «Сложные структуры данных» учебника были рассмотрены средства, которые поддерживает ассемблер для работы с данными. Но это деление весьма условно и не дает представления о том, как реализуется общая концепция понятий «данные», «тип данных» и «структура данных» в контексте программирования на языке ассемблера. Это обстоятельство существенно влияет на эффективность изучения и использования языка ассемблера. Учитывая сложность и практическую важность данного вопроса, есть смысл изложить его более систематично.

Проблема представления и организации эффективной работы с данными возникла одновременно с идеей разработки первой вычислительной машины. Вычислительная машина функционирует согласно некоторому алгоритму. А если есть алгоритм, то должны быть и данные, с которыми он работает. Аналогично известной дилемме о курице и яйце возникает вопрос о первичности алгоритма и данных. Схожий вопрос неявно заложен в название неизвестной книги Никлауса Вирта [4] — «Алгоритмы + структуры данных = программы», два издания которой были выпущены издательством «Мир» в 1985 и 1989 годах.

Что же представляют собой понятия «данные», «тип данных», «структура данных»? Попытаемся привести некую классификационную характеристику этих понятий.

Данные — набор байтов, рассматриваемый безотносительно к заложенному в них смыслу. Понятие «обработка данных» характерно для процессора как исполнительного устройства. При этом «данные» рассматриваются как совокупность двоич-

ных разрядов, которыми манипулирует определенная машинная команда. Для человека подобную интерпретацию вряд ли можно считать удобной. Для него более естественной является логическая интерпретация данных, которая базируется на понятии «тип данных».

Тип данных можно определить множеством значений *данных* и набором операций над этими значениями. В учебнике с точки зрения типа данные были разделены на две группы — простые и сложные. Данными простого типа считаются элементарные, неструктурированные данные, которые могут быть описаны с помощью одной из директив резервирования и инициализации памяти. Примером таких данных являются целые и вещественные числа различной размерности. В языках высокого уровня в качестве простых данных используются еще и данные символьного, логического, указательного типов. Данные простого типа называют *упорядоченными* (или *скалярными*), так как теоретически можно перечислить все значения, которые они могут принять.

Отличительная особенность данных простого типа — их неструктурированность. В контексте конкретной задачи, исходя из смысла и логики обработки между некоторыми простыми данными, могут существовать определенные отношения и связи, что позволяет рассматривать их как определенным образом организованные совокупности. Обобщенное название таких совокупностей — *структуры данных*. В общем случае отдельная структура данных может содержать не только простые данные, но и другие структуры данных.

С известной долей условности можно сказать, что тип данных и структура данных — понятия, независимые от компьютерной платформы. Вполне можно абстрагироваться от представления в программе данных простого или сложного типа и проводить теоретические рассуждения относительно действий с целыми и вещественными числами, массивами, списками, деревьями и т. д. Поэтому такие структуры данных называют структурами *данных уровня представления, абстрактными* или *логическими структурами*. Для машинной обработки абстрактные типы и структуры данных необходимо некоторым способом представить в оперативной памяти, то есть в виде *физической структуры данных (структуры хранения)*, которая, в свою очередь, отражает способ физического представления абстрактных данных на конкретной программно-аппаратной платформе. В качестве примера можно привести двумерный массив. Для программиста, пишущего программу для обработки этого массива, он видится в виде матрицы, содержащей определенное количество строк и столбцов. В оперативной памяти данный массив представляется в виде линейной последовательности ячеек. В данном случае логическая структура данных — двумерный массив, а соответствующая ему физическая структура — вектор (см. ниже). Таким образом, в большинстве случаев существует различие между логическими и соответствующими им физическими структурами данных. Задачей программиста фактически является написание кода, осуществляющего отображение логической структуры на физическую и наоборот. Этот код реализует различные операции логического уровня над структурой данных. Так, на логическом уровне доступ к элементу двумерного массива осуществляется указанием номеров столбца и строки в матрице, в которой расположен данный элемент. На физическом уровне эта операция выглядит по-другому. Для доступа к определен-

ному элементу массива программный код по известному номеру строки и столбца вычисляет смещение от начального адреса массива в памяти. Исходя из вышеприведенных рассуждений, конкретную структуру данных можно характеризовать ее логическим (абстрактным) и физическим (конкретным) представлениями, а также совокупностью операций на этих уровнях.

Язык ассемблера — язык уровня архитектуры конкретного компьютера. Память компьютеров с архитектурой Intel представляет собой упорядоченный набор непосредственно адресуемых машинных ячеек (байтов). Исходя из этого, номенклатура структур хранения данных архитектурно ограничена следующим набором: скаляр, вектор, список, сеть.

- **Скаляр** — поле, содержащее одиночное двоичное значение, размерностью один или несколько байтов. Количество байтов, составляющих скаляр, определяется допустимыми размерами операндов системы команд конкретного процессора.
- **Вектор** — конечное упорядоченное множество расположенных рядом скаляров одного типа, называемых элементами вектора. По сути дела вектор — это одномерный массив. Что у них общего? Геометрически вектор представляет собой состоящий из точек объект в пространстве, имеющий начальную точку, из которой он выходит, и конечную точку, в которую он приходит. Точки, лежащие в пространстве между начальной и конечной точками (элементами вектора), находятся между собой в единственно возможном отношении — отношении непосредственного следования. Такая строгая упорядоченность элементов вектора позволяет произвести их последовательную нумерацию. Аналогично и одномерный массив имеет началом и концом скаляры, расположенные по определенным адресам памяти. Между этими адресами последовательно расположены скаляры, составляющие элементы массива. Определенность с начальным и конечным адресами массива, а также с размерностью его элементов дает возможность однозначно идентифицировать любой его элемент.
- **Список** — набор элементов, каждый из которых состоит из двух полей. Одно поле содержит элемент данных или указатель на элемент данных, другое — указатель на следующий элемент списка, который, в свою очередь, тоже может быть начальным или промежуточным элементом другого списка. Наличие явного указания на упорядоченность элементов списка позволяет достаточно легко манипулировать содержимым списка, включая новые и исключая старые элементы списка без их фактического перемещения в памяти. Это свойство позволяет размещать в памяти динамически изменяющиеся структуры данных.
- **Сеть** — набор элементов, каждый из которых помимо информационного поля содержит несколько полей-указателей на другие элементы сети. С помощью сети удобно представлять такие структуры данных уровня представления, как деревья, ориентированные графы и т. п.

Как мы увидим ниже, эти четыре структуры хранения дают возможность представить в памяти машины практически все известные структуры уровня представления: массивы, записи, таблицы, стеки, очереди, деки, списки различной степени связности, деревья, ориентированные и неориентированные графы. Эти структуры можно классифицировать по следующим признакам.

- *Связность*, то есть отсутствие или наличие явно заданных связей между элементами структуры. К *несвязным* структурам уровня представления относятся массивы, символьные строки, стеки, очереди. К связным структурам уровня представления относятся связные списки.
- *Изменчивость*, то есть изменение числа элементов и (или) связей между элементами структуры. По этому признаку структуры делятся на статические (массивы, таблицы), *полустатические* (стеки, очереди, деки) и динамические (одно-, двух- и многосвязные списки).
- *Упорядоченность* — *линейные* (массивы, символьные строки, стеки, очереди, одно- и двухсвязные списки) и *нелинейные* (многосвязные списки, древовидные и графовые структуры).

Отображение структур представления в структуры хранения производится в соответствии с требованиями конкретной задачи и в зависимости от требований последней может производиться разными способами. В ходе дальнейшего изложения мы выясним возможности, которыми обладает язык ассемблера для представления и программной обработки наиболее полезных и часто используемых на практике структур представления (абстрактных типов данных).

Способы распределения памяти

Прежде чем приступить к рассмотрению того, каким образом поддерживается работа с различными структурами данных на ассемблере, необходимо определиться с тем, как будет решаться проблема распределения памяти. Это важный момент, так как фон-неймановская архитектура современных машин предполагает, что любые данные, обрабатываемые программой, располагаются в памяти. Традиционно в любом языке программирования высокого уровня существуют два способа распределения памяти для переменных и постоянных данных — *статический* и *динамический*. Соответственно данные будем называть *статическими* и *динамическими* объектами данных. В ассемблере в силу его специфики явно присутствует возможность только статического распределения памяти с помощью директив резервирования и инициализации памяти в сегменте данных. Статические объекты данных занимают отведенную им память на все время работы программы. До сих пор при написании ассемблерных программ мы использовали именно статическое распределение памяти. Недостаток этого очевиден — если объект данных занимает большую область памяти и имеет малый период жизни, то выделять память на все время работы программы неразумно. Гораздо эффективнее в таких случаях иметь возможность для временного выделения памяти объекту данных на период его жизни. В этом разделе мы уделим все внимание проблеме динамического распределения памяти в программах на ассемблере.

Динамическими объектами данных называются объекты данных, обладающие двумя особенностями:

- распределение/освобождение памяти для объекта данных производится по явному запросу программы;
- доступ к динамическим объектам данных производится не по их именам, а посредством указателей, содержащих ссылку (адрес) на созданный динамический объект.

Существует и третья особенность, о которой, возможно, не подозревают программирующие на языках высокого уровня, — технология выделения памяти для динамических объектов данных. Эта технология напрямую зависит от операционной среды, для которой разрабатывается программа. Так, в MS DOS динамическое выделение памяти во время работы приложения осуществляется с помощью двух функций прерывания `int 21h: 48h` (выделение блока памяти) и `49h` (освобождение блока памяти). Единицей выделения памяти при этом является параграф (16-байтовый блок памяти). С точки зрения современного программирования — это примитивно и не очень интересно. Гораздо большими возможностями обладает наиболее распространенная в настоящее время операционная система Windows.

В Windows существуют несколько механизмов динамического выделения памяти:

- виртуальная память;
- кучи;
- отображаемые в память файлы.

Пример использования последнего из перечисленных механизмов рассматривается в главе 7 этой книги, посвященной работе с файлами. Для нашего изложения представляют интерес, хотя и в разной степени, первые два механизма.

Механизм виртуальной памяти Windows

Механизм виртуальной памяти Windows реализуется с помощью функций API Win32 `VirtualAlloc` и `VirtualFree`.

```
LPVOID VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType,
                   DWORD flProtect)
BOOL VirtualFree(LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType)
```

С помощью функции `VirtualAlloc` приложение запрашивает в свое распоряжение область памяти (*регион*) в адресном пространстве с размером, указываемым параметром `dwSize`. Величина `dwSize` должна быть кратна 64 Кбайт, что, соответственно, является минимальным размером региона. Динамическое выделение памяти такими большими порциями может требоваться лишь для работы с большими массивами данных (структурами данных). Для нашего изложения этот способ выделения данных не подходит. Отметим лишь, что работа функции `VirtualAlloc` имеет следующую особенность. Имеются три варианта обращения к функции `VirtualAlloc`: резервирование региона в адресном пространстве процесса; выделение физической памяти в зарезервированном предыдущим вызовом функции `VirtualAlloc` регионе; резервирование региона с одновременной передачей ему физической памяти. Освобождение региона производится функцией `VirtualFree`. Более подробную информацию о параметрах функций `VirtualAlloc` и `VirtualFree` можно посмотреть в документации MSDN.

Механизм работы с кучами Windows

Этот механизм наиболее эффективен для поддержки работы с такими структурами данных, как связные списки, деревья и т. п. Как правило, отдельные элементы этих структур имеют небольшой размер, в то время как общее количество памяти, занимаемое такими структурами в разные моменты времени работы приложения,

может быть разным. Главное преимущество использования кучи — свобода в определении размера выделяемой памяти. В то же время это самый медленный механизм динамического выделения памяти.

Windows поддерживает работу с двумя видами куч: *стандартной* и *дополнительной*.

Во время инициализации процесса система выделяет ему *стандартную кучу* (или *кучу по умолчанию*), размер которой составляет 1 Мбайт. При желании можно указать компоновщику ключ /HEAP с новой величиной размера стандартной кучи. Создание и уничтожение стандартной кучи производится системой, поэтому в API не существует функций, управляющих этим процессом. Только одна функция должна вызываться перед началом работы со стандартной кучей — `GetProcessHeap`:

```
HANDLE GetProcessHeap(VOID)
```

`GetProcessHeap` возвращает дескриптор, используемый далее другими функциями работы с кучей.

Для более эффективного управления памятью и локализации структур хранения в адресном пространстве процесса предусмотрено создание дополнительных куч. Сделать это можно с использованием функции `HeapCreate`:

```
HANDLE HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize)
```

Размер создаваемой этой функцией кучи определяется параметрами `dwInitialSize` (начальный размер) и `dwMaximumSize` (максимальный размер). Возвращаемое функцией `HeapCreate` значение — дескриптор кучи, который используется затем другими функциями, работающими с данной кучей. Уничтожение дополнительной кучи осуществляется вызовом функции `HeapDestroy`, которой в качестве параметра передается дескриптор уничтожаемой кучи:

```
BOOL HeapDestroy(HANDLE hHeap)
```

Важно отметить, что на этапе создания как стандартной, так и дополнительных куч реального выделения памяти для них не производится. Главное — получить указатель и сообщить системе характеристики и размер создаваемой кучи.

После получения дескриптора работа со стандартной и дополнительной кучами осуществляется с помощью функций `HeapAlloc`, `HeapReAlloc`, `HeapSize`, `HeapFree`. Рассмотрим их подробнее.

Выделение физической памяти из кучи производится по запросу `HeapAlloc`:

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes)
```

Здесь параметр `hHeap` сообщает `HeapAlloc`, в пространстве какой кучи требуется выделение памяти размером `dwBytes` байтов. Параметр `dwFlags` представляет собой признаки, с помощью которых можно влиять на особенности выделяемой памяти. В случае успеха функция `HeapAlloc` возвращает адрес, который используется далее для доступа к физической памяти в выделенном блоке.

В ходе работы с выделенным блоком может сложиться ситуация, когда необходимо изменить его размер в большую или меньшую сторону. Для этого предназначена функция `HeapReAlloc`:

```
LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, SIZE_T dwBytes)
```

Параметр `hHeap` идентифицирует кучу, в которой изменяется размер блока, а параметр `lpMem` является адресом блока (полученным ранее с помощью `HeapAlloc`), размер которого изменяется. Новый размер блока указывается параметром `dwBytes`.

Играя размерами блоков, вы можете совсем запутаться. Функция `HeapSize` может вам определить текущий размер блока по адресу `lpMem` в куче `hHeap`.

`DWORD HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem)`

И наконец, когда блок по адресу `lpMem` в куче `hHeap` становится ненужным, его можно освободить вызовом функции `HeapFree`:

`BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem)`

Это минимальный набор функций Windows для работы с кучами. Он столь подробно был обсужден нами с целью дальнейшего использования этих функций для разработки приложений этого раздела.

Множество

Соня закрыла глаза и задремала. Но тут Бовванщик ее ущипнул, она взвизгнула и проснулась.

— ...начинается на М, — продолжала она. — Они рисовали мышеловки, математику, множество... Ты когда-нибудь видела, как рисуют множество?

— Множество чего? — спросила Алиса.

— Ничего, — отвечала Соня. — Просто множество!

— Не знаю, — начала Алиса, — может...

— А не знаешь — молчи, — оборвал ее Бовванщик.

Льюис Кэрролл. «Алиса в стране чудес»

Множество как структура уровня представления является совокупностью различных объектов, которые могут либо сами являться множествами, либо являться неделимыми элементами, называемыми атомами.

Множество как структура уровня хранения реализуется двумя способами:

- простым — в виде данных перечислимого типа; в языках высокого уровня этот тип данных реализуют с помощью типа «множество» (в Pascal) или констант перечислимого типа (в C);
- сложным — в виде вектора или связанного списка.

Отличие этих двух способов в том, что данные перечислимого типа ограниченно поддерживают понятие множества, представляя собой совокупность объектов, которым сопоставлены некоторые значения. При реализации множеств с помощью вектора или связанного списка становится возможным реализовать именно математическое понятие множества, согласно которому над множествами определен ряд операций: объединение, пересечение, разность, слияние, вставка (удаление) элемента и т. д. К данным перечислимого типа эти операции неприменимы.

В языке ассемблера также есть средства, позволяющие реализовать оба способа представления множеств. Так, описание данных перечислимого типа поддерживаются с помощью директивы `enum`. Как и в языках высокого уровня, данные перечислимого типа, вводимые этой директивой, являются константами, которым соответствуют уникальные символические имена. Директива `enum` имеет следующий синтаксис:

`символ_имя enum значение[, значение[, значение...]]`

Здесь значение представляет собой конструкцию `символ_имя [=число]`, а `символ_имя` — любое символическое имя, не совпадающее с ключевыми словами ас-

симвлера или другими ранее определенными в программе символическими именами. Следующие примеры описания множеств эквивалентны.

```
week enum Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
week enum Monday=0, Tuesday=1, Wednesday=2, Thursday=3, Friday=4, Saturday=5, Sunday=6
week enum Monday=0, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
week enum Sunday=6, Monday=0, Tuesday, Wednesday, Thursday, Friday, Saturday
```

Перечисление элементов множества может занимать несколько строк в программе.

Транслятор будет отводить под размещение каждого элемента множества столько байтов, сколько требуется для размещения значения самого большого элемента в этом множестве (байт, слово, двойное слово).

Если при описании очередного элемента множества число в некоторой конструкции **символ_имя** [=число] не задано, то транслятор присвоит этому элементу множества значение, на единицу большее предыдущего. Если ни одного значения не было задано, то первому элементу множества присваивается 0, второму 1 и т. д.

Работа с элементами множества в программе является завуалированной формой использования непосредственного операнда. В этом несложно убедиться, проанализировав листинг трансляции:

```
5          ;задаем множество:
6  =0000    =0001 =0002 + week      enum    Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday
7  =0003    =0004 =0005 +
8  =0006
...
20 0005 88 0006      mov ax, Sunday
```

Значение символического имени **символ_имя**, стоящего слева от директивы **enum**, равно количеству битов, необходимому для размещения максимального значения элемента справа от директивы **enum**:

```
5          ;задаем множество:
6  =0000    =0001 =0002 + week      enum    Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday
7  =0003    =0004 =0005 +
8  =0006
...
21 0005 B8 0007      mov ax, week
```

Перед использованием в программе необходимо определить и инициализировать экземпляр множества:

```
F_week    week Saturday
```

Размер этой переменной будет соответствовать размеру максимального элемента множества **week**. Сказанное иллюстрирует фрагмент листинга:

```
5          ;задаем множество:
6  =0000    =0001 =0002 + week enum Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday
7  =0003    =0004 =0005 +
8  =0006
9  0000 05          s    week Saturday
...
21 0005 A2 0000r    mov s, al
```

Хорошо видно, что в данном случае работа с экземпляром множества осуществляется как с байтовой переменной.

Возможности ассемблера для описания множества и работы с ним сложным способом (с помощью векторов и связанных списков) будут разъяснены в ходе дальнейшего изложения.

Массив

Все истинно великое совершается медленным, незаметным ростом.

Сенека

Описание массивов

Как структура представления, массив является упорядоченным множеством элементов определенного типа. Упорядоченность массива определяется набором целых чисел, называемых индексами, которые связываются с каждым элементом массива и однозначно конкретизируют его расположение среди других элементов массива. Локализация конкретного элемента массива — ключевая задача при разработке любых алгоритмов, работающих с массивами. Ее сложность прямо зависит от размерности массива.

Наиболее просто представляются одномерные массивы. Соответствующая им структура хранения — это вектор. Она однозначна и есть не что иное, как просто последовательное расположение элементов в памяти. Чтобы локализовать нужный элемент одномерного массива, достаточно знать его индекс. Так как ассемблер не имеет средств для работы с массивом как структурой данных, то для доступа к элементу массива необходимо вычислить его адрес. Для вычисления адреса i -го элемента одномерного массива можно использовать формулу:

$$A_i = AB + i \cdot len.$$

Здесь AB — адрес первого элемента массива размерностью n , i — индекс ($i = 0 \dots n - 1$), len — размер элемента массива в байтах. Процессор поддерживает режимы адресации, позволяющие легко работать с массивами, размерность элементов которых равна 2, 4 и 8 байтов [39]. Заметьте, что при таком определении можно не говорить о типе элементов массива. В общем случае они также могут быть структурированными объектами данных.

Представление двумерных массивов немного сложнее. Здесь мы имеем случай, когда структуры хранения и представления различны. О структуре представления говорить излишне — это матрица. Структура хранения остается прежней — вектор. Но теперь его нельзя без специальных оговорок интерпретировать однозначно. Все зависит от того, как решил разработчик программы «вытянуть» массив — по строкам или по столбцам. Наиболее естествен порядок расположения элементов массива — по строкам. При этом наиболее быстро изменяется последний элемент индекса. К примеру, рассмотрим представление на логическом уровне двумерного массива A_{ij} размерностью $n \times m$, где $0 \leq i \leq n - 1$, $0 \leq j \leq m - 1$:

$$\begin{array}{cccc} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{array}$$

Соответствующее этому массиву физическое представление в памяти — вектор — будет выглядеть так:

$$a_{00} a_{01} a_{02} a_{03} a_{10} a_{11} a_{12} a_{13} a_{20} a_{21} a_{22} a_{23} a_{30} a_{31} a_{32} a_{33}.$$

Номер конкретного элемента массива в этой, уже как бы ставшей линейной, последовательности определяется адресной функцией, которая устанавливает положение (адрес) в памяти этого элемента исходя из значения его индексов:

$$a_{ij} = n \cdot i + j.$$

Для вычисления адреса элемента массива в памяти необходимо полученное значение умножить на размер элемента и сложить с базовым адресом массива.

Аналогично осуществляется локализация элементов в массивах большей размерности. На рис. 2.1 показан трехмерный массив A_{ijk} размерностью $n \times m \times k$, где $n = 4, m = 4, k = 2$.

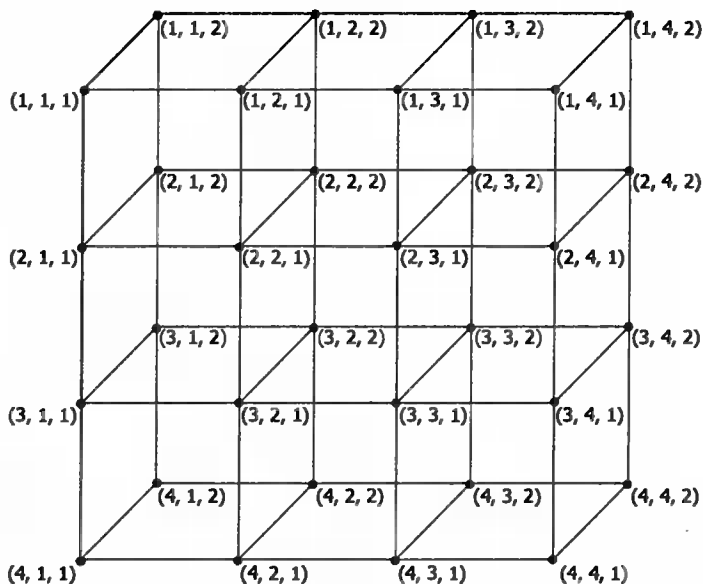


Рис. 2.1. Пример логической структуры трехмерного массива

Соответствующий этому массиву вектор памяти будет выглядеть так:

$$a_{000} a_{001} a_{010} a_{011} a_{020} a_{021} a_{030} a_{031} a_{100} a_{101} a_{110} a_{111} a_{120} a_{121} a_{130} a_{131} a_{200} a_{201} a_{210} a_{211} a_{220} a_{221} a_{230} a_{231} a_{300} a_{301} a_{310} a_{311} a_{320} a_{321} a_{330} a_{331}.$$

Соответственно номер элемента определяется так:

$$a_{ijk} = n \cdot m \cdot i + m \cdot j + z,$$

где $0 \leq i \leq n - 1, 0 \leq j \leq m - 1, 0 \leq z \leq k - 1$.

Для вычисления адреса осталось умножить полученное значение на размер элемента массива и сложить результат с базовым адресом массива.

Таким образом, преобразование многомерной, в общем случае логической структуры массива в одномерную физическую структуру производится путем ее линеаризации по строкам или столбцам. В первом случае быстрее всего изменяется по-

следний индекс каждого элемента, во втором — первый индекс. Недостаток описанного способа локализации элемента массива в том, что процесс вычисления адреса подразумевает выполнение операций сложения и умножения. Как известно, они не являются быстрыми. Существует метод Дж. Айлиффа, с помощью которого можно исключить операцию умножения из процесса вычисления индекса. Индексация в массиве производится с помощью иерархии дескрипторов, называемых векторами Айлиффа, так, как это показано на рис. 2.2. На этом рисунке j_1, j_2, j_3 обозначают соответственно первый, второй и третий индексы массива.

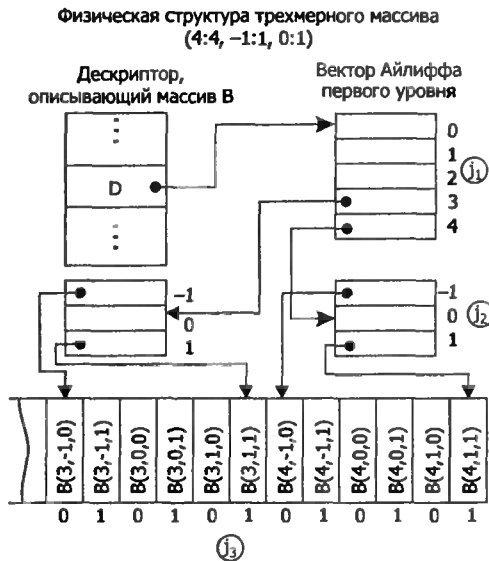


Рис. 2.2. Представление предыдущего трехмерного массива по методу Айлиффа

Из рисунка видно, что основной дескриптор массива содержит указатель на вектор Айлиффа первого уровня. Вектор первого уровня является единственным и содержит указатели векторов Айлиффа следующего уровня. То есть иерархия дескрипторов однозначно отражает размерность массива. Локализовать нужный элемент массива можно, руководствуясь его индексом логического уровня. Для этого нужно пройти по цепочке от основного дескриптора через соответствующие элементы векторов Айлиффа, извлекая из них и суммируя значения смещений с базовым адресом массива:

$$\text{Адрес } (B(j_1, j_2, \dots, j_n)) = (\dots(D) + j_1) + j_2 + \dots + j_n.$$

Здесь (D) — значение указателя вектора Айлиффа первого уровня, хранящееся в основном дескрипторе. При необходимости никто не мешает вам разместить в элементах векторов Айлиффа и другую служебную информацию, например диапазон изменения соответствующего индекса. Очевидный недостаток метода Айлиффа — увеличение общего объема памяти для структуры хранения многомерного массива в памяти.

Что касается описания массива в программе на ассемблере, то его варианты были подробно рассмотрены в учебнике. Перечислим их очень конспективно:

- перечисление элементов массива в поле операндов одной из директив описания данных;
- резервирование памяти для элементов массива с использованием оператора повторения `dup` (элементы массива формируются динамически в ходе выполнения программы);
- использование директив `label` и `rept`.

После рассмотрения возможностей по динамическому выделению памяти в операционных средах MS DOS и Windows можно к перечисленным трем вариантам добавить еще один — динамическое распределение памяти. Применительно к программированию для Windows возможности здесь очень широкие и вы вправе использовать наиболее подходящий способ из трех, обсуждавшихся в разделе «Способы распределения памяти».

Массив в разное время работы с ним может изменять свои характеристики — имя массива, адрес в памяти первого его элемента, количество и диапазон изменения индексов, тип элементов, размерность и т. п. В этом случае имеет смысл его физическое представление в памяти предварять специальным заголовком, или дескриптором, в котором указывать характеристики массива, чтобы программа могла правильно обрабатывать элементы массива. Особенно это важно, когда память для массива выделяется динамически.

Приемы практической работы с массивами продемонстрируем на примерах выполнения конкретных операций: сортировка массива, поиск в массиве, транспонирование матрицы и др. На примере работы с массивами очень удобно обсуждать особенности реализации этих операций, так как в общем случае элементы массива могут быть не просто скалярами, но и более сложными по структуре данными. Поэтому дальнейшее наше изложение, помимо демонстрации приемов работы с массивами в программах на ассемблере, будет посвящено рассмотрению возможных подходов к реализации нескольких классов важных на практике алгоритмов.

Сортировка массивов

Под *сортировкой* понимается процесс перестановки элементов некоторого множества в порядке убывания или возрастания. Если элементы множества — скаляры, то сортировка ведется относительно их значений. Если элементы множества — структуры, то каждая структура должна иметь поле (поля), относительно которого будет производиться упорядочивание местоположения элементов (то есть сортировка) относительно других элементов-структур множества.

Различают два типа алгоритмов сортировки: сортировку массивов и сортировку файлов. Другое их название — алгоритмы внутренней и внешней сортировки. Разница заключается в местонахождении упорядочиваемых объектов: для внутренней сортировки — это оперативная память компьютера, для внешней — файл. В данном разделе будут рассмотрены алгоритмы внутренней сортировки массивов. Способы внешней сортировки отложим до раздела, посвященного работе с файлами.

Существуют несколько алгоритмов сортировки массивов, которым следует уделить внимание в контексте изучения ассемблера. По критерию эффективности

алгоритмы сортировки массивов делят на простые и улучшенные. Среди простых методов, которые также называют сортировками «по месту», мы рассмотрим следующие:

- сортировка прямым включением;
- сортировка прямым выбором;
- сортировка прямым обменом.

Улучшенные методы в нашем изложении будут представлены следующими алгоритмами:

- сортировка Шелла;
- сортировка с помощью дерева;
- быстрая сортировка.

Сортировка прямым включением

Идея сортировки прямым включением (программа `prg4_96.asm`) заключается в том, что в упорядочиваемой последовательности `masi` длиной n ($i = 0 \dots n - 1$) последовательно выбираются элементы начиная со второго ($i \leq 1$) и ищутся их позиции в уже отсортированной левой части последовательности. При этом поиск места включения очередного элемента x в левой части последовательности `mas` может закончиться двумя ситуациями:

- найден элемент последовательности `mas`, для которого `masi < x`;
- достигнут левый конец отсортированной слева последовательности.

Первая ситуация разрешается тем, что последовательность `mas`, начиная с `masi`, раздвигается в правую сторону и на место `masi` перемещается x . Во втором случае следует сместить всю последовательность вправо и на место `mas0` переместить x .

Здесь для отслеживания условия окончания просмотра влево отсортированной последовательности используется прием «барьера». Суть его в том, что к исходной последовательности слева добавляется фиктивный элемент X . В начале каждого шага просмотра влево отсортированной части массива элемент X устанавливается в значение того элемента, для которого будет осуществляться поиск места вставки.

```

ПРОГРАММА prg4_96
// prg4_96 – программа на псевдоязыке сортировки прямым включением
// Вход: mas[n] – неупорядоченная последовательность байтовых двоичных значений.
// Выход: mas[n] – упорядоченная последовательность байтовых двоичных значений.
ПЕРЕМЕННЫЕ
INT BYTE n=8; // количество элементов в сортируемом массиве
INT BYTE mas[n]; // сортируемый массив размерностью n (0..n-1)
INT BYTE X; // барьер
INT BYTE i=0; j=0 // индексы
НАЧ ПРОГ
ДЛЯ i:=1 ДО n-1 // i изменяется в диапазоне 0..n-1
    НАЧ БЛОК_1
    // присвоение исходных значений для очередного шага
    X:=mas[i]; mas[0]:=X; j:=i-1
    ПОКА (X<mas[j]) ДЕЛАТЬ
        НАЧ БЛОК_2
        mas[j+1]:=mas[j]; j:=j-1
    КОН БЛОК_2

```

```

        mas[j+1]:=X
    КОН БЛОК_1
КОН ПРОГ

```

```

+-----+
:| Программа: prg4_96.asm. Сортировка прямым включением. |
+-----+
.data
mas      db      44, 55, 12, 42, 94, 18, 06, 67 : задаем массив
n = $ - mas
X        db      0                          : барьер
.code
        mov     cx, n - 1                    : цикл по i
        mov     si, 1                        : i=2
;----- присвоение исходных значений для очередного шага
cyc13:   mov     al, mas[si]
        mov     x, al                        : X:= mas[i]; mas[0]:=X; j:=i-1
        push    si                          : временно сохраним i, теперь j=i
;----- еще один цикл, который перебирает элементы до барьера x=mas[i]
cyc12:   mov     al, mas[si-1]
        cmp     x, al                        : сравнить x < mas[j-1]
        ja      ml                            : если x > mas[j-1]
;----- если x < mas[j-1], то
        mov     al, mas[si-1]
        mov     mas[si], al
        dec     si
        jmp     cyc12
ml:      mov     al, x
        mov     mas[si], al
        pop     si
        inc     si
        loop    cyc13
        ...

```

Этот способ сортировки не очень экономен, хотя логически он выглядит очень естественно.

Сортировка прямым выбором

В алгоритме сортировки прямым выбором (программа prg4_99.asm) на каждом шаге просмотру подвергается неупорядоченная правая часть массива. В ней ищется очередной минимальный элемент. Если он найден, то производится его перемещение на место крайнего левого элемента несортированной правой части массива.

```

ПРОГРАММА prg4_99
// prg4_99 – программа на псевдоязыке сортировки прямым выбором
// Вход: mas[n] – неупорядоченная последовательность байтовых двоичных значений.
// Выход: mas[n] – упорядоченная последовательность байтовых двоичных значений.
ПЕРЕМЕННЫЕ
INT_BYTE n=8; // количество элементов в сортируемом массиве
INT_BYTE mas[n]; // сортируемый массив размерностью n (0...n-1)
INT_BYTE X; i=0; j=0; k=0 // i, j, k – индексы
НАЧ ПРОГ
ДЛЯ i:=1 ДО n-1 // i изменяется в диапазоне 1..n-1
    НАЧ БЛОК_1
    // присвоение исходных значений для очередного шага
    k:=i; X:= mas[i]
    ДЛЯ j:=i+1 ДО n // j изменяется в диапазоне i+1..n
        ЕСЛИ mas[j]<X ТО
            НАЧ БЛОК_2
            k:=j; x:= mas[j]
        КОН БЛОК_2
    КОН БЛОК_1

```

```

        mas[k]:= mas[i]; mas[i]:=x
КОН_БЛОК_1
КОН_ПРОГ

;-----
;| Программа: prg4_99.asm. Сортировка прямым выбором.
;-----
.data
mas      db      4, 55, 12, 42, 94, 18, 06, 67 ; задан массив
n= $ - mas
X        db      0
K        dw      0
.code
;----- внешний цикл – по i
mov      cx, n - 1
xor      si, si
cyc11:   push     cx
        mov      k, si          ; k:=i
        mov      al, mas[si]
        mov      x, al          ; x:=mas[i]
        push     si             ; временно сохраним i – теперь j = i + 1
        inc      si             ; j = i + 1
;----- вложенный цикл – по j
mov      al, n
sub      ax, si
cyc12:   mov      cx, ax          ; число повторов внутреннего цикла по j
        mov      al, mas[si]
        cmp      al, x
        ja       ml             ; k:=j
        mov      k, si
        mov      al, mas[si]
        mov      x, al          ; x:=mas[k]
ml:      inc      si             ; j:=j+1
        loop     cyc12
        pop      si
        mov      al, mas[si]
        mov      di, k
        mov      mas[di], al     ; mas[k]:=mas[i]
        mov      al, x
        mov      mas[si], al     ; mas[i]:=x
        inc      si
        pop      cx
        loop     cyc11
;....

```

Первый вариант сортировки прямым обменом

Этот метод основан на сравнении и обмене пар соседних элементов до их полного упорядочивания (программа prg4_101.asm). В обиходе этот метод называется методом пузырьковой сортировки. Действительно, если упорядочиваемую последовательность расположить не слева направо, а сверху вниз («слева» — это «верх»), то визуально каждый шаг сортировки будет напоминать всплытие легких (меньших по значению) пузырьков вверх.

```

ПРОГРАММА prg4_101
// prg4_101 – программа на псевдоязыке сортировки прямым обменом 1
// Вход: mas[n] – неупорядоченная последовательность байтовых двоичных значений.
// Выход: mas[n] – упорядоченная последовательность байтовых двоичных значений.
ПЕРЕМЕННЫЕ
INT BYTE n=8; // количество элементов в сортируемом массиве
INT BYTE mas[n]; // сортируемый массив размерностью n (0..n-1)
INT BYTE X; i=0; j=0 // i, j – индексы

```

```

НАЧ ПРОГ
ДЛЯ i:=1 ДО n-1    // i изменяется в диапазоне 1..n-1
  НАЧ БЛОК_1
  ДЛЯ j:=n-1 ДОВНИЗ 1 // j изменяется в диапазоне i+1..n
    ЕСЛИ mas[j-1]< mas[j] ТО
      НАЧ БЛОК_2
      x:=mas[j-1]; mas[j-1]:=mas[j]; mas[j]:=x
      КОН БЛОК_2
    КОН БЛОК_1
  КОН БЛОК_1
КОН ПРОГ

```

```

;+-----+
;| Программа: prg4_101.asm. Сортировка прямым выбором 1. |
;+-----+
.data
mas      db      4, 55, 12, 42, 94, 18, 06, 67 ; задаем массив
n= $ - mas
x        db      0
.code
;----- внешний цикл — по i
mov      cx, n - 1
mov      si, 1
cyc11:   push     cx
mov      cx, n
sub      cx, si      ; число повторов внутреннего цикла
push     si          ; временно сохраним i — теперь j=n
mov      si, n - 1
;----- цикл по j с декрементом n-i раз
cyc12:   mov      al, mas[si-1] ; ЕСЛИ mas[j-1]< mas[j] ТО
cmp      mas[si], al
ja       ml         ;
mov      x, al      ; x:=mas[j-1]
mov      al, mas[si]
mov      mas[si-1], al ; mas[j-1]= mas[j]
mov      al, x
mov      mas[si], al ; mas[j]=x
ml:      dec      si
loop     cyc12
pop      si
inc      si
pop      cx
loop     cyc11
;...

```

Второй вариант сортировки прямым обменом

Эту сортировку называют шейкерной (программа prg4_104.asm). Она представляет собой вариант пузырьковой сортировки и предназначена для случая, когда последовательность почти упорядочена. Отличие шейкерной сортировки от пузырьковой в том, что на каждой итерации меняется направление очередного прохода: слева направо, справа налево.

```

ПРОГРАММА prg4_104
// prg4_104 — программа на псевдоязыке сортировки прямым обменом (шейкерной)
// Вход: mas[n] — неупорядоченная последовательность байтовых двоичных значений.
// Выход: mas[n] — упорядоченная последовательность байтовых двоичных значений.
ПЕРЕМЕННЫЕ
INT BYTE n=8; // количество элементов в сортируемом массиве
INT BYTE mas[n]; // сортируемый массив размерностью n (0..n-1)
INT BYTE X; i=0; j=0; r=0; l=0; k=0 // i, j, r, l, k — индексы
НАЧ ПРОГ
l:=2; r:=n; k:=n
ПОВТОРИТЬ

```

```

для j:=r ДОВНИЗ 1 // j изменяется от 1 до r
  ЕСЛИ mas[j-1]< mas[j] ТО
    НАЧ_БЛОК_1
      x:=mas[j-1]: mas[j-1]:=mas[j]: mas[j]:=x: k:=j
    КОН_БЛОК_1
для j:=1 ДОВНИЗ r // j изменяется от r до 1
  ЕСЛИ mas[j-1]< mas[j] ТО
    НАЧ_БЛОК_2
      x:=mas[j-1]: mas[j-1]:=mas[j]: mas[j]:=x: k:=j
    КОН_БЛОК_2
  r:=k-1
ПОКА (i>r)
КОН ПРОГ
+-----+
+| Программа: prg4_104.asm. Сортировка прямым выбором (шейкерная). |
+-----+
.data
mas      db      4, 55, 12, 42, 94, 18, 06, 67 : задаем массив
n:= $ - mas
x        db      0
L        dw      1
R        dw      n
k        dw      n
.code
:-----: для j:=r ДОВНИЗ 1 ; l:=2: r:=n: k:=n
сycl3:   mov     si, R ; j:=R
        push    si
        sub     si, L
        mov     cx, si ; число повторов цикла cycl1
        pop     si
        dec     si
cycl1:   mov     al, mas[si-1] ; ЕСЛИ mas[j-1]< mas[j] ТО
        cmp     al, mas[si]
        jna     m1
        mov     al, mas[si-1]
        mov     x, al ; x:=mas[j-1]
        mov     al, mas[si]
        mov     mas[si-1], al ; mas[j-1]:=mas[j]
        mov     al, x
        mov     mas[si], al ; mas[j]:=x
        mov     k, si ; k:=j
        dec     si ; j:=j-1
m1:      loop    cycl1
        mov     ax, k
        inc     ax
        mov     L, ax ; L:=k+1
:-----: цикл cycl2 : для j:=1 ДОВНИЗ r
        mov     si, L ; j:=L
        mov     ax, R
        sub     ax, L
        mov     cx, ax ; число повторов цикла cycl2
cycl2:   mov     al, mas[si-1] ; ЕСЛИ mas[j-1]< mas[j] ТО
        cmp     al, mas[si]
        jna     m2
        mov     al, mas[si-1]
        mov     x, al ; x:=mas[j-1]
        mov     al, mas[si]
        mov     mas[si-1], al ; mas[j-1]:=mas[j]
        mov     al, x
        mov     mas[si], al ; mas[j]:=x
        mov     k, si ; k:=j
        inc     si ; j:=j+1
m2:      loop    cycl2

```



```

mov    ax, k
dec    ax
mov    R, ax      : R:=k-1
cmp    L, ax      : L>R - ?
jae    $ + 5
jmp    cyc13
....

```

Улучшение классических методов сортировки

Приведенные выше четыре метода сортировки — базовые. Их обычно рассматривают как основу для последующего обсуждения и совершенствования. Ниже мы рассмотрим несколько усовершенствованных методов сортировки, после чего вы сможете оценить эффективность их всех.

Сортировка Шелла

Приводимый ниже алгоритм сортировки (программа `prg4_107.asm`) носит имя автора, который предложил его в 1959 году. Суть метода состоит в том, что сортировке подвергаются не все подряд элементы последовательности, а только отстоящие друг от друга на определенном *расстоянии* h . На каждом шаге значение h изменяется, пока не станет (обязательно) равно 1. Важно правильно подобрать значения h для каждого шага. От этого зависит скорость работы алгоритма. В качестве значений h можно использовать следующие числовые последовательности: (4, 2, 1), (8, 4, 2, 1) и даже (7, 5, 3, 1). Последовательности чисел можно вычислять аналитически. Так, Кнут предлагает следующие соотношения:

$$N_{k-1} = 3 \cdot N_k + 1,$$

в результате получается последовательность смещений: 1, 4, 13, 40, 121...

$$N_{k-1} = 2 \cdot N_k + 1,$$

в результате получается последовательность смещений: 1, 3, 7, 15, 31...

Подобное аналитическое получение последовательности смещений дает возможность разрабатывать программу, которая динамически подстраивается под конкретный сортируемый массив.

Отметим, что если $h = 1$, то алгоритм сортировки Шелла вырождается в сортировку прямыми включениями.

Существуют несколько вариантов алгоритма данной сортировки. Вариант, рассмотренный ниже, основывается на алгоритме, предложенном Кнутом.

```

ПРОГРАММА prg4_107
// prg4_107 — программа на псевдоязыке сортировки Шелла
// Вход: mas_dist=(7,5,3,1) — массив смещений: mas[n] — неупорядоченная
// последовательность байтовых двоичных значений.
// Выход: mas[n] — упорядоченная последовательность байтовых двоичных значений.
ПЕРЕМЕННЫЕ
INT BYTE t=4: // количество элементов в массиве смещений
INT BYTE mas[n]: // сортируемый массив размером n (0..n-1)
INT BYTE mas_dist[t]=(7,5,3,1): // массив смещений размером t (0..t-1)
INT BYTE h=0 // очередное смещение из mas_dist[]
INT BYTE X: i=0; j=0; s=0 // i, j, s — индексы
НАЧ ПРОГ
  для s:=0 до t-1
    НАЧ БЛОК_1
      h:=mas_dist[s]
      для j:=h до n-1

```

```

НАЧ_БЛОК_2
    i:=j-h; X:=mas[j]
    @@d4:    ЕСЛИ X>= mas[i] ТО ПЕРЕЙТИ_НА @@d6
    mas[i+h]:=mas[i]; i:=i-h
    ЕСЛИ i>=0 ТО ПЕРЕЙТИ_НА @@d4
    @@d6:    mas[i+h]:=X

```

```

    КОН_БЛОК_2

```

```

    КОН_БЛОК_1
КОН_ПРОГ

```

```

;-----+
;| Программа: prg4_107.asm. Сортировка Шелла. |
;-----+
.data
mas      db      4, 55, 12, 42, 94, 18, 06, 67 ; задаем массив
n = $ - mas
x        db      0
mas_dist db      7, 5, 3, 1 ; задаем массив расстояний
t = $ - mas_dist ; число элементов в массиве расстояний

.code
xor      bx, bx

;d1
mov      cx, t ; цикл по t (внешний)
mov      si, 0 ; индекс по mas_dist[]

@@d2:    push     cx
mov      bl, mas_dist[si] ; в bx — очередное смещение из mas_dist[]
inc      si
push     si ; для j:=h ДО n-1
mov      di, bx ; di — это j, j:=h+1 — это неавно
; для нумерации массива с нуля
; j=<n?
@@d3:    cmp      di, n - 1 ; конец итерации при очередном mas_dist[]
ja       @@m1
mov      si, di
sub      si, bx ; i:=j-h; si — это i
mov      al, mas[di] ; x:=mas[j]
mov      x, al ; x:=mas[j]

@@d4:    mov      al, x ; ЕСЛИ X>= mas[i] ТО ПЕРЕЙТИ_НА @@d6
cmp      al, mas[si]
jae      @@d6
push     di ; d5 — mas[i+h]:=mas[i]; i:=i-h
push     si
pop      di
add      di, bx ; i+h
mov      al, mas[si] ; mas[i+h]:=mas[i]
mov      mas[di], al ; mas[i+h]:=mas[i]
pop      di
sub      si, bx ; i:=i-h
cmp      si, 0 ; ЕСЛИ i>=0 ТО ПЕРЕЙТИ_НА @@d4
jge      @@d4

@@d6:    push     di ; mas[i+h]:=x
push     si
pop      di
add      di, bx ; i+h
mov      al, x
mov      mas[di], al ; mas[i+h]:=x
pop      di
inc      di ; j:=j+1
jmp      @@d3

@@m1:    pop      si
pop      cx
loop     @@d2

@@exit:
;...

```

Сортировка с помощью дерева

Следующий алгоритм (программа prg10_229.asm) является улучшением сортировки прямым выбором. Автор алгоритма (1964 год) сортировки с помощью дерева — Дж. Уильямс. В литературе эта сортировка носит название «пирамидальной». Если обсужденные выше сортировки интуитивно понятны, то алгоритм данной сортировки необходимо пояснить подробнее. Этот алгоритм предназначен для упорядочивания последовательности чисел, которые являются отображением в памяти дерева специальной структуры — пирамиды. Пирамида — помеченное двоичное дерево заданной высоты h , обладающее тремя свойствами:

- каждая конечная вершина имеет высоты h или $h - 1$;
- каждая конечная вершина высоты h находится слева от любой конечной вершины высоты $h - 1$;
- метка любой вершины больше метки любой следующей за ней вершины.

На рис. 2.3 изображено несколько деревьев, из которых лишь одно — T_4 — является пирамидой.

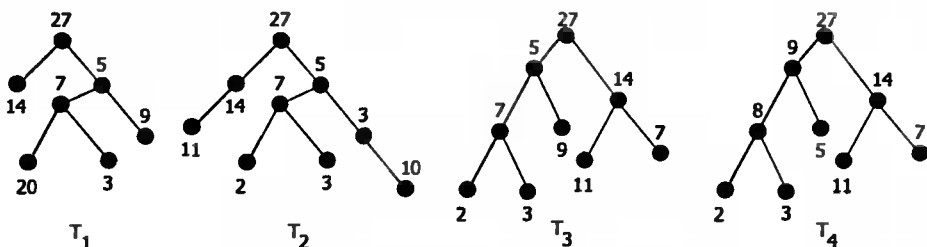


Рис. 2.3. Примеры деревьев (T_4 — пирамида)

Такая структура пирамид позволяет компактно располагать их в памяти. Например, пирамида, показанная на рисунке, в памяти будет представлена следующим массивом: 27, 9, 14, 8, 5, 11, 7, 2, 3. Оказывается, что эта последовательность чисел легко подвергается сортировке.

Таким образом, сортировка массива в соответствии с алгоритмом пирамидальной сортировки осуществляется в два этапа: на первом этапе массив преобразуется в отображение пирамиды; на втором выполняется собственно сортировка. Соответственно, нами должны быть разработаны две процедуры для решения задач каждого из этих двух этапов.

```

ПРОЦЕДУРА insert_item_in_tree (i, mas, N)
// insert_item_in_tree — процедура на псевдоязыке вставки элемента
// на свое место в пирамиду
// Вход: mas[n] — сформированная не до конца пирамида; i — номер добавляемого
// элемента в пирамиду из mas[n] (с конца); n — длина пирамиды
// Выход: действие — элемент добавлен в пирамиду.
НАЧ ПРОГ
    j:=i
@m1:   k:=2*j; l:=k+1
        ЕСЛИ (l<N И (mas[j]<mas[k] ИЛИ mas[j]<mas[l])) ТО ПЕРЕЙТИ_НА @m6
        ИНАЧЕ ПЕРЕЙТИ_НА @m2
        КОН ЕСЛИ
@m6:   ЕСЛИ mas[k]>mas[l] ТО ПЕРЕЙТИ_НА @m4
        ИНАЧЕ ПЕРЕЙТИ_НА @m3

```

```

      КОН_ЕСЛИ
@@m4:   x:=mas[j]
      mas[j]:=mas[k]
      j:=k
      mas[k]:=x
ПЕРЕЙТИ_НА @@m1
@@m3:   x:=mas[j]
      mas[j]:=mas[1]
      mas[1]:=x
      j:=1
ПЕРЕЙТИ_НА @@m1
@@m2:   ЕСЛИ (k==n И mas[j]<mas[k]) ТО ПЕРЕЙТИ_НА @@m7
      ИНАЧЕ ПЕРЕЙТИ_НА @@m8
      КОН_ЕСЛИ
@@m7:   x:=mas[j]
      mas[j]:=mas[n]
      mas[n]:=x
@@m8:
      ВОЗВРАТ
КОН ПРОГ
ПРОГРАММА prg10_229
// prg10_229 - программа на псевдоязыке пирамидальной сортировки
// Вход: mas[n] - неупорядоченная последовательность байтовых двоичных значений
// Выход: mas[n] - упорядоченная последовательность байтовых двоичных значений.
ПЕРЕМЕННЫЕ
INT BYTE n=16; // число элементов в mas[]
INT BYTE mas[n]; // сортируемый массив размерностью n (0..n-1)
INT BYTE X; // временная переменная
INT BYTE i=0; j=0; l=0; k=0; m=0 // индексы
НАЧ ПРОГ
      i:=n/2; l:=i; m:=n
// строим пирамиду на основе входного массива
      для k:=1 до 1
      НАЧ БЛОК 1
      insert_item_in_tree (i, mas, m)
      i:=i-1
      КОН БЛОК 1
// сортируем пирамиду
      для k:=2 до n
      НАЧ БЛОК 1
      x:=mas[1]
      mas[1]:=mas[m]
      mas[m]:=x
      m=m-1
      insert_item_in_tree (1, mas, m)
      КОН БЛОК 1
КОН ПРОГ

```

```

:-----+
:| Программа: prg10_229.asm. Пирамидальная сортировка. |
:-----+
.data
mas      db      17h, 05h, 99h, 3h, 33h, 7h, 13h, 27h, 77h
          db      9h, 69h, 11h, 81h, 14h, 2h, 8 ; задаем массив
          n = $ - mas
x         db      0 ; временная переменная
l         dw      0
l         dw      0
m         dw      0
j         dw      0
k         dw      0
.code
:-----+
:| Процедура: insert_item_in_tree (i, mas, N). |
:-----+

```

```

+-----+
: Вход: si - номер элемента i в последовательности (с конца).
: mas - массив элементов
: m=n/2 - значение, равное половине числа элементов массива mas.
+-----+
insert_item_in_tree proc
    push    si
    push    cx
    mov     j, si          ; j:=i
    mov     si, j          ; i->si
    mov     ax, j          ; k:=2*j; l:=k+1
    shl     ax, 1          ; j*2
    mov     k, ax          ; k:=j*2
    inc     ax             ; l:=k+1
    mov     l, ax          ; l:=k+1
    cmp     ax, m          ; l<m?
    ja      @m2
    mov     al, mas[si-1]   ; ax:=mas[j]
    mov     di, k
    cmp     al, mas[di-1]
    jna     @m6
    inc     di
    cmp     al, mas[di-1]
    jna     @m6
    jmp     @m2

:----- условие выполнилось:
: 2j+1<M AND (mas[j]<mas[2j] OR mas[j]<mas[2j+1])
: обмен с mas[j]
@m6:    mov     di, k
    mov     al, mas[di-1]   ; ax:=mas[k]
    inc     di
    cmp     al, mas[di-1]   ; mas[k]>mas[l]
    jna     @m3
    mov     al, mas[si-1]
    mov     x, al           ; x:=mas[j]
    dec     di
    mov     al, mas[di-1]
    mov     mas[si-1], al   ; mas[j]:=mas[k]
    mov     j, di          ; j:=k
    mov     al, x
    mov     mas[di-1], al   ; mas[k]:=x
    jmp     @m4

: @m3:
@m3:    mov     al, mas[si-1] ; x:=mas[j] ПЕРЕЙТИ НА @m1
    mov     x, al           ; mas[k] <= mas[l]
    mov     al, mas[di-1]   ; x:=mas[j]
    mov     mas[si-1], al   ; mas[j]:=mas[l]
    mov     j, di          ; j:=l
    mov     al, x
    mov     mas[di-1], al   ; mas[l]:=x
    jmp     @m4

:----- условие не выполнилось:
: 2j+1<M AND (mas[j]<mas[2j] OR mas[j]<mas[2j+1])
@m2:    mov     ax, k
    cmp     ax, m
    jne     @m1
    mov     di, k
    mov     al, mas[di-1]   ; al:=mas[k]
    cmp     mas[si-1], al   ; mas[j]<mas[k]
    jae     @m1
    mov     al, mas[si-1]
    mov     x, al           ; x:=mas[j]
    mov     di, n
    mov     al, mas[di-1]
    mov     mas[si-1], al   ; mas[j]:=mas[n]

```

```

        mov     al, x
        mov     mas[di-1], al    : mas[n]:=x
@@m1:   pop     cx
        pop     si
        ret
insert_item_in_tree endp
main:
        ;...
        mov     ax, n            : n-1
        mov     m, ax            : m:=n
        shr     ax, 1
        mov     i, ax            : i:=n\2
        mov     l, ax            : l:=i
        ;----- строим пирамиду на основе входного массива
        mov     cx, l            : цикл по k:=1..l
        mov     si, i
@@cyc1: mov     i, si
        call    insert_item_in_tree
        dec     si                : i:=i-1
        loop    @@cyc1
        ;----- а теперь собственно сортировка:
@@cyc2: mov     cx, n - 1          : n-2
        xor     si, si
        mov     al, mas[si]
        mov     x, al             : x:=mas[l]
        mov     di, m
        mov     al, mas[di-1]
        mov     mas[si], al        : mas[l]:=mas[m]
        mov     al, x
        mov     mas[di-1], al      : mas[m]:=x
        dec     m
        inc     si                : i:=1
        call    insert_item_in_tree
        loop    @@cyc2
        ;...
end main

```

Нахождение медианы

В этом месте обсуждение примеров реализации на ассемблере методов сортировки будет прервано с тем, чтобы обсудить один вспомогательный алгоритм — нахождение медианы числовой последовательности (программа `prg4_123.asm`). Его применение эффективно в контексте «быстрой сортировки», которая будет рассмотрена нами в следующем разделе.

Медиана — элемент последовательности, значение которого не больше значений одной половины этой последовательности и не меньше значений другой половины этой последовательности. Например, медианой последовательности чисел **38 7 5 90 65 8 74 43 2** является **38**. Соответствующая отсортированная последовательность будет выглядеть так: **2 5 7 8 38 43 65 74 90**.

Задачу нахождения медианы можно решить просто — предварительно отсортировать исходный массив и выбрать средний элемент. Но К. Хоор предлагает метод, который решает задачу нахождения медианы быстрее и, соответственно, может рассматриваться как вспомогательный для реализации других задач. Достоинство метода К. Хоора заключается в том, что с его помощью можно эффективно находить не только медиану, но и значение k -го по величине элемента последовательности. Например, третьим по величине элементом приведенной выше последовательности будет **7**.

Значение k -го элемента массива определяется по формуле $k = n/2$, где n — длина исходной последовательности чисел.

Ниже приведена программа `prg4_123.asm`, которая обнаруживает элемент-медиану массива. Аналогичную функцию выполняет и имеющаяся среди файлов, прилагаемых к книге, процедура `median`, которая отличается тем, что ее можно вызывать динамически во время работы программы, в которой она используется.

ПРОГРАММА `prg4_123`

// `prg4_123` — программа на псевдоязыке нахождения k -го по величине элемента массива `mas` длиной n . Для нахождения медианы $k=n/2$.

// Вход: `mas[n]` — неупорядоченная последовательность двоичных значений;

// k — номер искомого по величине элемента `mas[n]`.

// Выход: x — значение k -го по величине элемента последовательности `mas[n]`.

ПЕРЕМЕННЫЕ

INT_BYTE `n`; // длина `mas[n]`

INT_BYTE `mas[n]`; // сортируемый массив размерностью n ($0..n-1$)

INT_BYTE `x`; `w`; `i`=0; `j`=0; `l`=0; `r`=0 // `j`: `l`; `r` — индексы

НАЧ_ПРОГ

`l`:=1; `r`:=`n`

ПОКА `l`<`r` ДЕЛАТЬ

НАЧ_БЛОК_1

`x`:=`a[k]`; `i`:=1; `j`:=`r`

ПОВТОРИТЬ

ПОКА `a[i]`<`x` ДЕЛАТЬ `i`:=`i`+1

ПОКА `x`< `a[j]` ДЕЛАТЬ `j`:=`j`-1

ЕСЛИ `i`≤`j` ТО

НАЧ_БЛОК_2

`w`:=`a[i]`; `a[i]`:=`a[j]`; `a[j]`:=`w`

`i`:=`i`+1; `j`:=`j`-1

КОН_БЛОК_2

ПОКА (`i`>`j`)

ЕСЛИ `j`<`k` ТО `l`:=`i`

ЕСЛИ `k`<`i` ТО `r`:=`j`

КОН_БЛОК_1

КОН_ПРОГ

```

;+-----+
;| Программа: prg4_123. Поиск k-го элемента массива mas длиной n.      |
;| Для нахождения медианы k=n/2.                                         |
;+-----+

```

.data

```

mas      db      3Bh, 08h, 16h, 06h, 79h, 76h, 57h, 24h, 56h, 02h
          db      58h, 4Bh, 04h, 70h, 45h, 47h ; задаем массив
          n = $ - mas
L         dw      1
R         dw      n
k         dw      9
x         db      0

```

.code

;----- проверка, что $k \leq n-1$

`mov` `ax`, `n` - 1

`cmp` `k`, `ax`

`jg` `exit`

;----- адресация элементов массива с 0

`dec` `L`

`dec` `R`

;----- цикл(1)начало, пока $L < R$

@m8: `mov` `ax`, `L`

`cmp` `ax`, `R`

`jge` `exit`

```

mov     bx, k           : temp:=mas[k]
mov     al, mas[bx]     : al — это x
mov     x, al
mov     si, L           : I:=L si это i
mov     di, R           : J:=R di это j
;----- цикл ПОВТОРИТЬ начало
@em7:   cmp     mas[si], al : mas[i]<temp?
        jae     @em3
        inc     si
        jmp     @em7
;----- ПОКА x< a[j] ДЕЛАТЬ j:=j-1
@em3:   cmp     al, mas[di] : temp<mas[j]?
        jae     @em5
        dec     di
        jmp     @em3
@em5:   cmp     si, di     : ЕСЛИ i≤j TO
        ja      @em6
;----- если I=<J, то обмен mas[i]<->mas[j]
        mov     dl, mas[si] : w:=a[i]; a[i]:=a[j]; a[j]:=w
        xchg    mas[di], dl
        xchg    mas[si], dl
        inc     si         : i:=i+1
        dec     di         : j:=j-1
;----- цикл(2) конец, пока I=<J
@em6:   cmp     si, di
        jg      @em1      : ja нельзя!!!
        jmp     @em7
@em1:   cmp     di, k
        jge     $ + 6      : ЕСЛИ j<k TO l:=I
        mov     L, si      : L<-I
        cmp     k, si      : ЕСЛИ k<i TO r:=j
        jge     $ + 6
        mov     R, di      : R<-J
        jmp     @em8      : цикл(1)конец
;....

```

Быстрая сортировка

Последний рассматриваемый нами алгоритм (программа prg10_223.asm) является улучшенным вариантом сортировки прямым обменом. Этот метод разработан К. Хоором в 1962 году и назван им «быстрой сортировкой». Эффективность быстрой сортировки зависит от степени упорядоченности исходного массива. Для сильно неупорядоченных массивов — это один из лучших методов сортировки. В худшем случае, когда исходный массив почти упорядочен, его быстрая сортировка по эффективности не намного лучше сортировки прямым обменом.

Идея метода быстрой сортировки состоит в следующем. Первоначально среди элементов сортируемого массива $mas[1...n]$ выбирается один $mas[k]$, относительно которого выполняется переупорядочивание остальных элементов по принципу — элементы $mas[i] < mas[k]$ ($i = 0...n - 1$) помещаются в левую часть mas ; элементы $mas[i] > mas[k]$ ($i = 0...n - 1$) помещаются в правую часть mas . Далее процедура повторяется в полученных левой и правой подпоследовательностях и т. д. В конечном итоге исходный массив будет правильно отсортирован. В идеальном случае элемент $mas[k]$ должен являться медианой последовательности, то есть элементом последовательности, значение которого не больше значений одной части и не меньше значений оставшейся части этой последовательности. Нахождение медианы обсуждалось в предыдущем разделе. В следующей программе элементом $mas[k]$ является самый левый элемент подпоследовательности.


```

ПРОГРАММА prg27_136
// prg27_136 (по Кнуту) – программа на псевдоязыке «быстрой сортировки» массива.
// Вход: mas[n] – неупорядоченная последовательность двоичных значений длиной n.
// Выход: mas[n] – упорядоченная последовательность двоичных значений длиной n.
ПЕРЕМЕННЫЕ
INT BYTE n: // длина mas[n]
INT BYTE mas[n]: // сортируемый массив размерностью n (0..n-1)
INT BYTE k: TEMP: i=0: j=0: l=0: r=0: // i, j, l, r – индексы
INT WORD M=1 // длина подпоследовательности, для которой производится
// сортировка методом прямых включений
// для отладки возьмем M=1
НАЧ ПРОГ
    ЕСЛИ M=N ТО ПЕРЕЙТИ_НА q9
    l:=1: r:=n
    ВКЛЮЧИТЬ_В_СТЕК (0ffffh) // 0ffffh – признак пустого стека
q2: i:=1: j:=r+1: k:=mas[l]
q3: ЕСЛИ i=<j-1 ТО ПЕРЕЙТИ_НА qq3
    ПЕРЕЙТИ_НА q4 //итерация прекращена
qq3: i:=i+1
    ЕСЛИ mas[i]<k ТО ПЕРЕЙТИ_НА q3
q4: j:=j-1
    ЕСЛИ j<i ТО ПЕРЕЙТИ_НА q5
    ЕСЛИ k<mas[j] ТО ПЕРЕЙТИ_НА q4
    ЕСЛИ j>i ТО ПЕРЕЙТИ_НА q6
// обмен mas[l]:=mas[j]: mas[j]:=TEMP
    TEMP:=mas[l]: mas[l]:=mas[j]: mas[j]:=TEMP
ПЕРЕЙТИ_НА q7
q6: // обмен mas[i]<-> mas[j]
    TEMP:=mas[i]: mas[i]:=mas[j]: mas[j]:=TEMP
ПЕРЕЙТИ_НА q3
q7: ЕСЛИ r-j≥j-l>M ТО
    НАЧ БЛОК_1
        ВКЛЮЧИТЬ_В_СТЕК (j+1,r)
        r:=j-1
        ПЕРЕЙТИ_НА q2
    КОН БЛОК_1
    ЕСЛИ j-l>r-j>M ТО
    НАЧ БЛОК_1
        ВКЛЮЧИТЬ_В_СТЕК (l,j-1)
        l:=j+1
        ПЕРЕЙТИ_НА q2
    КОН БЛОК_1
    ЕСЛИ r-j>M≥j-l ТО
    НАЧ БЛОК_1
        l:=j+1
        ПЕРЕЙТИ_НА q2
    КОН БЛОК_1
    ЕСЛИ j-l>M≥r-j ТО
    НАЧ БЛОК_1
        r:=j-1
        ПЕРЕЙТИ_НА q2
    КОН БЛОК_1
    ИЗВЛЕЧЬ ИЗ СТЕКА (l,r)
    ЕСЛИ r<0ffffh ТО ПЕРЕЙТИ_НА q2
// далее, если M>1, должен быть фрагмент программы сортировки методом прямых
// включений (для экономии места опущен)
КОН ПРОГ

```

```

:-----+
: | Программа: prg27_136. Быстрая сортировка массива по Кнуту. |
:-----+
.data
mas      db      50h, 08h, 52h, 06h, 90h, 17h, 89h, 27h, 65h, 42h
          db      15h, 51h, 61h, 67h, 76h, 70h : задаем массив

```

```

k      n = $ - mas
L      db      1
R      dw      0          ; адресация элементов массива с 0
M      dw      n-1        ; адресация элементов массива до n
      dw      1          ; длина подпоследовательности
      ; (для отладки возьмем 1)

I      dw      0
J      dw      0
temp   db      0
.code

:----- ЕСЛИ M≤N ТО ПЕРЕЙТИ_НА q9
:q1     cmp     M, n        ; l:=1; r:=n
      jae     q9          ; на шаг 9
      push    0ffffh      ; признак пустого стека

:----- i:=1; j:=r+1; k:=mas[l]
q2:     mov     si, l        ; i(si)=L
      mov     di, R
      inc     di            ; j(di)=r+1
      xor     ax, ax
      mov     al, mas[si]
      mov     k, al

:----- ЕСЛИ i≤j-1 ТО ПЕРЕЙТИ_НА q3
q3:     inc     si          ; i:=i+1
      cmp     si, di        ; i≤j?
      jle     qq3
      dec     si            ; сохраняем i=j
      jmp     q4            ; ПЕРЕЙТИ_НА q4 // итерация прекращена
q3:     mov     al, k        ; i:=i+1
      cmp     mas[si], al   ; ЕСЛИ mas[i]<k ТО ПЕРЕЙТИ_НА q3
      jb      q3
q4:     dec     di          ; j:=j-1
      cmp     di, si        ; j>=i-1
      jb      q5

:----- ЕСЛИ k<mas[j] ТО ПЕРЕЙТИ_НА q4
      mov     al, k
      cmp     al, mas[di]
      jb      q4

:----- ЕСЛИ j>i ТО ПЕРЕЙТИ_НА q6
q5:     cmp     di, si        ; j<=i?
      jge     q6
:----- // обмен mas[l]:=mas[j]
      mov     bx, l
      mov     di, mas[bx]
      xchg    mas[di], di
      xchg    mas[bx], di
      jmp     q7            ; ПЕРЕЙТИ_НА q7

:----- mas[i]↔mas[j]
q6:     mov     di, mas[si]
      xchg    mas[di], di
      xchg    mas[si], di
      jmp     q3            ; ПЕРЕЙТИ_НА q3

:----- ЕСЛИ r-j≥j-1>M ТО
q7:     mov     ax, r
      sub     ax, di        ; r-j->ax
      mov     bx, di
      sub     bx, 1         ; j-1->bx
      cmp     ax, bx        ; r-j≥j-1?
      jl      q7_m4
      cmp     bx, M         ; j-1>M?
      jle     q7_m3

:----- r-j≥j-1>M = в стек (j+1,r): r:=j-1; перейти на шаг q2
      mov     ax, di
      inc     ax
      push    ax

```

```

push    r
mov     r, di
dec     r                ; r:=j-1
q7_m4:  cmp     ax, M      ; r-j>M?
        jg     q7_m2
        cmp     bx, M
        jle    q8
        :----- 4. j-l>M≥r-j - r:=j-1; перейти на шаг q2
        mov     r, di
        dec     r
q7_m3:  jmp     q2
        cmp     ax, M
        jle    q8
        :----- 3. r-j>M≥j-l - l:=j+1; перейти на шаг q2
        mov     l, di
        inc     l
        jmp     q2
q7_m2:  :----- 2. j-l>r-j>M - в стек (l,j-1); l:=j+1; перейти на шаг q2
        push    l
        mov     ax, di
        inc     ax
        push    ax
        mov     l, di
        inc     l
        jmp     q2
q8:     :----- извлекаем из стека очередную пару (l,r)
        pop     r
        cmp     r, 0ffffh ; ЕСЛИ r<>0ffffh ТО ПЕРЕЙТИ_НА q2
        je     q9
        pop     l
        jmp     q2
        :----- сортировка методом прямых включений -
        :       при M=1 этот шаг можно опустить
        :       (что и сделано для экономии места)
q9:     :...

```

Обратите внимание на каскад сравнений шага q7, целью которых является проверка выполнения следующих неравенств:

- $r - j \geq j - l > M$ — в стек ($j + 1, r$); $r := j - 1$; перейти на шаг q2;
- $j - l > r - j > M$ — в стек ($l, j - 1$); $l := j + 1$; перейти на шаг q2;
- $r - j > M \geq j - l - l = j + 1$; перейти на шаг q2;
- $j - l > M \geq r - j - r = j - 1$; перейти на шаг q2.

Проверка этих неравенств реализуется в виде попарных сравнений, последовательность которых выглядит так, как показано на рис. 2.4.

За подробностями алгоритма можно обратиться к тому 3 «Сортировка и поиск» книги Кнута [27].

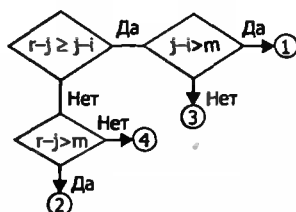


Рис. 2.4. Последовательность сравнений шага q7

Существует другой вариант алгоритма этой сортировки — у Вирта в книге «Алгоритмы + структуры данных = программы» [4]. Но у него используется понятие медианы последовательности. На задаче нахождения медианы мы останавливались выше. Она интересна еще и тем, что, по сути, является одной из задач поиска, рассмотрению которых посвящен следующий раздел.

Поиск в массивах

Поиск информации является одной из самых распространенных проблем, с которыми сталкивается программист в процессе написания программы. Правильное решение задачи поиска для каждого конкретного случая позволяет значительно повысить эффективность исполнения программы. Для общего случая обычно предполагается, что поиск ведется среди массива записей определенной структуры. В каждой записи есть уникальное ключевое поле. Абстрагируясь, можно сказать, что массив записей — это массив ключевых полей. В этом случае задачу поиска в массиве записей можно свести к задаче поиска в массиве ключевых слов. В этом разделе мы обсудим проблему поиска в массивах и пути ее решения.

В отличие от методов сортировки, классификация методов поиска не отличается особым разнообразием. Уверенно можно сказать, что методы поиска будут различными для упорядоченных и неупорядоченных массивов. Неупорядоченными массивами здесь считаются массивы, о порядке следования элементов которых нельзя сделать никаких предположений. Для таких массивов особых способов поиска нет, кроме как последовательно просматривать все их элементы. В теории такой поиск называется линейным. Если элементы массивов каким-то образом отсортированы, то речь идет об упорядоченных массивах и для поиска в них существует определенный набор методов. В следующих разделах мы рассмотрим ассемблерную реализацию наиболее интересных методов поиска, применяемых в тех случаях, когда ключевое поле — некоторое число. Большой интерес представляют методы поиска в строке, которые рассматриваются в главе 4. И наконец, существует третий класс методов поиска, основанный на арифметическом преобразовании исходных ключей — «хэшировании». Эти методы мы рассмотрим в разделе «Поиск в таблице».

Неупорядоченный поиск

Для выполнения неупорядоченного (линейного) поиска нет определенных методов. Чтобы эффективно реализовать эту операцию, необходимо уметь использовать средства языка программирования. Если поиск ведется в массиве чисел (а не записей с числовым ключевым полем), то наибольшей эффективности для реализации линейного поиска в программе на языке ассемблера можно достичь, если использовать цепочечные команды. Поэтому мы рассмотрим варианты реализации линейного поиска, предполагая, что массив, в котором ведется поиск, на самом деле является массивом записей и использование цепочечных команд не представляется возможным. Для вставки приведенных ниже фрагментов в свои реальные программы вы должны скорректировать соответствующие смещения.

Первый вариант неупорядоченного поиска

Этот вариант программы реализации линейного поиска предполагает обычный перебор до тех пор, пока не будет найден нужный элемент или не будут перебраны все элементы.

```

ПРОГРАММА prg27_429
// prg27_429 – программа на псевдоязыке линейного поиска в массиве (вариант 1).
// Вход: mas[n] – неупорядоченная последовательность двоичных значений длиной n;
// k – искомое значение
// Выход: i – позиция в mas[n] (0<i<n-1), соответствующая найденному символу.
ПЕРЕМЕННЫЕ
INT BYTE n; // длина mas[n]
INT BYTE mas[n]; // массив для поиска размерностью n (0..n-1)
INT BYTE k; // искомый элемент
INT WORD i=0 // индекс
НАЧ_ПРОГ
    i:=0
s2: ЕСЛИ (k==mas[i]) ТО ПЕРЕЙТИ_НА _exit
    i:=i+1
    ЕСЛИ (i<n) ТО ПЕРЕЙТИ_НА s2
_exit: // вывести значение i по месту требования
КОН_ПРОГ

```

```

+-----+
+| Программа: prg27_429.asm. Линейный поиск в массиве (вариант 1). |
+-----+
.data
mas      db      50h, 0Bh, 52h, 06h, 90h, 17h, B9h, 27h, 65h, 42h
          db      15h, 51h, 61h, 67h, 76h, 70h ; задаем массив
          n = $ - mas
k         db      15h
.code
          xor     si, si           ; i=(si):=0
          mov     al, k
s2:       cmp     al, mas[si]      ; ЕСЛИ k==mas[i] ТО ПЕРЕЙТИ_НА _exit
          je      ok
          inc     si              ; i:=i+1
          cmp     si, n - 1        ; ЕСЛИ i<n ТО ПЕРЕЙТИ_НА s2
          jbe     s2
          ; реакция на неудачный результат поиска
          ....
          jmp     _exit
ok:       ; реакция на удачный результат поиска
          ....
          jmp     _exit
          ....

```

Второй вариант неупорядоченного поиска

Программу первого варианта линейного поиска можно немного усовершенствовать вводом дополнительного элемента — *барьера*.

```

ПРОГРАММА prg27_430
// prg27_430 – программа на псевдоязыке линейного поиска в массиве (вариант 2).
// Вход: mas[n] – неупорядоченная последовательность двоичных значений длиной n+1;
// k – искомое значение
// Выход: i – позиция в mas[n] (0<i<n-1), соответствующая найденному символу.
ПЕРЕМЕННЫЕ
INT BYTE n; // длина mas[n]
INT BYTE mas[n+1]; // массив для поиска размерностью n+1 (0..n)
INT BYTE k; // искомый элемент
INT WORD i=0 // индекс
НАЧ_ПРОГ

```

```

i:=0; mas[n]:=k    // установить барьер
s2: ЕСЛИ (k==mas[i]) ТО ПЕРЕЙТИ_НА _exit
    i:=i+1; ПЕРЕЙТИ НА s2
_exit: ЕСЛИ i>n ТО ПЕРЕЙТИ_НА exit_error
// вывести значение i по месту требования
exit_error: // вывести сообщение об отсутствии элемента
КОН_ПРОГ

```

```

-----+
:| Программа: prg27_430. Линейный поиск в массиве (вариант 2). |
-----+
.data
mas      db      50h, 0Bh, 52h, 06h, 90h, 17h, B9h, 27h, 65h, 42h
          db      15h, 51h, 61h, 67h, 76h, 70h : задаем массив
          n = $ - mas
          db      0h                               : дополнительный элемент для барьера
k         db      15h
.code
xor       si, si
mov       al, k
mov       mas[n], al    ; i:=0; mas[n]:=k // установить барьер
s2:       cmp     al, mas[si] ; ЕСЛИ k==mas[i] ТО ПЕРЕЙТИ_НА _exit
          je      ok
          inc     si
          jmp     s2      ; i:=i+1; ПЕРЕЙТИ НА s2
ok:       cmp     si, n    ; ЕСЛИ i>n ТО ПЕРЕЙТИ_НА exit_error
          je      exit_error
          ;----- реакция на удачный результат поиска
          ....
          jmp     exit
          ;----- реакция на неудачный результат поиска
exit_error: ....

```

Упорядоченный поиск

На практике массивы элементов (записей) обычно определенным образом упорядочиваются. Это облегчает задачу поиска в них, так как появляется возможность формализации этого процесса. Записи в массиве в зависимости от значений ключевых полей могут быть отсортированы в числовом или лексикографическом порядке.

Двоичный поиск

Этот вид поиска (программа prg10_242.asm) предназначен для выполнения поиска в упорядоченном массиве (записей), содержащем N числовых ключей: $k_1 < k_2 < \dots < k_N$. Упорядоченный массив чисел 02h, 04h, 06h, 08h, 16h, 24h, 38h, 45h, 47h, 48h, 56h, 57h, 58h, 70h, 76h, 79h можно представить в виде двоичного дерева (рис. 2.5).

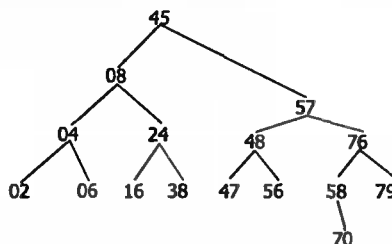


Рис. 2.5. Представление массива в виде двоичного дерева

Для этого необходимо написать программу, которая в цикле выполняет следующие действия. Вначале определяется элемент, расположенный в середине исходного массива, — он будет вершиной двоичного дерева. Далее в стеке запоминаются интервалы ключей подмассивов, расположенных слева и справа от элемента-вершины двоичного дерева. После этого в цикле происходит извлечение интервалов ключей подмассивов из стека, определение серединных элементов в этих подмассивах, разбиение на очередные подмассивы, запоминание их границ в стеке, извлечение и т. д. В результате будет построено двоичное дерево, подобное изображенному на рисунке.

Абстрактно задачу поиска ключа с определенным значением K можно сравнить с обходом двоичного дерева начиная с его вершины. Если K меньше значения ключа вершины, то идем к узлу дерева по левой ветке, если больше — то по правой. Значение ключа в очередном узле соответствует среднему элементу подмассива, лежащему слева (справа) от элемента, соответствующего вершине дерева. Для среднего элемента подмассива процедура сравнения и принятия решения о переходе повторяется. Процесс заканчивается, когда обнаружен узел, ключ которого равен K , либо очередной узел является листом, и двигаться больше некуда.

```

+-----+
+| Программа: prg10_242.asm. Двоичный (бинарный) поиск. |
+-----+
+| Вход: mas[n] — упорядоченная последовательность двоичных чисел длиной N. |
+|      k — искомое значение. |
+-----+
+| Выход: i — позиция в mas[n] (0<i<n-1), соответствующая найденному символу. |
+-----+
.data
mas      db      02h, 04h, 06h, 08h, 16h, 24h, 38h, 45h, 47h, 48h, 57h
          db      56h, 58h, 70h, 76h, 79h ; задаем массив
          n = $ - mas
k         db      4 ; искомое значение
.code
:----- в si и di индексы первого и последнего элементов последней
:      просматриваемой части последовательности
          mov     si, 0
          mov     di, n - 1
          xor     ax, ax
          mov     al, k ; искомое значение в ax
:----- проверка на окончание (неуспех): si>di
cont_search: cmp     si, di
              ja     exit_bad
:----- получим центральный индекс
          mov     bx, si
          add     bx, di
          shr     bx, 1 ; делим на 2
          adc     bx, 0 ; округляем в большую сторону
          cmp     mas[bx], al ; сравниваем с искомым значением
          je      exit_good ; искомое значение найдено
          ja      @@m1 ; mas[bx]>k
          mov     si, bx ; mas[bx]<k:
          inc     si
          jmp     cont_search
@@m1:     mov     di, bx
          dec     di
          jmp     cont_search
exit_bad: nop ; вывод сообщения об ошибке
          ....

```

```
exit_good:    nop                ; вывод сообщения об успехе
              ....
```

Такой подход очень эффективен для поиска в предварительно отсортированном массиве (необходимое условие). Другая возможность представления двоичного дерева — список. В списке каждый узел, кроме уникального ключа, содержит информационную часть и ссылки на последующий и, возможно, предыдущий элементы списка. Достоинство представления двоичного дерева в таком виде в том, что списком достаточно легко описывать дерево, в которое включаются и выключаются узлы. При представлении двоичного дерева в виде массива это сделать труднее. Ниже мы рассмотрим представление деревьев в программах на языке ассемблера.

Действия с матрицами

Рассмотренные выше операции над массивами были реализованы исходя из предположения, что массивы имеют одну размерность. Но существует группа задач, в которой работа осуществляется с массивами большей размерности. Прежде всего это действия с матрицами. Кстати, такая структура данных, как сеть (или граф), представляется с помощью матрицы [10].

Транспонирование прямоугольной матрицы

Приемы работы с массивами размерностью больше 1 удобно рассматривать на типовой задаче, например такой, как транспонирование матрицы.

Суть задачи транспонирования матрицы $A = \{a_{ij}\}$ заключается в замене строк столбцами и столбцов строками в соответствии с формулой $a'_{ij} = a_{ji}$, где a'_{ij} — элементы транспонированной матрицы $A' = \{a'_{ij}\}$. Максимальная величина индексов i и j задается константами m (количество строк) и n (количество столбцов), соответственно диапазон их значений составляет: $i = 0 \dots m - 1, j = 0 \dots n - 1$. Элементы матрицы определяются статически — в сегменте данных.

Выше уже отмечалось то, каким образом производится локализация в памяти элемента многомерного массива исходя из его логического номера при условии, что размерность элементов — 1 байт. Локализация элемента матрицы A относительно базового адреса производится по формуле:

$$a_{ij} = n \cdot i + j.$$

Соответствующий элемент в транспонируемой матрице будет расположен по адресу:

$$a'_{ij} = m \cdot i + j.$$

Например, рассмотрим матрицу 3×4 :

```
02h 04h 06h 08h
16h 24h 38h 45h
47h 4Bh 57h 56h
```

Эта матрица в памяти будет выглядеть так:

```
02h. 04h. 06h. 08h. 16h. 24h. 38h. 45h. 47h. 4Bh. 57h. 56h
```

Транспонированный вариант матрицы:

```
02h 16h 47h
04h 24h 4Bh
06h 38h 57h
08h 45h 56h
```


Транспонированный вариант матрицы в памяти будет выглядеть следующим образом:

02h, 16h, 47h, 04h, 24h, 48h, 06h, 38h, 57h, 08h, 45h, 56h

Для решения задачи «в лоб» по формулам $a_{ij} = n \cdot i + j$ и $a'_{ij} = m \cdot i + j$ требуется выделять в памяти область для хранения транспонированной матрицы, совпадающую по размеру с исходной.

```

-----+
:| Программа: prg29_102.asm. Транспонирование матрицы. |
-----+
:| Вход: mas[n] – матрица m × n. |
-----+
:| Выход: _mas[n] – транспонированная матрица n × m. |
-----+
.data
m          dw      3           : i=0...2
n          dw      4           : j=0...3
mas        db      02h, 04h, 06h : задаем матрицу 3×4
           db      08h, 16h, 24h
           db      38h, 45h, 47h
           db      48h, 57h, 56h
           s mas = $ - mas
_mas       db      s mas dup (0ffh)
temp       db      0
.code
           mov     cx, m
           xor     si, si       : i:=0
m1:        push    cx           : цикл по i
           mov     cx, n
           xor     di, di       : j:=0
           :----- локализуем masij по формуле: masij=n*i+j
m2:        mov     ax, n
           mul     si           : предполагаем, что результат лишь в ax
           add     ax, di       : n*i+j
           mov     bx, ax
           mov     al, mas[bx]
           mov     temp, al
           :----- локализуем местоприменик в _masij по формуле: _masij=masij=m*i+j
           mov     ax, m
           mul     di           : предполагаем, что результат лишь в ax
           add     ax, si       : _masij=masij=m*i+j
           mov     bx, ax
           mov     al, temp
           mov     _mas[bx], al
           inc     di           : j:=j+1
           loop    m2
           inc     si
           pop     cx           : восстанавливаем счетчик внешнего цикла
           loop    m1
           ....

```

Отметим, что для транспонирования прямоугольной матрицы необязательно ее моделировать так, как это сделано в предыдущей программе. Кнут приводит соотношение, которое позволяет транспонировать матрицу в линейном порядке, зная только значения m и n . Для этого используется соотношение, при котором значение из ячейки i (для $0 \leq i < N = m \cdot n - 1$) исходной матрицы переводится в ячейку $(m \cdot x) \bmod N$ транспонированной матрицы. Пример программы транспонирования матрицы в линейном порядке prg13_3_217.asm присутствует среди файлов, прилагаемых к книге.

Структуры

Чтобы правильно использовать машину, важно добиться хорошего понимания структурных отношений, существующих между данными, способов представления таких структур в машине и методов работы с ними.

Д. Кнут

Структура (запись) — конечное упорядоченное множество элементов, в общем случае имеющих различный тип. Элементы, входящие в состав структуры, называются *полями*. Каждое поле структуры имеет имя, по которому производится обращение к содержимому поля. Пример структуры — совокупность сведений о некотором человеке: номер ИНН, фамилия, имя, отчество, дата рождения, место работы, трудовой стаж и т. п. Подробно порядок описания структур и некоторые приемы работы с ними описаны в учебнике. Поэтому мы, чтобы не повторяться, рассмотрим те практически важные моменты, которые там были упущены: вложенность структур; использование структур для моделирования таблиц и организация работы с ними.

Вложенные структуры

В общем случае поле структуры само является структурой. Тогда структура представляет собой иерархическую конструкцию, которую можно изобразить в виде дерева (рис. 2.6).

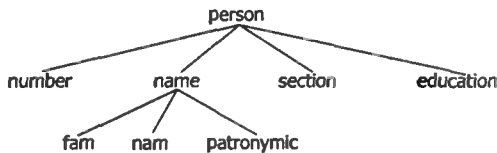


Рис. 2.6. Иерархическая структура сложной записи

Листы в этом дереве являются той полезной информацией, к которой необходимо получить доступ. Для этого нужно некоторым образом указать последовательность идентификаторов, в которой первым идентификатором является имя структуры, далее следуют идентификаторы промежуточных узлов и в последнюю очередь — идентификатор нужного листа. Но в ассемблере подобная схема реализована очень ограниченно — можно указать только имена первого и последнего идентификаторов (промежуточных попросту нет). Рассмотрим пример. Как обычно, прежде чем работать со структурой, необходимо задать ее шаблон. Далее для работы со структурой в программе необходимо создать ее экземпляр. В листинге ниже шаблон структуры называется `element`, а соответствующий ему экземпляр структуры — `s1`. Отметим лишь, что не стоит искать в приведенных ниже примерах какой-либо смысл, так как они предназначены только для демонстрации механизма вложенности структур (и объединений).

```

;+-----+
;| Программа: prg02_01.asm. Демонстрация описания и использования структур. |
;+-----+
element      struc

```

```

INN          dd      0           : ИНН
name         db      30 dup ( ' ' ) : Ф.И.О.
y_birthday   dw      1962        : год рождения
m_birthday   db      05         : месяц рождения
d_birthday   db      30         : день рождения
nationality  db      20         : национальность
ends
.data
sl           element <>
.code

:....
mov         al, sl.m_birthday
:....

```

Информацию о дате рождения можно оформить в виде отдельной структуры, вложенной в текущую структуру, так, как это сделано в программе ниже:

```

:-----+
:| Программа: prg02_02.asm. Демонстрация вложения одной структуры в другую. |
:-----+
Element      struc
INN          dd      0           : ИНН
fio          db      30 dup ( ' ' ) : Ф.И.О.
struc
y_birthday   dw      1962        : год рождения
m_birthday   db      05         : месяц рождения
d_birthday   db      30         : день рождения
ends
nationality  db      20         : национальность
ends
.data
sl           element <>
.code

:....
mov         al, sl.m_birthday
:....

```

Синтаксис ассемблера допускает вынести описание вложенной структуры за пределы текущей структуры. При этом можно инициализировать необходимые поля структуры birthday внутри отдельного экземпляра структуры element. Но работает такое описание только в режиме IDEAL, что и продемонстрировано в программе ниже.

```

:-----+
:| Программа: prg02_03.asm. Демонстрация вложения одной структуры в другую. /
:-----+
Birthday     struc
y_birthday   dw      1962        : год рождения
m_birthday   db      05         : месяц рождения
d_birthday   db      30         : день рождения
ends
element      struc
INN          dd      0           : ИНН
Birthday     struc {m_birthday=06, d_birthday=21}
fio          db      30 dup ( ' ' ) : Ф.И.О.
nationality  db      20         : национальность
ends
.data
sl           element <,<>>
.code

ideal
mov         al, sl.m_birthday
masm

:....

```

В описание структур допускается вкладывать не только описания других структур, но и объединений так, как это показано в следующей программе.

```
+-----+
:| Программа: prg02_04.asm. Демонстрация взаимного вложения |
:| объединений и структур.                                     |
+-----+
Element          struc
INN              dd      0           ; ИНН
Fio              db      30 dup ( ' ' ) ; Ф.И.О.
union
struc
y_birthday_1    dw      1962        ; год рождения
m_birthday_1    db      06         ; месяц рождения
d_birthday_1    db      30         ; день рождения
ends
struc
d_birthday_2    db      ?          ; день рождения
m_birthday_2    db      ?          ; месяц рождения
y_birthday_2    dw      ?          ; год рождения
ends
ends ;конец объединения
nationality      db      20         ; национальность
ends
.data
sl              element <>
.code

:...
mov     al, sl.m_birthday_1
mov     si, sl.m_birthday_2.0fffh
mov     al, si.m_birthday_2
:...
```

Массивы структур — таблицы

Обычной практикой является размещение одинаковых по содержанию экземпляров структур в одном месте, последовательно — друг за другом. В результате образуется логическая структура данных, называемая *таблицей*. Каждый экземпляр структуры, входящий в таблицу, называется *элементом таблицы*.

Как физическая структура данных таблица представляет собой линейную последовательность ячеек памяти, число которых определяется количеством и размером полей каждого элемента, а также числом элементов таблицы.

Над таблицей можно определить следующие операции:

- включение нового элемента путем расширения таблицы или его вставки на свободное место;
- поиск элемента для последующей его обработки;
- исключение элемента из таблицы.

Скорость доступа к элементам таблицы при выполнении этих операций зависит от двух факторов — способа организации поиска нужного элемента и размера таблицы.

Поиск в таблице

Перечисленные выше операции с таблицей предполагают, что для однозначного доступа к ее элементам последние должны содержать один или более признаков, отличающих их от других элементов этой таблицы. С этой целью в логическую

структуру элемента включается по крайней мере одно поле — *ключ*, содержимое которого уникально для любого из элементов, входящих в таблицу. В общем случае работа с таблицами сводится к методам решения задачи поиска нужного элемента таблицы по заданному значению ключа. Результат решения — получение адреса искомого элемента или заключение о его отсутствии. Для локализации элемента таблицы необходимо знать два адресных компонента — адрес таблицы и смещение нужного элемента относительно ее начала. Адрес таблицы получить несложно, так как он является адресом ее первого элемента. Что же касается определения второго компонента адреса, то для этого существует ряд методов, рассмотрением которых мы сейчас и займемся.

В ряде случаев наряду с ключевым полем определяют еще одно служебное поле — *поле текущего состояния элемента*, которое может принимать три значения:

- *элемент свободен* — после инициализации таблицы элемент ни разу не использовался для хранения полезной информации;
- *элемент используется* для хранения полезной информации;
- *элемент удален* — после инициализации таблицы элемент хотя бы один раз использовался для хранения полезной информации, после чего был освобожден.

Целесообразность использования поля текущего состояния элемента будет видна из дальнейших рассуждений.

С точки зрения организации поиска таблицы делятся на:

- неупорядоченные;
- древовидные;
- упорядоченные;
- с вычисляемыми входами (хэш-таблицы).

Неупорядоченные таблицы

Это простейший вид организации таблиц. Элементы располагаются непосредственно друг за другом, без пропусков. Процесс поиска нужного элемента выполняется очень просто — последовательным перебором элементов таблицы начиная с первого. При этом анализируется ключевое поле и в случае удовлетворения его условию поиска нужный элемент считается обнаруженным. Такой способ поиска называется *последовательным* или *линейным*. Для добавления нового элемента необходимо найти первую свободную позицию, после чего выполнить операцию добавления. Процесс удаления можно реализовать следующим образом. После локализации нужного элемента его поле текущего состояния нужно установить в состояние «элемент удален». Тогда процесс поиска места для добавления нового элемента будет заключаться в нахождении такого элемента, у которого поле текущего состояния элемента имеет значение «элемент удален» или «элемент свободен». Можно еще более оптимизировать работу с неупорядоченной таблицей путем добавления в начало таблицы (или перед ней) дескриптора, поля которого содержали бы информацию о размере таблицы, положении первого свободного элемента и т. д.

Обычно неупорядоченные таблицы используют в качестве временных для хранения небольшого количества элементов (до 20). Для таблиц большего размера такая организация поиска неэффективна, поэтому для них следует применять другие способы локализации нужного элемента.

Приемы работы с неупорядоченной таблицей продемонстрируем на примере следующей задачи. Требуется прочесть содержимое файла `maket.txt` и обо всех десятичных и шестнадцатеричных числовых константах собрать следующую информацию: значение константы и номер строки, в которой данная константа встретилась. После чего нужно вывести информацию о десятичных константах на экран. Для того чтобы избежать возни с несущественными деталями, введем следующие ограничения:

- содержимое файла — идентификаторы и константы, разделенные не более чем одним пробелом; перед первым и последним идентификатором или константой в строке также следует по одному пробелу;
 - для удобства преобразования предполагаем, что длина и количество строк файла `maket.txt` находятся в диапазоне 0...99, а общая длина файла — не более 240 байтов.
- Поле текущего состояния представляет собой запись, битовые поля которой означают следующее:
- биты 0 и 1 — состояние элемента: 00 — свободен; 01 — используется; 10 — удален;
 - бит 2 — тип константы: 0 — десятичная константа; 1 — шестнадцатеричная константа;
 - бит 3 — 0 — не последний элемент таблицы; 1 — последний элемент таблицы.

```

+-----+
:| Программа: prg02_05.asm. Демонстрация работы с неупорядоченной таблицей. |
+-----+
:| Вход: файл maket.txt с идентификаторами, среди которых присутствуют |
:| десятичные и шестнадцатеричные константы. |
+-----+
:| Выход: вывод информации о десятичных константах на экран. |
+-----+

state_tab      struc
last_off       dw      0           ; адрес первого байта за концом таблицы
elem_free      dw      0           ; адрес первого свободного элемента
                                   ; (0ffffh — все занято)

ends
constant       struc
state          db      0           ; поле состояния элемента таблицы
               db      02dh        ; форматирование вывода на экран
key            db      10 dup (' ') ; ключ, он же значение константы
               db      02dh        ; форматирование вывода на экран
line           db      2 dup (' ') ; строка файла, в которой
                                   ; встретилась константа
end_line       db      0dh, 0ah, '$' ; для удобства вывода на экран
ends

.data
s_tab          state_tab <>
               constant 19 dup (<>)
               constant <B,>           ; последний элемент — бит 3-1
end_tab = $ - tab

filename       db      'maket.txt', 0
handle         dw      0           ; файловый манипулятор
buf            db      240 dup (' ')
xlat_tab       db      0dh dup (00), 0dh ; признак конца строки

```

```

        db      '0'-0eh dup (0)
        db      ':' - '0' + 1 dup ('0') : признак цифры 0...9
        db      'H' - ':' dup (0), '' : признак буквы 'H'
        db      'h' - 'H' dup (0), 'h' : признак буквы 'h'
        db      0ffh - 'h' dup (00)
cur_line
.code
:----- открываем файл
        mov     ah, 3dh
        lea     dx, filename
        int     21h
        jc      exit : ошибка (cf=1)
        mov     mov     handle, ax
:----- читаем файл
        mov     ah, 3fh : функция установки указателя
        mov     mov     bx, handle
        mov     cx, 240 : читаем максимум 240 байтов
        lea     dx, buf
        int     21h
        jc      exit
        mov     cx, ax : фактически считано байтов в cx
:----- инициализируем дескриптор таблицы s_tab
        lea     si, tab : адрес таблицы в si
        mov     s_tab.elem_free, si : первый элемент таблицы свободен
        add     si, end_tab
        mov     s_tab.last_off, si : адрес за концом таблицы
        lea     bx, xlat_tab
        lea     di, buf
:----- сканируем до первого пробела
        push    ds
        pop     es
        mov     al, ''
cycl1:   repne   scasb : сканирование до первого пробела
        jcxz    displ : таблица заполнена
        push    cx
:----- классифицируем символ после пробела (команда XLAT)
        mov     al, [di]
        xlat
        cmp     al, '0' : первый символ за пробелом - 0
        je      m1
        cmp     al, 0dh : первый символ за пробелом - 0dh
        je      m2
:----- все остальное либо идентификаторы, либо неверно записанные числа
        pop     cx
        jmp     cycl1
:----- первый символ после пробела - 0...9
m1:      mov     si, di : откуда пересылать
        mov     al, ''
        push    di
        repne   scasb : сканирование до первого пробела
        mov     cx, di
        dec     cx
        sub     cx, si : сколько пересылать
        lea     di, tab
        cmp     s_tab.elem_free, 0ffffh : есть свободные элементы?
        je      displ : свободных элементов нет
        mov     di, s_tab.elem_free : адрес первого свободного элемента
        push    di
        lea     di, [di].key
        rep     movsb : пересылаем в элемент таблицы
        dec     di
:----- Какого типа эта константа?
        cmp     byte ptr [di], 'h'
        pop     di
        je      m4

```

```

        and    [di].state, 0fbh ; десятичная константа
        jmp    $ + 5
m4:      or     [di].state, 100b ; шестнадцатеричная константа
        mov    al, cur_line     ; текущий номер строки в a1
        aam                     ; преобразуем в символьное представление
        or     ah, 030h
        mov    [di].line, ah
        or     al, 030h
        mov    [di+1].line, al ; и в элемент таблицы
        or     [di].state, 1b   ; помечаем элемент как используемый
;----- теперь нужно поместить в поле s_tab.elem_free
;      адрес нового свободного элемента
m5:      cmp    di, s_tab.last_off
        ja     displ
        add    di, type constant ; к следующему элементу
        test   [di].state, 1b
        jnz    m5               ; используется - к следующему элементу
        mov    s_tab.elem_free, di
        pop    di
        pop    cx
m2:      jmp    csc11
        inc    cur_line         ; увеличить номер строки
        jmp    csc11
;----- отображение на экране элементов таблицы
displ:   lea    di, tab
m6:      test   [di].state, 100b
        jnz    m7
        mov    ah, 9            ; функция вывода строки
        mov    dx, di
        int    21h
m7:      add    di, type constant
        cmp    di, s_tab.last_off
        jb     m6
        ;...

```

В этой программе показаны основные приемы работы с неупорядоченной таблицей. Усложнить данный листинг можно, например, дополнив набор сведений о каждой константе информацией о ее длине. Далее можно попрактиковаться в этом вопросе, поставив себе задачу удалить из таблицы информацию о нулевых константах (обоих типов — 0 и 0h) и вывести окончательное содержимое таблицы на экран. В качестве ключа для доступа к ячейке таблицы годится значение самой константы (размером 10 байтов).

Обратите внимание на еще два момента, отраженные в этой программе.

- Использование команды XLAT для классификации символов в качестве цифр и остальных символов (букв). Среди других кодов в таблице перекодировки особо выделен байт со значением 0dh, который является первым байтом в паре 0d0ah. Как вы помните, эта пара вставляется редакторами ASCII-текстов для обозначения конца строки и перехода на другую строку.
- Организация таблицы. Ей предшествует четырехбайтовый префикс, содержащий информацию о конце таблицы и первом свободном элементе таблицы. В целях оптимизации область памяти для буфера buf и таблицы перекодировки лучше выделять динамически и после анализа файла удалять.

Древовидные таблицы

Древовидные таблицы являются несколько улучшенным вариантом неупорядоченных таблиц. Новые записи в древовидную таблицу по-прежнему поступают

неупорядоченно. Но на сей раз место, занимаемое этими записями среди других записей в таблице, определяется значением ключевого поля. Принцип здесь такой же, как и при сортировке с помощью дерева (см. выше).

В общем случае каждая запись в древовидной таблице сопровождается двумя указателями (рис. 2.7): один указатель хранит адрес записи с меньшим значением ключа (адрес узла-предка), второй — с большим (адрес узла-потомка).

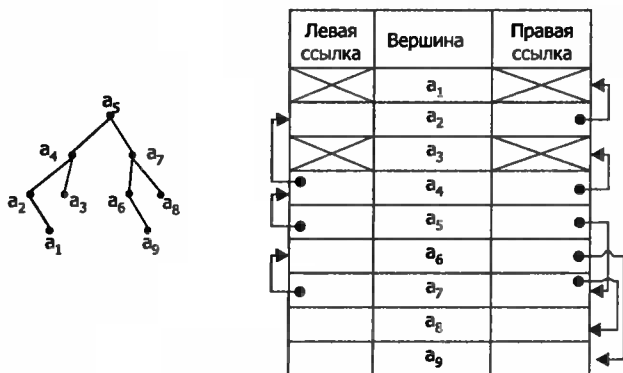


Рис. 2.7. Древовидная организация таблицы

Процесс добавления нового элемента в древовидную таблицу производится следующим образом. Значение ключа нового элемента сравнивается со значением ключа первого элемента дерева. По результатам сравнения ключей новый элемент «поступает» в левое или правое поддерево первого элемента дерева. Далее процесс сравнения и поступления в левое или правое поддерево очередного узла продолжается аналогичным образом и до тех пор, пока не будет найден пустой элемент в нужном направлении просмотра. Новая запись помещается в данную пустую позицию, при этом настраивается указатель с адресом узла-предка.

Преимущества древовидной организации таблицы проявляются на этапе поиска нужного элемента. Сам процесс подобен действиям на этапе вставки нового элемента в таблицу, являясь при этом в отличие от неупорядоченного поиска целенаправленным. Поиск прекращается либо при совпадении ключевого поля, либо при обнаружении пустого указателя, что означает отсутствие нужного элемента в таблице.

Самое сложное в использовании древовидных таблиц — реализация процесса удаления элементов, так как при этом необходимо производить их переупорядочивание.

Реализовать таблицу древовидной организации можно как с использованием полей указателей (см. выше), так и без них. Если отказаться от полей указателей, то таблица может представлять собой массив из n структур, которые пронумерованы от 0 до $n - 1$. Тогда структура с ключом, соответствующим корню дерева, должна находиться на месте структуры с номером $m = (n - 1)/2$ (целая часть от деления). Соответственно, левое поддерево будет располагаться в левом подмассиве (элементы $0 \dots m - 1$), а правое — в правой части массива (элементы $m + 1 \dots n - 1$). Корень левого поддерева — элемент $m'_l = (m - 1)/2$, корень правого поддерева —

элемент $m'_r = m + 1 + (n - 1)/2$ и т. д. Этот способ экономичнее с точки зрения расходования памяти, но сложнее алгоритмически. Предполагается, что соответствующее таблице дерево сбалансировано и максимальное количество элементов в таблице известно.

Наглядный пример организации древовидной таблицы — расстановка слов в лексикографическом порядке. Эта операция выполняется с помощью лексикографических деревьев, понятие о которых приведено ниже — в разделе, посвященном деревьям. Там же имеется соответствующий пример.

Упорядоченные таблицы

Работа с таблицей оказывается более эффективной, если элементы в ней некоторым образом упорядочены. Наиболее часто таблица сортируется по значению ключевого поля. Но не исключено и использование другого принципа упорядочивания, к примеру по частоте обращения к элементам таблицы и т. п. Упорядочивание таблицы и последующий поиск в ней производится методами, рассмотренными выше. Порядок выполнения этих и других операций с отсортированной по значению ключа таблицей рассмотрим на следующем примере.

Пусть необходимо заполнить элементы таблицы информацией о 10 словах, вводимых с клавиатуры. Длина слов — не более 20 символов. Структура элемента таблицы следующая: поле с количеством символов в слове; поле с самим словом. После ввода информации о словах необходимо упорядочить элементы таблицы по признаку длины слов. Затем вывести на экран элемент таблицы, содержащий первое из слов длиной 5 символов, удалить этот элемент из таблицы и вставить в нее новое слово, введенное с клавиатуры.

При такой постановке задачи нам придется решить весь комплекс проблем, возникающих при работе с упорядоченными таблицами.

```

+-----+
: | Программа: prg02_06.asm. Демонстрация работы с упорядоченной таблицей. |
+-----+
: | Вход: 10 слов, вводимых с клавиатуры. Длина слов — не более 20 символов. |
+-----+
: | Выход: вывод на экран элемента таблицы, содержащего первое из слов |
: | длиной 5 символов, удаление этого элемента из таблицы и вставка в нее |
: | нового слова, введенного с клавиатуры. |
+-----+
element_tab struc
len          db      0           ; длина слова
simv_id      db      20 dup (20h) ; само слово
ends
buf_0ah      struc
len_buf      db      24           ; длина buf_0ah
len_in       db      0           ; действительная длина введенного слова
; (без учета 0dh)
buf_in       db      21 dup (20h) ; буфер для ввода (с учетом 0dh)
ends
+-----+
: | Макрокоманда: s_movsb. Пересылка строки. |
: | Вход: in_str — откуда, out_str — куда. len_movs — сколько пересылать. |
+-----+
s_movsb      macro out_str, in_str, len_movs
ifdif <cx>, <len_movs>
    push     cx
endif

```

```

ifdif <si>, <in_str>
    push    si
endif
ifdif <di>, <out_str>
    push    di
endif
ifdif <si>, <in_str>
    lea     si, in_str      ; откуда пересылать
endif
ifdif <di>, <out_str>
    lea     di, out_str     ; куда пересылать
endif
ifdif <cx>, <len_movs>
    mov     cx, len_movs    ; сколько пересылать
endif
    rep     movsb           ; пересылаем строку
ifdif <di>, <out_str>
    pop     di
endif
ifdif <si>, <in_str>
    pop     si
endif
ifdif <cx>, <len_movs>
    pop     cx
endif
endm
.data
tab
    element_tab 10 dup (<>)
len_tab = $ - tab
buf_0ah <>
key      db 5
prev     element_tab <>      ; предыдущий элемент
x         element_tab <>      ; временная переменная для сортировки
.code

    lea     di, tab
    lea     si, buf.buf_in
    mov     cx, 10
    lea     dx, buf
    mov     ah, 0ah
    push    ds
    pop     es
;----- вводим слова с клавиатуры в буфер buf
m1:
    ....
    int     21h             ; сохраняем регистры
    mov     cl, buf.len_in
    mov     [di].len.cl      ; длина слова -> tab.length
    add     di, simv_id
    rep     movsb           ; пересылка слова в элемент таблицы
    ....
    add     di, type element_tab ; восстанавливаем регистры
    loop    m1
;----- упорядочиваем таблицу методом пузырьковой сортировки
n = 10
mov     cx, n - 1
mov     si, 1
;----- внешний цикл - по i
@@cyc11:
    push    cx
    mov     cx, n
    sub     cx, si          ; количество повторов внутреннего цикла
    push    si              ; временно сохраним i - теперь j=n
    mov     si, n - 1
;----- цикл по j с декрементом n-i раз
@@cyc12:
    push    si
    mov     ax, type element_tab

```

```

mul     si
mov     si, ax
mov     di, ax
sub     di, type element_tab ; в di адрес предыдущей записи
mov     al, [di].len
cmp     [si].len, al        ; сравниваем последующий с предыдущим
ja      @m1
;----- обмениваем ( после mas[j]=x)
s_movsb x, [di], <type element_tab> ; x=mas[j-1]
s_movsb [di], [si], <type element_tab> ; mas[j-1]=mas[j]
s_movsb [si], x, <type element_tab>
@m1:    pop     si
dec     si
loop    @@cyc12
pop     si
inc     si
pop     cx
dec     cx
jz      m2
jmp     @@cyc11
;----- ищем элемент путем двоичного поиска
m2:     mov     si, 0          ; индексы первого и последнего элементов
mov     di, n - 1            ; последней просматриваемой части
                                ; последовательности
cont_search: cmp     si, di    ; проверка на окончание (неуспешное): si>di
ja      exit
;----- получим центральный индекс
mov     bx, si
add     bx, di
shr     bx, 1                ; делим на 2
push    bx
mov     ax, type element_tab
mul     bx
mov     bx, ax
mov     al, key              ; искомое значение в ax
cmp     [bx].len, al         ; сравниваем с искомым значением
je      @m4                  ; искомое значение найдено
ja      @m3                  ; [bx].len>k
pop     bx                   ; здесь [bx].len<k
mov     si, bx
inc     si
jmp     cont_search
@m3:    pop     bx            ; 1
mov     di, bx
dec     di
jmp     cont_search
@m4:    mov     ax, type element_tab
mul     si
mov     si, ax               ; конец поиска - в si адрес элемента таблицы
                                ; со словом длиной 5 байтов
;----- выводим его на экран
mov     al, [si].len
xor     cx, cx
mov     cl, al               ; длина для movsb
aam
or      ax, 03030h           ; в ax длина в символьном виде
mov     buf.len_buf, ah
mov     buf.len_in, al
push    si                   ; сохраним указатель на эту запись
add     si, simv_id
lea     di, buf.buf_in
rep     movsb
mov     byte ptr [di], '$' ;конец строки для вывода
lea     dx, buf

```

```

        mov     ah, 09h
        int     21h ; выводим
;----- удаляем запись
        pop     si           ; восстановим указатель на запись
        mov     di, si
        add     si, type element_tab
        mov     cx, len_tab
        sub     cx, si       ; в cx сколько пересылать
        rep     movsb
;----- обнуляем последнюю запись
        xor     al, al
        mov     cx, type element_tab
        rep     stosb
insert: ;----- вводим слово с клавиатуры
        lea     dx, buf
        mov     ah, 0ah
        int     21h
;----- с помощью линейного поиска ищем место вставки,
; в котором выполняется условие buf.len_in <= [si].len
        lea     si, tab
        mov     al, buf.len_in
@@m5:   cmp     al, [si].len
        jbe     @@m6
        add     si, type element_tab
        jmp     @@m5
@@m6:   push    si           ; запоминаем позицию вставки
;----- раздвигаем таблицу, последний элемент теряется
        add     si, type element_tab
        mov     cx, len_tab
        sub     cx, si       ; сколько пересылать
        std
        lea     si, tab
        add     si, len_tab
        mov     di, si
        sub     si, type element_tab
        rep     movsb
        cld
;----- формируем и вставляем новый элемент
        pop     di           ; восстанавливаем позицию вставки
        push    di
        xor     al, al
        mov     cx, type element_tab
        rep     stosb       ; обнуляем место вставки
        pop     di
        lea     si, buf.buf_in
        mov     cl, buf.len_in
        mov     [di].len, cl
        add     di, simv_id
        rep     movsb       ; вставляем
        ...

```

Таблицы с вычисляемыми входами

Ранее мы отмечали, что скорость доступа к элементам таблицы зависит от двух факторов — способа организации поиска нужного элемента и размера таблицы. Для маленьких таблиц любой метод доступа будет работать быстро. С ростом размера таблицы способ организации доступа приходится выбирать прежде всего исходя из критерия скорости локализации нужного элемента таблицы. Элементы таблицы отличаются друг от друга уникальным значением ключевого поля. При этом ключевыми могут являться не только одно, но и несколько полей элемента таблицы. Ключ, в том числе и символьный, в памяти представляется последова-

тельностью байтов. Исходя из того что ключ уникален, соответствующая двоичная последовательность также будет уникальной. А нельзя ли использовать это уникальное значение ключа для вычисления адреса местоположения элемента в таблице? Оказывается, можно, а в ряде приложений это оказывается очень эффективно, так как в идеальном случае доступ к нужному элементу таблицы осуществляется всего за одно обращение к памяти. С другой стороны, на практике часто возникает необходимость размещения элементов таблицы с достаточно большим диапазоном возможных значений ключевого поля, в то время как программа реально использует лишь небольшое подмножество значений этих ключей. Например, значение ключевого поля может быть в диапазоне 0...3999, но задача постоянно востребует не более 50 значений. В этом случае крайне неэффективным было бы резервировать память для таблицы размером в 4000 элементов, а реально использовать чуть больше 1% отведенной для нее памяти. Гораздо лучше иметь возможность воспользоваться некоторой процедурой, отображающей задействованное пространство ключей на таблицу размером, близким к значению 50. Большинство подобных задач решается при помощи методики, называемой хэшированием. Ее основу составляют различные алгоритмы отображения значения ключа в значение адреса размещения элемента в таблице. Непосредственное преобразование ключа в адрес производится с помощью *функций расстановки (хэш-функций)*. Адреса, получаемые из ключевых слов с помощью *хэш-функций*, называются *хэш-адресами*. Таблицы, для работы с которыми используются методы хэширования, называются таблицами с *вычисляемыми входами*, *хэш-таблицами* или таблицами с прямым доступом.

Основополагающую идею хэширования можно пояснить на следующем примере. Предположим, необходимо подсчитать, сколько раз в тексте встречаются слова, первый символ которых — одна из букв английского алфавита (или русского — это не имеет значения, можно в качестве объекта подсчета использовать любой символ из кодовой таблицы). Для этого в памяти нужно организовать таблицу, количество элементов в которой будет соответствовать количеству букв в алфавите. Далее следует составить программу, в которой текст будет анализироваться с помощью цепочечных команд. При этом нас интересуют разделяющие слова пробелы и первые буквы самих слов. Так как символы букв имеют определенное двоичное значение, то на его основе вычисляется адрес в таблице, по которому располагается элемент, в минимальном варианте состоящий из одного поля. В этом поле ведется подсчет количества слов в тексте, начинающихся с данной буквы. В следующей программе с клавиатуры вводится 20 слов (длиной не более 10 символов), производится подсчет английских слов, начинающихся с определенной строчной буквы, и результат подсчета выводится на экран. Хэш-функция (функция расстановки) имеет вид: $A = (C - 97) \cdot L$, где A — адрес в таблице, полученный на основе двоичного значения символа C ; L — длина элемента таблицы (для нашей задачи $L = 1$); 97 — десятичное смещение в кодовой таблице строчного символа «a» английского алфавита.

```

+-----+
:| Программа: prg02_07.asm. Подсчет количества слов, начинающихся
:| с определенной строчной буквы
+-----+
:|
:| Вход: ввод с клавиатуры 20 слов (длиной не более 10 символов).
:|

```

```

:-----+-----
: | Выход: вывод результата подсчета на экран. |
:-----+-----
buf_0ah      struc
len_buf      db      11          : длина buf_in
len_in       db      0           : действительная длина введенного слова
                                : (без учета 0dh)
buf_in       db      11 dup (20h) : буфер для ввода (с учетом 0dh)
ends
.data
tab          db      26 dup (0)
buf          buf_0ah <
            db      0dh, 0ah, '$' : для вывода функцией 09h (int 21h)

.code
:----- вводим слова с клавиатуры
mov     cx, 20
lea     dx, buf
mov     ah, 0ah
m1:     int     21h
:----- анализируем первую букву введенного слова -
:         вычисляем хэш-функцию: A=C*1-97
mov     bl, buf.buf_in
sub     bl, 97
inc     [bx]
loop    m1
:----- выводим результат подсчета на экран
push    ds
pop     es
xor     al, al
lea     di, buf
mov     cx, type buf_0ah
rep     stosb      : чистим буфер buf
mov     cx, 26
:----- символ в buf.buf_in
lea     dx, buf
mov     bl, 97
m2:     push    bx
mov     buf.buf_in, bl
:----- опять вычисляем хэш-функцию: A=C*1-97 и преобразуем
:         "количество" в символьный вид
sub     bl, 97
mov     al, [bx]
aam
or      ax, 03030h      : в ax длина в символьном виде
mov     buf.len_in, al
mov     buf.len_buf, ah
:----- теперь выводим:
mov     ah, 09h
int     21h
pop     bx
inc     bl
loop    m2
:....

```

Таким образом, относительно сложная с первого взгляда задача очень просто реализуется с помощью методов хэширования. При этом обеспечивается высокая скорость выполнения операций доступа к элементам таблицы. Это обусловлено тем, что адреса, по которым располагаются элементы таблицы, являются результатами вычислений простых арифметических функций от содержимого соответствующих ключевых слов.

Перечислим области, где методы хэширования оказываются особенно эффективными.

- Разработка компиляторов, программ обработки текстов, пользовательских интерфейсов и т. п. В частности, компиляторы значительную часть времени обработки исходного текста программы затрачивают на работу с различными таблицами — операций, идентификаторов, констант и т. д. Правильная организация работы компилятора с информацией в этих таблицах означает значительное увеличение скорости создания объектного модуля, может быть, даже не на один порядок выше. Кстати, другие системные программы — редакторы связей и загрузчики — также активно работают со своими внутренними таблицами.
- Системы управления базами данных. Здесь особенный интерес представляют алгоритмы выполнения операций поиска по многим ключам, которые также основаны на методе хэширования.
- Разработка криптографических систем [30].
- Поиск по соответствию. Методы хэширования можно применять в системах распознавания образов, когда идентификация элемента в таблице осуществляется на основе анализа ряда признаков, сопровождающих объект поиска, а не полного соответствия заданному ключу. Если рассматривать эту возможность в контексте задач системного программирования, то ее можно использовать для исправления ошибок операторов при вводе информации в виде ключевых слов. Подробная информация о поиске по соответствию приведена в литературе [32].

Но на практике не все так гладко и оптимистично. Для эффективной и безотказной работы метода хэширования необходимо очень тщательно подходить как к изучению задачи на этапе ее постановки, так и к возможности использования конкретного алгоритма хэширования в контексте этой задачи. Так, на стадии изучения постановки задачи, в которой для доступа к табличным данным планируется использовать хэширование, требуется проводить тщательные исследования по направлениям: диапазон допустимых ключей, максимальное количество элементов в таблице, вероятность возникновения коллизий и т. д. При этом нужно знать как общие проблемы метода хэширования, так и нюансы конкретных алгоритмов хэширования. Одну из таких проблем рассмотрим на примере задачи.

Пусть необходимо подсчитать количество двухсимвольных английских слов в некотором тексте. В качестве хэш-функции для вычисления адреса можно предложить функцию подсчета суммы двух символов, умноженной на длину элемента таблицы: $A = (C1 + C2) \cdot L - 97$, где A — адрес в таблице, полученный на основе суммы двоичных значений символов $C1$ и $C2$; L — длина элемента таблицы; 97 — десятичное смещение в кодовой таблице строчного символа «a» английского алфавита. Проведем простые расчеты. Сумма двоичных значений двух символов «a» равна $97 + 97 = 194$, сумма двоичных значений двух символов «z» равна $122 + 122 = 244$. Если организовать хэш-таблицу, как в предыдущем случае, то получится, что в ней должно быть всего 50 элементов, чего явно недостаточно. Более того, для сочетаний типа «ab» и «ba» хэш-сумма соответствует одному числовому значению. В случае когда функция хэширования вычисляет одинаковый адрес для двух и более различных объектов, говорят, что произошла коллизия, или столкновение. Исправить положение можно введением допущений и ограничений, вплоть до замены используемой хэш-функции.

Программист может либо применить один из известных алгоритмов хэширования (что, по сути, означает использование определенной хэш-функции), либо изобрести свой алгоритм, наиболее точно отображающий специфику конкретной задачи. При этом необходимо понимать, что разработка хэш-функции происходит в два этапа.

1. Выбор способа перевода ключевых слов в числовую форму.
2. Выбор алгоритма преобразования числовых значений в набор хэш-адресов.

Выбор способа перевода ключевых слов в числовую форму

Вся информация, вводимая в компьютер, кодируется в соответствии с одной из систем кодирования (таблиц кодировки). В большинстве таких систем символы (цифры, буквы, служебные знаки) представляются однобайтовыми двоичными числами. В последних версиях Windows (NT/2000/XP) используется система кодировки Unicode, в которой символы в частном и самом распространенном случае представляются в виде двухбайтовых величин. Как правило, ключевые поля элементов таблиц — строки символов, которые могут состоять из букв, цифр и других символов, например пробела, знаков препинания и т. д. Совокупность этих символов (в общем случае подмножества всех символов, описываемых таблицей кодировки) составляет алфавит. В этом алфавите каждому символу C поставлен в соответствие определенный номер $N_C = \{0, 1, 2, 3, \dots, n-1\}$. Тогда строку $s = d_i d_{i-1} \dots d_1 d_0$, образованную из символов этого алфавита, можно рассматривать как представление некоторого целого числа в системе с основанием n . Значение этого числа можно вычислить по формуле:

$$A = \sum_{i=0}^l d_i n_i.$$

Например, рассмотрим алфавит, состоящий из первых пяти английских букв и пробела: $\{a, b, c, d, e, ' '\}$. Этим символам поставим в соответствие номера: $a = 0$, $b = 1$, $c = 2$, $d = 3$, $e = 4$, $' ' = 5$. Заметим, что основание этой системы $n = 6$. В таких условиях строке «cba dea» по данной формуле соответствует ее числовое значение:

$$2 \cdot 6^0 + 1 \cdot 6^1 + 0 \cdot 6^2 + 5 \cdot 6^3 + 3 \cdot 6^4 + 4 \cdot 6^5 + 0 \cdot 6^6 = 36\,080.$$

Таким способом, расширив предварительно алфавит, можно переводить в числовую форму не только слова, но и целые словосочетания.

Исходя из условий задачи, можно выбрать иной способ перевода ключевых слов в числовую форму. Главное здесь — постараться получить равномерное (или близкое к нему) распределение числовых величин. На втором этапе используется один из выработанных на первом этапе алгоритмов преобразования числовых значений в набор хэш-адресов. Качество получаемых результатов в основном определяется алгоритмом хэширования и соответствующей этому алгоритму хэш-функцией.

Выбор алгоритма преобразования числовых значений в набор хэш-адресов

Алгоритмы хэширования делятся на два типа: открытые и закрытые. Алгоритмы открытого хэширования предполагают, что объем памяти для структур хэширова-

ния не ограничен. Пример такого хэширования — *метод цепочек* — мы рассмотрим ниже. Алгоритмы закрытого хэширования, наоборот, предполагают поиск местоположения элемента в ограниченном пространстве, отведенном под хэш-таблицу. Перед обсуждением наиболее распространенных алгоритмов хэширования отметим их тесную связь с алгоритмами генерации псевдослучайных чисел. Этому есть следующие причины. Во-первых, при заданном исходном значении последовательность псевдослучайных чисел детерминирована (полностью предсказуема), так как всегда производится по рекурсивной схеме. Согласно этой схеме результаты, полученные на предыдущих шагах, используются на последующих в качестве исходных параметров. По этой причине большинство алгоритмов получения хэш-адресов, задействующих в качестве исходных параметров ключевые слова, в основе своей являются алгоритмами получения последовательностей псевдослучайных чисел. Во-вторых, разрешение коллизий в ряде случаев также осуществляется с помощью алгоритмов генерации случайных чисел.

Отличие алгоритмов хэширования от алгоритмов генерации псевдослучайных чисел состоит в снижении требований к степени корреляции (взаимозависимости) соседних генерируемых псевдослучайных значений. Для алгоритмов хэширования качество псевдослучайных чисел особой роли не играет, оно влияет лишь на количество попыток размещения элемента в хэш-таблице и соответственно этим во многом определяет скорость работы конкретного алгоритма хэширования.

Наиболее известные алгоритмы закрытого хэширования основаны на следующих методах [32]:

- деления;
- умножения;
- извлечения битов;
- середины квадрата;
- сегментации;
- перехода к новому основанию;
- алгебраического кодирования;
- вычисления значения CRC (см. главу 9).

Далее мы рассмотрим только первые четыре метода. Для остальных отметим лишь, что их используют либо в случае значительной длины ключевых слов, либо когда ключ состоит из нескольких слов. Информацию об этих методах можно почерпнуть в литературе [32].

Рассмотрение методов хэширования будет произведено на примере одной задачи. Это позволит лучше понять их особенности, преимущества, недостатки и возможные ограничения.

Необходимо разработать программу — фрагмент компилятора, которая собирает информацию об идентификаторах программы. Предположим, что в программе может встретиться не более M различных имен. Длину возможных имен ограничим восемью символами. В качестве ключа используются символы идентификатора, какие и сколько — будем уточнять для каждого из методов. Элемент таблицы состоит из 10 байтов: 1 байт для признаков, 1 байт для хранения длины идентификатора и 8 байтов для хранения символов самого идентификатора.

Метод деления

Этот простой алгоритм закрытого хэширования основан на использовании остатка деления значения ключа K на число, равное числу элементов таблицы M или близкое к нему:

$$A(K) = K \bmod M$$

В результате деления образуется целый остаток $A(K)$, который и принимается за индекс блока в таблице. Чтобы получить конечный адрес в памяти, нужно полученный индекс умножить на размер элемента в таблице. Для уменьшения коллизий необходимо соблюдать ряд условий:

- Значение M выбирается равным простому числу.
- Значение M не должно являться степенью основания, по которому производится перевод ключей в числовую форму. Так, для алфавита, состоящего из первых пяти английских букв и пробела {a, b, c, d, e, ' } (см. пример выше), основание системы равно 6. Исходя из этого, число элементов таблицы M не должно быть степенью 6.

Важно отметить случай, когда число элементов таблицы M является степенью основания машинной системы счисления (для процессора Intel это 2). Тогда операция деления (достаточно медленная) заменяется на несколько операций сдвига (очень быстрых). При этом не нужно забывать об обозначенном чуть выше условии, когда значение M не должно являться степенью основания, по которому производится перевод ключей в числовую форму.

Уточним постановку обозначенной выше задачи для демонстрации данного метода. Положим: $M = 64$; в качестве ключа K будем использовать сумму кодов всех символов идентификатора. Идентификаторы вводим с клавиатуры, конец ввода — по нажатию клавиши пробела. Первый байт элемента таблицы — байт признаков, нулевой бит которого означает занятость элемента (единичное значение). Коллизии пока разрешать не будем, сделаем это при рассмотрении соответствующих методов повторного хэширования. При возникновении коллизий будем обозначать их выводом на экран идентификаторов, послуживших причиной этого.

```

+-----+
: | Программа: prg02_08.asm. Демонстрация метода деления на примере
: | задачи сбора информации об идентификаторах программы.
+-----+
: | Вход: ввод с клавиатуры слов (длиной не более 10 символов).
+-----+
: | Выход: вывод слов, повлекших коллизии, на экран.
+-----+
elem_tab      struc
state         db      0           ; байт признаков
len_id        db      0           ; длина идентификатора
buf_id        db      8 dup (20h) ; буфер для хранения идентификатора
ends
buf_0ah       struc
len_buf       db      9           ; длина buf_in
len_in        db      0           ; действительная длина введенного слова
; (без учета 0dh)
buf_in        db      9 dup (20h) ; буфер для ввода (с учетом 0dh)
ends
.data

```

```

tab          elem_tab 64 dup (<>)
len_tab      db        64
len_elem     dw        10
buf_0ah     <>
db          0dh, 0ah, '$' : для вывода функцией 09h (int 21h)

.code
m1:          ;----- вводим слова с клавиатуры
            lea        dx, buf
            mov        ah, 0ah
            int        21h
            ;----- анализируем первую букву введенного слова
            ;         если пробел, то на выход
            cmp        buf.buf_in, 20h
            je         exit
            ;----- ВЫДЕЛЕН ФРАГМЕНТ ВЫЧИСЛЕНИЯ ХЭШ-ФУНКЦИИ
            ;-----
            xor        bx, bx
            mov        cl, buf.len_in
            xor        si, si
            xor        ax, ax
m2:          mov        bl, buf.buf_in[si]
            add        ax, bx
            inc        si
            loop       m2
            shr        ax, 6          : делим на 64
            ;-----
            ;----- определяем адрес в таблице, по которому
            ;         будет размещен идентификатор
            mul        len_elem      ; умножаем на 10 (длина элемента таблицы)
            lea        di, tab
            add        di, ax
            ;----- анализируем занятость элемента
            ;         переход на отображение идентификатора, если коллизия
            test       [di].state, 1b
            jnz        displ
            ;----- формируем элемент таблицы
            ;----- установить бит 0 – занятость
            or         [di].state, 1b
            ;----- пересылка идентификатора и его длины в элемент таблицы
            push       ds
            pop        es
            lea        si, buf.len_in
            xor        cx, cx
            mov        cl, buf.len_in
            inc        cx :длину тоже нужно захватить
            add        di, len_id
            rep        movsb
            jmp        m1
displ:       ;----- выводим идентификатор, вызвавший коллизию, на экран
            lea        dx, buf
            mov        ah, 09h
            int        21h
            jmp        m1
            :...

```

Метод умножения

Для этого метода нет ограничений на длину таблицы, свойственных методу деления. Вычисление хэш-адреса происходит в два этапа:

1. Вычисление нормализованного хэш-адреса в интервале $[0..1]$ по формуле: $F(K) = (C \cdot K) \bmod 1$, где C — некоторая константа из интервала $[0..1]$, K — результат преобразования ключа в его числовое представление, $\bmod 1$ означает, что $F(K)$ является дробной частью произведения $C \cdot K$.

2. Конечный хэш-адрес $A(K)$ вычисляется по формуле $A(K) = [M \cdot F(K)]$, где M — размер хэш-таблицы, а скобки $[]$ означают целую часть результата умножения. Удобно рассматривать эти две формулы вместе:

$$A(K) = M \cdot (C \cdot K) \bmod 1.$$

Кнут в качестве значения C рекомендует «золотое сечение» — величину, равную $((\sqrt{5} - 1)/2 \approx 0,6180339887$. Значение $F(K)$ можно формировать с помощью как команд сопроцессора, так и целочисленных команд. Команды сопроцессора вам хорошо известны и трудностей с реализацией последней формулы не возникает. Интерес представляет реализация вычисления $A(K)$ с помощью целочисленных команд. Правда, в отличие от реализации сопроцессором здесь все же удобнее ограничиться условием, когда M является степенью 2. Тогда процесс вычисления с использованием целочисленных команд выглядит так:

1. Выполняем произведение $C \cdot K$. Для этого величину «золотого сечения» $C \approx 0,6180339887$ нужно интерпретировать как целочисленное значение, соответствующее длине используемого машинного слова b в битах. Если $b = 8$, то соответственно $C \approx 62$, для $b = 16$ — $C \approx 61\,803$, и т. д. В качестве результата этого шага берутся старшие b битов произведения, причем считается, что десятичная точка стоит слева от этих битов.
2. При размере таблицы, равном степени 2 ($M = 2^p$), в качестве конечного хэш-адреса берутся первые p из b цифр, получившихся на шаге 1.

За исключением фрагмента вычисления хэш-функции, сама программа такая же, как и в методе деления. Поэтому вставьте фрагмент, приведенный ниже, на место в программе, соответствующее вычислению хэш-функции (отчеркнуто линиями).

```

:----- ВЫДЕЛЕН ФРАГМЕНТ ВЫЧИСЛЕНИЯ ХЭШ-ФУНКЦИИ
:-----
      xor     bx, bx
      mov     cl, buf.len_in
      xor     si, si
      xor     ax, ax
m2:    mov     bl, buf.buf_in[si]
      add     ax, bx
      inc     si
      loop    m2          ; получили K
      mov     bx, 61803
      mul     bx
:----- для M=64 p=6; получаем произведение в edx из dx:ax
      mov     cx, 16
m3:    cld
      rcl     ax, 1
      rcl     edx, 1
      loop    m3
      bsr     ecx, edx    ; в ecx номер первой единичной позиции
                           ; в edx (относительно нулевой)

      inc     cl
      ror     edx, cl     ; значение у левого края edx
      mov     cx, 6       ;двигаем в ax слева p=6 бит
m4:    cld
      rcl     edx, 1
      rcl     ax, 1
      loop    m4
:-----

```

Из результатов тестирования видно, что этот метод лучше задействует таблицу, более равномерно размещая в ней элементы.

Следующие два метода будут рассмотрены только концептуально, так как сложностей с их реализацией нет, кроме сильной зависимости от ключей, используемых в конкретной задаче.

Метод извлечения битов

Данный метод считается самым старым. Он рассматривает ключевое слово как строку битов. Двоичное значение хэш-адреса образуется сцеплением такого количества битов, которое необходимо для адресации всех элементов хэш-таблицы. Позиции, из которых извлекаются биты, определяются на основе статистического анализа возможных ключевых слов, то есть из условий конкретной задачи. Необходимо стремиться к тому, чтобы появление 0 и 1 в выделяемых позициях было как можно более равновероятным. Здесь трудно дать рекомендации, просто нужно провести анализ как можно большего количества возможных ключей, разделив составляющие их байты на тетрады. Для формирования хэш-адреса нужно будет взять биты из тех тетрад (или целиком тетрады), значения в которых изменялись равномерно.

Метод середины квадрата

В литературе часто упоминается метод середины квадрата как один из первых методов генерации последовательностей псевдослучайных чисел. При этом он непременно подвергается критике за плохое качество генерируемых последовательностей. Но, как упомянуто выше, для процесса хэширования это не является недостатком. Более того, в ряде случаев это наиболее предпочтительный алгоритм вычисления значения хэш-функции. Суть метода проста: значение ключа возводится в квадрат, после чего берется необходимое количество средних битов результата. Возможны варианты — при различной длине ключа биты берутся с разных позиций. Для принятия решения об использовании метода середины квадрата для вычисления хэш-функции необходимо провести статистический анализ возможных значений ключей. Если они часто содержат большое количество нулевых битов, то это означает, что распределение значений битов в средней части квадрата ключа недостаточно равномерно. В этом случае использование метода квадрата неэффективно.

На этом мы закончим знакомство с методами хэширования, так как полное обсуждение этого вопроса не является предметом книги. Информацию об остальных методах (сегментации, перехода к новому основанию, алгебраического кодирования) можно получить из различных источников, среди которых следует выделить [32] и [27].

В ходе реализации хэширования с помощью методов деления и умножения возможные коллизии мы лишь обозначали без их обработки. Настало время разобраться с этим вопросом.

Обработка коллизий

Для обработки коллизий применяются две группы методов:

- закрытые — в качестве резервных используются ячейки самой хэш-таблицы;

- открытые — для хранения элементов с одинаковыми хэш-адресами выделяется отдельная область памяти.

Видно, что эти группы методов разрешения коллизий соответствуют классификации алгоритмов хэширования — они тоже делятся на открытые и закрытые. Яркий пример открытых методов — метод цепочек, который сам по себе является самостоятельным методом хэширования. Он несложен, и мы рассмотрим его несколько позже.

Закрытые методы разрешения коллизий более сложные. Их основная идея — при возникновении коллизии попытаться отыскать в хэш-таблице свободную ячейку. Процедура поиска свободной ячейки называют *опробыванием*, или *повторным хэшированием* (вторичным хэшированием). При возникновении столкновения к первоначальному хэш-адресу $A(K)$ добавляется некоторое значение p_i и вычисляется выражение:

$$A(K) = (A(K) + p_i) \bmod M,$$

где $i = 0 \dots M$.

Если новый хэш-адрес $A(K)$ опять вызывает коллизию, то выражение вычисляется при p_2 , и т. д.

Здесь $p_i = p_1, p_2, p_3, \dots$; M — размер таблицы; $A(K)$ — значения хэш-адреса (начальное и очередное), вычисленные при определенном p_i .

Перечисленные ниже методы отличаются способом вычисления значений p_i :

- линейное повторное хэширование;
- квадратичное повторное хэширование;
- случайное повторное хэширование;
- вторичное хэширование сложением.

Использование любого метода повторного хэширования предполагает, что доступ к определенной ячейке абсолютно предсказуем с данным значением хэш-адреса, так как для записи в хэш-таблицу и поиска в ней используются одни и те же процедуры.

Линейное повторное хэширование

Линейное повторное хэширование предполагает, что к начальному хэш-адресу прибавляется по единице до тех пор, пока не будет обнаружена незанятая ячейка (в данном разделе под «единицей» понимается не число 1, а размер элемента таблицы).

Первоначально значение p_i полагается равным 0. Тогда формула $A(K) = (A(K) + p_i) \bmod M$ соответствует методу деления, который по значению ключа K вычисляет значение хэш-функции $A(K)$, то есть адрес размещения элемента таблицы, к которому осуществляется доступ для записи или чтения. Если элемент таблицы свободен (или при поиске значение ключа элемента совпадает с K), то в него осуществляется запись. В случае занятости элемента (или если при поиске значение ключа элемента не совпадает с K) значение функции $A(K)$ вычисляется при значении $p_1 = 1$. В случае неудачи значение функции $A(K)$ рассчитывается при значении $p_2 = 2$, и т. д. Таким образом доступ осуществляется к следующим подряд элементам таблицы, начиная с некоторого начального, вычисленного по фор-

муле $A(K) = K \bmod M$ (см. метод деления). Этот метод повторного хэширования неэффективен, так как часто приводит к образованию длинных цепочек элементов, имеющих одинаковое значение хэш-функции. Для примера доработаем программу, использующую для работы с хэш-таблицей метод умножения. В результате получится программа `prg02_09.asm`, полный текст которой имеется среди файлов, прилагаемых к книге.

```

-----+
:| Программа: prg02_09.asm. Демонстрация линейного повторного хэширования |
:| на примере задачи сбора информации об идентификаторах программы.      |
-----+
.code
:----- определяем адрес в таблице, по которому
: будет размещен идентификатор
m5: mov     bx, ax          : сохраним его на случай коллизии
    mul     len_elem       : умножаем на 10 (длина элемента таблицы)
    lea     di, tab
    add     di, ax
:----- анализируем занятость элемента.
: переход на отображение идентификатора, если коллизия
    test    [di].state, 1b
    jnz     disp1
:----- формируем элемент таблицы
:----- установить бит 0 – бит занятости
    or      [di].state, 1b
:----- пересылка идентификатора и его длины в элемент таблицы
    push    ds
    pop     es
    lea     si, buf.len_in
    xor     cx, cx
    mov     cl, buf.len_in
    inc     cx              : длину тоже нужно захватить
    add     di, len_id
    rep     movsb
    jmp     m1
disp1: :----- выводим идентификатор, вызвавший коллизию, на экран
:....
:----- повторное хэширование.
: ищем место в таблице для идентификатора, вызвавшего коллизию.
: путем линейного повторного хэширования
    inc     bx
    mov     ax, bx
    jmp     m5
:....

```

Квадратичное повторное хэширование

Процедура квадратичного повторного хэширования предполагает, что процесс поиска резервных ячеек производится с привлечением некоторой квадратичной функции, например такой:

$$p_i = a_i^2 + b_i + c.$$

Хотя значения a , b , c можно задавать любыми, велика вероятность быстрого заикливания значений p_i . Поэтому в качестве рекомендации опишем один из вариантов реализации процедуры квадратичного повторного хэширования, позволяющий осуществить перебор всех элементов хэш-таблицы [32]. Для этого значения в формуле $p_i = a_i^2 + b_i + c$ положим: $a = 1$, $b = c = 0$. Размер таблицы желательно задавать равным простому числу, которое определяется формулой $M = 4 \cdot n + 3$, где n — целое число. Для вычисления значений p_i используют одно из соотношений:

$$p_i = (K + i^2) \bmod M;$$

$$p_i = (M + 2 \cdot K - (K + i^2) \bmod M) \bmod M,$$

где $i = 1, 2, \dots, (M - 1)/2$; K — первоначально вычисленный хэш-адрес. Адреса, формируемые на основе первого соотношения, покрывают половину хэш-таблицы, а адреса, формируемые при помощи второй формулы, — другую половину. Практически реализовать данный метод можно следующей процедурой:

1. Задание $i = -M$.
2. Вычисление хэш-адреса K одним из методов хэширования.
3. Если ячейка свободна или ключ элемента в ней совпадает с искомым ключом, то завершение процесса поиска. Иначе $i = i + 1$.
4. Вычисление $h = (h + |i|) \bmod M$.
5. Если $i < M$, то переход к шагу 3. Иначе ($i \geq M$) таблица полностью заполнена.

Программа та же, что приведена в методе линейного повторного хэширования, за исключением добавления одной команды для инициализации процесса повторного хэширования, самого фрагмента повторного хэширования и небольших изменений сегмента данных:

```

N = 21

.data
M          db      4 * n + 3      ; размер таблицы
tab        elem_tab 4*n+3 dup (<>)
len_tab    db      4 * n + 3
len_elem   dw      10
i          dw      0
buf        buf_0ah <>
           db      dh, 0ah, '$'   ; для вывода функцией 09h (int 21h)

.code
;----- вводим слова с клавиатуры
;....
m1:        mov     i, -(4 * n + 3) ; начальное значение
;----- повторное хэширование.
;         ищем место в таблице для идентификатора, вызвавшего коллизию.
;         путем квадратичного повторного хэширования (его индекс в bx)
inc     i
;----- определяем модуль I
mov     ax, i
bt      ax, 15      ; операнд отрицателен
jnc     m6          ; неотрицателен
neg     ax          ; вычисляем модуль
;----- сравниваем I<M. Если I>M, то на выход
m6:       cmp     al, M
jae     exit
;----- вычислим h:=(h+|i|) mod M
add     ax, bx
div     M
shr     ax, 8        ; в ax новое значение хэш-адреса
                     ; (индекса в таблице)
jmp     m5

```

Случайное повторное хэширование

Для метода случайного повторного хэширования размерность таблицы M должна быть степенью двойки $M = 2^k$. Значения p_i вычисляются по следующему алгоритму:

1. Первоначально $p_1 = 1$.

2. Каждое последующее p_i рассчитывается так: получить $p_i = 5 \cdot p_{i-1} - 1$, выделить младшие $k + 2$ разрядов p_i , результат поместить обратно в p_i , затем эту величину сдвинуть вправо на 2 разряда (разделить на 4). Результат будет являться очередным значением p_i .

Преимущество этого метода в том, что получаемые числа различны. Основа программы та же, что и для метода линейного повторного хэширования, поэтому ниже приведены лишь измененные фрагменты.

```
.data
M          db      64
pi         dw      ?
five       db      5
.code
:----- начальное значение для процесса повторного хэширования.
:
:   если он будет
:   mov     pi, 1
:----- вводим слова с клавиатуры
m1:        lea     dx, buf
:   ....
:----- повторное хэширование.
:   ищем место в таблице для идентификатора, вызвавшего коллизию.
:   с помощью случайного повторного хэширования (k=6 => k+2=8)
:   mov     ax, pi
:   add     ax, bx
:   div     M
:   shr     ax, 6           ; в ax новое значение хэш-адреса
:                           ; (индекса в таблице)
:----- готовим pi для второго и последующего шагов
:   повторного хэширования
:   push    ax
:   mov     ax, pi
:   mul     five
:----- так как k+2=8, то это байт, и выделять ничего не нужно.
:   при необходимости применяйте команду AND с соответствующей
:   маской, обозначим ее
:   and     ax, 0ffh       ; выделить младшие k+2 разряда pi
:   shr     ax, 2           ; разделить на 4
:   mov     pi, ax
:   pop     ax
:   jmp     m5
:   ....
```

Повторное хэширование сложением

Этот метод предполагает другой способ вычисления очередного значения p_i ; $p_i = (i \cdot h) \bmod N$, где $i = 1, 2, 3, 4, \dots, N$ — размер таблицы (должен быть простым числом); h — исходное значение хэш-функции в диапазоне $1 \dots N - 1$. Соответствующую программу для этого реализуйте самостоятельно.

Формат таблицы хэширования и ее элементов

Хорошо продуманные формат хэш-таблицы и структура ее элементов являются мощным резервом увеличения производительности метода хэширования. Приведем несколько общих рекомендаций, которые во многих случаях могут оказаться полезными.

Элемент любой хэш-таблицы логически можно считать состоящим из двух частей: ключевой и информационной. Для совместного хранения этих двух частей требуется большой объем памяти, так как закрытое хэширование предполагает,

что под таблицу отводится максимально потребный объем памяти, даже если реально используется небольшое количество элементов. В результате возрастает вероятность непроизводительного расходования памяти. Этот недостаток можно исправить, если хэш-таблицу физически разделить на две части: ключевую и информационную. Под ключевую часть по-прежнему отводится столько памяти, сколько необходимо для хранения максимального количества элементов хэш-таблицы. Элемент хэш-таблицы в ключевой части хранит сами ключи (или их внутреннее представление), поле признаков (см. ниже) и указатель на информационную часть данного элемента таблицы. Информационные части хранятся в отдельной области памяти, при этом в отличие от ключевой части таблицы нет необходимости хранить информационные части для неиспользуемых элементов. По мере заполнения таблицы память для хранения информационных частей элементов можно выделять динамически. Такая схема поддержания хэш-таблицы называется индексной.

Другой вариант формата хэш-таблицы может задействовать свойства страничной адресации. Это можно осуществить для линейного повторного хэширования, для которого характерно попадание резервных элементов поблизости от значения вычисленного хэш-адреса. Вероятность попадания можно еще увеличить, если в процедуре повторного хэширования адресные смещения вычислять циклически, по модулю, равному размеру таблицы.

И последний вариант построения хэш-таблиц — клеточная организация. Ее также применяют при линейном повторном хэшировании. Суть этого варианта в том, что группа элементов таблицы объединяется в образование, называемое клеткой (рис. 2.8). До тех пор пока в клетке есть свободные ячейки, в них производится размещение новых элементов, соответствующих одному хэш-адресу.

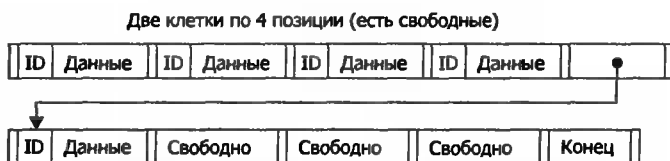


Рис. 2.8. Клеточная организация хэш-таблицы

Особенности формата таблицы нужно как-то описывать. Для этого в элемент таблицы, помимо ключевой и информационной частей, вводят поле признаков. В начале обсуждения поиска по таблице упоминалось назначение поля текущего состояния элемента таблицы. Поле признаков выполняет аналогичную функцию, но с учетом особенностей метода хэширования. Состав, структура и форма реализации этого поля определяются особенностями конкретной задачи. В большинстве случаев его удобно исполнять в виде битового поля и для обработки привлекать соответствующие средства ассемблера. Перечислим назначение некоторых битов, из которых может состоять поле признаков:

- занятости — может быть использован в процессе размещения нового объекта для быстрого определения занятости элемента таблицы;
- удаления (см. ниже раздел «Удаление — решение проблемы»);

- конца цепочки повторного хэширования, или коллизии, — предназначен для определения конца цепочки резервных ячеек, соответствующих определенному хэш-адресу;
- конца таблицы;
- связи — подходит для определения того, что содержит информационное поле: собственно информационную часть данного элемента таблицы или указатель на другую область памяти, где она размещена.

Некоторые из возможных структур элементов таблицы хэширования показаны на рис. 2.9.

На рисунке использованы следующие обозначения:

- ID — идентификатор ключевого слова;
- P_i — указатель области данных (индекс);
- P_0 — указатель области переполнения (или следующей записи в цепочке);
- T — терминальный символ;
- U — флаг занятости;
- D — флаг вычеркивания;
- C — флаг коллизии;
- L — флаг связи.

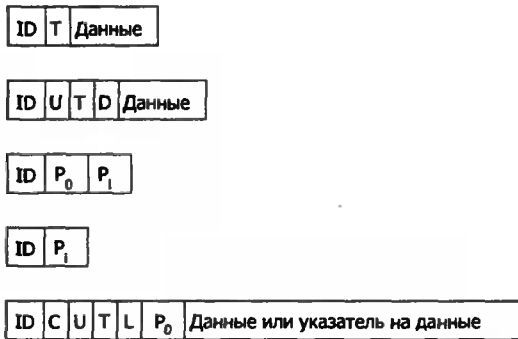


Рис. 2.9. Возможные структуры элементов таблицы хэширования

Удаление — решение проблемы

Выше мы разобрались с порядком выполнения двух основных операций с хэш-таблицей. Третья операция — удаление — не так проста в реализации. При удалении элемента внутри некоторой цепочки повторного хэширования его нельзя пометить как свободный, так как потеряются остальные элементы с таким же значением хэш-адреса. Сдвиг всех элементов цепочки к ее началу — процесс громоздкий и неэффективный. Гораздо производительнее использовать биты в поле признаков. Например, для линейного повторного хэширования проблема решается просто — удаляемый элемент помечается как «удаленный» и впоследствии может рассматриваться как свободный для размещения новых элементов с тем же зна-

чением хэш-адреса. Для того чтобы удаленный элемент сделать свободным, необходимо проверить следующий возможный элемент в цепочке. Если он пуст, то и удаляемую ячейку можно пометить как свободную. Для случайного повторного хэширования можно использовать подобные методы, но при этом необходимо помнить, что для него в отличие от линейного метода процесс прохождения по цепочке повторного хэширования необратим.

Метод цепочек

«На закуску» рассмотрим суть еще одной схемы хэширования, которая одновременно может выступать в качестве метода повторного хэширования. Метод цепочек является методом открытого хэширования и свободен от многих недостатков и проблем моделей закрытого хэширования.

В основе метода цепочек лежат три объекта: хэш-таблица, указатель ближайшей свободной ячейки и область памяти для размещения информационных частей элементов хэш-таблицы. Модель метода цепочек показана на рис. 2.10. Хэш-таблица здесь является таблицей указателей на начало цепочек элементов с определенным значением хэш-адреса. После вычисления хэш-адреса посредством соответствующего ему элемента хэш-таблицы программа получает доступ к началу цепочки элементов с данным значением хэш-адреса. Если цепочка пуста, то из указателя ближайшей свободной ячейки извлекается адрес, который заносится в элемент хэш-таблицы. В указатель *ближайшей свободной ячейки* заносится другой адрес, который будет впоследствии использоваться для размещения очередного нового элемента. Если цепочка не пуста, то при поиске программа просто идет по цепочке указателей и проверяет значение ключей до тех пор, пока не будет найден нужный элемент или не встретится элемент, отмеченный последним в цепочке. При добавлении элемента в цепочку с определенным значением хэш-адреса в соответствующий элемент хэш-таблицы заносится адрес из указателя ближайшей свободной ячейки. В поле связи нового элемента помещается указатель из элемента хэш-таблицы для данного хэш-адреса. Таким образом, добавление новых элементов производится в начало цепочек, что оправданно, так как статистически обращение к новым элементам происходит чаще.

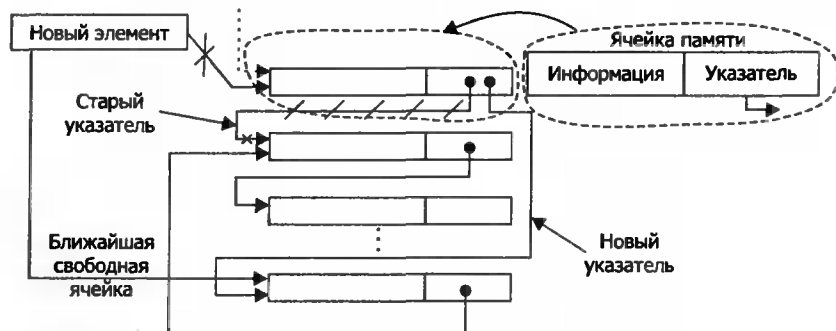


Рис. 2.10. Модель метода цепочек

Следует отметить, что метод цепочек значительно проще реализует проблему удаления элементов путем манипуляции битами в поле признаков и указателями.

Пример для демонстрации метода цепочек мы приводить не будем, так как для этого необходимы знания по работе со списками, которые являются предметом изучения следующего раздела. Метод цепочек потребуются нам при изучении вопросов компиляции, к тому времени мы уже будем обладать необходимыми знаниями и сможем безболезненно написать соответствующий программный код.

Перечисленные выше методы хэширования не являются догмой, так как, подчеркнем еще раз эту мысль, выбор алгоритма хэширования определяется прежде всего постановкой задачи. Поэтому программист вполне может изобрести новый метод хэширования (по сути, разработать свою хэш-функцию), наиболее точно отображающий специфику конкретной задачи. Но хэширование не панацея, и есть ряд областей, где его использование затруднено, недопустимо или просто невозможно. Прежде всего это касается временного фактора. Для метода хэширования на разных наборах исходных данных время доступа к элементам таблицы будет различным и зачастую трудно предсказуемым. Поэтому альтернативой для задач, для которых должно быть известно максимальное время доступа к данным, могут являться методы, основанные на деревьях поиска, и т. п. Наибольший эффект от хэширования — при поиске по заданным идентификаторам или дескрипторам, что характерно для задач баз данных, обработки документов и т. д. Для задач, в которых поиск ведется сравнением или вычислением сложных логических функций, лучше использовать традиционные методы сортировки и поиска.

Для того чтобы совершить плавный переход к рассмотрению следующей структуры данных — списков, вернемся еще раз к одной проблеме, связанной с массивами. Среди массивов можно выделить массивы специального вида, которые называют *разреженными*. В этих массивах большинство элементов равны нулю. Отводить место для хранения всех элементов расточительно. Естественно, возникает желание сэкономить. Что для этого можно предпринять?

Техника обработки массивов предполагает, что все элементы расположены в соседних ячейках памяти. Для ряда приложений это недопустимое ограничение.

Обобщенно можно сказать, что все перечисленные выше структуры имеют общие свойства:

- постоянство структуры данных на всем протяжении ее существования;
- память для хранения отводится сразу всем элементам структуры и все элементы находятся в смежных ячейках памяти;
- отношения между элементами просты настолько, что можно исключить потребность в средствах хранения информации об их отношениях в какой бы то ни было форме.

Исходя из этих свойств, означенные структуры данных и называют статическими.

Снять подобные ограничения можно, используя другой тип данных — списки. Для них подобных ограничений не существует.

Список

Если у веревки есть один конец, значит, у нее должен быть и другой.

Закон Микша (Прикладная Мерфология)

В общем случае под списком понимается линейно упорядоченная последовательность элементов данных, каждый из которых представляет собой совокупность одних и тех же полей. Упорядоченность элементов данных может быть двоякой.

- элементы списка располагаются последовательно как в структуре представления, так и в структуре хранения — список такого типа называется *последовательным*;
- порядок следования элементов задается с помощью специальных указателей, которые расположены в самих элементах и определяют их соседей справа и/или слева — подобные списки называются *связными*.

Последовательные списки

Если количество элементов в списке постоянно, то в зависимости от их типа список вырождается в вектор, массив, структуру или таблицу. Отличие списка от этих структур данных — в длине. Для списков она является переменной. Поэтому программист, разбираясь с самими списками, должен уделить внимание тому, каким образом ему следует выделять память для хранения списка. Обычно языки высокого уровня имеют соответствующие средства, в отличие от языка ассемблера. При написании программы на ассемблере программист должен уметь напрямую обратиться к операционной системе для решения проблемы динамического управления памятью. В примерах мы будем использовать соответствующие функции API Win32 как более современные, хотя для общего случая это не принципиально. В MS DOS имеются аналогичные функции (с точки зрения цели использования) — это функции 48h, 49h, 4ah прерывания 21h. Вопрос динамического управления памятью в силу своей важности в контексте настоящего рассмотрения требует особого внимания.

Отличие последовательных списков от связанных состоит в том, что добавление и удаление элементов возможно только по концам структуры. В зависимости от алгоритма изменения различают такие виды последовательных списков, как стеки, очереди и деки. Эти виды списков обычно дополняются служебным дескриптором, который может содержать указатели на начало и конец области памяти, выделенной для списка, указатель на текущий элемент.

Зачем нужен, к примеру, стек? Необходимость использования своего стека в программе вполне вероятна, хотя бы исходя из того, что элементы, для хранения которых он создается, могут иметь размер, отличный от 2/4 байта.

Важно понимать, что внутренняя структура элемента практически не зависит от типа последовательного списка. Списки отличаются набором операций, которые производятся над ними. Поэтому рассмотрим их отдельно для каждого типа последовательного списка.

Стек

Стек — последовательный список, в котором все включения и исключения элементов производятся на одном его конце — по принципу LIFO (Last In First Out — последним пришел, первым ушел).

Для стека определены следующие операции:

- создание стека;
- включение элемента в стек;
- исключение элемента из стека;
- очистка стека;
- проверка объема стека (числа элементов в стеке);
- удаление стека.

Создание стека должно сопровождаться инициализацией специального дескриптора стека, который может содержать следующие поля: имя стека, адреса нижней и верхней его границ, указатель на вершину стека.

Иллюстрацию организации и работы стека произведем на примере задачи, анализирующей правильность согласования скобок в некотором тексте. Причем условимся, что скобки могут быть нескольких видов: (), { }, [], < >. Программа реализована в виде приложения Win32 с использованием функций API Win32 для работы с кучей (из нее выделяется память для стека).

```

+-----+
: | Программа: prg12_51.asm. Проверка правильности расстановки скобок
: | в тексте — иллюстрируется работа со стеком.
+-----+
: | Вход: строка символов.
+-----+
: | Выход: сообщение на экран о том, согласованы скобки или нет.
+-----+
: | Имя стека — имя экземпляра структуры. Перед обращением к макрокомандам
: | работы со стеком необходимо инициализировать поля size_stk и size_item.
+-----+
desc_stk      struc      ; дескриптор стека
p_start      dd          0      ; адрес блока (начала области памяти
                                ; для стека) из общей кучи процесса
size_stk      dd          0      ; размер стека в байтах
p_top        dd          0      ; адрес вершины стека
size_item     dd          1      ; размер элемента стека в байтах
                                ; (по умолчанию 1 байт)
Hand_Head     dd          0      ; описатель общей кучи процесса
ends
.data
string        db          "a<a(kk{k)p>pp" ; строка для тестирования
              l_string = $ - string
              ;----- описание данных и строк сообщений
              ....
.code
              ;----- описание макрокоманд работы со стеком
              ; (см. файлы, прилагаемые к книге)
              ;----- создание стека
create_stk     macro descr:REQ, SizeStk:=<256>
              ....
endm
              ;----- очистка стека
init_stk       macro descr:REQ

```



```

endm
;....
;----- удаление стека
delete_stk macro descr:REQ
;....
endm
;----- добавление элемента в стек
push_stk macro descr:REQ, adr_item:REQ
;....
endm
;----- проверка стека на пустоту
TestEmptyStk macro descr:REQ, label_err:REQ
;....
endm
;----- извлечение элемента из стека в область памяти
pop_stk macro descr:REQ, adr_item:REQ
;....
endm
;----- извлечение элемента из вершины стека без его удаления оттуда
TestTopStk macro descr:REQ, adr_item:REQ
;....
endm
start proc near ; точка входа в программу
;....
create_stk char_stk ; создание стека
;----- анализируем строку
mov ecx, l_string
lea ebx, string
jmp cycl
e_exit: jmp exit
cycl: jcxz e_exit
cmp byte ptr [ebx], "("
je m_push
cmp byte ptr [ebx], "["
je m_push
cmp byte ptr [ebx], "{"
je m_push
cmp byte ptr [ebx], "<"
je m_push
cmp byte ptr [ebx], ")"
jne m1
;----- извлекаем элемент из вершины стека и анализируем его
TestEmptyStk char_stk, mes_error
pop_stk char_stk, <offset temp>
cmp temp, "("
jne mes_error
jmp r_next
m1: cmp byte ptr [ebx], "]"
jne m2
;----- извлекаем элемент из вершины стека и анализируем его
TestEmptyStk char_stk, mes_error
pop_stk char_stk, <offset temp>
cmp temp, "["
jne mes_error
jmp r_next
m2: cmp byte ptr [ebx], "}"
jne m3
;----- извлекаем элемент из вершины стека и анализируем его
TestEmptyStk char_stk, mes_error
pop_stk char_stk, <offset temp>
cmp temp, "{"
jne mes_error
jmp r_next
m3: cmp byte ptr [ebx], ">"
jne r_next

```

```

:----- извлекаем элемент из вершины стека и анализируем его
TestEmptyStk char_stk, mes_error
pop_stk char_stk, <offset temp>
cmp     temp, "<"
jne     mes_error
jmp     r_next

:----- включение скобки в стек
m_push: push_stk char_stk, ebx
r_next:  add     ebx, char_stk.size_item
        dec     ecx
        jmp     cycl

:----- вывод на экран сообщения об ошибке mes_e
mes_error:
        ....
        jmp     exit_exit

:----- проверяем стек на пустоту
exit:   pop_stk char_stk, <offset temp>
        jnc     mes_error ; стек не пуст

:----- вывод на экран сообщения mes_ok
        ....

:----- финализация
exit_exit: delete_stk char_stk ; удаляем блок памяти со стеком
        ....

```

Код, выполняющий работу со стеком, оформлен в виде макрокоманд. При необходимости код можно сделать еще более гибким. Для этого нужно использовать функцию API Win32 `HeapReAlloc`, которая изменяет размер выделенного блока в сторону его увеличения или уменьшения. В принципе, полезной может оказаться операция определения объема стека, то есть количества элементов в нем. Попробуйте реализовать ее самостоятельно.

Очередь

Очередь [2, 10, 15] — последовательный список, в котором включение элементов производится на одном конце, а исключение — на другом, то есть по принципу FIFO (First In First Out — первым пришел, первым ушел). Сторона очереди, на которой производится включение элементов, называется *хвостом*. Соответственно, противоположная сторона — *голова* очереди. Очередь описывается дескриптором, который может содержать поля: имя очереди, адреса верхней и нижней границ области памяти, выделенной для очереди, указатели на голову и хвост.

Набор операций для очереди:

- создание очереди (`create_que`);
- включение элемента в очередь (`push_que`);
- исключение элемента из очереди (`pop_que`);
- проверка пустоты очереди (`TestEmptyQue`);
- очистка очереди без освобождения памяти для нее (`init_que`);
- удаление очереди (`delete_que`).

При помощи очереди удобно моделировать некоторые обслуживающие действия, например это может быть некоторая очередь запросов к критическому ресурсу или очереди заданий на обработку. К очереди так же, как и к стеку, применимы такие параметры, как *емкость* очереди и *размер элемента* очереди.

На практике используются очереди двух типов — простые и кольцевые. Очередь обслуживается двумя указателями — головы (P_1) и хвоста очереди (P_2) (рис. 2.11).

Указатель головы P_1 идентифицирует самый старый элемент очереди, указатель хвоста — первую свободную позицию после последнего включенного в очередь элемента. Сами элементы физически по очереди не двигаются. Меняется лишь значение указателей. При включении в очередь новый элемент заносится в ячейку очереди, на которую указывает P_2 . Операция исключения подразумевает извлечение элемента из ячейки очереди, указываемой P_1 . Кроме извлечения элемента, операция исключения производит корректировку указателя P_1 так, чтобы он указывал на следующий самый старый элемент очереди. Таким образом, указатель хвоста простой очереди всегда ссылается на свободную ячейку очереди и рано или поздно он выйдет за границы блока, выделенного для очереди. И это случится несмотря на то, что в очереди могут быть свободные ячейки (ячейки A_1, A_2 на рис. 2.11, сверху). Чтобы исключить данное явление, очередь организуется по принципу кольца (рис. 2.11, снизу). В обоих способах организации очереди важно правильно определить ее емкость. Недостаточный объем очереди может привести в определенный момент к ее переполнению, что чревато потерей новых элементов, претендующих на включение в очередь.

Для иллюстрации порядка организации и работы с очередью рассмотрим пример. Пусть имеется строка символов, которая следующим образом моделирует некоторую вычислительную ситуацию: символы букв означают запросы на некоторый ресурс и подлежат постановке в очередь (имеющую ограниченный размер). Если среди символов встречаются символы цифр в диапазоне 1–9, то это означает, что необходимо изъять из очереди соответствующее значению цифры количество элементов. Если очередь полна, а символов цифр все нет, то происходит потеря заявок (символов букв). В нашей программе будем считать, что очередь кольцевая и ее переполнение, помимо потери новых элементов, приводит к выводу соответствующего сообщения. Для кольцевой очереди возможны следующие соотношения значений указателей P_1 и P_2 (рис. 2.11, снизу): $P_1 < P_2$, $P_1 = P_2$ (пустая очередь), $P_1 > P_2$. Память для очереди в нашей задаче выделяется динамически средствами API Win32.

Заметим, что, исходя из потребностей конкретной задачи, можно изменить дисциплину обслуживания очереди. Например, установить, что при переполнении очереди потеря подлежат старые заявки (в голове очереди) и т. п.

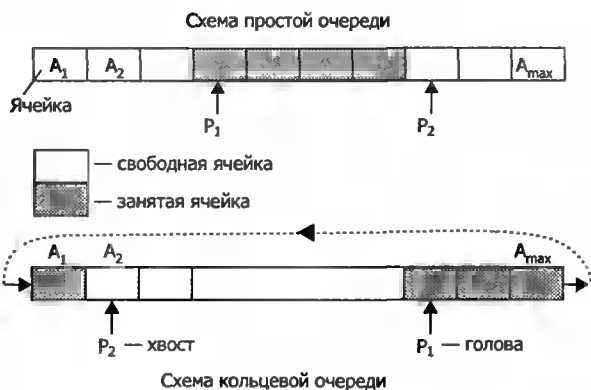


Рис. 2.11. Структура простой и кольцевой очередей

```

+-----+
: | Программа: prg02_10.asm. Пример приложения для Win32 (работа с кольцевой, |
: | очередью) с использованием функций API Win32 для работы с кучей. |
+-----+
: | Вход: строка символов, содержащая буквы и цифры. |
+-----+
: | Выход: нет. Работа программы наблюдается под управлением отладчика. |
+-----+
: | Имя очереди — имя экземпляра очереди. Перед обращением к макрокомандам |
: | необходимо проинициализировать поля структуры size_que и size_item_que. |
+-----+
desc_que      struc      ; дескриптор очереди
p_start_que   dd         0      ; адрес блока (начала области памяти
                                ; для очереди) из кучи процесса (A1)
p_end_que     dd         0      ; адрес конца блока в общей куче
                                ; процесса (Amax)
size_que      dd         0      ; размер очереди в байтах
p_head       dd         0      ; указатель головы очереди (P1)
p_tail       dd         0      ; указатель хвоста очереди (P2)
size_item_que dd         1      ; размер элемента очереди в байтах
                                ; (по умолчанию 1 байт)
Hand_Head     dd         0      ; описатель общей кучи процесса
ends

.data
char_que      desc_que <>
string        db          "gfvfvs4gdfdf2gfd9gffff7ddf3gdf1teeeeeeg6gfd9sdgdf4"
               l_string = $ - string
               ....
mes_e         db          "Очередь переполнена"
               len_mes_e = $ - mes_e

.code
:----- описание макрокоманд работы с очередью
:
:----- создание очереди
create_que    macro descr:REQ, SizeQue:REQ
               ....
endm

:----- удаление очереди
delete_que    macro descr:REQ
               ....
endm

:----- добавление элемента в очередь [2]
push_que     macro descr:REQ, adr_item:REQ
               ....
endm

:----- извлечение элемента из очереди [2]
pop_que      macro descr:REQ, adr_item:REQ
               ....
endm

:----- очистка очереди (удаление всех элементов
: без удаления самой очереди)
init_que     macro descr:REQ
               ....
endm

:----- проверка очереди на пустоту
TestEmptyQue macro descr:REQ
               ....
endm

start        proc near      ; точка входа в программу
               ....
               create_que char_que, 10 ; создание очереди
:----- читаем символы строки
: +1 для обработки всех элементов, включая последний
               lea     esi, string
               mov     ecx, l_string+1

```

```

                push    ecx
                jmp     cyc1
e_xit:          jmp     exit
                ;----- анализируем очередной символ строки
cyc1:           pop     ecx
                jcxz    e_xit
                lodsb                    ; в al очередной символ строки
                dec     ecx
                push    ecx
                cmp     al, 31h
                jb      m1
                cmp     al, 39h
                ja      m1
                ;----- удаляем из очереди элементы
                xor     ecx, ecx
                mov     cl, al
                sub     cl, 30h           ; преобразуем в двоичный эквивалент
m2:             pop_que char_que, <offset temp>
                jc      cyc1           ; если очередь пуста
                loop    m2
                jmp     cyc1
                ;----- добавляем элементы в очередь
m1:             mov     temp, al
                push_que char_que, <offset temp>
                jnc     cyc1           ; успех
                ;----- вывод на экран сообщения об ошибке —
                :         отсутствие места в очереди
                :         ....
                jmp     cyc1
                ;----- выход из приложения, удаляем блок памяти с очередью
exit:           delete_que char_que
                :....

```

Как и в случае со стеком, код, выполняющий работу с очередью, оформлен в виде макрокоманд. Программа выдает единственное сообщение о переполнении очереди. Чтобы наблюдать работу программы в динамике, необходимо выполнить следующие действия.

1. Загрузить исполняемый модуль программы в отладчик `td32.exe`.
2. Отследить адрес начала блока памяти, который будет выделен системой по нашему запросу для размещения очереди. Для этого подвергните пошаговому выполнению в окне CPU содержимое макрокоманды `create_que`.
3. Выяснив адрес начала блока для размещения очереди, отобразите его в окне Dump и нажмите клавишу F7 или F8. Не отпуская этой клавиши, наблюдайте за тем, как изменяется состояние очереди.

Дека

Дека [2, 13] — очередь с двумя концами. Дека является последовательным списком, в котором включение и исключение элементов может осуществляться с любого из двух концов списка. Логическая и физическая структуры деки аналогичны обычной очереди, но относительно деки следует говорить не о хвосте и голове, а о правом и левом конце очереди. Реализацию операций включения и исключения в деку следует также дополнить признаками левого или правого конца. В литературе можно встретить понятие деки с ограниченным вводом. Это дека, в которой элементы могут вставляться с одного конца, а удаляться с обоих.

Связные списки

Связный список — набор элементов, каждый из которых структурно состоит из двух частей — содержательной и связующей. Содержательная часть представляет собой собственно данные или указатель на область памяти, где эти данные содержатся. Связующая часть предназначена для связи элементов в списке и определяет очередность их следования. Благодаря связующей части в каждом из элементов становятся возможными «путешествия» по списку путем последовательных переходов от одного элемента к другому.

В отличие от последовательного списка связный список относится к динамическим структурам данных. Для них характерны непредсказуемость в числе элементов, которое может быть нулевым, а может и ограничиваться объемом доступной памяти. Другая характеристика связного списка — необязательная физическая смежность элементов в памяти.

Относительно связующей части различают следующие типы связных списков.

- *Односвязный список* — список, начало которого определяется указателем начала, а каждый элемент списка содержит указатель на следующий элемент. Последний элемент должен содержать признак конца списка или указатель на первый элемент списка. В последнем случае мы имеем односвязный кольцевой список. Его преимущество состоит в том, что просмотр всех элементов списка можно начинать с любого элемента, а не только с головы. Более того, в случае кольцевого односвязного списка указателем его начала может быть любой элемент списка. Этот вид списка всегда линейный, так как однозначно задает порядок своего просмотра.
- *Двусвязный список* — список, связующая часть которого состоит из двух указателей — на предыдущий и последующий элементы. Начало и конец списка определяются отдельными указателями. Последние элементы двусвязного списка в каждом из направлений просмотра содержат признаки конца. Если замкнуть эти указатели друг на друга, то мы получим кольцевой двусвязный список. В этом случае потребность в указателе конца списка отпадает. Такой вид списка может быть как линейным (иметь одинаковый порядок просмотра в обоих направлениях), так и нелинейным (второй указатель каждого элемента списка задает очередность просмотра, которая не является обратной порядку, устанавливаемому первым указателем).
- *Многосвязный список* — список, связующая часть элементов которого в общем случае содержит произвольное количество указателей. В этом виде списка каждый элемент входит в такое количество односвязных списков, сколько он имеет в себе полей связи.

В общем случае для связанных списков определены следующие операции [3]:

- создание связного списка;
- включение элемента в связный список, в том числе и после (перед) определенным элементом;
- доступ к элементу связного списка (поиск в списке);
- исключение элемента из связного списка;

- упорядочение (перестройка) связного списка;
- очистка связного списка;
- проверка объема списка (числа элементов в связном списке);
- объединение нескольких списков в один;
- разбиение одного списка на несколько;
- копирование списка;
- удаление связного списка.

Связные списки очень важны для представления различных сетевых структур, в частности деревьев, что и будет рассмотрено нами чуть ниже. Пока же рассмотрим работу с некоторыми из обозначенных нами типов связных списков на практических примерах.

Односвязные списки

В простейшем случае односвязный список можно организовать в виде двух одномерных массивов, длины которых совпадают и определяются максимально допустимой длиной списка. Поля первого массива содержат собственно данные, а соответствующие поля второго массива представляют собой указатели. Значением каждого из этих указателей является индекс элемента первого массива, который логически следует за данным элементом, образуя таким образом связанный список. В качестве примера рассмотрим программу, которая в некотором массиве связывает все ненулевые элементы, а затем как полезную работу подсчитывает их количество. В последний единичный элемент в качестве признака конца списка заносим 0ffh.

```

+-----+
| Программа: prg3_10.asm. Пример реализации односвязных списков |
| с помощью двух массивов. |
+-----+
| Вход: массивы с данными и указателями. |
+-----+
| Выход: нет. Работа программы наблюдается под управлением отладчика. |
+-----+
.data
mas          db      0, 55, 0, 12, 0, 42, 94, 0, 18, 0, 6
              db      67, 0, 58, 46      ; задаем массив
              n = $ - mas
point        db      0                  ; указатель списка – индекс первого
                                              ; ненулевого элемента в mas
mas_point     db      n dup (0)         ; определим массив указателей
.code
              lea     si, mas           ; в si адрес mas_point
              xor     bx, bx            ; в bx будут индексы – кандидаты на
                                              ; включение в массив указателей
              :----- ищем первый ненулевой элемент
cyc11:        mov     cx, n - 1
              cmp     byte ptr [si][bx], 0
              jne     m1                ; если нулевые элементы, продолжим
              inc     bx
              loop    cyc11
              jmp     exit              ; если нет ненулевых элементов
m1:           mov     point, bl          ; запоминаем индекс первого ненулевого
              mov     di, bx            ; в di индекс предыдущего ненулевого
              :----- просматриваем далее (cx тот же)

```

```

cyc12:      inc     bx
            cmp     byte ptr [si][bx], 0
            je      m2          ; нулевой => пропускаем
;----- формируем индекс
            mov     mas_point[di], bl ; индекс следующего ненулевого
                                      ; в элемент mas_point предыдущего
m2:         mov     di, bx        ; запоминаем индекс ненулевого
            inc     bx
            loop    cyc12
            mov     mas_point[di], 0ffh ; индекс следующего ненулевого
                                      ; в элемент mas_point
;----- а теперь подсчитаем единичные, не просматривая все, —
; результат в ax
            xor     ax, ax
            cmp     point, 0
            je      exit
            inc     ax
            mov     bl, point
            cmp     mas_point[bx], 0ffh
            je      exit
            inc     ax
            mov     bl, mas_point[bx]
            jmp     cyc13
;----- результат подсчета в ax, с ним нужно что-то делать,
; иначе он будет испорчен
;....

```

Если количество нулевых элементов велико, то можно сократить объем хранения данного одномерного массива в памяти (см. ниже материал о разреженных массивах).

Кстати, в качестве еще одного реального примера использования односвязных списков вспомните, как реализован учет распределения дискового пространства в файловой системе MS DOS (FAT).

Далее рассмотрим другой, более естественный вариант организации связанного списка. Его элементы содержат как содержательную, так и связующую части.

Рассуждения относительно содержательной составляющей пока опустим — они напрямую зависят от конкретной задачи. Уделим внимание связующему компоненту. Понятно, что это адреса. Но какие? Будем считать, что существуют два типа адресов, которые могут находиться в связующей части: абсолютные и относительные. Абсолютный адрес в конкретном элементе списка — это, по сути, смещение данного элемента относительно начала сегмента данных или блока данных при динамическом выделении памяти и принципиально он лишь логически связан со списком. *Относительный* адрес формируется исходя из внутренней нумерации элементов в списке и имеет смысл лишь в контексте работы с ним. Пример относительной нумерации рассмотрен выше при организации списка с помощью двух массивов. Поэтому далее этот способ мы рассматривать не будем, а сосредоточимся на абсолютной адресации.

Для начала разработаем код, реализующий применительно к односвязным спискам основные из обозначенных нами выше операций со связанными списками. Односвязные списки часто формируют с головы, то есть включение новых элементов производится в голову списка.

Первая проблема — выделение памяти для списка. Это можно выполнить либо статическим способом, зарезервировав место в сегменте данных, либо динамически, то есть так, как мы это делали выше при реализации операций со стеками и оче-

редями. Недостатки первого способа очевидны, хотя в реализации он очень прост. Гораздо интереснее и предпочтительнее второй вариант, который предполагает использование кучи для хранения связанного списка. Для придания большей реалистичности изложению рассмотрим простую задачу: ввести с клавиатуры символьную строку, ограниченную символом "." (точка), и распечатать ее в обратном порядке.

```

:-----+
:| Программа: prg15_102.asm. Инвертирование строки с односвязными списками. |
:-----+
:| Вход: символьная строка с клавиатуры. |
:-----+
:| Выход: вывод на экран инвертированной строки. |
:-----+
item_list      struc          : элемент списка
next           dd            0 : адрес следующего элемента
info           db            0 : содержательная часть (у нас – символ)
ends
.data
mas            db            80 dup (' ') : в эту строку вводим
mas_rev        db            80 dup (' ') : из этой строки выводим
len_mas_rev    = $ - mas_rev
:....
mes1           db            'Введите строку символов (до 80) для инвертирования '
               db            '(с точкой на конце):'
               len_mes1 = $ - mes1
.code
:----- описание макрокоманд работы со связанным списком
:----- создание списка – формирование головы списка и первого элемента
create_list    macro descr:REQ, head:REQ
:....
endm
:----- добавление элемента в связанный список
add_list       macro descr:REQ, head:REQ, H_Head:REQ
:....
endm
:----- создание элемента в куче для последующего добавления в список
create_item    macro descr:REQ, H_Head:REQ
:....
endm
start          proc          near : точка входа в программу
:....
:----- вывод строки текста – приглашения на ввод строки
: для инвертирования
:....
:----- вводим строку в mas
:....
:----- создаем связанный список
create_list item_list, Head_list
:----- первый элемент обрабатываем отдельно
lea esi, mas
mov al, [esi]
cmp al, "."
je rev_out
mov [ebx].info, al
:----- вводим остальные символы строки с клавиатуры до тех пор,
: пока не встретится "."
cycl:          inc esi
               mov al, [esi]
               cmp al, "."
               je rev_out
               add_list item_list, Head_list, Hand_Head
               mov [ebx].info, al

```

```

        jmp      cyc1
;----- вывод строки в обратном порядке
rev_out:  lea     esi, mas_rev
        mov     ebx, Head_list
cyc12:   mov     al, [ebx].info
        mov     [esi], al
        inc     esi
        mov     ebx, [ebx].next
        cmp     ebx, 0fffffffh
        jne     cyc12
;----- выводим инвертированную строку на экран
;....

```

Недостаток такого способа построения списка — в порядке включения элементов. Хорошо видно, что он является обратным в очередности поступления элементов. Для исправления данной ситуации можно, конечно, ввести еще один указатель на последний элемент списка. Есть и другой вариант — организация двусвязного списка. Уделим теперь внимание другим операциям с односвязным списком.

Включение в список

Для включения нового элемента в список необходимо предварительно локализовать элемент, до или после которого будет производиться это включение. Для того чтобы вставить новый элемент после локализованного, необходимо выполнить действия, отраженные фрагментом кода:

```

item_list  struc                ; элемент списка
next       dd      0            ; адрес следующего элемента
info       db      0            ; содержательная часть (у нас — символ)
ends

;----- предполагаем, что адрес локализованного элемента
; находится в регистре EBX, а адрес нового элемента — в EAX
;....
mov     edx, [ebx].next
mov     [eax].next, edx
mov     [ebx].next, eax
;....

```

При вставке нового элемента после локализованного возникает проблема, источник которой в однонаправленности односвязного списка, так как, по определению, проход к предшествующему элементу невозможен. Можно, конечно, включить новый элемент, как показано выше, а затем попросту обменять содержимое этих элементов. К примеру, это может выглядеть так:

```

item_list  struc                ; элемент списка
next       dd      0            ; адрес следующего элемента
info       db      0            ; содержательная часть (у нас — символ)
ends

;----- предполагаем, что адрес локализованного элемента находится
; в регистре EBX, а адрес нового элемента — в EAX
;....
mov     edx, [ebx].next
mov     [eax].next, edx
mov     edx, [ebx].info
mov     [eax].info, edx
mov     [ebx].next, eax
;----- осталось заполнить поле info нового элемента
;....

```

Но если производится включение в упорядоченный список, то логичнее работать с двумя указателями — на текущий и предыдущий элементы списка так, как

это показано ниже. При этом предполагается, что новый элемент создается описанной в предыдущей программе макрокомандой `create_item`.

```

item_list      struc                : элемент списка
next           dd      0            : адрес следующего элемента
info           db      0            : содержательная часть (у нас – символ)
ends
.data
ins_item       item_list <.,15>    : вставляемый элемент (поле info
                                   : содержит значение – критерий вставки)
Head_list      dd      0fffffffh    : указатель на начало списка
                                   : (0fffffffh – список пуст)
Hand_Head      dd      0            : переменная для дескриптора кучи
.code
;----- Инициализация списка и работа с ним.
;       список упорядочен по возрастанию
;...
;----- ищем место вставки
;----- 1 – выбираем первую ячейку
mov     ebx, Head_list
xor     eax, eax                  : будет указатель на предыдущий элемент
m1:     cmp     ebx, 0fffffffh    : 2 - последняя ячейка?
        je      no_item          : список пуст
;----- 3 – новая ячейка до очередной выбранной?
mov     dl, [ebx].info
cmp     dl, ins_item.info
ja      next_item
test    eax, eax
jnz     into
;----- вставить первым
create_item item_list, Hand_Head : создание элемента в куче
mov     Head_list, edx           : адрес нового в голову списка
mov     [edx].next, ebx          : настройка указателей
jmp     exit                     : на выход
;----- вставить внутри списка
into:    create_item item_list, Hand_Head : создание элемента в куче
mov     [eax].next, edx          : адрес нового в поле next предыдущего
mov     [edx].next, ebx          : в поле next нового адрес текущего
jmp     exit                     : на выход
;----- выбор очередного элемента
next_item: mov     eax, ebx          : адрес текущего в eax
          mov     ebx, [ebx].next
          jmp     m1
;----- 4 – список пуст или нет больше элементов
no_item:  test     eax, eax
          jnz     no_empty         : список непустой
;----- список пуст
mov     Head_list, edx           : адрес нового в голову списка
mov     [edx].next, 0fffffffh    : пока единственный элемент
jmp     exit                     : на выход
;----- список не пуст – новая ячейка в конец списка
no_empty: mov     [eax].next, edx
          mov     [edx].next, 0fffffffh : последний элемент в списке
;----- выход
exit:     ;...

```

Исключение из списка

Алгоритм предполагает наличие двух указателей — на текущую и предыдущую ячейки. Для разнообразия этот алгоритм подразумевает прямой порядок следования элементов в списке. В качестве упражнения читателю предлагается, если нужно, переписать приводимый ниже фрагмент для организации обратного порядка следования элементов.

```

item_list    struc                ; элемент списка
next         dd      0            ; адрес следующего элемента
info         db      0            ; содержательная часть (у нас – символ)
ends
.data
search_b     db      0            ; критерий поиска (поле info экземпляра
                                   ; структуры item_list)
Head_list    dd      0fffffffh    ; указатель на начало списка
                                   ; (0fffffffh – список пуст)

.code
;----- Инициализация списка и работа с ним.
;       список упорядочен по возрастанию
;....
;----- ищем ячейку, подлежащую удалению
;----- 1 – выбираем первую ячейку
mov     ebx, Head_list
xor     eax, eax                ; будет указатель на предыдущую ячейку
;----- 2 - последняя ячейка?
cmp     ebx, 0fffffffh
je      no_item
;----- сравниваем с критерием поиска
cysl:   mov     dl, search_b
        cmp     [ebx].info, dl
        jne     ch_next_item    ; нашли?
;----- да, нашли!
        test    eax, eax
        jnz     no_fist         ; это не первая ячейка
;----- первая ячейка(?) => изменяем указатель
;       на начало списка и на выход
        mov     edx, [ebx].next
        mov     Head_list, edx
        jmp     exit
no_fist: mov     [eax].next, ebx ; перенастраиваем указатели =>
                                   ; элемент удален
        jmp     exit           ; на выход
;----- выбор следующего элемента
ch_next_item: mov     eax, ebx    ; запомним адрес текущей ячейки
                                   ; в указателе на предыдущую
        mov     ebx, [ebx].next ; адрес следующего элемента
        jmp     cysl
;----- обработка ситуации отсутствия элемента
no_item: ;....

```

Остальные обозначенные нами выше операции очевидны и не должны вызвать у читателя трудностей в реализации.

Другой интересный и полезный пример использования односвязных списков — работа с разреженными массивами. Разреженные массивы представляют собой массивы, у которых много нулевых элементов. Представить разреженный массив можно двумя способами: с использованием циклических списков с двумя полями связи (рис. 2.12) и посредством хэш-таблицы.

С разреженными массивами можно работать, используя методы хэширования. Для начала нужно определиться с максимальным размером хэш-таблицы (M) для хранения разреженного массива. Это значение будет зависеть от максимально возможного количества ненулевых элементов. Ключом для доступа к элементу хэш-таблицы выступает пара индексов (i, j) , где $i = 1 \dots p, j = 1 \dots q$. Числовое значение ключа вычисляется по одной из следующих формул:

$$K = i + q \cdot (j - 1) - 1,$$

$$K = j + p \cdot (i - 1) - 1.$$

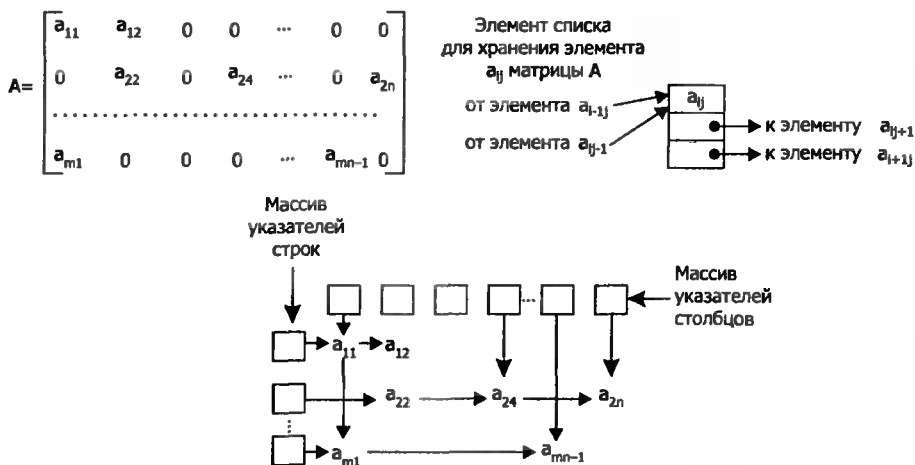


Рис. 2.12. Структура хранения разреженной матрицы при помощи списков

Остальные действия зависят от избранного способа вычисления хэш-функции (см. выше соответствующий раздел о методах хэширования).

Двусвязные списки

Двусвязный список представляет собой список, связующая часть которого состоит из двух полей. Одно поле указывает на левого соседа данного элемента списка, другое — на правого. Кроме этого, со списком связаны два указателя — на голову и хвост списка (рис. 2.13, сверху). Более удобным может быть представление списка, когда функции этих указателей выполняет один из элементов списка, одно из полей связи которого указывает на последний элемент списка (рис. 2.13, снизу).

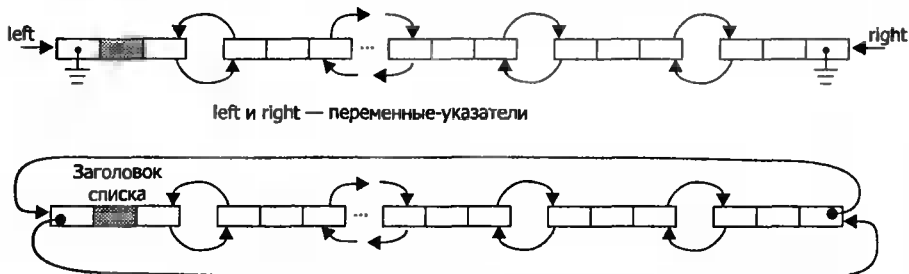


Рис. 2.13. Схемы организации двусвязных списков

Преимущество использования двусвязных списков — в свободе передвижения по списку в обоих направлениях, в удобстве исключения элементов. Возникает вопрос о том, как определить начало списка. Для этого существуют по крайней мере две возможности: определить два указателя на оба конца списка или определить голову списка, указатели связей которой адресуют первый и последний элемент списка. В случае двусвязного списка добавление и удаление элементов производится простой перенастройкой указателей. Проиллюстрируем это на примере задачи с обратным выводом вводимой с клавиатуры строки, рассмотренной в раз-

деле, посвященном односвязным спискам. Это позволит сравнить порядок выполнения типовых операций над списками и эффективность работы с ними. Структуру списка определим как двусвязный список с головой, содержащей указатели на первый и последний элементы списка.

```

+-----+
+| Программа: prg15_102.asm. Инвертирование строки (двусвязные списки). |
+-----+
+| Вход: символьная строка с клавиатуры. |
+-----+
+| Выход: вывод на экран инвертированной обратной строки. |
+-----+
item_list      struc          ; элемент списка
prev           dd            0      ; адрес предыдущего элемента
info           db            0      ; содержательная часть у нас – символ)
next           dd            0      ; адрес следующего элемента
ends
Head_list      struc          ; заголовок списка
first          dd            0      ; адрес первого элемента списка
info           db            0ffh   ; 0ffh – признак заголовка списка
last           dd            0      ; адрес последнего элемента списка
ends
.data
mas            db            80 dup (' ') ; в эту строку вводим
mas_rev        db            80 dup (' ') ; из этой строки выводим
len_mas_rev    = $ - mas_rev
mes1           db            'Введите строку символов (до 80) для инвертирования '
               db            '(с точкой на конце):'
               len_mes1 = $ - mes1
.code
;----- описание макрокоманд работы со связанным списком
; (см. файлы, прилагаемые к книге)
;----- создание двусвязного списка
create_doubly_list macro head:REQ
;...
endm
;----- добавление элемента в двусвязный список
add_list        macro descr:REQ, head:REQ, H_Head:REQ
;...
endm
start           proc near          ; точка входа в программу
;...
;----- вывод строки текста – приглашение
; на ввод строки для инвертирования
;...
;----- вводим строку текста для инвертирования
;...
;----- создаем связанный список, для чего
; инициализируем заголовок списка
create_doubly_list Doubly_Head_list
;----- вводим символы строки с клавиатуры до тех пор.
; пока не встретится "."
lea esi, mas
cyc1:           mov al, [esi]
               cmp al, "."
               je rev_out
               add_list item_list, Doubly_Head_list, Hand_Head
               mov [ebx].info, al
               inc esi
               jmp cyc1
;----- вывод строки в обратном порядке
rev_out:        lea esi, mas rev
               mov ebx, Doubly_Head_list.last
cyc12:          mov al, [ebx].info

```

```

mov     [esi], al
inc     esi
mov     ebx, [ebx].prev
cmp     [ebx].info, 0ffh ; дошли до последнего элемента списка?
jne     cyc12
:----- выводим инвертированную строку на экран
:....

```

Включение в список

Для добавления нового элемента в список необходимо предварительно локализовать элемент, до или после которого будет производиться это включение. Для того чтобы вставить новый элемент после локализованного, то есть «справа», необходимо выполнить действия, отраженные фрагментом кода:

```

item_list      struc          ; элемент списка
prev           dd             0          ; адрес предыдущего элемента
info           db             0          ; содержательная часть у нас – символ)
next           dd             0          ; адрес следующего элемента
ends

:----- предполагаем, что адрес локализованного элемента
: находится в регистре EBX, а адрес нового элемента – в EAX
:....
push         [ebx].next
pop          [eax].next          ; [ebx].next->[eax].next
mov          [eax].prev, ebx     ; адрес предыдущего элемента->[eax].prev
mov          [ebx].next, eax     ; адрес следующего элемента->[ebx].next
:----- будьте внимательны – меняем указатель предыдущего элемента
: в следующем за новым элементом
mov          ebx, [eax].next     ; адрес следующего элемента->[ebx].next
mov          [ebx].prev, eax     ; адрес предыдущего элемента->[ebx].prev
:....

```

Включение элемента перед локализованным элементом выполняется аналогично. Простота работы с двусвязными списками в отличие от односвязных достигается за счет отсутствия проблем с передвижением по списку в обе стороны.

Исключение из списка

Аналогично включению, исключение из двусвязного списка не вызывает проблем и достигается перенастройкой указателей. Фрагмент программы, демонстрирующей эту перенастройку, показан ниже.

```

item_list      struc          ; элемент списка
prev           dd             0          ; адрес предыдущего элемента
info           db             0          ; содержательная часть у нас – символ)
next           dd             0          ; адрес следующего элемента
ends

:----- предполагаем, что адрес локализованного элемента
: находится в регистре EBX, а адрес нового элемента – в EAX
:....
mov          eax, [ebx].prev     ; адрес предыдущего элемента -> eax
push         [ebx].next
pop          [eax].next
mov          eax, [ebx].next     ; адрес следующего элемента -> eax
push         [ebx].prev
pop          [eax].prev
:....

```

В общем случае одно- и двусвязные списки представляют собой линейные связанные структуры. Но в некоторых ситуациях второй указатель двусвязного списка может задавать порядок следования элементов, не являющийся обратным по отношению к первому указателю (рис. 2.14).

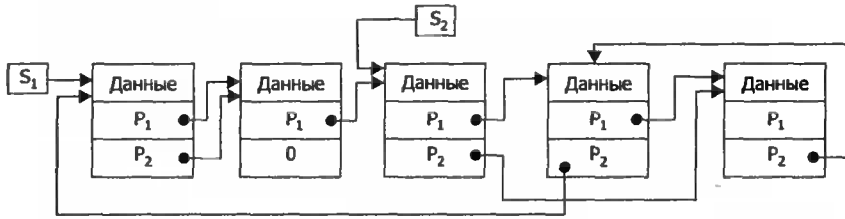


Рис. 2.14. Логическая структура нелинейного двусвязного списка

Граф

Неважно, что кто-то идет неправильно,
возможно, это хорошо выглядит...

Первый закон Скотта (Прикладная Мерфология)

Выше мы уделили достаточно внимания работе с матрицами и списками, и это сделано не случайно. Еще более обобщая понятие списка, мы придем к определению многосвязного списка, для которого характерно наличие произвольного количества указателей на другие элементы списка. Можно говорить, что каждый конкретный элемент входит в такое количество односвязных списков, сколько у него есть указателей. Многосвязный список получается «прошитым» односвязными списками, поэтому такие многосвязные списки называют *прошитыми*, или *плексами*. С помощью многосвязных списков можно моделировать такую структуру данных, как *граф (сеть)*. Частный случай графа — дерево. Рассмотрим варианты представления в памяти компьютера этих структур данных.

Графом называется кортеж $G = (V, E)$, где V — конечное множество вершин, E — конечное множество ребер, соединяющих пары вершин из множества V . Две вершины u и v из множества V называются *смежными*, если в множестве E существует ребро (u, v) , соединяющее эти вершины. Граф может быть *ориентированным* и *неориентированным*. Это зависит от того, считаются ли ребра (u, v) и (v, u) разными. В практических приложениях часто ребрам приписываются веса, то есть некоторые численные значения. В этом случае граф называется *взвешенным*. Для каждой вершины v у графа есть множество смежных вершин, то есть таких вершин u_i ($i = 1 \dots n$), для которых существуют ребра (v, u_i) . Это далеко не все определения, касающиеся графа, но для нашего изложения их достаточно, так как его цель — иллюстрация средств ассемблера для работы с различными структурами данных. Поэтому рассмотрим варианты представления графа в памяти компьютера в виде, пригодном для обработки. Какой из этих вариантов лучше, зависит от конкретной задачи. Мы также не будем проводить оценку эффективности того или иного вида представления.

■ **Матрица смежности.** Граф, имеющий M вершин, можно представить в виде матрицы размерностью $M \times M$. При условии, что вершины помечены как $v_1, v_2, \dots, v_m$, значение матрицы $a_{ij} = 1$ говорит о существовании ребра между вершинами v_i и v_j . Иначе говоря, эти вершины являются смежными. Для ориентированного графа матрица смежности будет симметричной.

- **Матрица инцидентности.** В основе этого представления также лежит матрица, но строки в ней соответствуют вершинам, а столбцы — ребрам (рис. 2.15). Из рисунка видно, что каждый столбец содержит две единицы в строках, причем одинаковых столбцов в матрице нет.

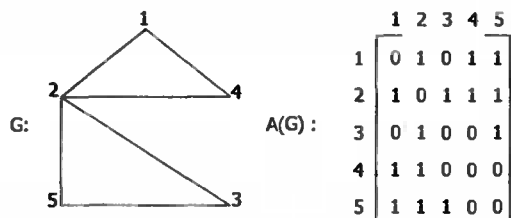


Рис. 2.15. Представление графа матрицей инцидентности

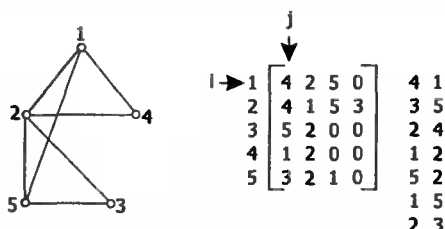


Рис. 2.16. Представление графа векторами смежности

- **Векторы смежности.** Этот вариант представления также матричный (рис. 2.16). Все вершины пронумерованы. Каждой вершине соответствует строка матрицы, в которой перечислены номера вершин, смежных с данной.
- **Списки смежностей.** Каждой вершине графа сопоставлен односвязный список, элементы которого включают в себя по крайней мере два поля — с номером смежной вершины и указателем на следующий элемент.

В качестве примера рассмотрим использование графа для представления диаграмм состояний дискретных систем. Здесь под дискретной системой понимается любая система, представляемая моделью с конечным числом состояний [16]. На вход системы может поступать некоторое количество входных переменных, которые призваны оказать определенное воздействие на дискретную систему. Реакция последней выражается в том, что устанавливаются в определенные значения выходные переменные. Значения всех переменных (входных и выходных) меняются дискретно, то есть через определенные тактовые моменты, между которыми поведение системы не изменяется. Подобные системы удобно изображать в виде «черного ящика». Элементарный пример дискретной системы — настольная лампа с выключателем. Входная переменная — положение выключателя со значениями «включено» и «выключено», выходная переменная — состояние лампы «горит» и «не горит». Строго говоря, такое определение дискретной системы больше подходит для специального класса систем, называемых конечными автоматами. Для нас они важны тем, что их используют при разработке компиляторов для распознавания строк входного языка.

Формально конечный автомат задается следующим кортежем: $M = \{K, A, P, S, F\}$, где:

- K — конечное множество состояний;
- A — конечный входной алфавит;
- P — множество переходов;
- S — начальное состояние;
- F — множество последних состояний.

Действительно, представим, что есть некоторая программа — сканер, на вход которой поступают лексемы из исходного текста программы. Назначение сканера — дать ответ о принадлежности некоторой лексемы входному языку компилятора. Принципиально сканер функционирует так. Он имеет конечное множество состояний, одно из которых является начальным S , и несколько конечных F . Перед началом обработки символов очередной лексемы сканер находится в состоянии S . По мере считывания очередной литеры лексемы состояние сканера меняется. Эти переходы также заранее определены множеством правил перехода P . После окончания чтения лексемы автомат должен оказаться в одном из конечных состояний, некоторые из которых могут соответствовать состоянию ошибки. Исходя из того, в каком из состояний оказался сканер, он делает вывод о принадлежности лексемы входному языку. Если лексема ошибочная, то компилятором формируется соответствующее диагностическое сообщение, если же лексема корректная, то далее она в зависимости от своего типа подвергается дальнейшей обработке.

Существуют два подхода к программированию конечного автомата — задавать его поведение в табличном виде и с помощью многосвязного списка. Мы не будем обсуждать их достоинства и недостатки, тем более что у них одна цель. Воспользуемся же мы тем, что программирование поведения конечного автомата является удобным примером для иллюстрации применения многосвязных списков.

В качестве примера рассмотрим фрагмент сканера для распознавания вещественных чисел в директивах ассемблера dd , dq , dt . Правило описания этих чисел в виде синтаксической диаграммы приведено в главе 17 «Архитектура и программирование сопроцессора» учебника. Ему соответствует показанное ниже регулярное выражение и детерминированный конечный автомат (рис. 2.17):

$(+|-)dd*.dd*e(+|-)dd*$

Здесь d^* обозначает цифру 0–9 или пустое значение.

Программа будет состоять из двух частей:

- построение списка — здесь выполняется построение многосвязного списка по заданному регулярному выражению;
- обработка входной строки на предмет выяснения того, является ли она правильной записью вещественного числа в директивах ассемблера dd , dq , dt .

При реализации первого пункта возникает проблема — как задавать элемент многосвязного списка, если в общем случае различные элементы списка могут иметь разное количество связей? Как показано на рисунке, различные состояния имеют разное количество связей. Можно предложить разные подходы к выбору программной реализации этих связей. Выберем следующий. Организуем все

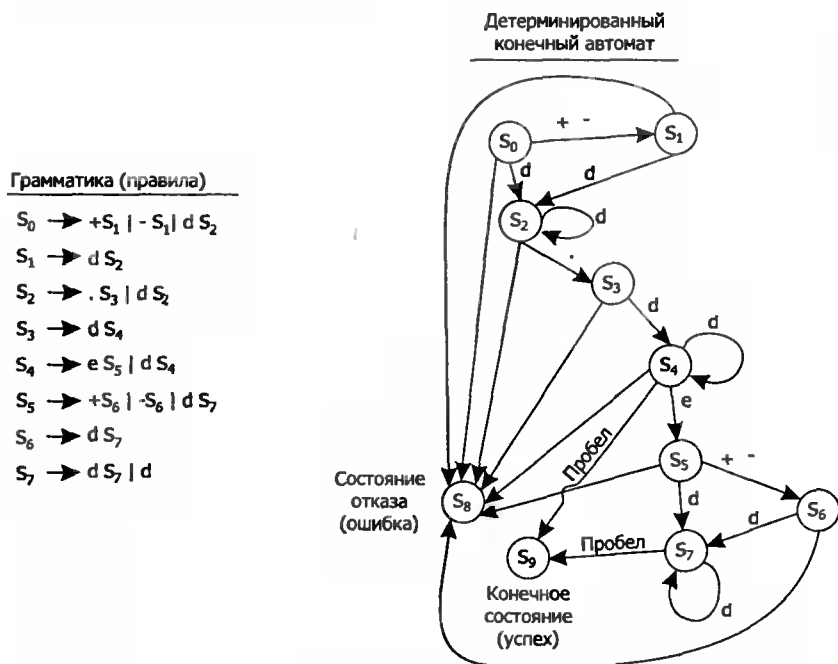


Рис. 2.17. Грамматика языка описания вещественных чисел в директиве dd и соответствующий ей детерминированный конечный автомат

исходящие связи каждого конкретного элемента в односвязный список. В этом случае многосвязный список будет содержать элементы двух типов:

- описывающие состояния автомата — они организованы в двусвязный список;
- описывающие ссылки или переходы от данного состояния к другим состояниям в зависимости от очередного входного символа — эти ссылки структурированы в виде односвязных списков для каждого из состояний.

В случае нелинейной организации двусвязного списка его построение таит ряд проблем. Часть из них снимается при статическом способе построения списка, который для конечных автоматов является наилучшим. При необходимости, используя приведенные ниже структуры, читатель может самостоятельно описать такой список в своем сегменте данных. Далее мы займемся динамическим, более сложным вариантом построения нелинейного списка, реализующим конечный автомат. Напомним, что его задачей является распознавание лексем — описания вещественного числа в директивах ассемблера dd, dq, dt. Для построения нелинейного многосвязного списка разработаем ряд макрокоманд. С их помощью построение проводится в два этапа:

- создание двусвязного списка состояний конечного автомата;
- создание односвязных списков переходов.

В приведенной ниже программе двусвязный список состояний конечного автомата строится сразу. Далее для каждого состояния, в котором может находиться автомат, конструируются односвязные списки переходов. По мере своего созда-

ния односвязные списки переходов подсоединяются к соответствующим элементам двусвязного списка состояний конечного автомата. В конце программы производится распознавание строки на предмет ее принадлежности к множеству строк, описываемых приведенным выше регулярным выражением для вещественного числа.

Создание двусвязного списка состояний конечного автомата

Опишем структуру элементов списка состояний и макрокоманду для его построения. Особенность в том, что указатель списка содержит ссылку на последний выделенный элемент списка, что на самом деле особого значения не имеет

```

item_list_state struc                ; элемент списка состояний
prev_state   dd    0                ; адрес предыдущего элемента состояния
id_state_state db    0              ; идентификатор состояния –
                                      ; двоичное значение (0...n)
current_state dd    0                ; адрес элемента текущего состояния
point_cross  dd    0                ; указатель на начало списка переходов
                                      ; для этого состояния
ends

;-----+-----+
;| Макрокоманда: create_list_state. Создание списка состояний. |
;|-----+-----+
;| вход: Hand_Head – дескриптор кучи процесса по умолчанию.   |
;| descr – имя структуры-элемента списка состояний.           |
;| N_state – количество состояний.                             |
;|-----+-----+
;| выход: head – переменная для хранения указателя на конец |
;| списка состояний.                                           |
;|-----+-----+
create_list_state macro Hand_Head:REQ, descr:REQ, head:REQ, N_state:REQ
;----- сохраняем регистры
pushad
;----- используем кучу, выделяемую процессу по умолчанию (1 Мбайт).
; но при необходимости можно создать и дополнительную кучу с помощью
; HeapCreate HANDLE и GetProcessHeap (VOID):
call GetProcessHeap
mov Hand_Head, eax ; сохраняем дескриптор
;----- запрашиваем и инициализируем блоки памяти из кучи
xor ebx, ebx ; здесь будут указатели на предыдущие элементы
xor ecx, ecx ; c1 – номер элемента состояния (двоичный)
rept N_state
;----- LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)
push ecx
push type descr ; размер структуры
push 0 ; флаги не задаем
push Hand_Head ; описатель кучи
call HeapAlloc
mov [eax].prev_state, ebx ; запоминаем адрес предыдущего
; (если ebx=0, то это первый)
mov ebx, eax ; запоминаем адрес текущего в ebx
mov [eax].current_state, eax ; и в descr.current_state
pop ecx
mov [eax].id_state_state, c1
inc c1
endm

;----- указатель на последний элемент списка состояний
; в поле-указатель на начало списка head
mov head, ebx
;----- восстанавливаем регистры
popad
endm

```

Создание односвязного списка переходов для состояния конечного автомата

Также опишем структуру элемента списка переходов и макрокоманду для создания этого элемента. Особенность в том, что, в отличие от списка состояний, макрокоманда строит не весь список, а только один его элемент. Другая особенность — указатель на полученный односвязный список является ссылкой на последний выделенный элемент списка и именно к этому элементу осуществляется привязка элемента состояния к своему списку переходов, что на самом деле особой роли не играет.

```

item_list_cross struc          ; элемент списка переходов
symbol db 0                   ; входной символ, при котором автомат
                               ; переходит в состояние ниже
                               ; (поля id_state_cross и next_item)
id_state_cross db 0           ; идентификатор целевого состояния
                               ; в списке состояний
point_state dd 0              ; адрес элемента целевого состояния
next_item dd 0                ; адрес следующего элемента в списке
                               ; переходов для этого состояния
ends

```

ends

```

;-----+-----
; Макрокоманда: create_item_cross. Создание элемента списка переходов
; для определенного состояния.
;-----+-----

```

```

; вход: EBX — адрес предыдущего (для поля descr.next_item).
;        Для первого должен быть равен нулю.
; sim — символ ASCII, при поступлении которого производится переход
;       в состояние state.
; descr — имя структуры-элемента списка переходов.
; state — номер состояния, в которое производится переход.
;         (если двузначное, то в скобках <>).
; head — имя переменной для указателя на конец списка состояний.
; Hand_Head — дескриптор кучи процесса по умолчанию.
;-----+-----

```

```

; выход: EBX — адрес созданного элемента списка переходов.
; CF = 1 — ошибка: нет такого состояния.
;-----+-----

```

```

create_item_cross macro sim:REQ, state:REQ, descr:REQ, head:REQ, Hand_Head:REQ

```

```

    local ml, @cyc1, exit_m
;----- сохраняем регистры
    push    eax
;----- запрашиваем и инициализируем блок памяти из кучи
; LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)
    push    type descr      ; размер структуры
    push    0               ; флаги не задаем
    push    Hand_Head       ; описатель кучи
    call    HeapAlloc
    mov     [eax].next_item, ebx ; адрес предыдущего
    mov     ebx, eax         ; запоминаем адрес текущего
    mov     [eax].symbol, "&sim" ; инициализируем поле symbol
                                ; текущего элемента
    mov     [eax].id_state_cross, state ; номер состояния в descr
;----- теперь нужно определить адрес элемента в списке состояний
; state для выполнения дальнейших переходов и инициализации
; поля point_state
    push    ebx
    mov     ebx, head
    cld
@cyc1:    cmp     [ebx].id_state_cross, state
    je      ml
    jc      exit_m

```

```

mov     ebx, [ebx].prev_state ; адрес предыдущего состояния
                                ; в списке состояний
test    ebx, ebx              ; последний элемент?
jnz     @@cyc1
stc
jmp     @@cyc1
;----- нашли!
m1:     mov     [eax].point_state, ebx
;----- восстанавливаем регистры
exit_m: pop     ebx
        pop     eax
endm

```

Далее приведена вспомогательная макрокоманда, которая по номеру состояния определяет адрес соответствующего элемента в списке состояний

```

+-----+
:| Макрокоманда: def_point_item_state.
:| Определение элемента в списке состояний по номеру состояния.
+-----+
:| Вход: N_state – номер состояния.
:| head – имя ячейки, где хранится указатель на список состояний.
+-----+
:| Выход: EBX – адрес элемента в списке состояний.
+-----+
def_point_item_state macro N_state:REQ, head:REQ
local   @@cy, @@m1
mov     eax, head
@@cy:   cmp     [eax].id_state_state, N_state
        je     @@m1          ; нашли?
        mov    eax, [eax].prev_state ; адрес следующего состояния
        test   eax, eax      ; последний элемент?
        jnz    @@cy
        stc                  ; CF=1, если состояния с таким номером
                                ; не существует
        jmp    @@cy
;----- нашли!
@@m1:   pop
endm

```

Собственно программа `prg02_11.asm`, выполняющая построение и инициализацию конечного автомата для распознавания лексемы вещественного числа в директивах `dd`, `dq` и `dt`, достаточно велика. Полный ее текст можно найти среди файлов, прилагаемых к книге.

Дерево

То, что неясно, следует выяснить. То, что трудно творить, следует делать с великой настойчивостью.

Конфуций

Деревом называется сетевая структура, обладающая следующими свойствами:

- среди всех узлов существует один, у которого отсутствуют ребра, приходящие от других узлов, — этот узел называется корнем;
- все узлы, за исключением корня, имеют одно и только одно входящее ребро;
- ко всем узлам дерева имеется путь, начало которого находится в корне дерева.

Графически дерево изображают так, как показано на рис. 2.18 (как выразился классик криптографии Брюс Шнайер, «в компьютерных лесах дерева растут сверху вниз»).

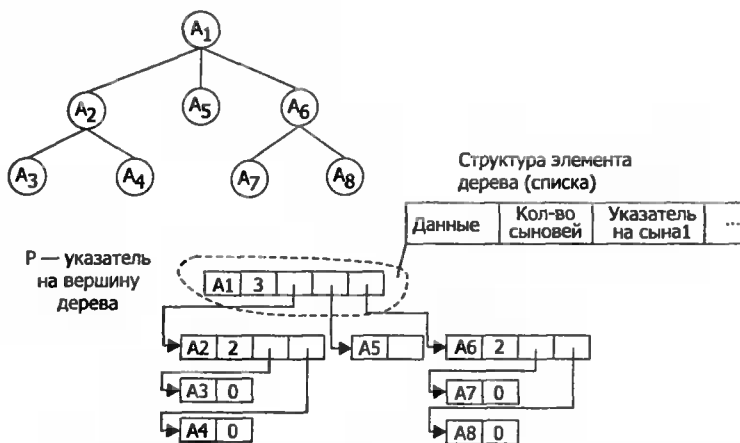


Рис. 2.18. Изображение дерева и соответствующего ему связанного списка

Следуя терминологии дерева, можно ввести некоторые определения, касающиеся его структуры. Путем из узла n_i в узел n_k называется последовательность узлов $n_1, n_2, \dots, n_k$, где для всех $i, 1 \leq i = k$ узел n_i является родителем узла n_{i+1} . Длинной пути называется число, на единицу меньшее числа узлов, составляющих этот путь. Оконечные узлы дерева, которые не ссылаются на другие узлы, называются *листьями*. Другие узлы дерева, за исключением корня, называются *узлами ветвления*. Две смежные вершины дерева состоят в «родственных» отношениях. Вершина X , находящаяся на более высоком уровне дерева по отношению к вершине Y , называется *отцом*. Соответственно, вершина Y по отношению к X называется *сыном*. Если вершина X имеет несколько сыновей, то по отношению друг к другу последние называются *братьями*. В принципе, вместо этих терминов можно использовать следующие: родитель, потомок, предок и т. п. Классификация деревьев производится по степени исхода ветвей из узлов деревьев. Дерево называется m -арным, если степень исхода его узлов не превышает значения m .

В практических приложениях широко используется специальный класс деревьев — *бинарные деревья*. Бинарное дерево — m -арное дерево при $m = 2$. Это означает, что степень исхода его вершин не превышает 2. В случае когда m равно 0 или 2, имеет место полное бинарное дерево. Важно то, что любое m -арное дерево можно преобразовать в эквивалентное ему бинарное дерево. В свою очередь, в силу ограничений по степени исхода вершин бинарное дерево легче представлять в памяти и обрабатывать. По этой причине мы основное внимание уделим работе с бинарными деревьями.

Перечислим операции, которые обычно выполняются при работе с деревьями:

- представление дерева в памяти;
- обход или прохождение дерева (поиск в дереве);
- включение в дерево и исключение из дерева определенного узла;
- балансировка дерева и операции со сбалансированным деревом.

Представление дерева в памяти

Естественным является представление дерева в памяти компьютера в форме многосвязного списка. Это наиболее динамичная форма. В случае постоянства структуры дерева могут использоваться и другие способы представления дерева в памяти, например в виде массива, как это было во время рассмотрения нами двоичной сортировки. В случае представления дерева в форме многосвязного списка указатель на начало списка должен указывать на элемент списка, являющийся корнем. В свою очередь, узлы дерева связываются между собой так, чтобы корень дерева был связан с корнями его поддеревьев и т. д. При этом можно выделить три случая.

- Связь узлов в дереве осуществляется таким образом (рис. 2.19, а), что каждый узел-сын содержит ссылку на вышестоящий узел (узел-отец). В этом случае корень будет содержать нулевой указатель на месте соответствующего указателя. Имея такой тип связей узлов в дереве, можно подниматься от нижних уровней дерева к его корню, что бывает полезным при реализации определенных видов синтаксического разбора.
- Каждый узел дерева содержит указатели на свои поддеревья (рис. 2.19, б), то есть каждый отец ссылается на своих сыновей. Этот способ применим для представления деревьев небольшой «арности», например бинарных деревьев. Имея указатель на корень дерева, можно достичь любого узла дерева.
- Каждый узел дерева ссылается на свои поддеревья с помощью списка, при этом каждый узел содержит два указателя — для представления списка поддеревьев и для связывания узлов в этот список (рис. 2.19, в).

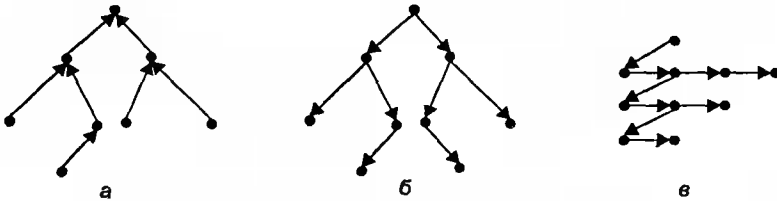


Рис. 2.19. Варианты связи узлов дерева между собой

Бинарное дерево обычно представляется с помощью второго способа (рис. 2.19, б). Минимально для представления узла достаточно трех полей: содержательной части узла и двух указателей — на левого (старшего) и правого (младшего) сыновей. Таким образом, описать бинарное двоичное дерево можно с помощью нелинейного двусвязного списка. Структура узла в программе может выглядеть так:

```
node tree      struc
symbol         db      0          : содержательная часть
l_son          dd      0          : указатель на старшего (левого) сына
r_son          dd      0          : указатель на младшего (правого) сына
ends
```

Рассмотрим на простом примере реализацию основных операций над деревом. Ранее при обсуждении сортировок упоминался двоичный поиск. При этом говорилось, что сам алгоритм совпадает с алгоритмом прохождения пути от вершины

двоичного дерева к заданной его вершине. Реализуем данный вид поиска, но уже посредством действительно двоичного дерева. Для начала построим двоичное дерево, используя произвольный массив чисел.

Построение двоичного дерева

Как уже отмечалось выше, для реализации двоичного поиска необходим упорядоченный массив чисел. Поэтому уже на этапе построения двоичное дерево необходимо специальным образом организовать. Двоичное дерево поиска организуется так, что между его узлами существуют следующие соотношения: для каждого узла *a* все элементы левого поддерева меньше *a*, а элементы правого поддерева больше либо равны *a*.

```

+-----+
:| Программа: prg02_12.asm. Построение и инициализация двоичного дерева. |
+-----+
:| Вход: произвольный массив чисел в памяти. |
+-----+
:| Выход: двоичное дерево в памяти. |
+-----+
node_tree      struc          ; узел дерева
symbol         db      0      ; содержательная часть
l_son          dd      0      ; указатель на старшего (левого) сына
r_son          dd      0      ; указатель на младшего (правого) сына
ends
.data
mas            db      37h, 12h, 8h, 65h, 4h, 54h, 11h, 02h, 32h, 5h
               db      4h, 87h, 7h, 21h, 65h, 45h, 22h, 11h, 77h, 51h
               db      26h, 73h, 35h, 12h, 49h, 37h, 52h
               l_mas = $ - mas

.code
BuildBinTree   proc
:----- формируем корень дерева и указатель на дерево HeadTree.
: Для размещения дерева используем кучу, выделяемую процессу
: по умолчанию (1 Мбайт), но при необходимости можно создать
: и дополнительную кучу с помощью HeapCreate
:----- HANDLE GetProcessHeap (VOID)
               call    GetProcessHeap
               mov     Hand_Head, eax ; сохраняем дескриптор
:----- запрашиваем и инициализируем блок памяти
: из кучи для корня дерева
               xor     ebx, ebx      ; будет указатель на предыдущий узел
:----- LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)
               push    type node_tree ; размер структуры для узла дерева
               push    0             ; флаги не задаем
               push    eax           ; описатель кучи
               call    HeapAlloc
               mov     HeadTree, eax ; запоминаем указатель на корень
               mov     ebx, eax      ; и в ebx
:----- подчистим выделенную область памяти в куче
               push    ds
               pop     es
               mov     edi, eax
               mov     ecx, type node_tree
               xor     al, al
               cld
               rep stosb
               lea     esi, mas      ; первое число из mas в корень дерева
               lodsb             ; число в al
               mov     [ebx].symbol, al
:----- далее в цикле работаем с деревом и массивом mas

```

```

mov     ecx, mas - 1
;----- запрашиваем блок памяти из кучи для узла дерева:
; LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes)
@@cyc11: push    ecx           ; HeapAlloc портит ecx, возвращая
                                ; в нем некоторое значение
        push    type node_tree ; размер структуры для узла дерева
        push    0              ; флаги не задаем
        push    Hand Head      ; описатель кучи
        call    HeapAlloc
        pop     ecx
        mov     ebx, eax        ; запоминаем указатель на узел дерева в ebx
        mov     NewNode, eax    ; и во временную переменную
;----- подчистим выделенную область памяти в куче
        mov     edi, eax
        push    ecx
        mov     ecx, type node_tree
        xor     al, al
        cld
        rep     stosb
        pop     ecx
;----- читаем очередное число из mas и записываем его в новый узел
        lodsb                ; число в al
        mov     [ebx].symbol, al
;----- ищем место в дереве согласно условиям упорядочивания
; и настраиваем указатели в узлах дерева
        mov     ebx, HeadTree
m_search: cmp     al, [ebx].symbol
        mov     edi, ebx        ; продублируем
        jae     ml              ; если al >= [ebx].symbol
;----- если меньше, то идем по левой ветке
        mov     ebx, [ebx].l_son
        test    ebx, ebx
        jnz     m_search
;----- если этого сына нет, то добавляем его к «папе»
        mov     ebx, NewNode
        mov     [edi].l_son, ebx
        jmp     end_cyc11
;----- если больше или равно, то по правой ветви
ml:      mov     ebx, [ebx].r_son
        test    ebx, ebx
        jnz     m_search
;----- если этого сына нет, то добавляем его к «папе»
        mov     ebx, NewNode
        mov     [edi].r_son, ebx
end_cyc11: loop   @@cyc11
        ret                                ; двоичное дерево поиска построено
BuildBinTree endp
start      proc    near                    ; точка входа в программу
        call    BuildBinTree
        :...
end

```

В результате мы должны получить дерево, обойдя которое в определенном порядке, сформируем упорядоченный массив чисел:

2h, 4h, 4h, 5h, 7h, 8h, 11h, 11h, 12h, 12h, 21h, 22h, 26h, 32h, 35h, 37h, 37h, 45h, 49h, 51h, 52h, 54h, 65h, 65h, 73h, 77h, 87h.

Убедимся в этом, разработав программу обхода двоичного дерева.

Обход узлов дерева

Возможны три варианта обхода дерева:

■ сверху вниз — корень, левое поддерево, правое поддерево;

- слева направо — левое поддереву, корень, правое поддереву;
- снизу вверх — левое поддереву, правое поддереву, корень.

Понятно, что процедура обхода рекурсивна. Под термином «обход дерева» понимается то, что в соответствии с одним из определенных выше вариантов посещается каждый узел дерева и с его содержательной частью выполняются некоторые действия. Для нашей задачи годится только второй вариант, так как только в этом случае порядок посещения узлов будет соответствовать упорядоченному массиву. В качестве полезной работы будем извлекать значение из узла двоичного дерева и выводить его в массив в памяти. Отметим особенности функции LRBeat, производящей обход дерева. Во-первых, она рекурсивная, во-вторых, в ней для хранения адресов узлов используется свой собственный стек.

```

:-----+
:| Програма: prg02_13.asm. Обход двоичного дерева поиска (слева направо). |
:-----+
:| Вход: двоичное дерево в памяти (строится по массиву чисел). |
:-----+
:| Выход: массив чисел в памяти. |
:-----+
node tree      struc      : узел дерева
symbol         db         0      : содержательная часть
l_son          dd         0      : указатель на старшего (левого) сына
r_son          dd         0      : указатель на младшего (правого) сына
ends
desc_stk       struc      : дескриптор программного стека
:....
ends

:----- описание макрокоманд работы со стеком
:----- создание стека
create_stk     macro      HandHead:REQ, descr:REQ, SizeStk:<=256>
:....
endm

:----- добавление элемента в стек
push_stk      macro      descr:REQ, rg_item:REQ
:....
endm

:----- извлечение элемента из стека
pop_stk       macro      descr:REQ, rg_item:REQ
:....
endm

.data
mas           db         37h, 12h, 8h, 65h, 4h, 54h, 11h, 02h, 32h, 5h, 4h
              db         87h, 7h, 21h, 65h, 45h, 22h, 11h, 77h, 51h, 26h
              db         73h, 35h, 12h, 49h, 37h, 52h ; исходный массив
              l mas = $ - mas
ordered_array  db         1_mas dup (0) : упорядоченный массив
              : (результат см. в отладчике)
              :....
doubleWord_stk desc_stk <> : дескриптор стека
count_call    dd         0      : счетчик рекурсивного вызова процедуры
.code
BuildBinTree  proc          : см. prg02_12.asm
:....
:----- двоичное дерево поиска построено
ret
BuildBinTree  endp
LRBeat        proc
:----- рекурсивная процедура обхода дерева — слева направо
: (левое поддереву, корень, правое поддереву)
inc count_call : подсчет количества вызовов процедуры

```

; (для согласования количества записей
; и извлечений из стека)

```

test    ebx, ebx
jz      @@exit_p
mov     ebx, [ebx].l_son
test    ebx, ebx
jz      @@m1
push_stk doubleWord_stk, ebx
@@m1:   call LRBeat
pop_stk doubleWord_stk, ebx
jnc     @@m2
;----- подчистим за собой стек и на выход
mov     ecx, count_call
dec     ecx
pop     NewNode      ; pop "в никуда"
loop    $ - 6
jmp     @@exit_p
@@m2:   mov     al, [ebx].symbol
cld
stosb
mov     ebx, [ebx].r_son
test    ebx, ebx
jz      @@m1
push_stk doubleWord_stk, ebx
call    LRBeat
@@exit_p: dec count_call
ret
LRBeat  endp
Start   proc      near      ; точка входа в программу
call    BuildBinTree ; формируем двоичное дерево поиска
;----- обходим узлы двоичного дерева слева направо
; и извлекаем значения из содержательной части.
; нам понадобится свой стек (размер 256 байтов устроит,
; но макроопределение мы подкорректировали)
create_stk Hand_Head, doubleWord_stk
push    ds
pop     es
lea     edi, ordered_array
mov     ebx, HeadTree
push_stk doubleWord_stk, ebx ; указатель на корень в наш стек
call    LRBeat
;....

```

Еще одно замечание о рекурсивном механизме. Реализация рекурсии в программах ассемблера лишена той комфортности, которая характерна для языков высокого уровня. В данном варианте реализации для рекурсивной процедуры LRBeat возникает несбалансированность стека. В результате этого после последнего ее выполнения на вершине стека лежит не тот адрес и команда RET отработывает неверно. Для устранения подобного эффекта нужно вводить корректирующий код, суть которого заключается в следующем. Процедура LRBeat подсчитывает количество обращений к ней и легальных, то есть через команду RET, выходов из нее. При последнем выполнении анализируется счетчик обращений count_call и производится корректировка стека. Данный способ подходит не для всех возможных практических задач. Поэтому лучшим способом будет аккуратное поддержание стека в процессе рекурсивного вызова процедур. Предлагаю читателям самостоятельно реализовать этот вариант программы.

Для полноты изложения осталось только показать, как изменится процедура LRBeat для других вариантов обхода дерева. Варианты обеих операций, UDBeat и RLBeat, можно найти среди файлов, прилагаемых к книге.

Построенное выше бинарное дерево теперь можно использовать для дальнейших операций, в частности поиска. Для достижения максимальной эффективности поиска необходимо, чтобы дерево было сбалансированным. Так, дерево считается идеально сбалансированным, если число вершин в его левых и правых поддеревьях отличается не более чем на единицу. Более подробные сведения о сбалансированных деревьях вы можете получить, изучая соответствующую литературу [4]. Здесь же будем считать, что сбалансированное дерево уже построено. Разберемся с тем, как производить включение и исключение узлов в подобном, заранее построенном упорядоченном дереве.

Включение узла в упорядоченное бинарное дерево

Задача включения узла в дерево уже была решена нами при построении дерева. Осталось оформить соответствующий фрагмент программы в виде отдельной процедуры `addNodeTree`. Чтобы не дублировать код, разработаем рекурсивный вариант процедуры включения — `addNodeTree`. Он хоть и не так нагляден, как нерекурсивный вариант кода в программе `prg_03.asm`, но выглядит достаточно профессионально. Текст процедуры `addNodeTree` вы найдете среди файлов, прилагаемых к книге.

Работу процедуры `addNodeTree` можно проверить с помощью программы `prg02_13.asm` (в файлах, сопровождающих книгу, ей соответствует программа `prg02_14.asm`).

В результате проведенных работ мы получили дублирование кода, строящего и дополняющего дерево. В принципе, для этого достаточно одной процедуры `addNodeTree`. Но в учебных целях мы ничего изменять не будем, чтобы иметь два варианта построения дерева — рекурсивный и нерекурсивный. При необходимости читатель сам определит, какой из вариантов более всего отвечает его задаче, и в соответствии с ней доработает код.

Исключение узла из упорядоченного бинарного дерева

Эта процедура более сложная. Причиной тому то обстоятельство, что существует несколько вариантов расположения исключаемого узла в дереве: узел представляет собой лист, узел имеет одного потомка и узел имеет двух потомков. Самый непростой случай — последний. Процедура удаления элемента `delNodeTree` является рекурсивной и, более того, содержит вложенную процедуру `del`, которая также является рекурсивной. Текст процедуры `delNodeTree` находится среди файлов, прилагаемых к книге.

Работу этой процедуры можно проверить с помощью программы `prg02_13.asm` (в комплекте файлов, прилагаемых к книге, ей соответствует программа `prg02_15.asm`).

Графически процесс исключения из дерева выглядит так, как показано на рис. 2.20. Исключаемый узел помечен стрелкой.

Для многих прикладных задач, ориентированных на обработку символьной информации, важное значение могут иметь так называемые лексикографические деревья. Для нашего изложения они важны тем, что находят эффективное применение при разработке трансляторов, чему мы также уделим внимание в этой книге.

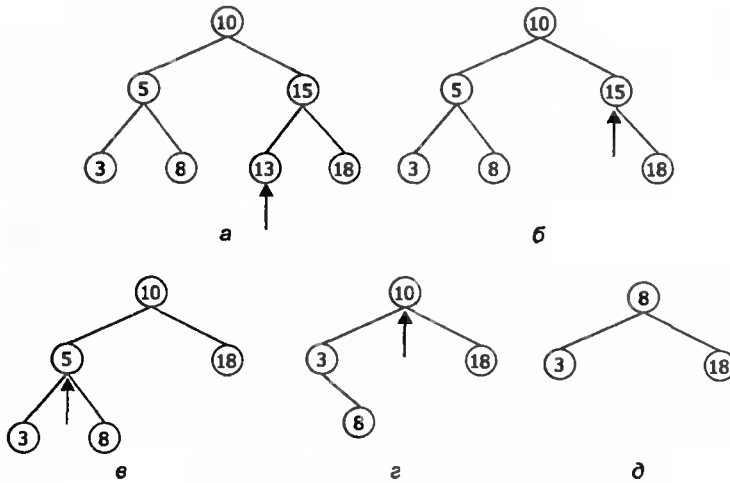


Рис. 2.20. Исключение из дерева

Лексикографическое дерево

Деревья поиска, подобные лексикографическим, являются альтернативой методам поиска, основанным на хэшировании. Структурно лексикографическое дерево представляет собой бинарное дерево поиска. Рассмотрим задачу, которая заключается в анализе некоторого текстового файла и сборе статистики о частоте встречаемости слов в этом тексте. В качестве итога требуется вывести на экран информацию о частоте повторяемости каждого слова. В такой постановке задачи программа будет состоять из следующих частей.

- Чтение текстового файла и построение соответствующего ему лексикографического дерева. Для этого будем читать и проверять на предмет новизны каждое слово в анализируемом тексте. Если слово встретилось впервые, то необходимо сформировать и внести в дерево узел, который структурно состоит из двух полей: поля для хранения самого слова и поля для хранения частоты встречаемости этого слова в тексте.
- Для вывода на экран статистики требуется совершить левосторонний обход дерева. Если изменить условия вывода статистики, например упорядочить слова по частоте повторяемости в тексте, то необходимо выполнить перестройку дерева и его последующий правосторонний обход.

Для простоты преобразования положим, что каждое слово может встречаться в тексте не более 9 раз. Длина слова — не более 10 символов (для упрощения контроль количества вводимых символов не производится). Слова в файле разделяются одним пробелом. Также для сокращения текста программы считаем, что имя входного файла фиксированно — `infile.txt`.

Программа построения лексикографического дерева `prg_12_78.asm` имеет достаточно большой размер, полный ее текст можно найти среди файлов, прилагаемых к книге.

Бинарные деревья — не единственный вид деревьев, которые могут быть использованы в ваших программах. В литературе [2, 4, 13] можно ознакомиться с представлениями деревьев, отличных от бинарных. Мы не будем развивать далее проблему представления деревьев, хотя, честно говоря, очень хочется. За кадром остались такие важные проблемы, как балансировка деревьев, работа с В-деревьями и т. д. После столь обстоятельного введения и при наличии соответствующей литературы вы сможете без особого труда решить эти и другие проблемы. Тем более, что на худой конец любое дерево поддается преобразованию в бинарное и работе с ним при помощи приемов, описанных в настоящем разделе.

В преддверии рассмотрения следующего материала отметим, что разработанная в этом разделе программа будет очень полезна в процессе построения различных транслирующих программ, частным случаем которых являются компиляторы. Лексикографические деревья можно с успехом использовать в процессе работы сканера, задачей которого, в частности, является выяснение принадлежности некоторого идентификатора к ключевым словам некоторого языка. Введите в файл `infile.txt` список ключевых слов (разделенных одним пробелом), запустите программу и в памяти вы получите лексикографическое дерево.

Глава 3

Процедуры в программах на ассемблере

Если бы строители строили здания так же, как программисты пишут программы, первый залетевший дятел разрушил бы цивилизацию.

Второй закон Вейнберга (Прикладная Мерфология)

В учебнике достаточно полно был рассмотрен вопрос организации работы с процедурами, но некоторые проблемы остались за кадром. В этой главе мы остановимся на трех из них: реализации рекурсивных и вложенных процедур на ассемблере, а также разработке динамических библиотек (DLL).

Реализация рекурсивных процедур

В предыдущей главе, посвященной структурам данных, использовались рекурсивные процедуры. Данный вопрос требует дополнительного пояснения.

Процедура называется *рекурсивной*, если она прямо или косвенно обращается к себе самой. Рекурсия является естественным свойством для большого числа математических и вычислительных алгоритмов. Важно отметить, что любой рекурсивный алгоритм можно сделать итеративным, но это не всегда целесообразно. В теории программирования рекурсия, как правило, воспринималась неоднозначно. В конечном итоге была выработана следующая рекомендация — рекурсию следует избегать в случаях, когда имеется очевидное итерационное решение. Как мы убедимся из приведенного ниже обсуждения, очевидная рекурсивная задача вычисления факториала не дает никакого выигрыша относительно другой программной реализации. Настоящий эффект возникает в тех задачах, где рекурсия используется в определении обрабатываемых данных. Такими данными могут являться, например, динамические структуры данных — стеки, деревья, списки, очереди и т. п.

Как показал материал, посвященный структурам данных, это действительно верно. При небольшом усилии в программе на ассемблере можно достаточно эффективно организовать рекурсивную процедуру, подобную процедуре обхода дерева LRBeat из главы 2.

В ассемблере нет средств прямой поддержки рекурсивных алгоритмов, но есть косвенные — нужно только немного подыграть этому процессу. По сравнению с языками высокого уровня, имеющими встроенную поддержку рекурсии, в программе ассемблера все действия по ее организации приходится предусматривать самому программисту. А для этого необходимо иметь представление о процессах, происходящих при рекурсивном вызове процедуры.

Планируя использование рекурсивных процедур, необходимо продумывать следующие вопросы:

- способы передачи параметров в процедуру и возврата результатов ее работы;
- способ сохранения локальных переменных процедуры;
- организацию выхода из процедуры.

Подробно эти вопросы обсуждались в главе 15 «Модульное программирование» учебника. Но при организации рекурсии они приобретают особый смысл, так как требуется не просто вызвать процедуру, а вызвать ее из себя несколько раз, обрабатывая при каждом вызове свои локальные данные, и в конечном итоге возвратить управление в точку программы, расположенную следом за первым вызовом данной процедуры.

Как правило, для работы с локальными данными процедуры и параметрами, передаваемыми в процедуру, задействуется стек. При этом не обязательно использовать для этого только системный стек, иногда удобнее создать его модель, работу с которой осуществлять средствами самой программы. Пример организации такого программного стека мы реализовали при написании программы обхода дерева посредством рекурсивной процедуры **LRBeat**.

Рассмотрим характерные моменты рекурсивного вызова процедур на примере классической задачи по теме — вычисления факториала. Вспомним алгоритм вычисления факториала:

$$F(0) = 1; F(i) = i \times F(i - 1)$$

Как было отмечено выше, с точки зрения скорости работы кода рекурсивный вариант вычисления факториала неэффективен, его лучше вычислять итеративно в цикле, но в учебных целях этот пример оправдан.

```

.data
r_fact    dw    0
.code
fact:
    proc
    push   bp
    mov    bp, sp
    mov    cx, [bp+4]
    mov    ax, cx
    mul    r_fact
    mov    r_fact, ax
    dec    cx
    jz     end_p
    push   cx
    call   fact
end_p:
    mov    sp, bp
    pop    bp
    ret
fact      endp
main:
    ....

```

```

mov     r_fact.1
push    word ptr 5
call    fact
:....

```

В общем-то ничего необычного в этом коде нет. Передача параметров в рекурсивную процедуру производится через стек. При этом в него необязательно помещать все данные. Возможен вариант, когда локальные данные определяют в сегменте данных или в выделенной динамической области памяти, а в стек помещается только указатель на них. В любом случае, начало рекурсивной процедуры будет содержать код пролога, подобный приведенному в программе выше:

```

fact      proc
           push    bp
           mov     bp, sp

```

Смысл этого фрагмента легче понять, наблюдая поведение программы вычисления факториала в отладчике. Как было сказано, перед вызовом процедуры в стек помещаются данные (или указатель на них), информация о местонахождении которых должна быть сохранена в интересах как вызывающей, так и вызываемой процедуры. В нашем случае в процедуру `fact` передается переменная факториала. После этого производится вызов процедуры, в результате чего в стек заносится адрес возврата. В вызванной процедуре к данным переменным необходимо получить доступ. Для этого предназначен регистр `EBP/BP`. Перед использованием его содержимое должно быть также сохранено в стеке. Для первого вызова его значение несущественно. В этот момент весь предыдущий контекст работы программы сохранен. Команда `mov bp, sp` загружает в регистр `BP` указатель на текущую вершину стека, после чего можно обращаться к данным, переданным в процедуру. По сути, сейчас мы с вами сформировали *кадр стека*. Следующий рекурсивный вызов этой функции придает действию сохранения регистра `BP` особый смысл. Команда `push bp` сохраняет в стеке адрес кадра стека для предыдущего вызова рекурсивной процедуры. Теперь для выхода из процедуры достаточно выполнить приведенные ниже команды эпилога, позволяющие корректно отработать обратную цепочку возврата в основную программу:

```

end_p:    :....
           mov     sp, bp
           pop     bp
           ret
fact      endp

```

Проследите в отладчике за тем, как происходит удаление кадров стека из процедуры и возврат в программу на первую после иницилирующей рекурсивный вызов команды `CALL` команду.

Если вам не нравится обращаться к параметрам в кадре стека посредством регистра `EBP/BP`, то вы можете использовать средства `TASM` для более удобной организации этого процесса. Они подробно описаны в главе 15 «Модульное программирование» учебника. Но суть применения этих средств та же, в чем легко убедиться с помощью глашатая истины — отладчика.

С параметрами все ясно, но как быть с локальными переменными, с которыми работает рекурсивно вызываемая функция? Для их размещения необходимо планировать место в стеке и предусматривать код, который помещает локальные переменные в стек, а затем восстанавливает их. Для примера рассмотрим программу,

которая рисует узор на основе окружностей. В основе этого графического построения лежит алгоритм [45], суть которого показывают приведенные ниже фрагменты текста процедуры на псевдоязыке:

```
// DrawPattern – рекурсивная процедура DrawPattern (вариант 1)
// вывода на экран узора из окружностей на псевдоязыке (фрагмент)
// Вход: x и y – координаты центра окружности; r – радиус окружности;
// p – порядок узора, hwnd – идентификатор окна, HDC – идентификатор контекста
VOID DrawPattern (
    HWND hwnd, HDC hdc, INT_DWORD x, INT_DWORD y, INT_DWORD r, INT_DWORD p)
ПЕРЕМЕННЫЕ
HWND hwnd; HDC hdc;
INT_DWORD hdc, x, y, r, p
НАЧ_ПРОГ
    ЕСЛИ (p) TO //пока p≠0
        НАЧ_БЛОК_1
// Ellipse – функция Win32 API для вывода эллипса (окружности), вписанного
// в прямоугольник (квадрат) с координатами правого верхнего угла (x_up, y_up)
// и левого нижнего угла (x_low, y_low):
// Ellipse
// (HDC hdc, INT_DWORD x_up, INT_DWORD y_up, INT_DWORD x_low, INT_DWORD y_low)
// так как для рисования нужны координаты прямоугольника,
// а не центра окружности, то преобразуем их при вызове Ellipse:
Ellipse(hdc, x_up-r, y_up-r, x_low+r, y_low+r)
// рисуем еще четыре окружности с центрами по краям этой окружности
DrawPattern (hwnd, hdc, x-r, y, r, p-1)
DrawPattern (hwnd, hdc, x, y-r, r, p-1)
DrawPattern (hwnd, hdc, x+r, y, r, p-1)
DrawPattern (hwnd, hdc, x, y+r, r, p-1)
        КОН_БЛОК_1
    КОН_ПРОГ
```

Вызов данной процедуры из программы на псевдоязыке может быть следующим:

```
//...
// рисовать узор 5-го порядка
DrawPattern (hwnd, hdc, 200, 140, 60, 5)
//...
```

Для того чтобы избежать рассмотрения лишних деталей, эта функция интегрирована в среду оконного приложения prg_17_1 из главы 17 «Архитектура и программирование сопроцессора» учебника. Поэтому детали реализации оконного Windows-приложения вы можете посмотреть там. Сейчас же сосредоточим все внимание на рекурсивной функции DrawPattern. Эту функцию мы реализовали в двух вариантах — без использования локальных переменных (DrawPattern_1) и с использованием локальных переменных (DrawPattern_2). Тексты обеих функций мы поместили в DLL-библиотеку maket_dll.DLL. Работа с DLL-библиотеками будет рассмотрена в этой главе ниже. Далее сравним работу функций DrawPattern_1 и DrawPattern_1.

Вызов функции DrawPattern_1 из основной программы осуществляется следующим фрагментом кода (полный текст имеется среди материалов, прилагаемых к книге, в каталоге программ для данной главы).

```
+-----+
| Программа: prg3_1.asm. Фрагмент оконного приложения. |
| вызывающего рекурсивную процедуру DrawPattern_1. |
+-----+
;----- объявление пользовательских процедур (из maket_dll.DLL)
extrn DrawPattern_1:PROC
```

```

extrn      DrawPattern_2:PROC
.data
;----- определение констант для фигуры "Узор из окружностей"
p          dd      5          ; порядок узора
r          dd      60         ; радиус окружности
y_P        dd      140        ; начальная y-координата центра окружности
x_P        dd      200        ; начальная x-координата центра окружности
.code
;-----+
;| Процедура: MenuProc. Обработка сообщений от меню. |
;-----+
MenuProc    proc
arg         @hwnd:DWORD, @wparam:DWORD, @hdc:DWORD, @hbrush:DWORD
uses        eax, ebx
mov         ebx, @wparam      ; в bx идентификатор меню
;...
cmp         bx, IDM_DLL_LACES_1
je          @@idmdl1laces_1
cmp         bx, IDM_DLL_LACES_2
je          @@idmdl1laces_2
jmp         @exit
;...
@@idmdl1laces_1:
;...
;----- рисуем узор из окружностей. рекурсивная функция для рисования
; находится в DLL-модуле:
; DrawPattern_1(hwnd, hdc, x, y, r, p) -
; функция не работает с локальными переменными
push        p                ; порядок узора
push        r                ; радиус окружности
push        y_P              ; y-координата центра окружности
push        x_P              ; x-координата центра окружности
push        memdc            ; идентификатор контекста устройства
push        @hwnd
call        DrawPattern_1
jmp         @exit
;...

```

Фрагмент файла **maket_dll.DLL**, содержащий процедуру **DrawPattern_1**, приведен ниже:

```

;-----+
;| Библиотека: maket_dll.DLL. |
;| Фрагмент DLL, содержащей рекурсивную процедуру DrawPattern_1. |
;-----+
;----- объявление процедур DLL-библиотеки общедоступными
publicdll  WriteCon
publicdll  DrawPattern_1
publicdll  DrawPattern_2
.code
;-----+
;| Процедура: DrawPattern_1. Рекурсивная процедура рисования узора |
;| (без использования локальных переменных). |
;-----+
DrawPattern_1 proc
arg         @hwnd:dword, @hdc:dword, @x:dword, @y:dword, @r:dword, @p:dword
;----- рисуем окружность
; рекурсивно вызываем DrawPattern_1(hwnd,hdc,x,y,r,p)
; BOOL Ellipse (HDC hdc, int nLeftRect, int nTopRect, int nRightRect,
;              int nBottomRect);
; готовим параметры в стеке для вызова Ellipse
;...
call        Ellipse          ; рисуем окружность
;----- и еще четыре меньшего порядка
dec         @p

```

```

        cmp     @@p, 0
        je      @@End_Draw
        shr     @@r, 1          : делим на 2
:----- (x-r,y)
        :...                  : готовим параметры в стеке
                                : для вызова DrawPattern_1
        call    DrawPattern_1
:----- (x+r,y)
        :...                  : готовим параметры в стеке
                                : для вызова DrawPattern_1
        call    DrawPattern_1
:----- (x,y-r)
        :...                  : готовим параметры в стеке
                                : для вызова DrawPattern_1
        call    DrawPattern_1
:----- (x,y+r)
        :...                  : готовим параметры в стеке
                                : для вызова DrawPattern_1
        call    DrawPattern_1
:----- генерация сообщения WM_PAINT
        : для вывода изображения на экран
@@End_Draw:
        :...
        call    InvalidateRect
        ret
endp     DrawPattern_1

```

Такой вариант процедуры не требует внимания к параметрам, которые передаются в стеке при вызове рекурсивной процедуры, так как после возврата из нее они попросту не нужны и удаляются из стека. Но стоит нам в процедуре `DrawPattern` изменить порядок обращения к процедуре `Ellipse`, как ситуация резко меняется. Рассмотрим второй вариант организации процедуры `DrawPattern`.

```

// DrawPattern – рекурсивная процедура DrawPattern (вариант 2) вывода
// на экран узора из окружностей на псевдоязыке (фрагмент)
// Вход: x и y – координаты центра окружности; r – радиус окружности;
// p – порядок узора, hwnd – идентификатор окна, HDC – идентификатор контекста.
VOID DrawPattern (
    HWND hwnd, HDC hdc, INT_DWORD x, INT_DWORD y, INT_DWORD r, INT_DWORD p)
ПЕРЕМЕННЫЕ
HWND hwnd; HDC hdc;
INT_DWORD hdc, x, y, r, p
НАЧ_ПРОГ
    ЕСЛИ (p) TO // пока p≠0
        НАЧ_БЛОК_1
        // рисуем еще четыре окружности с центрами по краям этой
        DrawPattern (hwnd, hdc, x-r, y, r, p-1)
        DrawPattern (hwnd, hdc, x, y-r, r, p-1)
        DrawPattern (hwnd, hdc, x+r, y, r, p-1)
        DrawPattern (hwnd, hdc, x, y+r, r, p-1)
        // Ellipse – функция Win32 API для вывода эллипса (окружности), вписанного
        // в прямоугольник (квадрат) с координатами правого верхнего угла (x_up, y_up)
        // и левого нижнего угла (x_low, y_low):
        // Ellipse(
        // HDC hdc, INT_DWORD x_up, INT_DWORD y_up, INT_DWORD x_low, INT_DWORD y_low)
        // так как для рисования нужны координаты прямоугольника,
        // а не центра окружности, то преобразуем их при вызове Ellipse:
        Ellipse(hdc, x-up-r, y-up-r, x_low+r, y_low+r)
        КОН_БЛОК_1
    КОН_ПРОГ

```

Если в первом варианте процедуры `DrawPattern` — `DrawPattern_1` окружности рисовались перед очередной рекурсивной передачей управления в процедуру `Draw-`

Pattern, то во втором варианте это делается в последнюю очередь — во время обратного хода по цепочке вызовов процедуры DrawPattern. Это уже требует наличия локальных переменных в процедуре и их сохранения на период, пока осуществляются рекурсивные вызовы процедуры DrawPattern. Приведем соответствующие фрагменты основной программы и функции DrawPattern_2 из DLL-библиотеки maket_dll.DLL.

```

+-----+
:| Программа: prg3_1.asm. Фрагмент оконного приложения.
:| вызывающего рекурсивную процедуру DrawPattern_2.
+-----+
:
:
+-----+
:| Процедура: MenuProc. Обработка сообщений от меню.
+-----+
MenuProc      proc
arg            @hwnd:DWORD, @wparam:DWORD, @hdc:DWORD, @hbrush:DWORD
uses          eax, ebx
              mov     ebx, @wparam      ; в bx идентификатор меню
              :
              cmp     bx, IDM_DLL_LACES_1
              je      @idmdlllaces_1
              cmp     bx, IDM_DLL_LACES_2
              je      @idmdlllaces_2
              jmp     @exit
              :
:----- рисует узор из окружностей. рекурсивная функция для рисования
: находится в DLL-библиотеке:
: DrawPattern_2(hwnd,hdc,x,y,r,p) —
: функция работает с локальными переменными
@idmdlllaces_2:
              :
              : для начала очистим окно
:----- теперь можно рисовать: DrawPattern_2(hwnd,hdc,x,y,r,p)
              push    p                  ; порядок узора
              push    r                  ; радиус окружности
              push    y_P                ; y-координата центра окружности
              push    x_P                ; x-координата центра окружности
              push    memdc              ; идентификатор контекста устройства
              push    @hwnd
              call     DrawPattern_2
:----- генерация сообщения WM_PAINT для вывода изображения на экран
              :
              call     InvalidateRect
              jmp     @exit
              :
+-----+
:| Библиотека: maket_dll.DLL. Фрагмент DLL-библиотеки.
:| содержащей рекурсивную процедуру DrawPattern_2.
+-----+
:
:
+-----+
:| Процедура: DrawPattern_2. Рекурсивная процедура рисования узора
:| (с использованием локальных переменных).
+-----+
DrawPattern_2  proc
arg            @hwnd:dword, @hdc:dword, @x:dword, @y:dword, @r:dword, @p:dword
              :----- рисуем окружность -
              : рекурсивно вызываем DrawPattern_2(hwnd,hdc,x,y,r,p)
              dec     @p
              cmp     @p, 0
              je      @End_Draw
              shr     @r, 1              ; делим на 2

```

```

:----- готовим параметры в стеке для вызова Ellipse
:....
:----- вызываем DrawPattern для отображения еще четырех
: окружностей меньшего порядка
:----- (x-r,y)
:....
: готовим параметры в стеке
: для вызова DrawPattern_2
call DrawPattern_2
:----- (x+r,y)
:....
: готовим параметры в стеке
: для вызова DrawPattern_2
call DrawPattern_2

:----- (x,y-r)
:....
: готовим параметры в стеке
: для вызова DrawPattern_2
call DrawPattern_2
:----- (x,y+r)
:....
: готовим параметры в стеке
: для вызова DrawPattern_2
call DrawPattern_2
:----- рисуем окружности на обратном ходе, для чего организуем
: доступ в стеке к локальным параметрам для Ellipse
@@End_Draw:
push ebp
mov ebp, [esp]
push dword ptr [ebp-4]
push dword ptr [ebp-8]
push dword ptr [ebp-12]
push dword ptr [ebp-16]
push dword ptr [ebp-20]
call Ellipse : рисуем окружность
pop ebp
ret
endp DrawPattern_2

```

Из этого фрагмента отчетливо видно, в чем разница между размещением параметров, передаваемых в рекурсивную процедуру, и локальными переменными этой процедуры. Для доступа к параметрам используются положительные смещения относительно адреса в BP (это скрыто от нас с помощью директивы ARG), а для доступа к локальным параметрам — отрицательные смещения.

Разница в изображениях возникла из-за разных мест в программе, где вызывается функция InvalidateRect. Попробуйте самостоятельно исправить этот «дефект».

Разработка динамических библиотек (DLL)

Библиотеки динамической компоновки (Dynamic Link Libraries, DLLs) представляют собой хранилище общедоступных процедур. Механизм DLL-библиотек является неотъемлемой частью операционной системы Windows. Суть этого механизма в том, что в процессе компоновки исполняемого модуля с использованием внешних процедур в него помещаются не сами процедуры, а только их названия (номера) вместе с названиями DLL-библиотек, в которых они содержатся. В главе 15 «Модульное программирование» учебника для связи модулей на разных языках рассматривались стандартные соглашения по передаче параметров, которые специфическим образом реализовывались на уровне конкретных компиляторов языков программирования. Этот аппарат был, пожалуй, единственным средством связи разноразличных модулей при программировании для MS DOS. В среде Windows

более естественным является механизм DLL-модулей. Он позволяет, в частности, разработать набор процедур на ассемблере и затем использовать их в программах на языках высокого уровня, поддерживающих механизм динамического (позднего) связывания.

Как правило, если язык программирования поддерживает разработку Windows-приложений, то он имеет средства для разработки и использования DLL-библиотек. Ассемблер не является исключением. Общие принципы DLL-библиотек для всех языков одинаковы, так как эти библиотеки являются универсальным средством, не зависящим от конкретного языка. Поэтому, разрабатывая DLL-модуль, необходимо учитывать общие требования к таким библиотекам. Структурно DLL-библиотека представляет собой обычную программу, включающую некоторые специфические элементы. Рассмотрим процесс создания и использования DLL-библиотеки на языке ассемблера. Для этого разработаем консольное приложение, которое выводит некоторую строку на экран 10 раз. На каждой итерации вывода меняются атрибуты этой строки. За основу взята программа `prg05_11.asm` из главы 5. Только теперь строка с выводимым сообщением находится в приложении, а сама процедура вывода — внутри DLL. Для демонстрации передачи и возврата параметров в процедуру передаются длина и адрес строки, а возвращаются значения `0xffffffff` в четырех регистрах — `EAX`, `EBX`, `ECX`, `EDX`. Обсудим процесс по шагам.

Шаг 1. Разработка текста DLL-библиотеки

Как мы уже отметили, DLL-библиотека представляет собой обычную программу на языке ассемблера. Выбор примера поэтому неслучаен. Тем самым мы подтвердим тезис о том, что обычная программа и DLL-библиотека имеют много общего. С точки зрения структуры, DLL-библиотека является набором процедур, переменных и констант, а также необязательного кода инициализации, которые оформлены в соответствии с требованиями ассемблера. Ниже приведен пример DLL-библиотеки для нашей задачи.

```

+-----+
: | Модуль: maket_dll.asm. Текст DLL-библиотеки. |
: | Содержит одну функцию — WriteCon.           |
+-----+
.486
locals
.model flat, STDCALL ; модель памяти flat
;----- Объявление внешними используемых в данной программе
; функций Win32 (ASCII)
;----- объявление процедуры WriteCon общедоступной
public dll WriteCon
;...
.data
;...
.code
DllMain
arg @@hInst:dword, @@event:dword, @@no_use:dword
@@m: mov eax, 1
ret
endp
DllMain
WriteCon proc ; см. материалы, прилагаемые к книге.
; и prg05_11.asm из главы 5
arg @@adr_str:dword, @@len_str:dword

```



```

        :...
        ret
        WriteCon
endp
end DllMain

```

Хорошо видно, что текст для DLL-библиотеки является действительно обычным файлом ассемблера. Есть все, даже имя точки входа, указываемое в последней директиве **END**. Но здесь и начинаются странности. На самом деле это не обычная точка входа, которую мы привыкли указывать в любой программе на ассемблере, а адрес команды в DLL-библиотеке, получающей управление в строго определенных случаях. Эта команда является первой в цепочке команд, составляющих так называемый *код инициализации* DLL-библиотеки. Назначение этого кода — выполнить необходимые действия по инициализации DLL-библиотеки при наступлении определенных событий. Его наличие необязательно, и при его отсутствии нет необходимости указывать соответствующую метку в заключительной директиве **END**. Если все же код инициализации присутствует в DLL-библиотеке, то он должен быть разработан с учетом определенных требований.

■ Во-первых, этот код должен быть рассчитан на то, что он получает управление в одном из четырех случаев. О наступлении каждого из этих случаев операционная система извещает DLL-библиотеку путем передачи ей одного из четырех предопределенных значений — *флагов*. Значения этих флагов перечислены в файле `winnt.h`. Рассмотрим данные флаги и возможные действия при их поступлении в DLL-библиотеку.

- **DLL_PROCESS_ATTACH** = 1 — передается операционной системой DLL-библиотеке при проецировании последней в адресное пространство процесса. Передача этого флага производится всего один раз, обычно при загрузке приложения, использующего данную DLL-библиотеку. Если позже другой процесс попытается загрузить ту же библиотеку, то система попросту увеличит ее счетчик использования без отправки флага **DLL_PROCESS_ATTACH**. Получив данный флаг, DLL-библиотека должна выполнить действия по созданию необходимой среды функционирования для своих процедур. Например, обеспечить их кучей.
- **DLL_THREAD_ATTACH** = 2 — передается операционной системой DLL-библиотеке при создании нового потока в процессе. Этим библиотеке предоставляется возможность нужным образом обработать факт создания нового потока. Следует иметь в виду, что описываемое действие не является обратимым, то есть если DLL-библиотека загружается в процесс, когда в нем уже функционируют потоки, то ни одному из них не посылается флаг **DLL_THREAD_ATTACH**.
- **DLL_THREAD_DETACH** = 3 — передается операционной системой DLL-библиотеке при выгрузке потоком DLL-библиотеки.
- **DLL_PROCESS_DETACH** = 0 — передается операционной системой DLL-библиотеке при выгрузке DLL-библиотеки из адресного пространства процесса. Логично, что при этом требуется провести завершающие действия по освобождению всех ресурсов, которыми владеет DLL-библиотека. Обычно эти действия являются обратными по отношению к предпринятым при инициализации библиотеки (см. флаг **DLL_PROCESS_ATTACH**).

- Во-вторых, имя точки входа DLL-библиотеки может быть любым, но уникальным в пределах одной DLL. Главное, чтобы при наличии кода инициализации это имя было указано в директиве **END**.
- В-третьих, оформление кода инициализации в виде отдельной процедуры обязательно. Главное, выполнить два основных действия кода инициализации DLL-библиотеки (при его наличии):
 - вернуть единицу в регистре **EAX**;
 - удалить из стека три параметра, которые передаются DLL-библиотеке при передаче описанных выше флагов: **hInstDLL** — дескриптор DLL-библиотеки, назначенный ей системой при загрузке библиотеки в адресное пространство процесса; **event** — значение флага, передаваемого в DLL-библиотеку; **fImpLoad** — параметр не равен 0, если библиотека загружена неявно (см. ниже), и равен 0 в противном случае.

Структура полного варианта кода инициализации выглядит так:

```

include      WindowConA.inc          : проверьте значения флагов в этом файле
DllMain
arg          ....
             proc
             hInstDLL:dword,  event:dword, fImpLoad:dword
             cmp      [event]. DLL_PROCESS_ATTACH
             jne      m
             :----- выполняем действия для DLL_PROCESS_ATTACH
             ....
             cmp      [event]. DLL_THREAD_ATTACH
             jne      m
             :----- выполняем действия для DLL_THREAD_ATTACH
             ....
             cmp      [event]. DLL_THREAD_DETACH
             jne      m
             :----- выполняем действия для DLL_THREAD_DETACH
             ....
             cmp      [event]. DLL_PROCESS_DETACH
             jne      m
             :----- выполняем действия для DLL_PROCESS_DETACH
             ....
m:           mov      eax, 1
             ret
DllMain     endp

```

Минимальный вариант может выглядеть так, как это сделано в нашем примере:

```

DllMain     proc
arg          hInstDLL:dword,  event:dword, fImpLoad:dword
m:          mov      eax, 1
             ret
DllMain     endp

```

Или так:

```

DllMain:
m:          mov      eax, 1
             ret     12

```

Не забывайте, что директива **ARG** приводит к тому, что в код, генерируемый транслятором, вставляются команды **ENTER** и **LEAVED** (см. выше раздел «Реализация рекурсивных процедур» и раздел «Реализация вложенных процедур» среди материалов, выложенных на сайт). Кроме этого, команда **RET** процедуры дополняется значением, равным сумме длин параметров, указанных в директиве **ARG**. Исполни-

ние такой команды приводит к удалению из стека количества байтов, равного этому сформированному значению. Что касается кода функций (процедур), составляющих DLL-библиотеку, то для их написания используются стандартные правила разработки программ. Описание данных также ничем не отличается от обычной программы ассемблера. Ведь в конечном итоге код и данные процедур DLL-библиотеки оказываются в адресном пространстве процесса наравне с его кодом и данными.

Последнее, что необходимо отметить, — все экземпляры данных и имена процедур, которые должны быть видны вне пределов DLL-библиотеки, объявляются общими при помощи одной из директив — PUBLIC или PUBLICDLL.

Шаг 2. Трансляция и компоновка исходного текста DLL-библиотеки

После того как подготовлен исходный текст библиотеки, его транслируют обычным для программ ассемблера образом. Что же касается компоновки, то необходимо помнить, что ее целью является получение файла с расширением DLL, а не обычного файла с расширением EXE. Весь этот процесс удобно обсуждать на примере реального файла `makefile`, текст которого приведен ниже:

```
NAME = maket_dll
OBJS = $(NAME).obj
DEF = $(NAME).def
RES = $(NAME).res
TASMOPT=/m3 /mx /z /q /DWINVER=0400 /D_WIN32_WINNT=0400
!if $(DEBUG)
TASMDEBUG=/zi
LINKDEBUG=/v
!else
TASMDEBUG=/l
LINKDEBUG=
!endif
!if $(MAKEDIR)
IMPORT=import32
!else
IMPORT=import32
!endif
$(NAME).EXE: $(OBJS) $(DEF)
    tlink32 /Tpd /aa /c $(LINKDEBUG) $(OBJS).$(NAME).. $(IMPORT). $(DEF)
.asm.obj:
    tasm32 $(TASMDEBUG) $(TASMOPT) $&.asm
```

Запуск данного файла производится командной строкой:

```
make -DDEBUG -fmakefile_dll.mak >p.txt
```

В результате формируются несколько файлов, перечень которых определяется тем, насколько успешно отработали программы транслятора `tasm32` и компоновщика `tlink32`. Для быстрой оценки этого результата мы перенаправили весь вывод в файл `p.txt`. Просмотрев этот файл, можно оценить успешность создания DLL-библиотеки, не анализируя другие файлы (например, листинг). При наличии синтаксических ошибок необходимо исправить их и повторить запуск MAKE-файла на исполнение.

Для успешной компоновки необходим еще один файл — с расширением DEF. Необходимое и достаточное содержимое файла `maket_dll.def` приведено ниже:

```
LIBRARY      maket_dll
```

```
DESCRIPTION 'Win32 DLL'
EXPORTS      WriteCon @1
```

В этом файле следует обратить внимание на макропеременную `EXPORTS`, которая содержит имена экспортируемых функций DLL-библиотеки и их ординалы, то есть порядковые номера этих функций в DLL-библиотеке. Последние использовались в 16-разрядных версиях Windows, в современных версиях этой операционной системы их использование необязательно, и Microsoft настоятельно рекомендует этого не делать.

О том, что компоновщик должен создать именно DLL-библиотеку, объявляют с помощью ключа `/Prd`.

Шаг 3. Создание LIB-файла

Как указать приложению местонахождение внешних функций, расположенных в DLL-библиотеках? Если бы приложение работало только с одной DLL-библиотекой, то проблем бы не было — указывай нужную и продолжай процесс сборки приложения. Если количество необходимых приложению DLL-библиотек больше одной, а тем более если их десятки, то ситуация требует иного решения, нежели простое перечисление нужных приложению DLL-библиотек. Для централизованного хранения информации о размещении используемых приложением функций в DLL-библиотеках применяют LIB-файлы. Эти файлы представляют собой своеобразный справочник о размещении функций в DLL-библиотеках. При этом не указывается никаких путей, так как при обращении к DLL-библиотеке операционная система ищет ее по следующему алгоритму.

1. В каталоге, содержащем EXE-файл приложения.
2. В текущем каталоге процесса.
3. В системном каталоге Windows.
4. В основном каталоге Windows.
5. В каталогах, указанных в переменной окружения `PATH`.

В пакете TASM для создания LIB-файла предназначена утилита `Implib.exe`. Для создания LIB-файла в нашем примере необходимо выполнить следующую команду строку:

```
IMPLIB.EXE maket_dll.lib maket_dll.DLL >p.txt
```

Как видите, мы опять используем перенаправление вывода в файл `p.txt` для быстрой оценки результата работы программы `IMPLIB.EXE`. Если выполнение этой утилиты было успешным, то формируется файл `maket_dll.lib`, который в дальнейшем используется для сборки целевого приложения.

Шаг 4. Сборка приложения с использованием DLL-библиотеки

Приведем содержимое MAKE-файла для сборки целевого приложения:

```
NAME = maket
OBSJ = $(NAME).obj
DEF = $(NAME).def
!if $(DEBUG)
TASMDDEBUG=/zi
```

```

LINKDEBUG=/v
!else
TASMDEBUG=
LINKDEBUG=
!endif
TASMOPT=/m3 /z /q # /DWINVER=0400 /D_WIN32_WINNT=0400
# /mx
!if $d(MAKEDIR)
IMPRT=$(MAKEDIR)\import32+maket_dll
!else
IMPRT=import32+maket_dll
!endif
$(NAME).EXE: $(OBSJ) $(DEF)
tlink32 /Tpe /aa /x /c $(LINKDEBUG) $(OBSJ).$(NAME).. $(IMPORT). $(DEF)
.asm.obj:
del $(NAME).EXE
tasm32 $(TASMDEBUG) /ml $(TASMOPT) $&.asm...
```

Теперь, имея два MAKE-файла (для сборки файлов .DLL и .EXE), можно провести сравнительный анализ их содержимого. Отметим два момента:

- в макропеременной IMPORT указываются имена (без расширений) LIB-файлов, содержащих сведения о нужных приложению функциях в DLL-библиотеках (если LIB-файлов несколько, то они перечисляются с использованием знака +);
- для сборки EXE-приложения используется ключ /Tre редактора связей.

Содержимое DEF-файла maket.def приложения:

```

NAME          maket
DESCRIPTION   'Assembly Console Windows Program'
CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE MULTIPLE
EXPORTS
```

И наконец, содержимое самого файла maket.asm, использующего функцию из разработанной нами DLL-библиотеки maket_dll.dll.

```

;+-----+
;| Программа: maket.asm. Вызов функции WriteCon из файла maket_dll.dll. |
;+-----+
.4B6
includelib    maket_dll.lib          ; необязательно
.data
TitleText    db      "Строка выводится процедурой из DLL"
              Len_TitleText = $ - TitleText
.code
start        proc    near            : точка входа в программу
              ....
              push    Len_TitleText
              push    offset TitleText
              call     WriteCon
exit:        ....                    : выход из приложения
```

Импортируемую из DLL-библиотеки функцию необходимо объявить внешней с помощью директивы `extrn WriteCon:PROC`.

Шаг 5. Проверка работоспособности приложения с использованием DLL-библиотеки

Для проверки работоспособности полученного на предыдущем шаге приложения хорошо подходит отладчик `td32.exe`. Кстати, когда вы будете в нем работать, обратите внимание на то, как происходит переход из DLL-библиотеки на код в про-

цедуре. Вы увидите, что помощь в этом оказывает «неизвестно откуда» появившаяся команда JMR. Причину этого вы можете выяснить, прочитав раздел «Секция описания импортируемых функций PE-файла» главы «Форматы исполняемых файлов» книги [40].

При разработке DLL-библиотек естественным образом возникает вопрос о совместимости с приложениями, написанными на других языках. Это тем более актуально, если речь идет о продуктах разных фирм-производителей программного обеспечения. Проверить, насколько совместима сформированная нами с помощью средств TASM DLL-библиотека, можно с помощью утилиты DUMPBIN.EXE из пакета Microsoft Visual Studio. Запустите ее командной строкой вида:

DUMPBIN.EXE -exports maket_dll.dll>p.txt

Тогда в файле p.txt вы получите отчет о содержимом раздела экспорта DLL-библиотеки maket_dll.dll. Проанализировав полученные результаты, вы убедитесь, что проблем с распознаванием нашей DLL-библиотеки у этого программного средства фирмы Microsoft не возникло. Это дает основание полагать, что данную библиотеку при соответствующем наполнении полезными функциями можно использовать при программировании на VisualC/C++, VisualBasic и т. п. При этом необходимо иметь в виду, что есть вероятность искажения имен функций при использовании компиляторов различных фирм. Подробнее об этом можно узнать в соответствующей литературе [11].

Не следует забывать, что на практике допустимы три формы загрузки DLL-библиотеки в адресное пространство процесса: неявная, явная и отложенная. Описанный выше способ сборки приложения на самом деле был неявным и предполагал, что загрузка DLL-библиотеки производится при запуске самого приложения. Явный способ загрузки DLL-библиотеки предполагает ее загрузку во время работы приложения. Для этого в Win32 API существуют специальные функции:

```
HINSTANCE LoadLibrary( LPCTSTR lpLibFileName );
HMODULE LoadLibraryEx(LPCTSTR lpLibFileName, HANDLE hFile, DWORD dwFlags);
```

Третий способ подключения DLL-библиотек — отложенная загрузка. Этот способ предполагает, что DLL-библиотека не будет загружена в адресное пространство процесса до тех пор, пока приложению не потребуются осуществить доступ к любому экспортируемому из данной DLL-библиотеки объекту (переменной, константе, процедуре). Подробнее об этом и других вопросах разработки и использования DLL-библиотек можно прочитать в литературе [11].

Глава 4

Обработка цепочек элементов

Против незнания есть только одно средство — знание. Истинное же знание может быть достигнуто только через личное совершенствование.

Л. Н. Толстой

Представленный ниже материал является дополнением к главе 12 «Цепочечные команды» учебника. Из этой главы следуют выводы о том, что, во-первых, цепочечные команды являются мощным инструментом обработки последовательностей элементов размером 1/2/4 байтов и, во-вторых, это единственное средство процессора для обработки данных по схеме *память-память*. В процессе разработки программ для учебника и этой книги мы достаточно часто использовали команды процессора этой группы. Но цепочечные команды — это примитивы, и, как любые примитивы, они являются лишь основой для построения более сложных алгоритмов обработки цепочек элементов. Особенно это чувствуется при разработке программ для задач обработки текстов и, в частности, задачи поиска. Для решения озвученной проблемы существует ряд классических алгоритмов, оптимизирующих этот процесс. Ниже будет приведено несколько программ, демонстрирующих алгоритмы поиска данных в строках символов, то есть цепочках элементов размером 1 байт. Так как все они построены на основе стандартных байтовых цепочечных команд процессора, то при необходимости обработки последовательностей элементов большей размерности (2 и 4 байта) их доработка не составит вам особого труда.

Организацию поиска одиночного символа в программе на ассемблере мы рассматривать не будем, так как это делается самими цепочечными командами без привлечения посторонней алгоритмической поддержки. Информацию об этом можно получить из учебника.

Более подробно мы разберем организацию поиска текстовой подстроки в строке символов, превышающей размер искомой подстроки. При всей кажущейся простоте этого вида поиска его реализация в программах на языке ассемблера сопряжена с рядом проблем.

Введем некоторые обозначения:

- P — строка-аргумент поиска, размерность строки P — M байтов, j — индекс символа в строке P , $0 \leq j < M - 1$;
- S — строка, в которой ведется поиск строки P , размерность строки S — N байтов, i — индекс символа в строке S , $0 \leq i < N - 1$.

Все алгоритмы поиска в текстовой строке можно разбить на два класса: прямые и учитывающие особенности объектов поиска. Последний класс алгоритмов предполагает предварительный анализ искомой подстроки и формирование на его основе некоторой информации, управляющей процессом поиска.

Прямой поиск в текстовой строке

Цель поиска некоторой строки P в строке большего размера S — определить первый индекс элемента в строке S , начиная с которого все символы S совпадают с символами строки P . Для этого алгоритм поиска последовательно просматривает символы строки S , проводя одновременное сравнение ее очередного символа с первым символом строки P . После возникновения такого совпадения алгоритм производит последовательное сравнение соответствующих элементов строк S и P до возникновения одного из следующих условий:

- в процессе поиска соответствия достигнут конец строки P — это означает, что строка P совпадает с некоторой подстрокой строки S ;
- достигнут конец строки S при незавершенном или не начатом просмотре строки P — это означает, что строке P не соответствует ни одна из подстрок S .

Одна из главных проблем, которую приходится решать при написании программы обработки символьной строки, — определение конца строки S . Здесь возможны два варианта:

- статический — размер строки фиксирован некоторым значением N ;
- динамический (характерен для обработки массивов символьных строк) — длина строки определяется значением, являющимся либо первым элементом очередной строки, либо концевым (служебным) символом, значение которого заранее определено и не может совпадать ни с одним символом строки.

Начнем обсуждение прямого способа поиска с программы поиска в строке с фиксированной длиной. Для экономии места ограничим число вхождений P в S единицей.

```

+-----+
+| Программа: prg4_67_f.asm. Поиск строки P в строке S. Длина S фиксирована. |
+-----+
+| Вход: S и P — массивы символов размером N и M байтов (M=<N). |
+-----+
+| Выход: сообщение о количестве вхождений строки P в строку S. |
+-----+
.data
s      db      "Ах, какой был яркий день!"
      db      "  Лодка, солнце, блеск и тень."
      db      "  и везде цвела сирень." ; задаем массив S
      Len_S = $ - s
      db      "$"
```



```

mes      db      "Вхождений строки - "
p        db      "ень"          ; задаем массив P - аргумент поиска
        len_p = $ - p
        db      "$ - "
Count    db      0, "$"         ; счетчик вхождений P в S
.code

:....
cld
mov      cx, len_s
lea      di, s
mov      al, p                  : P[0]->al
next_search:
lea      si, p                  : на следующий символ
inc
repne    scasb
jcxz     exit
push     cx
mov      cx, len_p - 1
repe     cmpsb
jz       eq_substr              : строка p <= подстроке в s
mov      bx, len_p - 1
sub      bx, cx
pop      cx
sub      cx, bx                 : учли пройденное при сравнении cmpsb
jmp      next_search

:----- далее можно выйти, если поиск однократный.
:      но мы упорные, поэтому продолжаем...
eq_substr:
pop      cx
sub      cx, len_p - 1          : учли пройденное при сравнении cmpsb
inc      count
jmp      next_search
exit:    add      count, 30h
:----- вывод сообщения mes на экран
:....

```

Из программы видно, что когда размер строки фиксирован, то проблема ее конца решается просто. Но чаще приходится иметь дело с задачами, выполняющими поиск подстроки в строке, длина которой заранее не известна. Это характерно, в частности, для приложений обработки файлов. Но и с файлами не так все просто. Для текстовых ASCII-файлов особых проблем нет — в них строки заканчиваются символами 0dh, 0ah. Сложнее дело обстоит с обработкой двоичных файлов, где с равной степенью вероятности могут встретиться любые символы кодовой таблицы. В подобных случаях проблему локализации места в файле, где осуществляется поиск, нужно решать исходя из постановки конкретной задачи. Несмотря на это сами приемы поиска не сильно отличаются от рассмотренных в этом разделе.

В случае когда поиск осуществляется в строке или в массиве строк в памяти, длина которых заранее не известна, то для обозначения их окончаний нужно ввести некоторый служебный символ. Например, в ряде языков существует понятие строки ASCIIZ, представляющей собой обычную символьную строку с завершающим нулевым символом. По этому символу и определяется конец строки. Подобные служебные символы можно использовать и для обработки строк в памяти. Другая возможность задания границы строки — введение в структуру строки специального поля, обычно первого байта (2 байтов), содержащего длину этой строки.

Следующая программа демонстрирует возможную организацию поиска в текстовом файле. Для этого содержимое файла читается в динамически выделяемую область памяти. После небольшой модернизации данную программу можно рассматривать как основу для других программ поиска в строках памяти, ограничен-


```

        cmp     eax, 0xffffffffh
        je      exit      : если неуспех
        mov     hFile, eax : файловый манипулятор
;----- определим размер файла
;      DWORD GetFileSize(HANDLE hFile, LPDWORD lpFileSizeHigh):
;      ....
        call    GetFileSize
        test    eax, eax
        jz      exit      : если неуспех
        mov     FileSize, eax : сохраним размер файла
;----- запрашиваем блок памяти из кучи
;      LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes):
;      ....
        call    HeapAlloc
        mov     buf_start, eax : адрес блока с текстом программы
                                : из общей кучи процесса
;----- читаем файл
;      BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer,
;      :      DWORD nNumberOfBytesToRead,
;      :      LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped):
;      ....
        call    ReadFile
        test    eax, eax
        jz      exit      : если неуспех
;-----
        push    ds
        pop     es
        cld
        mov     ecx, FileSize
        mov     edi, buf_start : адрес буфера с текстом файла в edi
cycl1:   push    ecx
        push    edi
        mov     ebx, ecx
        mov     al, 0dh        : 0dh -> a1
        repne  scasb
        jcxz   e_exit
e_exit:  jmp     $ + 7
;      ....
        jmp     exit
        pop     edi
        sub     ebx, ecx
        xchg    ebx, ecx
        mov     al, p          : P[0]->a1
next_search: repne  scasb
        jcxz   end_str        : достигнут конец строки
;----- проверяем возможное совпадение
        push    ecx
        lea     esi, p
        inc     esi
        mov     ecx, len_p - 1
        repe  cmpsb
        jz      eq_substr     : строка p <= подстроке в s
        mov     edx, len_p - 1
        sub     edx, ecx
        pop     ecx
        sub     ecx, edx       : учли пройденное при сравнении cmpsb
        jmp     next_search
end_str: inc     edi
        inc     ebx
        xchg    ebx, ecx
;----- преобразуем в символьное представление
;      :      счетчик вхождений count
;      ....
;----- вывод на экран строки mes
;      ....

```

```

call    WriteConsoleA
:....
mov     count, 0          : обнуляем счетчик вхождений в строку
pop     ecx               : восстанавливаем
jmp     cycl1
eq_substr: pop     ecx
         sub     ecx, len_p - 1 : учли пройденное при сравнении cmpsb
         inc     count
         jmp     next_search
exit:   pop     ecx
:----- выход из программы — задержим ввод до нажатия любой клавиши
:....

```

Этой программой мы проиллюстрировали оба варианта поиска с динамическим определением размера строки.

Алгоритмы, реализованные выше, можно использовать для организации поиска в строке S небольшой длины, так как попытки повысить эффективность приведут к излишним накладным расходам. Для строки S большой размерности (потокковые данные для приложений мультимедиа) прямые алгоритмы поиска могут быть неэффективными. Положение можно исправить рассмотренными ниже алгоритмами поиска с предварительным анализом искомой подстроки.

Поиск с предварительным анализом искомой подстроки

Основу материала этого раздела составляет алгоритм КМП-поиска. Имя «КМП» является выборкой первых букв фамилий его создателей: Д. Кнута, Д. Мориса и В. Пратта. В основе алгоритма КМП-поиска лежит идея о том, что в процессе просмотра строки S с целью поиска вхождения в нее образца P становится известной информация о просмотренной части. Работа алгоритма производится в два этапа.

1. Анализируется строка-образец P . По результатам анализа заполняется вспомогательный массив смещений D .
2. Производится поиск в строке S с использованием строки-образца P и массива смещений D .

Ниже приведена программа, реализующая алгоритм КМП-поиска.

```

// prg4_73 КМП — программа на псевдоязыке поиска строки P в строке S
// по алгоритму КМП-поиска. Длина S фиксирована.
// Вход: S и P — массивы символов размером N и M байтов (M<=N).
// Выход: сообщение о количестве вхождений строки P в строку S.
ПЕРЕМЕННЫЕ
INT_BYTE s[n]; // 0<=i<N-1
INT_BYTE p[m]; // 0<=j<M-1
INT_BYTE d[m]; // массив смещений
INT_BYTE k=-1; i=0; j=0 // индексы
НАЧ ПРОГ
//этап 1: формирование массива смещений d
j:=0; k:=-1; d[0]:=-1
ПОКА j<M-1 ДЕЛАТЬ
    НАЧ БЛОК 1
        ПОКА ((k>=0)И(p[j]<>p[k])) k:=d[k]
        j:=j+1; k:=k+1
        ЕСЛИ p[j]==p[k] ТО d[j]:=d[k]
        ИНАЧЕ d[j]:=k
    КОН БЛОК 1

```

```
// этап 2: поиск
i:=0; j:=0; k:=0
ПОКА ((j<M)И(i<N)) ДЕЛАТЬ
    НАЧ_БЛОК_1
        ПОКА ((j>=0)И(s[i]<>p[j])) j:=d[j]
        j:=j+1; i:=i+1
    КОН_БЛОК_1
ЕСЛИ j=M ТО вывод сообщения об удаче поиска
ИНАЧЕ вывод сообщения о неудаче поиска
КОН_ПРОГ
```

```
-----+
:| Программа: prg4_73_KMP.asm. Поиск строки P в строке S
:| по алгоритму КМП-поиска. Длина S фиксирована.
:-----+
.data
s      db      "fgcabefabcaabcdabc!bdeededegjfkababcdabces"
      db      "abeabcdabce;j:koiaabcabe" ; задаем массив S
      Len_S = $ - s      ; N=Len_S
      db      "$"
mes     db      0dh, 0ah, "Вхождений строки - "
p       db      "abcdabce" ; задаем массив P - аргумент поиска
      Len_P = $ - p      ; M=Len_P
      db      " _ "
Count  db      0, "$" ; счетчик вхождений P в S
d       db      Len_p dup (0) ; массив смещений
k       db      0
.code
;----- этап 1 - заполнение массива смещений: j:=0; k:=-1; d[0]:=-1
xor     si, si      ; si - это j
mov     k, -1
mov     d, -1
;----- ПОКА j<M-1 ДЕЛАТЬ
cyc11:  cmp     si, len_p - 1 ; j<M-1
      jge     exit_d
      cmp     k, 0 ; ПОКА ((k>=0)И(p[j]<>p[k])) k:=d[k]
      jl      falsee
      mov     bl, k
      mov     al, p[si]
      cmp     al, p[bx]
      je      falsee
      mov     bl, d[bx] ; k:=d[k]
      jmp     cyc11
;----- j:=j+1; k:=k+1
falsee: inc     si      ; j:=j+1
      inc     k
      mov     bl, k
      mov     al, p[si] ; ЕСЛИ p[j]==p[k] ТО d[j]:=d[k]
      cmp     al, p[bx]
      jne     elsee
      mov     al, d[bx]
      mov     d[si], al
      jmp     cyc11 ; $+6
elsee:  mov     d[si], bl ; ИНАЧЕ d[j]:=k
      jmp     cyc11
;----- этап 2: поиск
exit_d: xor     di, di ; i:=0;
      xor     si, si ; j:=0;
m3:     cmp     si, Len_p ; ПОКА ((j<M)И(i<N)) ДЕЛАТЬ
      jge     m1
m34:    cmp     di, Len_s
      jge     m1
      cmp     si, 0 ; ПОКА ((j>=0)И(s[i]<>p[j])) j:=d[j]
```

```

jl      m4
mov     al, s[di]
cmp     al, p[si]
je      m4
movzx   al, d[si]
mov     si, ax
m4:     inc     si      : j:=j+1
        inc     di      : i:=i+1
        jmp     m3
m1:     ;----- ЕСЛИ j=M TO вывод сообщения об удаче поиска
        cmp     si, len_p
        jne     exit_f   ; ИНАЧЕ вывод сообщения о неудаче поиска
        inc     count
        cmp     di, len_s
        jge     exit_f
        xor     si, si
        jmp     m34
exit_f:  add     count, 30h
        ;----- вывод сообщения mes на экран
        ;....

```

Подробно, хотя и не очень удачно, алгоритм КМП-поиска описан у Вирта [4]. Этот алгоритм достаточно сложен для понимания, хотя в конечном итоге его идея проста. Центральное место в алгоритме КМП-поиска играет вспомогательный массив D . Поясним его назначение. Массив D содержит значения, которые нужно добавить к текущему значению j в позиции первого несовпадения символов в строке S и подстроке P (рис. 4.1).

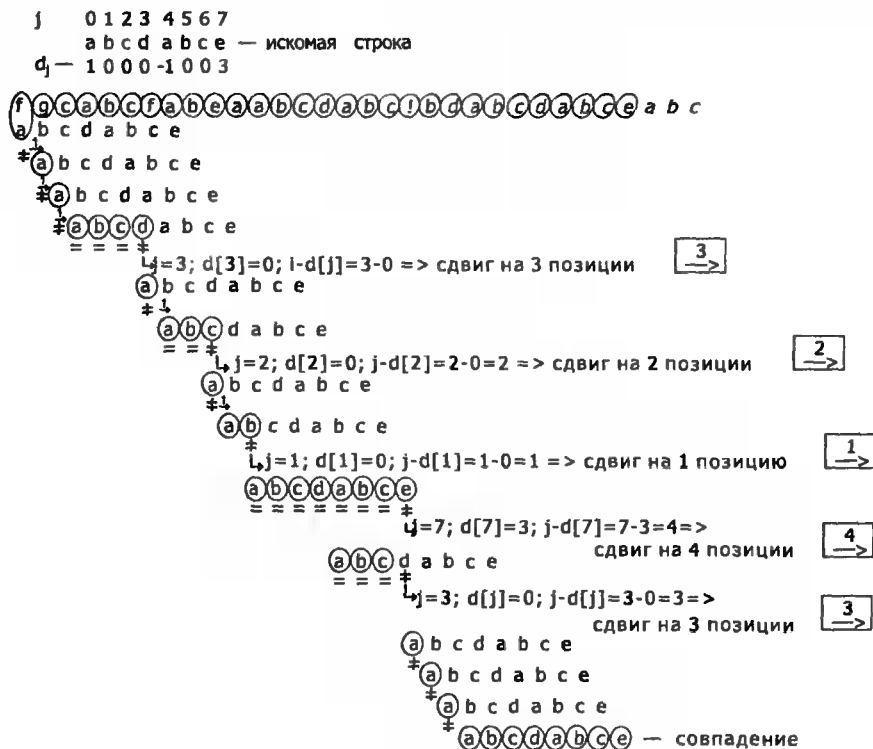


Рис. 4.1. Пример КМП-поиска

Из проведенного обсуждения можно сделать два вывода — один приятный, другой — не очень. Во-первых, явное достоинство этого метода в том, что исключены возвраты назад. Во-вторых, эффективность использования КМП-поиска зависит от исходных данных — реальное ускорение поиска (сдвиг образа P относительно строки S более чем на один символ) получается, когда в строке S достаточно часто встречаются последовательности символов, совпадающие с началом образа P .

Следующий (и последний) алгоритм поиска в строке носит название Боуера и Мура [4] — БМ-поиск. Он в значительной степени устраняет недостатки, присущие методу КМП-поиска.

```
// prg4_83_BP — программа на псевдоязыке поиска строки P в строке S
// по алгоритму БМ-поиска. Длина S фиксирована.
// Вход: S и P — массивы символов размером N и M байтов (M<=N).
// Выход: сообщение о количестве вхождений строки P в строку S.
ПЕРЕМЕННЫЕ
INT BYTE M; // M — длина образца
INT BYTE s[n]; // 0<=i<N-1
INT BYTE p[m]; // 0<=j<M-1
INT BYTE d[256]; // вспомогательный массив с размером = длине кодовой таблицы
INT BYTE k=0; i=0; j=0 // индексы
НАЧ ПРОГ
// этап 1: формирование массива d
для j=0 до 255 ДЕЛАТЬ
    НАЧ БЛОК 1
        d[j]=M
    КОН БЛОК 1
для j=0 до M-2 ДЕЛАТЬ
    НАЧ БЛОК 1
        d[p[j]]=M-j-1
    КОН БЛОК 1
// этап 2: поиск
i:=M
ПОВТОРИТЬ
    j:=M; k:=i
    ПОВТОРИТЬ
        K:=k-1; j:=j-1
        ПОКА (j>=0)ИЛИ(p[j]==s[k])
            i:=i+d[s[i-1]]
        ПОКА (j>=0)ИЛИ(i<N)
        ЕСЛИ j<0 ТО вывод сообщения об удаче поиска
        ИНАЧЕ вывод сообщения о неудаче поиска
    КОН ПРОГ
```

```
-----+
;| Программа: prg4_83_BP.asm. Поиск строки P в строке S
;| по алгоритму БМ-поиска. Длина S фиксирована.
;|-----+
.data
mes      db      0dh, 0ah, "Вхождений строки — "
p        db      "ab"                ; задаем массив P — аргумент поиска
        Len_P = $ - p                ; M=Len_P
        db      " — в строку — "
s        db      "fgcabceabcaab"     ; задаем массив S
        Len_S = $ - s                ; N=Len_S
        db      " — "
Count    db      0                    ; счетчик вхождений P в S
        db      " раз(a)$"
d        db      255 dup (0)         ; вспомогательный массив
k        db      0
.code
;----- этап 1 — заполнение массива D значением M —
```

```

:      размером образца для поиска
mov     cx, 255      ; размер кодовой таблицы ASCII
mov     al, len_p     ; для j=0 до 255 ДЕЛАТЬ
lea     di, d
rep     stosb         ; d[j]:=M
:----- цикл просмотра образца и замещение некоторых элементов d
:      значениями смещений (см. пояснение за текстом программы)
xor     si, si        ; j:=0
:----- для j=0 до M-2 ДЕЛАТЬ
cyc11:  cmp     si, len_p - 1
        jge     e_cyc11
        mov     al, p[si]
        mov     di, ax
        xor     bx, bx
        mov     bl, len_p
        dec     bl
        sub     bx, si
        mov     d[di], bl      ; d[p[j]]:=M-j-1
        inc     si
        jmp     cyc11
:----- //этап 2: поиск
e_cyc11: mov     di, len_p      ; i:=M
:----- ПОВТОРИТЬ - цикл пока (j>=0)ИЛИ(I<n)
cyc12:  mov     si, len_p      ; j:=M
        mov     bx, di        ; k:=I
:----- цикл пока (j>=0)ИЛИ(p[j]==p[k])
cyc13:  dec     bx              ; k:=k-1
        dec     si              ; j:=j-1
        cmp     si, 0          ; пока (j>=0)ИЛИ(p[j]==p[k])
        jl      m2
        mov     al, p[si]
        cmp     s[bx], al
        jne     m2
        jmp     cyc13
:----- i:=i+d[s[i-1]]
m2:     push    di
        dec     di
        xor     ax, ax
        mov     al, s[di]
        mov     di, ax
        mov     al, d[di]
        pop     di
        add     di, ax          ; I:=I+d[s[I-1]]
        cmp     si, 0          ; цикл пока (j>=0)ИЛИ(I<n)
        jl      m1
        cmp     di, len_s
        jg      exit_f
        jmp     cyc12
:----- вывод сообщения о результатах поиска
m1:     inc     count
        jmp     cyc12
exit_f: add     count, 30h
        lea     dx, mes
        mov     ah, 09h
        int     21h
exit:   ;...

```

Идея алгоритма БМ-поиска в том, что сравнению подвергаются не первые, а последние символы образца P и очередного фрагмента строки S . Если они не равны, то сдвиг в строке S осуществляется сразу на всю длину образца. Если последние символы равны, то сравнению подвергаются предпоследние символы, и т. д. При несовпадении очередных символов величина сдвига извлекается из таблицы D , которая, таким образом, выполняет ключевую роль в этом алгоритме. Заполнение

таблицы происходит на основе конкретной строки-образца P следующим образом. Размер таблицы определяется размером алфавита, то есть количеством кодов символов, которые могут появляться в тексте. В нашем случае мы отвели под таблицу D память длиной, равной длине кодовой таблицы ASCII. Таким образом, строки S и P могут содержать любые символы. Первоначально все байты кодовой таблицы заполняются значением, равным длине строки-образца для поиска P . Далее последовательно извлекаются символы строки-образца P начиная с первого. Для каждого символа определяется позиция его самого правого вхождения в строку-образец P . Это значение и заносится в таблицу D на место, соответствующее этому символу. Подробнее получить представление о заполнении таблицы можно по фрагменту программы на псевдоязыке:

```
// этап 1: формирование массива d
ДЛЯ j=0 ДО 255 ДЕЛАТЬ
    НАЧ БЛОК 1
        d[j]:=M
    КОН БЛОК 1
ДЛЯ j=0 ДО M-2 ДЕЛАТЬ
    НАЧ БЛОК 1
        d[p[j]]:=M-j-1
    КОН БЛОК 1
```

Так, для строки *abcdabce* процесс и результаты формирования таблицы D показаны на рис. 4.2.

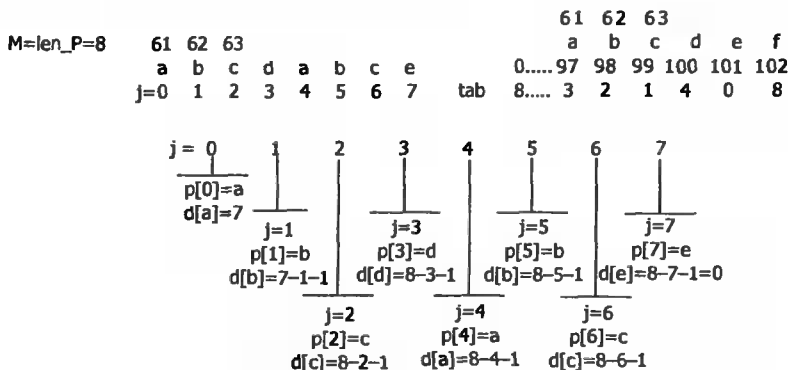


Рис. 4.2. Формирование массива D в программе БМ-поиска

Куда в двух последних программах пристроить цепочечные команды? К сожалению, некуда. Честно говоря, когда автор писал текст этих программ, он настолько увлекся, что напрочь забыл как о них, так и о цели настоящего раздела — показать особенности использования цепочечных команд при организации поиска информации. А когда вспомнил, то сделал вывод, что пристроить их вроде бы и некуда. А вы как думаете? Окончательный вывод об эффективности можно сделать по результатам работы профилировщика.

Развитие этой темы лежит в плоскости обсуждения проблемы обработки файлов, к которой мы когда-нибудь вернемся.

Глава 5

Работа с консолью в программах на ассемблере

Бросая в воду камешки, смотри на круги, ими образуемые; иначе бросание будет пустой забавой.

Козьма Прутков

На практике редко возникает необходимость разработки программы как «вещи в себе». В подавляющем большинстве случаев программа должна взаимодействовать с пользователем, получая от него данные посредством клавиатуры и выдавая результаты своей работы на экран. При знакомстве с новым языком программирования одним из первых вопросов, на которые ищет ответа программист, является выяснение средств этого языка для выполнения операций обмена с консолью (*консоль* — управляющий терминал, клавиатура и монитор). Что касается языка ассемблера, то собственных средств обмена с консолью у него нет. Чтобы выполнить подобную операцию, программа использует возможности самого компьютера (прерывания BIOS) и операционной системы, в среде которой эта программа работает. Каждый программист самостоятельно ищет решение проблемы обмена с консолью. Так как эта задача актуальна всегда, то имеется необходимость на конкретных примерах показать порядок использования средств BIOS и ОС для обмена с консолью в программах на ассемблере. Примеры не очень сложны, и читатель легко сможет быстро встроить их в свои программы.

Функции BIOS для работы с консолью

В контексте нашего изложения ROM BIOS (Read Only Memory Basic Input-Output System) представляет собой совокупность программ в энергонезависимой памяти компьютера, одной из задач которых является устранение специфики аппаратных компонентов компьютера для функционирующего на нем программного обеспечения, включая операционную систему. Обслуживание клавиатуры и монитора выполняют программы BIOS, называемые *драйверами*. Структурно драйверы состоят из ряда подпрограмм, называемых *функциями*, каждая из которых отвечает за определенные действия. Обращение к функциям BIOS производится аналогично обращению к функциям MS DOS. Для работы с клавиатурой и экраном BIOS

содержит два программных прерывания — 16h и 10h, обращение к которым, исходя из вышесказанного, является обращением к драйверам этих устройств. Для вызова прерываний, как обычно, используется команда INT — int 16h или int 10h. Для выполнения определенной операции в регистр AH заносится номер функции. При необходимости в других регистрах может указываться дополнительная (параметрическая) информация. Ниже рассмотрим подробнее возможности BIOS для работы с консолью.

Функции BIOS для работы с клавиатурой

Прерывание 16h BIOS имеет функции для различных типов клавиатур: обычной — 84 клавиши, и двух типов расширенной клавиатуры — 101–102- и 122-клавишной. Выяснить функциональные возможности клавиатуры позволяет функция 09h:

Вход: AH = 09h.

Выход: AL = битовое поле, установленные биты которого обозначают поддерживаемые функции: 7 — резерв; 6 — поддержка клавиатуры со 122 клавишами (и функций 20h–22h (int 16h)); 5 — поддержка расширенной клавиатуры со 101–102 клавишами (и функций 10h–12h (int 16h)); 4 — поддержка функции 0Ah (int 16h); 3 — поддержка функции 0306h (int 16h); 2 — поддержка функции 0305h (int 16h); 1 — поддержка функции 0304h (int 16h); 0 — поддержка функции 0300h (int 16h).

Прежде чем вызывать эту функцию, необходимо удостовериться в том, что она поддерживается данной версией BIOS. Сделать это можно, вызвав функцию 0c0h прерывания int 15h.

Вход: AH = C0h получить конфигурацию.

Выход: CF = 1 — BIOS не поддерживает эту функцию; CF = 0 — в случае успеха: ES:BX — адрес конфигурационной таблицы в ROM-памяти; AH = состояние (00h — успех; 86h — функция не поддерживается).

Формат конфигурационной ROM-таблицы:

Смещение	Размер	Описание
00h	2 байта	Число байтов в этой таблице
02h	1 байт	Модель BIOS
03h	1 байт	Подмодель BIOS
04h	1 байт	Издание BIOS: 0 — 1-я редакция, 1 — 2-я редакция и т. д.
05h	1 байт	1-й байт свойств
06h	1 байт	2-й байт свойств
07h	1 байт	3-й байт свойств
08h	1 байт	4-й байт свойств
09h	1 байт	5-й байт свойств

Если в результате этого вызова бит 6 второго байта свойств установлен, то BIOS поддерживает функцию 09h прерывания int 16h, с помощью которой определяют функциональные возможности клавиатуры.

```

:-----+
:| Программа: prg05_01.asm. |
:| Определение функциональных возможностей клавиатуры. |
:-----+
.code
main:
        mov     ah, 0c0h
        int     15h
        push    ds
        push    es
        pop     ds
        movzx   ax, byte ptr [bx+6]
        bt      ax, 6
        pop     ds
        jnc     exit          ; функция 09h int 16h не поддерживается
                                ; (см. замечание ниже)
        mov     ah, 09h      ; функция 09h int 16h поддерживается
        int     16h          ; анализируем AL/AH (см. замечание ниже)
        ....

```

Относительно использования функций 0c0h int 15 и 09h int 16 следует сделать следующее замечание. Системы BIOS различных производителей реализуют эти функции неоднозначно. На компьютере автора с AMI BIOS функция 0c0h int 15 возвращает нулевой второй байт свойств. Что же касается результатов вызова функции 09h int 16, то эта функция возвращает битовое поле не в регистре AL, а в AH. Поэтому будьте внимательны при использовании этих функций и, если необходимо, прокомментируйте команду jnc exit в приведенном выше фрагменте.

Чтение символа (0h, 10h, 20h int 16h)

Вход: AH = 00h чтение символа с ожиданием (для 84-клавишной клавиатуры).

Выход: для обычных клавиш (AH = скан-код BIOS; AL = символ ASCII); для клавиш и комбинаций с расширенным ASCII-кодом (AH = расширенный ASCII-код; AL = 0).

Функция 00h int 16h считывает информацию об очередном символе из буфера клавиатуры, и если его там нет, то ждет его появления, приостанавливая выполнение программы. Результат работы функции необходимо оценивать по содержимому регистра AL: если AL <> 0, то нажата обычная клавиша клавиатуры и регистры AL и AH содержат соответственно символ ASCII и скан-код BIOS. В обратном случае на клавиатуре была нажата одна из клавиш (или их комбинация), соответствующая расширенному коду, и регистры AL и AH содержат соответственно нуль и расширенный код клавиши. Как известно, любой символ можно внести в буфер, если, удерживая нажатой клавишу Alt, набирать десятичный код символа на цифровой клавиатуре. В этом случае в регистре AH формируется нулевой скан-код, а в регистр AL заносится введенный код символа ASCII.

Для расширенных клавиатур (101–102 и 122 клавиши) необходимо использовать функции 10h или 20h int 16h, так как более ранние функции не поддерживают работу с дополнительными клавишами, введенными в состав этих клавиатур. При нажатии этих клавиш в регистр AL заносится код 0eh, а в регистр AH — расширенный ASCII-код.

Вход: AH = 10h, 20h чтение символа с ожиданием (для 101–102- и 122-клавишных клавиатур соответственно).

Выход: для обычных клавиш (AH = скан-код BIOS; AL = символ ASCII); для клавиш и комбинаций с расширенным кодом (AH = расширенный ASCII-код; AL = 0); для дополнительных клавиш (AH = расширенный ASCII-код; AL = 0eh).

Для ввода строки символов данные функции необходимо использовать в цикле. На примере показанной ниже программы, запустив отладчик, можно исследовать содержимое AX при нажатии различных клавиш и их комбинаций.

```

;-----+
;| prg05_02.asm. Ввод строки с использованием функции ввода символа 10h. |
;-----+
.data
string      db      5 dup (0)
len_string  dd      string - string
adr_string  dd
.code
m1:         mov     cx, len_string
            les     di, adr_string
            mov     ah, 010h
            int     16h
            stosb
            loop    m1
            ....

```

Программа вводит 5 символов и сохраняет их в строке str.

Проверка наличия символа (01h, 11h, 21h int 16h)

Вход: AH = 01h проверка наличия символа (для 84-клавишной клавиатуры).

Выход: если ZF = 0, то регистры AH и AL содержат: для обычных клавиш (AH = скан-код BIOS; AL = символ ASCII); для клавиш и комбинаций с расширенным ASCII-кодом (AH = расширенный ASCII-код; AL = 0); если ZF = 1, то буфер пуст.

Функция 01h получает информацию о символе, не удаляя его из буфера клавиатуры. Исключение составляют нажатия дополнительных клавиш на расширенных моделях, не совместимых с 83–84-клавишными клавиатурами. В процессе проверки функцией 01h они удаляются из буфера. Поэтому при работе с расширенными клавиатурами необходимо использовать функции 11h и 21h.

Вход: AH = 11h, 21h проверка наличия символа (для 101–102- и 122-клавишных клавиатур соответственно).

Выход: если ZF = 0, то регистры AH и AL содержат: для обычных клавиш (AH = BIOS скан-код; AL = символ ASCII); для клавиш и комбинаций с расширенным кодом (AH = расширенный ASCII-код; AL = 0); для дополнительных клавиш (AH = расширенный ASCII-код; AL = 0eh); если ZF = 1, то буфер пуст.

В большинстве случаев работу с результатами выполнения данной функции логично начинать с анализа флага ZF (командами JZ или JNZ). Что же касается содержимого регистра AX, то оно аналогично содержимому этого регистра в рассмотренной выше функции 00h int 16h. Совместно с функцией 00h данную функцию можно использовать в программах, управление которыми производится с клавиатуры. Частный случай таких программ — командные процессоры. Соответствующий фрагмент кода может быть следующим.

```

.data
str          db      10 dup (0)
adr_str      dd      str
.code
            ....

```

```

        les     di, adr_str
        xor     cx, cx
m1:     :----- проверяем наличие символа
        mov     ah, 01h
        int     16h
        jz      exit      : буфер пуст
        :----- извлекаем очередной символ
        mov     ah, 00h
        int     16h
        :----- пересылаем его
        stosb
        jmp     m1
        :...

```

Получение состояния флагов клавиатуры (02h, 12h, 22h int 16h)

BIOS предоставляет функцию 02h для получения состояния световых индикаторов клавиатуры и некоторых управляющих клавиш.

Вход: AH = 02h получить состояние флагов клавиатуры (для 84-клавишной клавиатуры).

Выход: AL = битовое поле, установленные биты которого соответствуют состоянию следующих флагов: 7 — режим вставки активен; 6 — индикатор Caps Lock включен; 5 — индикатор Num Lock включен; 4 — индикатор Scroll Lock включен; 3 — нажата клавиша Alt (любая клавиша Alt на 101–102-клавишной клавиатуре); 2 — нажата клавиша Ctrl (любая клавиша Ctrl на 101–102-клавишной клавиатуре); 1 — нажата левая клавиша Shift; 0 — нажата правая клавиша Shift.

Поддержка расширенных клавиатур осуществляется функциями 12h и 22h BIOS.

Вход: AH = 12h, 22h получить состояние флагов клавиатуры (для 101–102- и 122-клавишных клавиатур).

Выход: AL = первое битовое поле, установленные биты которого соответствуют состоянию флагов, возвращаемых в регистре AL функцией 02h; AH = второе битовое поле, установленные биты которого соответствуют следующему состоянию флагов: 7 — нажата клавиша Print Screen (SysRq); 6 — нажата клавиша Caps Lock; 5 — нажата клавиша Num Lock; 4 — нажата клавиша Scroll lock; 3 — нажата правая клавиша Alt; 2 — нажата правая клавиша Ctrl; 1 — нажата левая клавиша Alt; 0 — нажата левая клавиша Ctrl.

Кроме этого, состояние данных флагов можно прочесть из оперативной памяти по адресам: 0040h:0017h (AL) и 0040h:0018h (AH).

Запись символа в буфер клавиатуры (05h int 16h)

Вход: AH = 05h запись символа в буфер клавиатуры: CH = скан-код; CL = символ ASCII.

Выход: AL = состояние: 00h — успешная запись; 01h — ошибка (буфер клавиатуры заполнен).

С помощью этой функции можно подыгрывать программам, которые ожидают ввода с клавиатуры. Сам буфер клавиатуры организован по принципу кольца, имеет размер 16 байтов и занимает в памяти диапазон адресов 0040h:001Eh...0040h:003Dh. В ячейке 0040h:001Ah хранится адрес начала (головы) буфера, а в ячейке 0040h:001Ch — адрес конца (хвоста). Если содержимое этих ячеек равно, то буфер пуст.

Одному символу в буфере соответствует слово, в котором первый байт — скан-код клавиши, а второй — символ ASCII. Исследовать данную функцию можно с использованием конвейерной операции (!) MS DOS. Для этого оформим фрагмент кода для определения наличия символа в буфере и его ввода в виде отдельной программы.

```

:-----+
:| Программа: prg05_03_02.asm. Проверка наличия символа в буфере. |
:-----+
.code
:
:....
:----- проверяем наличие символа
m1:      mov     ah, 01h
:         int     16h
:         jz      exit      : буфер пуст
:----- извлекаем символ
:         mov     ah, 00h
:         int     16h
:----- выводим его средствами MS DOS (см. ниже)
:         mov     dl, al
:         mov     ah, 02h
:         int     21h
:         jmp     m1
:         :....

```

Также подготовим программу, помещающую символы из строки `str` в буфер клавиатуры.

```

:-----+
:| prg05_03_01.asm — программа, помещающая символы в буфер клавиатуры. |
:-----+
.data
str      db      "dfsh", 0
.code
:
:....
:         lea     di, str
m1:      xor     cx, cx
:         mov     ah, 05h
:         mov     cl, byte ptr [di]
:         jcxz    exit
:         int     16h
:         inc     di
:         jmp     m1
:         :....
exit:    :....

```

В командную строку MS DOS необходимо ввести строку:

```
prog_1.exe | prog_2.exe >p.txt
```

В результате всех этих действий создается файл `p.txt`, который и будет содержать строку `str` из файла `prog_1.asm`.

Функции BIOS для работы с экраном

Работа с экраном средствами BIOS производится с помощью набора функций прерывания `10h`. С помощью этих функций поддерживаются текстовый и графический режимы работы монитора. В данном разделе будут рассмотрены некоторые функции вывода текста в текстовом режиме.

Установка видеорежима (00h int 10h)

Любой дисплейный адаптер поддерживает несколько текстовых и графических режимов. Переключение между этими режимами производится с помощью функции 00h int 10h.

Вход: AH = 00h, установить видеорежим: AL = номер видеорежима (если бит 7 регистра AL = 0, то экран очищается, в обратном случае (AL.7 = 1) содержимое экрана не изменяется).

Номеров видеорежимов много, нумерация режимов с высоким разрешением (SVGA) зависит от производителя видеоадаптера, но на практике существует достаточно много SVGA-режимов, поддерживаемых в качестве стандартных. Мы не будем приводить никаких сведений по этому поводу, при необходимости информацию о нумерации видеорежимов можно получить из соответствующих источников.

Установка позиции курсора (02h int 10h)

Функция 02h позволяет изменить позицию курсора и сделать ее начальной для последующего вывода. Заметим, что среди функций MS-DOS нет подобной функции и функцию 02h int 10h BIOS можно использовать в комбинации со средствами MS-DOS для организации форматированного вывода на экран.

Вход: AH = 02h — установить позицию курсора; BH = номер видеостраницы (зависит от используемого видеорежима); DH = строка (00h — верх); DL = колонка (00h — левая).

Получение позиции курсора (03h int 10h)

Функция 03h позволяет получить текущую позицию курсора. Аналогично сказанному выше, среди функций MS-DOS нет подобной функции и функцию 03h int 10h BIOS также можно использовать в сочетании с функциями MS-DOS.

Вход: AH = 03h — получить позицию курсора; BH = номер видеостраницы (зависит от используемого видеорежима).

Выход: DH = строка текущей позиции курсора (00h — верх); DL = колонка текущей позиции (00h — левая); CH = номер начальной строки курсора; CL = номер последней строки курсора.

Запись символа и его атрибута в видеопамять (09h int 10h)

Функция 09h предназначена для записи ASCII-кода символа и его атрибута непосредственно в видеопамять, причем сделать это можно с количеством повторений, заданных в регистре CX.

Вход: AH = 09h — запись символа и его атрибута в текущую позицию курсора; BH = номер видеостраницы; AL = ASCII-код символа; BL = байт-атрибут; CX = число повторений.

Для вывода одного символа содержимое регистра CX должно быть равно 1. В текстовом режиме для CX > 1 вывод осуществляется до конца текущей строки, после чего переходит на другую строку.

Кодировка байта-атрибута в этой и других функциях производится в соответствии со следующими таблицами.

Номера битов	Значение
7	Мигающий символ
4–6	Цвет фона
3	Символ яркого цвета
0–2	Цвет символа

Кодировка цветов приведена в следующей таблице.

Биты	Цвет	Яркий цвет
000b	Черный	Темно-серый
001b	Синий	Светло-синий
010b	Зеленый	Светло-зеленый
011b	Циан	Светлый циан
100b	Красный	Светло-красный
101b	Малиновый	Светло-малиновый
110b	Коричневый	Желтый
111b	Светло-серый	Белый

Чтение символа и его атрибута из видеопамати (08h int 10h)

В памяти видеоадаптера каждый символ представлен двумя байтами, содержащими ASCII-код символа и его байт-атрибут. Функция 08h BIOS позволяет прочитать код символа и его атрибут непосредственно из видеопамати.

Вход: AH = 08h — чтение символа и его атрибута в текущей позиции курсора; BH = номер видеостраницы.

Выход: AL = ASCII-код символа; AH = байт-атрибут.

Ниже приведена программа, которая устанавливает курсор в заданную позицию и производит другие действия.

```

+-----+
:| Программа: prg05_04.asm. Установка курсора в заданную позицию,      |
:| вывод строки и чтение символа из текущей позиции.                  |
+-----+
.code
main:
    ;----- установка позиции курсора (10, 10)
    xor     bh, bh
    mov     dh, 10
    mov     dl, 10
    mov     ah, 02h
    int     10h

    ;----- записываем символ и атрибут в видеопамать
    mov     al, "a"
    mov     bl, 10001100b    ; атрибут — ярко-красный мигающий
    mov     cx, 5            ; повторить 5 раз

```

```

mov     ah, 09h
int     10h
;----- прочитаем символ из текущей позиции видеопамати
mov     ah, 08h
int     10h
;----- выясним текущую позицию курсора
xor     bh, bh
mov     ah, 03h
int     10h
;----- все результаты смотрим в отладчике
;....

```

Важно отметить, что текущая позиция курсора после выполнения функций 08h и 09h осталась неизменной. Отсюда следует важный вывод о том, что при использовании этих функций необходимо также заботиться и о движении курсора функцией 02h. BIOS предоставляет функцию 0Eh, которая отображает символы в режиме телетайпа, предполагающем автоматическую корректировку текущей позиции курсора после вывода символа.

Запись символа в видеопамать (0Ah int 10h)

Функция 0Ah предназначена для записи ASCII-кода символа с текущим значением атрибута в данной позиции непосредственно в видеопамать, причем сделать это можно с количеством повторений, заданных в регистре CX.

Вход: AH = 0Ah — запись символа в текущую позицию курсора; BH = номер видеостраницы; AL = ASCII-код символа; CX = число повторений.

Аналогично функции 09h, текущая позиция курсора не изменяется.

Запись символа в режиме телетайпа (0Eh int 10h)

Функция 0Eh выводит символ в текущую позицию курсора с автоматическим ее смещением (в отличие от функций 09h и 0Ah).

Вход: AH = 0Eh — запись символа в текущую позицию курсора; BH = номер видеостраницы; AL = ASCII-код символа; CX = число повторений.

Запись символа в последнюю позицию строки автоматически переводит курсор в первое знакоместо следующей строки.

Вывод строки (13h int 10h)

Эта функция появилась в BIOS компьютеров архитектуры AT.

Вход: AH = 13h вывод строки (AT); AL = режим записи: бит 0 — после вывода курсор перемещается в конец строки; бит 1 — каждый символ в строке представлен двумя байтами: байтом с ASCII-кодом и байтом-атрибутом; биты 2–7 — резерв; BH = номер видеостраницы; BL = байт-атрибут, если строка содержит только символы (AL.1 = 0); CX = число символов в строке; DH, DL = начальные строка и столбец на экране; ES:BP — адрес в памяти начала строки.

Обратите внимание, что содержимое строки для вывода может быть двух типов: с байтом-атрибутом, сопровождающим каждый символ строки, и без него. В последнем случае строка состоит из одних кодов символов с единым значением байта-атрибута, указываемым в регистре BL.

Как видно, многие функции BIOS работают непосредственно с видеопаматью. Из-за того что для видеопамати отводится определенный диапазон адресов (для текстового режима это начальный адрес 0b800h:0000h), доступ к ней можно произ-

водить обычными командами работы с памятью процессора, в том числе и цепочечными.

Прокрутка окна вверх (06h int 10h)

Функция 06h позволяет определить на экране окно, в котором возможно прокрутить определенное количество строк вверх. При такой прокрутке верхние строки исчезают и снизу добавляются пустые строки.

Вход: AH = 06h — перемещение строк в окне вверх; AL = число строк для заполнения снизу; BH = атрибут символов (цвет) в строке для заполнения; CH и CL = строка и столбец верхнего левого угла окна; DH и DL = строка и столбец нижнего правого угла окна.

Строки для заполнения снизу имеют цвет, определенный в BH. Если указать AL = 0, то окно очистится и заполнится строками с цветом, заданным байтом-атрибутом в BH.

Ниже приведена программа вывода нескольких строк на экран, которая определяет окно на экране и прокручивает его на несколько строк вверх.

```

:-----+
:| Программа: prg05_05.asm. Работа с окном на экране. |
:-----+
.data
string      db      "dfsh3453637869шораервванв"
len_string  = $ - string
adr_string  dd      string
.code
:....
ml:         mov     cx, 25
            mov     al, 1          ; после вывода — курсор в конец строки
            xor     bh, bh         ; номер видеостраницы
            mov     bl, 7          ; атрибут
            push    cx
            mov     cx, len_string ; длина выводимой строки
            les     bp, adr_string ; адрес строки в пару ES:BP
            mov     ah, 13h
            int     10h
            inc     dh              ; строка начала вывода
            inc     dl              ; столбец начала вывода
            pop     cx
            loop    ml
:----- определяем и прокручиваем окно вверх
            mov     al, 4           ; 4 строки
            mov     bh, 0
            mov     ch, 5
            mov     cl, 5
            mov     dh, 10
            mov     dl, 30
            mov     ah, 06h
            int     10h
:....

```

Заметьте, что функция 06h достаточно гибко работает с курсором.

Прокрутка окна вниз (07h int 10h)

Функция 07h позволяет определить на экране окно, в котором возможно прокрутить определенное количество строк вниз. При такой прокрутке нижние строки исчезают и сверху добавляются пустые строки.

Вход: AH = 07h — перемещение строк в окне вниз; AL = число строк для заполнения сверху; BH = атрибут символов (цвет) в строке для заполнения; CH и CL = строка и столбец верхнего левого угла окна; DH и DL = строка и столбец нижнего правого угла окна.

Строки для заполнения сверху имеют цвет, определенный в BH. Если указать AL = 0, то окно очистится и заполнится строками с цветом, заданным в BH. Структура байта-атрибута аналогична описанной выше.

Функции MS DOS для работы с консолью

Ценность программы прямо пропорциональна весу ее «выдачи».

Прикладная Мерфология

Функции MS DOS для работы с консолью сосредоточены в обработчике прерывания int 21h. Они представляют собой набор средств работы с консолью, занимающий промежуточное положение между программами пользователя и средствами BIOS. Для достижения большей эффективности некоторые из функций BIOS можно комбинировать с функциями MS DOS. Как пример такого полезного взаимодействия можно привести задействование возможностей BIOS по работе с курсором. Как будет видно из приведенного ниже материала, среди функций MS DOS подобные средства отсутствуют. При выполнении конкретных практических заданий можно найти и другие полезные примеры взаимодействия.

Функции MS DOS для ввода данных с клавиатуры

Для ввода данных с клавиатуры можно использовать два вида функций: универсальную функцию 3fh (ввод из файла) и группу специализированных функций MS DOS ввода с клавиатуры.

Подробно использование функции 3fh для ввода данных рассматривается в главе 7, а здесь сфокусируемся на второй группе, в которую входит семь функций, отличающихся друг от друга следующими характеристиками:

- ожиданием ввода при отсутствии символа в буфере клавиатуры или только проверкой буфера на наличие символов для ввода;
- количеством вводимых символов;
- наличием «эха» при вводе, то есть дублированием набираемого на клавиатуре символа на экране;
- восприимчивостью к сочетанию клавиш Ctrl+C (код 03h).

Чтение с эхом символа с клавиатуры (01h int 21h)

Функция 01h позволяет ввести один символ с клавиатуры. Если символа нет, то функция ожидает его ввода. Вводимый символ отображается на экране («эхо»).

Вход: AH = 01h — чтение символа с эхом.

Выход: AL = ASCII-код символа или 0.

На выходе функция помещает в регистр AL ASCII-код символа или 0. Наличие нуля в регистре AL говорит о том, что в буфере клавиатуры находится расширен-

ный ASCII-код и необходимо повторить вызов функции с тем, чтобы прочитать его второй байт. Также функция 01h проверяет наличие в буфере символов нажатия комбинации Ctrl+C (Ctrl+Break), при обнаружении которых производится вызов прерывания int 23h.

Для ввода нескольких символов данную функцию необходимо использовать в цикле.

```

+-----+
+| Программа: prg05_06.asm. Ввод нескольких символов функцией 01h 21h. |
+-----+
.data
string      db      5 dup (0)
len_string = $ - string
adr_string  dd      string
.code
;...
mov     cx, len_string
les     di, adr_string
m1:     mov     ah, 01h
        int     21h
        cmp     al, 0          ; расширенный код?
        jne     m2
;----- обрабатываем расширенный код
;...
jmp     m3
;----- формируем строку символов
m2:     ;...
        stosb
m3:     loop    m1
;...

```

Проверяя работу программы, вместо ввода очередного символа введите комбинацию Ctrl+C и посмотрите реакцию программы.

Прямой ввод с эхом символа с клавиатуры (06h int 21h)

Функция 06h также позволяет ввести один символ с клавиатуры. Но, в отличие от функции 01h, она не ожидает ввода при отсутствии символа в буфере. Вводимый символ отображается на экране («эхо»).

Вход: AH = 06h — чтение символа с эхом без ожидания; DL = 0ffh — признак того, что функция 06h используется для ввода; если DL <> 0ffh, то функция вызывается для вывода символа (см. ниже).

Выход: если ZF = 0, то AL = ASCII-код символа; если ZF = 1, то символа в буфере нет.

Результаты работы этой функции необходимо оценивать прежде всего по значению флага ZF. Если ZF = 0, то функция поместила в регистр AL ASCII-код символа или 0. Наличие нуля в регистре AL говорит о том, что в буфере клавиатуры находится расширенный ASCII-код и необходимо повторить вызов функции с тем, чтобы прочитать его второй байт. Функция 06h не проверяет наличие в буфере символов нажатия комбинации Ctrl+C (Ctrl+Break).

Чтение без эха символа с клавиатуры (07h int 21h)

Функция 07h аналогична функции 01h, за исключением того, что читает символ с клавиатуры без ожидания его ввода, без «эха» и без проверки нажатия комбинации Ctrl+C (Ctrl+Break).

Вход: AH = 07h — чтение символа без «эха».

Выход: AL = ASCII-код символа или 0 (см. описание функции 01h int 21h)

Наличие нуля в регистре AL говорит о том, что в буфере клавиатуры находится расширенный ASCII-код и необходимо повторить вызов функции с тем, чтобы прочесть его второй байт.

Чтение без эха символа с клавиатуры (08h int 21h)

Функция 08h аналогична функции 01h, за исключением того, что вводит символ с клавиатуры без отображения его на экране (без «эха»).

Вход: AH = 08h — чтение символа без «эха».

Выход: AL = ASCII-код символа или 0 (см. описание функции 01h int 21h).

Наличие нуля в регистре AL говорит о том, что в буфере клавиатуры находится расширенный ASCII-код и необходимо повторить вызов функции с тем, чтобы прочесть его второй байт. Отличие от предыдущей функции в том, что эта функция производит проверку нажатия комбинации Ctrl+C (Ctrl+Break), при наличии которого вызывается прерывание int 23h.

Ввод строки символов с клавиатуры (0Ah int 21h)

Функция 0Ah вводит строку символов в буфер памяти специального формата. Если символов в буфере клавиатуры нет, то функция ожидает их появления. Конец ввода — нажатие клавиши Enter (ASCII-код 0dh). Формат буфера:

- первый байт буфера содержит количество символов для ввода с учетом символа 0dh, прекращающего процесс ввода;
- второй байт содержит реальное число введенных символов, но уже без учета завершающего символа 0dh;
- начиная с третьего байта размещается строка введенных символов с завершающим символом 0dh, максимальная длина строки — 254 символа.

Вход: AH = 0ah — ввод строки в буфер (до 254 символов); DS:DX — адрес буфера, первый байт которого должен содержать количество символов для ввода.

Выход: введенная строка, начиная с третьего байта буфера по адресу в DS:DX, длина строки — во втором байте буфера.

Перед вызовом функции 0Ah в первый байт буфера необходимо поместить значение максимальной длины строки. Если первый байт равен нулю, то вызов функции игнорируется и программа продолжает выполнение без ожидания ввода строки. Функция производит проверку нажатия клавиатурной комбинации Ctrl+C (Ctrl+Break), при наличии которого вызывается прерывание int 23h. Вводимая строка отображается на экране. Буфер ввода для данной функции лучше оформлять в виде структуры.

```

+-----+
:| Программа: prg05_07.asm. Ввод строки функцией 0Ah int 21h. |
+-----+
buf_0ah      struc
len_buf      db          11      ; длина buf_0ah
len_in       db          0       ; действительная длина введенного
                                   ; слова (без учета 0dh)
buf_in       db      11 dup (20h) ; буфер для ввода (с учетом 0dh)
ends
    
```

```

.data
buf
adr_buf
.code
        buf_0ah <>
        dd      buf

;-----
;...
; вводим 10 символов с клавиатуры
        lds     dx, adr_buf
        mov     ah, 0ah
        int     21h
;-----
; обработка введенной строки
;...

```

Получить состояние клавиатуры (0Bh int 21h)

Функция 0Bh проверяет наличие в буфере символа для ввода.

Вход: AH = 0Bh — проверка состояния клавиатуры.

Выход: AL = 0ffh — буфер клавиатуры содержит символ для ввода; AL = 0 — буфер клавиатуры пуст.

Данная функция формирует только логический результат — присутствует символ в буфере или буфер пуст, поэтому вызов функции 0Bh необходимо комбинировать с одной из функций извлечения символа из буфера ввода. Использование этой функции удобно для программ, управление которыми производится с клавиатуры, — типа командного процессора. В процессе своей работы они постоянно ожидают ввода пользователем управляющих команд, в связи с чем периодически проверяют входной буфер.

Функция производит проверку нажатия комбинации Ctrl+C (Ctrl+Break), при наличии которого вызывается прерывание int 23h.

Ввод с клавиатуры с предварительной очисткой буфера (0Ch int 21h)

Функция 0Ch выполняет ввод, предварительно очищая буфер клавиатуры. Это удобно для предотвращения чтения из буфера оставшихся там символов, возможно, введенных ошибочно или случайно. Функция гарантирует, что программа получит именно те данные, которые ввел оператор. Важно отметить, что функция 0Ch выполняет только очистку буфера, ввод символа осуществляет одна из функций, номер которой указывается в регистре AL при вызове этой функции.

Вход: AH = 0Ch — ввод с клавиатуры с предварительной очисткой; AL = номер функции (01h, 06h, 07h, 08h, 0ah).

Выход: определяется функцией, указанной в AL при вызове функции.

Функция производит проверку нажатия комбинации Ctrl+C (Ctrl+Break), при наличии которого вызывается прерывание int 23h.

Функции MS DOS для вывода данных на экран

Для вывода данных на экран можно использовать два вида функций: универсальную функцию 40h (вывод в файл через дескриптор) и группу специализированных функций MS DOS вывода на экран.

Использование функции 40h будет рассматриваться в главе, посвященной работе с файлами. Материал, представленный ниже, посвящен второй группе функций — для вывода символов на экран средствами MS DOS. В группу входят три функции. Рассмотрим их.

Вывод символа на экран (02h int 21h)

Функция 02h позволяет вывести один символ на экран.

Вход: AH = 02h — вывод символа; DL = символ для вывода.

Функция 02h проверяет наличие в клавиатурном буфере символов нажатия комбинации Ctrl+C (Ctrl+Break), при обнаружении которых производится вызов прерывания int 23h. В процессе работы функция реагирует на управляющие символы, такие как 0dh (возврат каретки), 0ah (перевод строки), 08h (курсор назад на один символ), 07h (звуковой сигнал) и т. д.

Здесь, для того чтобы вывести строку, необходимо организовать цикл.

```

:-----+
:| Программа: prg05_08.asm. Посимвольный вывод строки функцией 02h int 21h. |
:-----+
.data
string      db      "Строка для вывода функцией 02h"
             len_string = $ - string
.code
             ;...
:----- выводим строку string на экран
             mov     cx, len_string ; длина строки
             lea     si, string      ; адрес строки
ml:          mov     ah, 02h
             mov     dl, [si]
             int     21h
             inc     si
             loop    ml
             ;...

```

Прямой вывод символа на экран (06h int 21h)

Функция 06h выводит один символ на экран. Эта функция универсальна, так как используется и для ввода (см. выше), и для вывода символа.

Вход: AH = 06h — вывод символа на экран; DL = символ для вывода (за исключением 0ffh).

Функция 06h не проверяет наличие в буфере символов нажатия комбинации Ctrl+C (Ctrl+Break). Порядок использования данной функции аналогичен таковому для функции 02h.

Вывод строки на экран (09h int 21h)

Функция 09h отображает строку символов на экране. Строка должна обязательно заканчиваться символом \$. Данную функцию удобно использовать для вывода на экран различных диагностических сообщений. Если требуется организовать вывод строк, длина которых формируется динамически, то лучше либо прибегнуть к упомянутой выше функции 40h, либо выводить их в цикле, тело которого содержит одну из функций — 02h или 06h.

Вход: AH = 09h — вывод строки на экран; DS:DX — адрес строки для вывода с завершающим символом \$.

Функция 09h проверяет наличие в клавиатурном буфере символов нажатия комбинации Ctrl+C (Ctrl+Break), при обнаружении которых производится вызов прерывания int 23h. В процессе вывода она реагирует на управляющие символы, такие как 0dh (возврат каретки), 0ah (перевод строки), 08h (курсор назад на один символ), 07h (звуковой сигнал) и т. д.

Приведенный ниже фрагмент показывает порядок применения функции 09h int 21h.

```

:-----+
: | Программа: prg05_09.asm. Вывод строки на экран функцией 09h int 21h. |
:-----+
.data
string      db      "Строка для вывода функцией 09h $"
adr_string  dd
.code
        :...
:-----+ выводим строку string на экран
        lds      dx, adr_string ; адрес строки в DS:DX
        mov      ah, 09h
        int      21h
        :...

```

Работа с консолью в среде Windows

Если ничто другое не помогает, прочтите, наконец, инструкцию.

Прикладная Мерфология

Windows поддерживает работу двух типов приложений — оконных, в полной мере использующих все достоинства графического интерфейса, и консольных, работающих исключительно в текстовом режиме. Поэтому не следует путать понятие «консоли», используемое выше, с понятием «консольного приложения» Windows. В предшествующем материале под «консолью» подразумевались средства для ввода информации с клавиатуры и вывода ее на экран. Для однозначности изложения далее под термином «консоль» мы будем иметь в виду либо само консольное приложение, либо его видимую часть — окно консольного приложения.

В главе 16 учебника были изложены основы программирования консольных приложений Windows. В этом материале была рассмотрена минимальная программа консольного приложения и средства для автоматизации высокоуровневого консольного ввода-вывода. Поэтому, для экономии места, этот материал здесь рассматриваться не будет. Будут показаны лишь средства для организации низкоуровневого ввода-вывода. Те читатели, которые не собираются приобретать учебник, могут найти необходимый материал среди файлов, прилагаемых к книге и выложенных на сайте. Организация ввода-вывода в оконном приложении Windows здесь также рассматриваться не будет, так как в главах 16 «Создание Windows-приложений на ассемблере» и 17 «Архитектура и программирование сопроцессора» учебника этот вопрос был рассмотрен очень подробно и полно. Что-либо существенное добавить к уже сказанному трудно.

Организация низкоуровневого консольного ввода-вывода

Низкий уровень консольного ввода-вывода, по сравнению с высоким уровнем, обладает более широкими и гибкими возможностями. Низкоуровневые функции консольного ввода-вывода обеспечивают прямой доступ к входному и экранным буферам консоли, предоставляя приложению доступ к событиям мыши и клави-

атуры, а также к информации об изменении размеров окна консоли. Функции низкоуровневого ввода-вывода позволяют приложению иметь доступ по чтению-записи к указанному числу последовательных символьных ячеек в экранном буфере или к прямоугольному блоку символьных ячеек в указанной позиции экранного буфера.

Обсудим возможности низкоуровневого ввода-вывода на примере работы с входным буфером (входной очередью) и буферами экрана. Отметим, для работы с ними существуют разные группы команд. Так, для операций с входным буфером используются функции низкоуровневого ввода-вывода — `WriteConsoleInput/ReadConsoleInput`. Группа функций для работы с буферами экрана будет конспективно описана в конце этого раздела.

Поддержка работы с мышью в консоли

Большое достоинство консольных приложений — встроенная средствами Windows поддержка мыши. Она реализуется с помощью функции `ReadConsoleInput`. Важно отметить, что эта функция предназначена для получения информации о событиях не только мыши, но и о событиях клавиатуры, изменении размера окна и т. д.

```
BOOL ReadConsoleInput(HANDLE hConsoleInput, PINPUT_RECORD lpBuffer, DWORD nLength,
    LPDWORD lpNumberOfEventsRead);
```

Параметры этой функции:

- `hConsoleInput` — стандартный дескриптор ввода, полученный функцией `GetStdHandle`;
- `lpBuffer` — указатель на буфер, в который записывается информация о событии мыши, — эта область памяти имеет структуру, называемую `INPUT_RECORD`, ее формат рассмотрен чуть ниже (необходимо заметить, что возможно групповое чтение информации из входного буфера, поэтому указатель `lpBuffer` может указывать на массив структур; информация о том, сколько событий будет читаться в этот массив структур, определяется параметром `nLength`);
- `nLength` — размер буфера (во входных записях), на который ссылается указатель `lpBuffer`;
- `lpNumberOfEventsRead` — определяет переменную, в которую записывается действительное число прочитанных записей входного буфера.

Запись входного буфера консоли имеет структуру, называемую `INPUT_RECORD`. Ее описание на языке C++ выглядит так:

```
typedef struct _INPUT_RECORD {
    WORD EventType;
    union {
        KEY_EVENT_RECORD KeyEvent;
        MOUSE_EVENT_RECORD MouseEvent;
        WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
        MENU_EVENT_RECORD MenuEvent;
        FOCUS_EVENT_RECORD FocusEvent;
    } Event;
} INPUT_RECORD;
```

В этой структуре первое поле `EventType` размером в слово содержит тип события, а второе поле `Event` является объединением различных структур. Поля какой из структур будут заполнены, регламентируется типом события, то есть первым полем, которое может принимать значения:

- **KEY_EVENT = 0001h** — поле **Event** содержит структуру **KEY_EVENT_RECORD** с информацией относительно события клавиатуры;
- **MOUSE_EVENT = 0002h** — поле **Event** содержит структуру **MOUSE_EVENT_RECORD** с информацией относительно движения мыши или нажатия ее кнопок;
- **WINDOW_BUFFER_SIZE_EVENT = 0004h** — поле **Event** содержит структуру **WINDOW_BUFFER_SIZE_RECORD** с информацией о новом размере экранного буфера;
- **MENU_EVENT = 0008h** — поле **Event** содержит структуру **MENU_EVENT_RECORD** (это событие используется внутри Windows и должно игнорироваться);
- **FOCUS_EVENT = 0010h** — поле **Event** содержит структуру **FOCUS_EVENT_RECORD** (используется внутри Windows и должно игнорироваться).

Для обработки события мыши структура **MOUSE_EVENT_RECORD** выглядит так:

```
typedef struct _MOUSE_EVENT_RECORD {
    COORD dwMousePosition;
    DWORD dwButtonState;
    DWORD dwControlKeyState;
    DWORD dwEventFlags;
} MOUSE_EVENT_RECORD;
```

Исходя из вышесказанного, структура **INPUT_RECORD** для обработки событий мыши в программе на ассемблере должна выглядеть следующим образом:

```
INPUT_RECORD struct
EventType      dw      0
dwMousePosition struct
x              dw      0
y              dw      0
ends
dwButtonState  dw      0
dwControlKeyState dw    0
dwEventFlags   dw      0
ends
```

Поле **EventType** для события мыши содержит значение **MOUSE_EVENT = 0002h**, а поля структуры **MOUSE_EVENT_RECORD** соответственно означают следующее:

- **dwMousePosition** — координаты мыши в окне консоли (в знакоместах);
- **dwButtonState** — состояние кнопок мыши в момент возникновения события, при нажатии кнопок устанавливаются следующие биты (при одновременном нажатии устанавливается несколько битов):
 - если установлен бит 0 поля **dwButtonState**, то в момент наступления события была нажата левая кнопка мыши;
 - если установлен бит 1 поля **dwButtonState**, то в момент наступления события была нажата правая кнопка мыши;
 - если установлен бит 2 поля **dwButtonState**, то в момент наступления события была нажата средняя кнопка мыши, если она есть;
- **dwControlKeyState** — поле описывает состояние управляющих клавиш клавиатуры в момент наступления события мыши (если одновременно нажато несколько клавиш, то значение в этом поле является результатом операции логического сложения (ИЛИ) перечисленных ниже значений):
 - **RIGHT_ALT_PRESSED = 0001h** — нажата правая клавиша Alt;
 - **LEFT_ALT_PRESSED = 0002h** — нажата левая клавиша Alt;

- `RIGHT_CTRL_PRESSED = 0004h` — нажата правая клавиша Ctrl;
- `LEFT_CTRL_PRESSED = 0008h` — нажата левая клавиша Ctrl;
- `SHIFT_PRESSED = 0010h` — нажата любая клавиша SHIFT;
- `NUMLOCK_ON = 0020h` — индикатор Num Lock включен;
- `SCROLLLOCK_ON = 0040h` — индикатор Scroll Lock включен;
- `CAPSLOCK_ON = 0080h` — индикатор Caps Lock включен;
- `ENHANCED_KEY = 0100h` — нажата клавиша расширенной клавиатуры (101 и 102 клавиши): Ins, Del, Home, End, Page Up, Page Down, ←, ↑, →, ↓, / или Enter;
- `dwEventFlags` — поле содержит одно из двух значений:
 - `MOUSE_MOVED = 0001h` — перемещение мыши;
 - `DOUBLE_CLICK = 0002h` — выполнен двойной щелчок кнопки мыши.

Среди материалов, прилагаемых к книге, имеется демонстрационная программа обработки событий мыши (`prg05_13.asm`), которые отслеживаются следующим образом: нажатие левой кнопки приводит к выводу сообщения в позиции нажатия, нажатие правой кнопки влечет за собой завершение работы программы.

В заключение обращаю внимание читателя на то, что API Win32 имеет функцию `Mouse_Event`, которая позволяет генерировать события, соответствующие реальным движениям мыши и щелчкам ее кнопок. Тем самым API Win32 предоставляет механизм для создания обучающих и демо-версий программ. Формат этой функции: `VOID mouse_event(DWORD dwFlags, DWORD dx, DWORD dy, DWORD dwData, DWORD dwExtraInfo)`

Расширенная поддержка клавиатуры в консоли

Функции работы с текстом высокого уровня не дают других возможностей оперирования клавиатурой, кроме как примитивного ввода текста. При разработке программ текстового режима часто требуется информация о состоянии управляющих клавиш, о факте удержания клавиши, что может свидетельствовать о желании пользователя повторить ввод некоторого символа или просто о желании получить тривиальный скан-код клавиши. Эти и другие события клавиатуры доступны программе посредством описанной выше функции `ReadConsoleInput`.

События клавиатуры генерируются при нажатии любой клавиши. Процесс их обработки аналогичен обработке событий мыши. В первую очередь заполняется экземпляр структуры `INPUT_RECORD`. При этом в поле `EventType` помещается значение `KEY_EVENT = 0001h` — это означает, что объединение `Event` содержит структуру `KEY_EVENT_RECORD` с информацией относительно события клавиатуры. Указанная структура включает в себя следующие поля:

```
typedef struct _KEY_EVENT_RECORD {
    BOOL bKeyDown;
    WORD wRepeatCount;
    WORD wVirtualKeyCode;
    WORD wVirtualScanCode;
    union {
        WCHAR UnicodeChar;
        CHAR AsciiChar;
    } uChar;
    DWORD dwControlKeyState;
} KEY_EVENT_RECORD, *PKEY_EVENT_RECORD;
```

Тот же код в переводе на язык ассемблера:

```

KEY_EVENT_RECORD struc
bKeyDown          db    0
wRepeatCount       dw    0
wVirtualKeyCode     dw    0
wVirtualScanCode   dw    0
union
UnicodeChar        dw    0
AsciiChar           db    0
ends
dwControlKeyState  dd    0
ends

```

Поля этой структуры означают:

- **bKeyDown** — содержит 1 («истина»), если клавиша была нажата, и 0 («ложь»), если клавиша была отпущена;
- **wRepeatCount** — содержит количество повторных кодов при удержании клавиши в нажатом состоянии;
- **wVirtualKeyCode** — содержит виртуальный код клавиши, который идентифицирует данную клавишу не зависящим от устройства способом;
- **wVirtualScanCode** — содержит виртуальный скан-код данной клавиши, который представляет собой аппаратно-зависимое значение, сгенерированное аппаратными средствами клавиатуры;
- объединение **AsciiChar** и **UnicodeChar** — содержит либо ASCII-код клавиши, либо Unicode-код;
- **DwControlKeyState** — описывает состояние управляющих клавиш (это поле содержит такие же значения, как и аналогичное поле в структуре, описывающей события мыши: **RIGHT_ALT_PRESSED**, **LEFT_ALT_PRESSED**, **RIGHT_CTRL_PRESSED**, **LEFT_CTRL_PRESSED**, **SHIFT_PRESSED**, **NUMLOCK_ON**, **SCROLLLOCK_ON**, **CAPSLOCK_ON**, **ENHANCED_KEY**).

Среди материалов, прилагаемых к книге, есть демонстрационная программа обработки событий клавиатуры (**prg05_14.asm**). Она построена на основе предыдущей программы и дополняет ее возможностью обработки событий клавиатуры, сообщая пользователю о нажатии некоторых управляющих клавиш. Для всех остальных клавиш просто фиксируется факт нажатия. При этом необходимо помнить, что однократному нажатию клавиши реально соответствуют два события — нажатие и отпускание клавиши. В связи с этим программа выводит два сообщения. На практике избыточности можно избежать, анализируя поле **bKeyDown**: **bKeyDown = 1**, когда клавиша нажата; **bKeyDown = 0**, когда клавиша отпущена. Выход из программы — при выполнении любых действий с мышью.

Окно консоли и экранный буфер

В заключение обсуждения особенностей работы с консольными приложениями разберемся, что представляет собой экранный буфер консоли и какие средства предоставляет API **Win32** для работы с ним.

Для того чтобы легко понять соотношение понятий «окно консоли» и «экранный буфер консоли», представьте себе офисный календарь, на котором текущее число отмечается квадратной рамкой, закрепленной на прозрачной целлофановой ленте и перемещающейся вдоль нее. Теперь представим, что содержимое листа

календаря вне этой рамки невидимо, то есть доступно только через окошко, которое образует рамка. Для того чтобы увидеть содержимое всего листа календаря, необходимо двигать рамку. В контексте этой ассоциации — лист календаря — это экранный буфер (логический экран), а площадь внутри рамки — окно консоли, то есть видимая часть экранного буфера.

Возможна поддержка нескольких экранных буферов, связанных с данной консолью, но только один из них может подвергаться отображению в окне консоли — его называют активным экранным буфером. Другие экранные буферы, если они были созданы, являются неактивными. Для создания экранного буфера используется функция `CreateConsoleScreenBuffer`. К неактивным экранным буферам можно обращаться для чтения и записи, но отображаться в окне консоли будет только активный экранный буфер (или его часть). Для того чтобы сделать экранный буфер активным, вызывается функция `SetConsoleActiveScreenBuffer`. Функция `CreateConsoleScreenBuffer` имеет показанный ниже формат.

```
HANDLE CreateConsoleScreenBuffer(DWORD dwDesiredAccess, DWORD dwShareMode,
    CONST LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwFlags,
    LPVOID lpScreenBufferData);
```

Параметры функции:

- **dwDesiredAccess** — определяет желаемый тип доступа к экранному буферу консоли, этот параметр может быть одним из следующих значений либо их комбинацией:
 - **GENERIC_READ** = 80000000h — запрашивается доступ по чтению к экранному буферу консоли для того, чтобы разрешить процессу прочитать данные из буфера;
 - **GENERIC_WRITE** = 40000000h — запрашивается доступ для записи к экранному буферу консоли для того, чтобы разрешить процессу записать данные в буфер;
- **dwShareMode** — определяет возможность разделения экранного буфера консоли; нулевое значение этого параметра указывает, что буфер не может быть разделен, ненулевое состояние параметра может быть одним из следующих значений или их комбинацией:
 - **FILE_SHARE_READ** — другие операции открытия могут быть выполнены для экранного буфера консоли с доступом для чтения;
 - **FILE_SHARE_WRITE** — другие операции открытия могут быть выполнены для экранного буфера консоли с доступом для записи;
- **lpSecurityAttributes** — указатель на структуру `SECURITY_ATTRIBUTES`, которая определяет, может ли возвращаемый функцией `CreateConsoleScreenBuffer` дескриптор наследоваться дочерними процессами, — если `lpSecurityAttributes` = `NULL`, то дескриптор не наследуется;
- **dwFlags** — определяет тип создаваемого экранного буфера консоли, в настоящее время поддерживается только один такой тип — `CONSOLE_TEXTMODE_BUFFER` = 1;
- **lpScreenBufferData** — зарезервирован и должен быть равен `NULL`.

Функция `CreateConsoleScreenBuffer` формирует дескриптор созданного экранного буфера, который затем используется функциями для доступа к этому буферу.

Для того чтобы сделать буфер активным, задействуют функцию **SetConsoleActiveScreenBuffer**.

```
BOOL SetConsoleActiveScreenBuffer(HANDLE hConsoleOutput);
```

Функция имеет единственный параметр — **hConsoleOutput** — дескриптор экранного буфера, созданного функцией **CreateConsoleScreenBuffer**. Как уже было отмечено, консоль в состоянии иметь много экранных буферов. Функция **SetConsoleActiveScreenBuffer** определяет, какой из них будет отображен. Приложение может производить запись в неактивный экранный буфер и затем обращаться к функции **SetConsoleActiveScreenBuffer** для отображения содержимого буфера. Чтение и запись в неактивный (и активный тоже) экранный буфер производится функциями низкоуровневого ввода-вывода — **WriteConsoleOutput/WriteConsoleOutputCharacter** и **ReadConsoleOutput/ReadConsoleOutputCharacter**, которым при вызове передается дескриптор нужного экранного буфера, полученного предварительно функцией **CreateConsoleScreenBuffer**.

Каждый из созданных экранных буферов поддерживает собственный текущий прямоугольник окна, определяемый координатами верхней левой и нижней правой символьных ячеек, которые будут отображены в окне консоли. Для определения видимого в окне консоли прямоугольника экранного буфера используется функция **GetConsoleScreenBufferInfo**.

```
BOOL GetConsoleScreenBufferInfo(HANDLE hConsoleOutput,
    PCONSOLE_SCREEN_BUFFER_INFO lpConsoleScreenBufferInfo);
```

Параметрами этой функции являются:

- **hConsoleOutput** — дескриптор экранного буфера, созданного функцией **CreateConsoleScreenBuffer**; этот идентификатор должен иметь тип доступа **GENERIC_READ**;
- **lpConsoleScreenBufferInfo** — указатель на структуру **CONSOLE_SCREEN_BUFFER_INFO**, в которую помещается информация об экранном буфере.

Структура **CONSOLE_SCREEN_BUFFER_INFO** имеет следующий вид:

```
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
    COORD    dwSize;                // размер экранного буфера в колонках и строках
    COORD    dwCursorPosition;     // координаты столбца и строки курсора
                                        // в экранном буфере
    WORD     wAttributes;          // цвет фона и текста, с которыми записываются
                                        // и отображаются символы в экранном буфере
                                        // функциями WriteFile/WriteConsole и
                                        // ReadFile/ReadConsole
    SMALL_RECT srWindow;           // определяет структуру SMALL_RECT, которая
                                        // содержит координаты левого верхнего и нижнего
                                        // правого углов экранного буфера,
                                        // видимого в окне консоли на экране дисплея
    COORD    dwMaximumWindowSize;  // определяет максимальный размер окна консоли
                                        // с учетом текущего размера экранного буфера
                                        // и шрифт
} CONSOLE_SCREEN_BUFFER_INFO ;
```

Для приложения интерес, в частности, представляет параметр **srWindow** с координатами видимой части экранного буфера. Далее, руководствуясь действиями пользователя (выполняющего прокрутку окна или изменение его размера) по отношению к окну консоли, приложение может изменять значения в структуре **SMALL_RECT** и передавать ее на вход функции **SetConsoleWindowInfo**, которая устанавливает текущий размер и позицию окна консоли относительно экранного буфера.

```
BOOL SetConsoleWindowInfo(HANDLE hConsoleOutput, BOOL bAbsolute,
    CONST SMALL_RECT *lpConsoleWindow);
```

Параметрами этой функции являются:

- **hConsoleOutput** — дескриптор экранного буфера, созданного функцией **CreateConsoleScreenBuffer**; этот идентификатор должен иметь тип доступа **GENERIC_WRITE**;
- **bAbsolute** — определяет порядок использования координат в структуре, указанной параметром **lpConsoleWindow**; если **bAbsolute = 1** («истина»), то координаты определяют новые левый верхний и нижний правый углы окна; если **bAbsolute = 0** («ложь»), то координаты — смещения относительно текущих координат углов окна;
- **lpConsoleWindow** — указатель на структуру **SMALL_RECT**, в которую записывается информация о новых координатах экранного буфера.

Структура **SMALL_RECT** имеет следующий вид:

```
typedef struct _SMALL_RECT {
    SHORT Left;      // x-координата верхнего левого угла
    SHORT Top;       // y-координата верхнего левого угла
    SHORT Right;     // x-координата нижнего правого угла
    SHORT Bottom;    // y-координата нижнего правого угла
} SMALL_RECT;
```

При работе с функцией **SetConsoleWindowInfo** следует иметь в виду, что она возвращает ошибку (нулевое значение), если координаты видимой части экранного буфера указывают за его действительные границы. Максимально допустимый размер окна для данной консоли можно получить с помощью функции **GetConsoleScreenBufferInfo**. Таким образом, обе эти функции подходят для листания экранного буфера.

Для закрытия экранного буфера используется функция **CloseHandle**, которой передается идентификатор закрываемого экранного буфера.

```
BOOL CloseHandle(HANDLE hObject);
```

Для того чтобы завершить рассмотрение функций, предназначенных для поддержки консольного приложения, перечислим те из них, что остались «за кадром».

Функция	Назначение
FillConsoleOutputAttribute	Устанавливает цвет текста и фона для указанного числа символьных ячеек, начинающихся по указанным координатам в экранном буфере
FillConsoleOutputCharacter	Запись символа в экранный буфер указанное число раз по указанным координатам
FlushConsoleInputBuffer	Сброс на диск входного буфера консоли. Все входные записи во входном буфере консоли до настоящего момента времени удаляются
GenerateConsoleCtrlEvent	Посылка сигнала, определенного этой функцией, совместно использующим консоль процессам
GetConsoleCursorInfo	Предоставление информации о размере и видимости курсора для указанного экранного буфера

продолжение ➤

Функция	Назначение
GetConsoleMode	Предоставление информации о текущем входном режиме входного буфера консоли или текущем режиме вывода экранного буфера консоли
GetConsoleTitle	Извлечение строки из области заголовка для текущего окна консоли
GetLargestConsoleWindowSize	Возвращает размер самого большого возможного окна консоли, основанного на текущем шрифте и размере изображения
GetNumberOfConsoleInputEvents	Возвращает число непрочитанных записей ввода во входном буфере
GetNumberOfConsoleMouseButtons	Возвращает число кнопок на мыши, используемых текущей консолью
PeekConsoleInput	Чтение данных из входного буфера консоли без их удаления
ScrollConsoleScreenBuffer	Перемещение блока данных в экранном буфере. Действие перемещения может быть ограничено путем определения отсекающего прямоугольника. Содержание экранного буфера вне отсекающего прямоугольника будет неизменным
SetConsoleCursorInfo	Установка размера и видимости курсора для указанного экранного буфера консоли
SetConsoleMode	Установка режима входного буфера консоли или режима вывода экранного буфера консоли
SetConsoleScreenBufferSize	Изменение размера указанного экранного буфера консоли
SetStdHandle	Установка некоторого дескриптора как дескриптора стандартного ввода, стандартного вывода или устройства ошибки. Может использоваться при перенаправлении ввода-вывода

Глава 6

Преобразование чисел

Внутри каждой большой задачи сидит маленькая, пытающаяся пробиться наружу.

Закон больших чисел Хоара (Прикладная Мерфология)

В предыдущей главе мы рассмотрели решение проблемы обмена данными с консолью. Данные, вводимые с консоли и выводимые на нее, кодируются операционной системой в соответствии с текущей таблицей кодировки. Отдельный предмет обсуждения при этом — процесс ввода-вывода числовой информации. В каждом языке программирования он реализован по-своему. Одна из целей, к которой стремятся разработчики компиляторов, — по возможности сделать этот процесс прозрачным для программиста. Язык ассемблера, в отличие от языков высокого уровня, не обладает средствами такого прозрачного ввода-вывода числовой информации. Но в этом и состоит его достоинство, так как при определенном опыте и квалификации программиста появляется хорошая возможность повышения эффективности конечного кода.

Для каждой конкретной задачи преобразование чисел между различными представлениями, допустимыми компьютером, может быть выполнено несколькими способами. Для эффективного решения задач обработки числовой информации программист должен знать эти способы и уметь выбрать наиболее подходящий из них для решения конкретной проблемы. Сразу следует отметить, что эта тема далеко не нова. Многие источники, посвященные ассемблеру, с той или иной степенью подробности рассматривают проблему преобразования чисел. И хотя здесь трудно придумать что-то новое, эта тема заслуживает того, чтобы ей было уделено место в книге, посвященной вопросам прикладного программирования на ассемблере. В данной главе сделана попытка систематизированно рассмотреть различные существующие подходы к решению проблемы ввода-вывода числовой информации и ее преобразованию во внутреннее представление в компьютере.

Начнем с того, что вспомним главу 2 «Программно-аппаратная архитектура МП IA32» учебника, где приведена классификация типов данных, допустимых процессорами Pentium III/4. Для нашего изложения важно то, что они делятся на две большие группы — данные целочисленного и вещественного типов. Причем целочисленные данные можно разделить на две подгруппы: двончные и двончно-де-

сятичные (BCD-числа). Исходя из этого, постараемся сформулировать направления преобразований числовой и символьной информации, востребованные на практике.

При обмене с консолью допустимы следующие преобразования:

- десятичные целые числа в символьном виде $\leftrightarrow$ внутреннее двоичное представление;
- шестнадцатеричные целые числа в символьном виде $\leftrightarrow$ внутреннее двоичное представление;
- двоичные целые числа в символьном виде $\leftrightarrow$ внутреннее двоичное представление;
- десятичная дробь в символьном виде $\leftrightarrow$ внутреннее представление в виде вещественного числа;
- дробное шестнадцатеричное число в символьном виде $\leftrightarrow$ внутреннее представление в виде вещественного числа;
- дробное двоичное число в символьном виде $\leftrightarrow$ внутреннее представление в виде вещественного числа.

Взаимное преобразование между внутренними представлениями:

- двоичное число $\leftrightarrow$ двоично-десятичное число;
- двоично-десятичное число $\leftrightarrow$ вещественное число;
- двоичное целое число $\leftrightarrow$ вещественное число.

Рассмотрим основные способы выполнения некоторых из этих видов преобразований. В своих рассуждениях будем предполагать, что числа положительные. Если вы внимательно изучили материал глав 6 и 8 учебника, а также главы 1 этой книги, посвященной программированию арифметических операций над двоичными и двоично-десятичными числами, то вы легко сможете дополнить приведенные ниже программы возможностью учета знака при выполнении соответствующих преобразований.

Ввод чисел с консоли

В этом разделе разберем способы преобразования десятичных целых и вещественных чисел, вводимых с консоли в символьном виде, в соответствующее им внутреннее двоичное представление. Начнем с целых десятичных чисел. Заметим, что этот вид преобразования является наиболее востребованным на практике. Далее будет обсуждена проблема преобразования вещественных чисел.

Выбор способа перевода десятичных целых чисел из символьного во внутреннее двоичное представление и обратно определяется диапазоном возможных исходных значений. Разберем два способа.

Ввод целых десятичных чисел из диапазона 0–99

Для значений из диапазона 0–99 взаимное преобразование между символьной десятичной и двоичной формами может производиться командами умножения и деления двоично-десятичных (BCD-чисел) — AAM и AAD.

```

+-----+
| Программа: prg06_01.asm. Ввод с консоли двоичного числа из диапазона 0-99 |
| в десятичном символьном представлении. |
+-----+
| Вход: число в десятичной системе счисления, вводимое в символьном виде |
| с клавиатуры. |
+-----+
| Выход: AL - двоичное число. |
+-----+
buf_0ah      struc
len_buf      db      3          : длина buf_0ah
len_in       db      0          : действительная длина введенного слова
                                : (без учета 0dh)
buf_in       db      3 dup (20h) : буфер для ввода (с учетом 0dh)
ends
.data
buf          buf_0ah <>
adr_buf      dd      buf
.code
;...
;----- вводим 2 символа с клавиатуры.
; контроль на допустимые значения не делаем
lds         dx, adr_buf
mov         ah, 0ah
int         21h
xor         ax, ax
cmp         buf.len_in, 2      : сколько чисел введено реально?
jne         ml
mov         ah, buf.buf_in
;----- преобразование в упакованное десятичное представление
ml:         mov         al, buf.buf_in+1
and         ax, 0f0fh
aad                     : в al двоичный эквивалент исходного
                        : двузначного десятичного значения
;...

```

Ввод целых десятичных чисел из диапазона 0–4 294 967 295

Если исходное значение выходит за диапазон 0–99, то здесь следует иметь в виду возможность возникновения ситуации, при которой значение вводимого десятичного числа превышает диапазон, допустимый форматами типов целочисленных данных, поддерживаемых, в частности, арифметическими командами процессора. Для Pentium III/4 это 8, 16 и 32 бита. Диапазоны значений для этих форматов (числа без знака):

- для операнда размером 8 битов — 0–255;
- для операнда размером 16 битов — 0–65 535;
- для операнда размером 32 бита — 0–4 294 967 295.

Как видите, максимальное число не такое уж и большое. Поэтому мы столько внимания уделили работе с числами большой размерности в главе 1, посвященной программированию арифметических операций. В ней данные большой размерности просто описывались в сегменте данных без какого-либо намека на возможность их ввода с консоли или отображения на ней. В этом разделе мы постараемся ликвидировать упущение, что несомненно поднимет привлекательность для читателя того и другого материала. Но вначале мы рассмотрим способы преобразования

значений, которые укладываются в указанные выше диапазоны. Для этого можно предложить два способа преобразования в символьном представлении десятичных чисел, вводимых с консоли: посредством десятичного полинома и с использованием возможностей сопроцессора по обработке данных.

В основе способа с десятичным полиномом лежит возможность представления десятичного числа суммой произведений на степени числа 10 составляющих его десятичных цифр, которые соответствуют позициям этих цифр в исходном числе:

$$A_{10} = a_{n-1} \times 10^{n-1} + a_{n-2} \times 10^{n-2} + \dots + a_1 \times 10^1 + a_0 \times 10^0.$$

Вычисление данного полинома лучше производить по схеме Горнера:

$$A_{10} = (\dots((0 + a_{n-1}) \times 10 + a_{n-2}) \times 10 + \dots + a_1) \times 10 + a_0.$$

Например, число 3405 по этим формулам может быть представлено так:

$$3405 = 3 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 5 \times 10^0 = (((0 + 3) \times 10 + 4) \times 10 + 0) \times 10 + 5.$$

Ниже приведена программа преобразования целого десятичного числа в символьном виде из диапазона 0–4 294 967 295 в эквивалентное двоичное представление. Для ввода числа с клавиатуры используем функцию 3fh MS DOS. Она удобна тем, что возвращает количество действительно введенных символов в регистре AL.

```

+-----+
:| Программа: prg06_02.asm. Преобразование целого десятичного числа |
:| в символьном виде из диапазона 0-4294967295 в двоичное представление. |
+-----+
:| Вход: ввод с клавиатуры числа в десятичной системе счисления |
:| (не более 10 цифр). |
+-----+
:| Выход: EAX - двоичное число - результат преобразования. |
+-----+
.data
string      db      12 dup (0)      ; максимальное число состоит из 10 цифр
                                         ; (12 - с учетом 0d0ah)
len_string = $ - string
ten         dd      10
string_e    db      0dh, 0ah, "Результат вышел за границы операнда !!!$"
adr_string_e dd      string_e
.code
:-----
: вводим с клавиатуры.
: в AL - количество действительно введенных десятичных цифр
mov     bx, 0          ; стандартный дескриптор - клавиатура
mov     cx, len_string
lea     dx, string     ; формируем указатель на строку string
mov     ah, 3fh        ; номер функции DDS
int     21h
jc      exit           ; переход в случае ошибки
:----- преобразование
mov     ecx, eax
sub     ecx, 2          ; корректируем счетчик цикла (чтобы
                        ; не учитывать 0d0ah, вводимые 3fh)
jecxz   exit           ; число не было введено
dec     ecx            ; не умножать на 10 последнюю цифру числа
jecxz   m2             ; введено однозначное число
lea     si, string     ; формируем указатель на строку string
xor     eax, eax       ; eax:=0
ml:     xor     edx, edx
        mov     dl, [si]
        and     dl, 0fh
        add     eax, edx

```

```

mul     ten
jc      exit_e      : результат вышел за границы операнда
inc     si
loop    m1
m2:     mov     di, [si]
and     di, 0fh      : преобразуем ASCII->BCD
add     eax, edx      : результат преобразования в регистре EAX
jnc     exit         : результат вышел за границы операнда
:----- выводим строку string_e на экран
exit_e:      ....

```

При необходимости вы можете изменить программу так, чтобы в ней использовались регистры меньшей разрядности.

Ввод целых десятичных чисел из диапазона 0–999 999 999 999 999 999

Второй способ преобразования десятичных чисел, хотя и выглядит несколько экзотически, вполне работоспособен. Данный подход предполагает использование особенностей некоторых команд сопроцессора. В материале главы 17 «Архитектура и программирование сопроцессора» учебника мы перечисляли форматы данных, которые поддерживает сопроцессор. Перечислим их еще раз:

- двоичные целые числа в трех форматах — 16, 32 и 64 бита;
- упакованные целые десятичные (BCD) числа — максимальная длина — 18 упакованных десятичных цифр (9 байтов);
- вещественные числа в трех форматах — коротком (32 бита), длинном (64 бита), расширенном (80 битов).

Для нас интерес представляют форматы целых двоичных и упакованных десятичных (BCD) чисел, а также команды обмена этими значениями с вершиной стека сопроцессора. Процесс преобразования десятичного целого числа, вводимого с клавиатуры, показан в программе ниже. Необходимо отметить, что этот способ преобразования позволяет расширить диапазон значений до 0–999 999 999 999 999 999.

```

+-----+
:| Программа: prg06_03.asm. Ввод целых десятичных чисел из диапазона |
:| 0-999 999 999 999 999 999 и преобразования их в двоичное представление. |
+-----+
:| Вход: ввод с клавиатуры числа в десятичной системе счисления |
:| в диапазоне значений 0-999 999 999 999 999 999. |
+-----+
:| Выход: string_bin - двоичное число - результат преобразования. |
+-----+
.data
db 0 : барьер, если введенное количество
: цифр нечетно
string db 20 dup (0) : максимальное исходное число состоит
: из 18 цифр (20 - с учетом 0d0ah)
len_string = $ - string
adr_string dd string
string_pack dt 0 : сюда упаковывается исходное значение
len_string_pack = $ - string_pack
adr_string_pack dd string_pack
:----- результат - двоичное значение различной разрядности
string_bin_byte label byte

```

```

string_bin_word label word
string_bin_dword label dword
string_bin dq 0 ; поле для полного результата –
; эквивалентного двоичного представления

;....

code
;....
;----- вводим с экрана
mov bx, 0 ; стандартный дескриптор – клавиатура
mov cx, len_string
lea dx, string ; формируем указатель на строку string
mov ah, 3fh ; номер функции DOS
int 21h
jc exit ; переход в случае ошибки
;----- преобразуем строку с десятичными числами в ее двоичный эквивалент
; в AL – количество действительно введенных десятичных цифр.
;----- сначала переведем в упакованное представление
mov cx, ax
sub cx, 2 ; корректируем счетчик цикла (чтобы
; не учитывать 0d0ah, вводимые 3fh)
jecxz exit ; число не было введено
lds si, adr_string
add si, cx
dec si ; указатель на последнюю введенную
; десятичную цифру
ml: les di, adr_string_pack
std ; флаг df=1 – работаем со строкой string,
; начиная с ее конца

xor ax, ax
lodsb
and al, 0fh
shl ax, 8
lodsb
shl al, 4
add al, ah ; в AL две очередные упакованные цифры
cld ; флаг df=1 – работаем со строкой string_pack
; с ее начала

stosb
sub cx, 2
cmp cx, 0
jg ml
;----- теперь преобразуем в эквивалентное двоичное представление
finit ; инициализируем сопроцессор
fbld string_pack ; помещаем в стек сопроцессора
fistp string_bin ; и извлекаем как целое
;....

```

Приведенная программа преобразует любое значение из диапазона $0-10^{18}-1$. Интересно отметить количественное значение максимальной двоичной величины, соответствующее верхней границе диапазона, — это `+0de0b6b3a763ffff16`. Оно пригодится нам при рассмотрении обратного преобразования для вывода на консоль — из двоичного в десятичное представление. Извлечь значение нужной разрядности можно, если ввести директивой `label` соответствующие идентификаторы в исходный текст программы (что и сделано в нашем сегменте кода):

```

string_bin_byte label byte
string_bin_word label word
string_bin_dword label dword
string_bin dq 0 ; поле для результата –
; эквивалентного двоичного представления

;....

```

Ввод целых десятичных чисел из диапазона 0—∞

Для преобразования десятичного числа произвольной разрядности из символьного представления в двоичное потрудиться придется несколько больше. Основа для этой работы была заложена в материале, посвященном арифметическим операциям для чисел произвольной разрядности. Поэтому наши действия при разработке программы преобразования напомнят игру с конструктором, когда из готовых компонентов будет создаваться новый продукт.

Исходными компонентами программы преобразования десятичного числа произвольной разрядности из символьного представления в двоичное будут являться макрокоманда умножения N -байтового числа на число размером M байтов и программа сложения чисел размером N байтов без учета знака. Алгоритм вычисления двоичного эквивалента будет таким же, как рассмотренный выше, — вычисление полинома по схеме Горнера. Ниже приведен вариант реализующей его программы. Расположение байтов результата — по схеме, естественной для процессоров Intel, то есть младший байт находится по младшему адресу.

```

+-----+
: | Программа: prg06_04.asm. Ввод целых десятичных чисел из диапазона 0..∞. |
+-----+
: | Ввод: ввод с клавиатуры числа в десятичной системе счисления |
: | длиной до 20 цифр. |
+-----+
: | Выход: string_bin - двоичное число - результат преобразования. |
+-----+
:
: ;----- см. описание макрокоманд add_unsign_N_1
: ; и mul_unsign_NM в главе 1
:
add_unsign_N_1 macro summand_1, summand_2, N
:
endm
mul_unsign_NM macro u, i, v, j, w
:
endm
.data
string db 22 dup (0) ; максимальное число состоит из 20 цифр
: ; (22 — с учетом 0d0ah)
:
len_string = $ - string
ten dd 10
string_bin db 10 dup (0) ; максимальная длина двоичного числа —
: ; 10 байтов
:
len_string_bin = $ - string_bin
carry db 0 ; перенос сложения последних байтов
adr_string_bin dd string_bin
string_bin_w db len_string_bin + 1 dup (0) ; результат умножения для
: ; mul_unsign_NM = len_string_bin+1 байтов
:
len_string_bin_w = $ - string_bin_w
adr_string_bin_w dd string_bin_w
k db 0 ; перенос  $0 \leq k < 255$ 
b dw 100h ; размер машинного слова
.code
:
: ;----- вводим десятичное число с клавиатуры
mov bx, 0 ; стандартный дескриптор — клавиатура
mov cx, len_string
lea dx, string ; формируем указатель на строку string
mov ah, 3fh ; номер функции DOS
int 21h

```



```

        jc      exit          ; переход в случае ошибки
;----- преобразуем строку с десятичными числами в ее двоичный эквивалент,
; в AL — количество действительно введенных десятичных цифр
        mov     ecx, eax
        sub     ecx, 2        ; корректируем счетчик цикла (чтобы
                               ; не учитывать 0d0ah, вводимые 3fh)

        jcxz    $ + 4         ; число не было введено
        jmp     $ + 5
        jmp     exit
cont_1:  dec     ecx          ; не умножать на 10 последнюю цифру числа
        jcxz    $ + 4         ; однозначное число
        jmp     $ + 5
        jmp     m2
        lea     si, string    ; формируем указатель на строку string
        xor     eax, eax      ; eax:=0
m1:     xor     edx, edx
        mov     dl, [si]
        and     dl, 0fh       ; преобразуем ASCII->BCD
        add     unsign_N1_string_bin, dl, len_string_bin
;----- умножаем на 10
        mul     unsign_NM_string_bin, len_string_bin, ten, 1, string_bin_w
;----- копируем string_bin_w в string_bin
        cld
        push    si
        push    cx
        lds     si, adr_string_bin_w
        les     di, adr_string_bin
        mov     cx, len_string_bin_w
        rep     movsb
        pop     cx
        pop     si
        inc     si
        dec     cx
        jcxz    m2
        jmp     m1
m2:     mov     dl, [si]
        and     dl, 0fh
        add     unsign_N1_string_bin, dl, len_string_bin
;----- результат преобразования — в строке string_bin
        ....

```

Одно из направлений совершенствования этой программы — динамическое выделение памяти для всех чисел с неизвестной длиной. Необходимо заметить, что способ преобразования длинных чисел универсален — его можно использовать и для значений, которые укладываются в представимые в процессоре диапазоны типов данных.

Ввод вещественных чисел

Теперь у нас все готово для того, чтобы выполнить ввод с клавиатуры символьного представления вещественного числа и преобразование его в соответствующий двоичный эквивалент. В главе 17 «Архитектура и программирование сопроцессора» учебника мы обсуждали понятие вещественного числа. Отмечалось, что вещественное число имеет две формы записи — с плавающей точкой (34.89) и научную (3.45e-3 = 3.45×10^{-3}). Для преобразования вещественного числа из символьного представления в эквивалентное двоичное можно предложить несколько способов. Самый простой — использовать возможность загрузки в сопроцессор упакованного BCD-числа. В этом случае алгоритм преобразования состоит в следующем. Символьная строка с вещественным числом вводится в память, где она преобразуется

в упакованное BCD-число. При вводе указанной символьной строки запоминается положение плавающей точки. Полученное упакованное BCD-число загружается в сопроцессор, после чего оно делится на 10 в степени, соответствующей положению плавающей точки в исходном числе. Для малых чисел (в диапазоне до $|10^{18}-1|$) этот способ вполне хорош. Его можно расширить, если вводить число в научном формате, при этом процесс перевода мантииссы аналогичен рассмотренному выше, но при подготовке к делению на степень 10 необходимо учесть значение степени, указанное после символа «Е». Но все равно, несмотря на расширения диапазона, разрядность мантииссы ограничена 18 цифрами. Устранить этот недостаток можно, используя операции с числами произвольной разрядности. Этот способ интересен своей универсальностью, поэтому уделим ему основное внимание.

Итак, напишем программу ввода вещественного числа с клавиатуры в одном из двух возможных форматов — простом формате с плавающей точкой. Доработать программу для использования научного формата для вас не составит труда.

В качестве знаков, разделяющих мантииссу на целую и дробную части, можно использовать как запятую, так и точку. Суть алгоритма преобразования состоит в следующем. Производится ввод с клавиатуры символов вещественного числа. После ввода анализируются символы буфера, куда были помещены символы введенного числа, на предмет выяснения положения плавающей точки. Обнаруженная позиция запоминается. Относительно нее введенные символы делятся на символы цифр целой и дробной частей. Привлекая алгоритм преобразования символьного представления десятичного числа в двоичный эквивалент, преобразуем целую часть вещественного числа. Дробная часть вещественного числа также переводится в двоичный эквивалент. Это преобразование выполняется с использованием сопроцессора по формуле:

$$(((... (u_{-m}/b + u_{-m-1})/b + ... + u_{-2})/b + u_{-1})/b,$$

где u_n — символы десятичных цифр дробной части вещественного числа $u_{-m} u_{-m-1} ... u_{-2} u_{-1}$, $b = 10$. После того как данное выражение вычислено (его результат находится в вершине стека сопроцессора), производится сложение результата с преобразованной целой частью вещественного числа. Все, теперь в вершине стека сопроцессора находится вещественное число — эквивалент своего исходного символьного представления. Текст программы преобразования вещественного числа из символьного представления достаточно велик и по этой причине приведен отдельно, среди материалов, прилагаемых к книге (prg06_05.asm). Заметьте, что с целью экономии места никаких проверок на правильность формата вводимого вещественного числа в программе не делается.

Последнее замечание — об ограничениях на размерность исходного числа. Здесь следует различать размерности целой и дробной частей. Что касается дробной части, то тут вообще ограничений нет, за исключением тех, которые накладывает сам сопроцессор на вводимые в его регистры значения. Для целой части узкое место — максимальная размерность операнда в команде целочисленного сложения FIADD, которая составляет 32 бита.

Ввод шестнадцатеричных и двоичных чисел мы обсуждать не будем, так как общие принципы их ввода аналогичны рассмотренным выше для десятичных чисел. Потребность во вводе с клавиатуры шестнадцатеричных и двоичных чисел возникает значительно реже, чем десятичных.

Вывод чисел на консоль

В этом разделе мы рассмотрим алгоритмы обратного преобразования чисел — из внутреннего двоичного представления в число в символьном виде, формат записи которого соответствует правилам требуемой системы счисления. Необходимо предупредить читателя, что рассмотрение обратного преобразования не будет симметричным прямому преобразованию. И в подтверждение этому начнем обсуждение проблемы вывода чисел на консоль с алгоритма преобразования шестнадцатеричных чисел в символьное представление.

Вывод шестнадцатеричных чисел

Умение работать с шестнадцатеричными числами — необходимое условие успешного программирования на низком уровне. Шестнадцатеричные числа, по сравнению с двоичными, являются более естественными для анализа внутреннего представления информации в компьютере. Вспомним, что каждый байт — это совокупность двух тетрад, а диапазон значений, представимых одной тетрадой, как раз совпадает с диапазоном значений, которые может принимать однозначное шестнадцатеричное число. Поэтому сам процесс преобразования шестнадцатеричных чисел в символьное представление особого труда не представляет. Например, алгоритм вывода на консоль содержимого одного байта состоит в выделении некоторым способом значений его младшей и старшей тетрад и дальнейшее их преобразование к символьному виду. Если нужно вывести на консоль символьное представление более чем одного байта, то процесс выделения тетрад и их преобразования выполняется последовательно необходимое количество раз.

В качестве полезной иллюстрации алгоритма преобразования шестнадцатеричной информации в символьное представление рассмотрим макрокоманду **SHOW**, которая переводит содержимое одного из четырех регистров — **AL**, **AH**, **AX**, **EAX** в символьное шестнадцатеричное представление. Этот макрос является универсальным средством, которое позволяет осуществить «подглядывание» за содержимым регистра или области памяти динамически, во время выполнения программы. С его помощью можно отобразить содержимое любого из доступных регистров или области памяти длиной до 32 битов. Для этого достаточно лишь переслать содержимое нужного объекта (регистра или ячейки памяти) с учетом его размера в один из регистров — **AL**, **AH**, **AX**, **EAX**. Имя одного из этих регистров указывается затем в качестве фактического аргумента макрокоманды **SHOW**. Второй аргумент этого макроопределения — позиция на экране. Задавая определенные значения, мы можем судить о том, какая именно макрокоманда **SHOW** сработала. Еще одна немаловажная особенность данного макроса состоит в его возможности работать как в реальном, так и защищенном режимах, с учетом допустимой адресации. Проверить работу данного макроопределения вы можете с помощью следующей программы.

```

:-----+
: | Программа: prg06_06.asm. Проверка работоспособности макрокоманды SHOW. |
:-----+
include      show.inc
.data
pole        dd      3cdf436fh

```

```
.code
;...
mov     ax, 1f0fh
show    al, 0
show    ah, 160
show    ax, 320
mov     eax, pole
show    eax, 480
;...
```

Ниже приведены фрагменты текста макрокоманды **SHOW**. Полный текст этой макрокоманды имеется среди материалов, прилагаемых к книге.

```
+-----+
+| Макрокоманда: show. Отображение содержимого регистров AL, AH, AX, EAX. |
+-----+
+| Вход: arg_n – имя одного из регистров AL, AH, AX, EAX. |
+|      n_poz – номер позиции на экране, по умолчанию – 1000. |
+-----+
Show      macro    arg_n, n_poz:=<1000>
            local  main_part, disp, pause, template, VideoBuffer
            local  p_mode, m1, m2
            ;----- переход на начало блока команд во избежание выполнения данных
            jmp     main_part
            ;----- некоторые константы и переменные
            ;...
            ;----- начало блока команд
            ;      сохранение в стеке используемых регистров:
            ;      EAX, EBX, ECX, EDX, EDI, DS, ES
main_part: ;...
            push    cs
            pop      ds

            lea      bx, cs:template ; в bx – адрес таблицы-шаблона (для xlat)
            xor      cx, cx           ; очистка cx
            ;----- начало блока определения того,
            ;      какой регистр был передан макросу
IFIDNI     <al>, <&arg_n>              ; если аргумент=al или AL.
            ?reg8bit = TRUE           ; установка флага 8-битового регистра
            mov      ah, al
ENDIF
IFIDNI     <ah>, <&arg_n>              ; если аргумент=ah или AH.
            ?reg8bit = TRUE           ; установка флага 8-битового регистра
ENDIF
            ;----- передан не al или AL
IFIDNI     <ah>, <&arg_n>              ; если аргумент=ah или AH.
            ?reg8bit = TRUE           ; установка флага 8-битового регистра
ENDIF
            ;----- передан не AH или ah
IFIDNI     <ax>, <&arg_n>              ; если аргумент равен ax или AX.
            ?reg16bit = TRUE          ; установка флага 16-битового регистра
ENDIF
            ;----- передан не ah, AH, ax или AX
            ;...
            ;----- обработка содержимого регистров AL, AH, AX, EAX
IF          (?reg8bit)                ; если передан al или ah
            push     eax
            and      ah, 0f0h          ; обращение к старшей четверке битов ah
            shr      ax, 12            ; сдвиг битов в начало (16 - 4 = 12)
            xlat     ; трансляция по таблице-шаблону
            ;----- помещение символа из al в edi
            movdi    ax
            shl      di, 8
            inc      cx
            pop      eax
            and      ax, 0f00h         ; обращение к младшей тетраде ah
            shr      ax, 8             ; сдвиг битов в начало (16 - (4 + 4) = 8)
            xlat     ; трансляция по таблице-шаблону
```

```

        or     di, ax           ; помещение очередного символа в DI
        shl   edi, 16
        inc   cx

ENDIF
IF      (?reg16bit)           ; если передан ax или ax
:----- начало обработки значения регистра AX
        push  eax
:----- обращение к старшей четверке битов ax
        and   ax, 0f00h
        shr   ax, 12          ; сдвиг битов в начало (16 - 4 = 12)
        xlat                     ; трансляция по таблице-шаблону
:----- помещение символа из al в старшую тетраду
:
        starшей половины EDI
        mov   di, ax
        shl   edi, 8
        inc   cx
        pop   eax
        push  eax
:----- обращение ко второй четверке битов ax
        and   ax, 0f00h
        shr   ax, 8           ; сдвиг битов в начало (16 - (4 + 4) = 8)
        xlat                     ; трансляция по таблице-шаблону
:----- помещение очередного символа в младшую тетраду
:
        starшей половины EDI
        or    di, ax
        shl   edi, 8
        inc   cx
        pop   eax
        push  eax
:----- обращение к старшей четверке битов в AL
        and   ax, 0f0h
        shr   ax, 4           ; сдвиг битов в начало
                                ; (16 - (4 + 4 + 4) = 4)
        xlat                     ; трансляция по таблице-шаблону
        or    di, ax           ; помещение очередного символа в EDI
        shl   edi, 8
        inc   cx
        pop   eax
:----- обращение к младшей четверке битов AX
        and   ax, 0fh
        xlat                     ; трансляция по таблице-шаблону
        or    di, ax           ; помещение очередного символа в EDI
        inc   cx

ENDIF
IF      (?reg32bit)           ; если передан EAX или EAX
:....                          ; аналогично AX

ENDIF

:----- вывод на экран результата с учетом режима работы процессора
:
: результат — в паре EDI:EBX, количество цифр — в CX
:....
:----- восстановление регистров
:....

ENDM

```

Вывод целых десятичных чисел из диапазона 0–99

Выше упоминалось, что для значений из диапазона 0–99 взаимное преобразование между символьной десятичной и двоичной формами может производиться командами умножения и деления двоично-десятичных (BCD-чисел) — AAM и AAD.

```

+-----+
: | Программа: prg06_07.asm. Ввод с консоли десятичного числа из диапазона
: | 0-99 и обратный его вывод на консоль.
+-----+

```

```

buf_0ah      struc
len_buf      db      3          ; длина buf_0ah
len_in       db      0          ; действительная длина введенного слова
                                   ; (без учета 0dh)
buf_in       db      3 dup (20h) ; буфер для ввода (с учетом 0dh)
ends
.data
buf          buf_0ah <>
adr_buf      dd      buf
.code
;...
;----- вводим 2 символа с клавиатуры.
; контроль на допустимые значения не делаем
lds         dx, adr_buf
mov         ah, 0ah
int         21h
xor         ah, ah
cmp         buf.len_in, 2      ; сколько чисел введено реально?
jne         ml
mov         ah, buf.buf_in
;----- преобразование в неупакованное десятичное представление
ml:         mov     al, buf.buf_in+1
            and     ax, 0f0fh
            aad                     ; в AL двоичный эквивалент исходного
                                   ; двузначного десятичного значения
;----- вывод на консоль содержимого AL
aam
mov         dx, ax
or          dx, 03030h
rol         dx, 8              ; выводим старшую цифру
mov         ah, 2
int         21h
rol         dx, 8              ; выводим младшую цифру
int         21h
;...

```

Для преобразования с целью последующего вывода на консоль больших двоичных значений можно использовать два способа: путем деления по модулю 10 (диапазон значений не ограничен) и с помощью сопроцессора ($0..10^{18} - 1$).

Вывод целых десятичных чисел из диапазона $0-\infty$

В основе алгоритма вывода двоичных значений из диапазона $0-\infty$ лежит положение о том, что цифры $(...U_2U_1U_0)$ десятичного представления, начиная с младшей, получаются последовательным делением исходного двоичного значения u на 10: $U_0 = u \bmod 10$; $U_1 = \lfloor u/10 \rfloor \bmod 10$; $U_2 = \lfloor \lfloor u/10 \rfloor / 10 \rfloor \bmod 10$ и т. д., до тех пор пока после очередного деления делимое не окажется равным нулю: $\lfloor \dots \lfloor u/10 \rfloor / 10 \rfloor \dots \rfloor = 0$. Здесь символы $\lfloor \dots \rfloor$ обозначают целую часть частного, округленного в меньшую сторону.

Почему в отличие от алгоритмов ввода с консоли для обратного преобразования нет такого разнообразия способов? Это объясняется особенностью команды DIV процессора, которая используется в описанном выше алгоритме для получения частного и остатка. Ее требование к соотношению значений делимого и делителя — размер частного должен быть в два раза меньше делимого. В противном случае возникает исключение #DE (ошибка деления), и программа аварийно завершается.

Исходя из этих условий, нам ничего не остается, кроме как воспользоваться программой беззнакового деления значения размером N байтов на значение размером 1 байт. Она была рассмотрена в главе 1, посвященной целочисленным арифметическим операциям. Для удобства эту программу мы оформим в виде макрокоманды.

```

+-----+
: | Программа: prg06_08.asm. Вывод целых десятичных чисел из диапазона 0..∞. |
+-----+
: | Вход: bin_dd - многобайтовое двоичное число |
: | для преобразования в области памяти. |
+-----+
: | Выход: вывод десятичного числа из диапазона 0..∞ на экран. |
+-----+

:----- div_unsign_N_1_I - макрокоманда деления N-разрядного
: беззнакового целого на одnorазрядное число размером 1 байт
: (порядок следования байтов - младший байт по младшему адресу
: (Intel)). См. главу 1 и материалы, прилагаемые к книге
div_unsign_N_1_I macro u, N, v, w, r
:....
endm
.data
string db 10 dup (0) ; пусть максимальное десятичное число
; состоит из 10 цифр
len_string = $ - string
adr_string dd string
bin_dd label byte
dd 0ffffffffh
len_bin_dd = $ - bin_dd
ten db 10
remainder dw 0
.code
:....
:----- значение для преобразования должно быть в памяти
les di, adr_string ; строка с десятичными символами
cld ; обработка в прямом направлении
continue:
div_unsign_N_1_I bin_dd, len_bin_dd, ten, bin_dd, remainder
mov ax, remainder
or ax, 30h ; преобразуем в символьное представление
stosb ; сохраняем в string
; очередную десятичную цифру
; подсчитываем количество цифр
inc cx
cmp bin_dd, 0
jne continue
:----- вывод на консоль с конца строки
mov ah, 2
std
mov si, di
dec si
m1: lodsb
mov dl, al
int 21h
loop m1
:....

```

В данной программе преобразованию подвергается значение в памяти. Причем мы в качестве исходного двоичного значения задали максимально возможное беззнаковое число размером в двойное слово. Результат преобразования — 4 294 967 295, что полностью согласуется с ожидаемым десятичным значением. Но задавать исходные значения в памяти не всегда удобно, хотелось бы, чтобы можно

было подвергать преобразованию значения прямо из регистров процессора. Для такого типа преобразований (значений в регистрах процессора) лучше подойдет способ с использованием сопроцессора. Рассмотрим его.

Вывод целых десятичных чисел из диапазона 0–999 999 999 999 999 999

Этот способ вывода основан на возможности сопроцессора работать с упакованными десятичными числами. Выше мы уже рассматривали преобразование десятичных чисел в двоичное представление на основе этой возможности. Система команд сопроцессора содержит команду **FBSTP**, которая сохраняет десятичное число из вершины стека сопроцессора в области памяти с одновременным преобразованием этого числа в формат десятичного числа. Область памяти, в которую происходит сохранение, должна быть описана директивой **DT**. Важно отметить, что команда **FILD**, с помощью которой вы будете помещать целое число в сопроцессор для дальнейшего преобразования, трактует целые числа как числа со знаком. Поэтому попытка задать целое число в виде 0ffffh (с единичным старшим разрядом операнда) приведет к тому, что в стек сопроцессора будет помещено значение -1 со всеми вытекающими отсюда последствиями для результата преобразования.

```

+-----+
:| Программа: prg06_09.asm. Вывод целого десятичного числа |
:| из диапазона 0-999 999 999 999 999 999 на экран. |
+-----+
:| Вход: выводимое значение – в поле string_bin_dword. |
+-----+
:| Выход: вывод десятичного числа на экран. |
+-----+
.data
:----- поле string_bin_dword содержит выводимое значение – с помощью
: идентификаторов, вводимых директивой label, это значение может
: трактоваться как значение различной разрядности
string_bin_byt label byte
string_bin_word label word
string_bin_dword label dword
string_bin_qword dq 0de0b6b3a763ffffh ; зададим максимально возможное для
: сопроцессора двоичное целое со знаком
: в string_pack исходное значение из string_bin_dword в упакованном десятичном формате
string_pack dt 0
len_string_pack = $ - string_pack
adr_string_pack dd string_pack
string db 20 dup (0) ; максимальный результат состоит
: из 18 десятичных цифр
len_string = $ - string
adr_string dd string
:....
.code
:....
:----- преобразуем bin->dec
finit
fild string_bin_qword ; заносим в сопроцессор
: двоичное целое десятичное
fbstp string_pack ; извлекаем упакованное десятичное
:----- распакуем в цикле
lds si, adr_string_pack
add si, len_string_pack - 2 ; на конец string_pack
: (18 упакованных десятичных цифр)
les di, adr_string

```



```

cysl:      mov     cx, 9           ; 9 пар упакованных десятичных цифр
xor        ax, ax
std        ; string_pack обрабатываем с конца
lods      ; в al очередные 2 упакованные
          ; десятичные цифры
:----- распаковываем — ah = младшая, al = старшая
shl        ax, 4
rol        al, 4
or         ax, 3030h             ; преобразуем в символьное представление
xchg       ah, al               ; ah = младшая, al = старшая
cld        ; в string записываем с начала
stosw
loop       cysl
:----- выводим на консоль
mov        bx, 1                 ; стандартный дескриптор — экран
mov        cx, len_string
lds        dx, adr_string        ; формируем указатель на строку string
mov        ah, 40h               ; номер функции DOS
int        21h
jc         exit                  ; переход в случае ошибки
:....

```

Вывод вещественных чисел

Последний алгоритм, который мы рассмотрим в этом разделе, — преобразование вещественного значения в вид, пригодный для его отображения на экране консоли. Ниже приведены только те фрагменты программы, которые относятся непосредственно к преобразованию. Полный текст программы находится среди материалов, прилагаемых к книге.

```

:-----+-----+
: | Программа: prg06_10.asm. Вывод вещественного числа (32 бита). |
:-----+-----+
: | Вход: выводимое значение - в поле float32. |
:-----+-----+
: | Выход: вывод вещественного числа короткого формата на экран. |
:-----+-----+
: | Обязательно наличие в программе процедур: |
: | read_cursor_position, set_cursor_position. |
:-----+-----+
.data
dec_bin_mant32 dt 0 ; мантисса в двоично-десятичном
                  ; представлении
dec_bin_har32 dt 0 ; характеристика в двоично-десятичном
                  ; представлении
cwr dw 0 ; переменная для сохранения состояния
                  ; регистра слова состояния
ten dw 10 ; константа
float32 dd 1.2345678e12 ; значение вещественного числа размером в 32 бита
                  ; для вывода
mant32 dd 0 ; мантисса в двоичном представлении
har32 dd 0 ; характеристика — вещественный формат
                  ; в двоичном представлении
int_har32 dd 0 ; характеристика — целое
                  ; в двоичном представлении
number db 0
char db 0
cursor_column db 0
cursor_line db 0
number_of_digits db 9
flag db 0
.code
:----- процедура сдвига курсора на одну позицию вправо

```

```

next_cursor_column proc
:....
next_cursor_column endp
:----- процедура позиционирования курсора
set_cursor_position proc
:....
set_cursor_position endp
:----- процедура определения текущей позиции курсора
read_cursor_position proc
:....
read_cursor_position endp
:----- процедура вывода символа с учетом цвета
print_char proc
:....
print_char endp
:----- выделение мантиссы из короткого формата (32 бита)
: и ее преобразование в двоично-десятичный формат
: (для положительной характеристики (результат в st(0)))
positive_har proc
fimul ten
sub int_har32. 6
lab_p_h: fidiv ten
cmp int_har32. 0
dec int_har32
jg lab_p_h
ret
positive_har endp
:----- выделение мантиссы из короткого формата (32 бита)
: и ее преобразование в двоично-десятичный формат
: (для отрицательной характеристики результат в st(0))
negative_har proc
fidiv ten
sub int_har32. 7
lab_n_h: fimul ten
cmp int_har32. 0
inc int_har32
jl lab_n_h
ret
negative_har endp
:----- вывод вещественного числа (32 бита) в десятичном виде
fprint32 proc
pusha
:----- установка размера мантиссы в 24 бита
finit
fstcw cwr
and cwr. 1111000011111111b
or cwr. 1111000011111111b
fldcw cwr
fld float32 ; загрузка 32-битного числа в стек
fextract ; выделение мантиссы (st0)
; и характеристики (st1)
fstp mant32 ; запоминаем мантиссу
fist har32 ; запоминаем характеристику
:----- перевод двоичной характеристики в десятичную характеристику
fldlg2 ; загрузка десятичного логарифма 2
fimul har32 ; умножение двоичной характеристики на log10(2)
frndint ; округление
fistp int_har32 ; сохранение десятичной характеристики
fild int_har32
fbstp dec_bin_har32 ; сохранение двоично-десятичного
; значения характеристики
:----- выбор процедуры по выделению мантиссы
fld float32
cmp har32. 0

```

```

jge     case1
call    negative_har    ; преобразования мантиссы.
                        ; если характеристика отрицательная
jmp     end_case
case1:   call positive_har    ; преобразование мантиссы.
                        ; если характеристика положительная
end_case: fbstp    dec_bin_mant32    ; сохранение двоично-десятичного
                        ; представления мантиссы

:----- вывод на экран вещественного числа
lea     si, dec_bin_mant32
add     si, 9
mov     al, [si]
:----- вывод знака числа
test    al, al
jz      zero
mov     char, "-"
call    print_char
call    next_cursor_column
:----- данный фрагмент пропускает байты с нулевым содержанием
: до первого байта со значащей цифрой
zero:    dec     si
        dec     number_of_digits
        mov     al, [si]
        test    al, al
        jnz     first_zero    ; найден первый ненулевой байт
        jmp     zero          ; байт имеет нулевое значение —
                                ; продолжаем поиск
:----- просмотр тетрад первого найденного ненулевого байта
first_zero: and    al, 11110000b
        test    al, al
        je      second_digit   ; если старший полубайт байта равен нулю,
                                ; начинаем вывод со второго байта
        jmp     first_digit     ; если старший полубайт байта не равен нулю,
                                ; начинаем вывод первого байта
:----- начало цикла вывода мантиссы
print_digits: dec    si
        dec     number_of_digits ; индекс выводимого байта
        mov     al, [si]
:----- вывод первого полубайта, содержащего цифру
first_digit: and    al, 11110000b
        shr     al, 4
        add     al, 30h
        mov     char, al
        call    print_char
        call    next_cursor_column
:----- если выводимая цифра первая, то выводим после нее точку
        cmp     flag, 0
        jne     second_digit
        mov     char, "."
        call    print_char
        call    next_cursor_column
        inc     flag
:----- вывод второго полубайта, содержащего цифру
second_digit: mov     al, [si]
        and     al, 00001111b
        add     al, 30h
        mov     char, al
        call    print_char
        call    next_cursor_column
:----- если выводимая цифра первая, то выводим после нее точку
        cmp     flag, 0
        jne     nonfirst_digit
        mov     char, "."
        call    print_char

```

```

        call    next_cursor_column
        inc     flag
nonfirst_digit: cmp     number_of_digits, 0
               jne     print_digits
               mov     flag, 0
               ;----- вывод характеристики числа
               mov     char, "E"
               call    print_char
               call    next_cursor_column
               lea     si, dec_bin_har32
               ;----- вывод знака числа
               add     si, 9
               mov     al, [si]
               test    al, al
               je      print_har
               mov     char, "-"
               call    print_char
               call    next_cursor_column
               ;----- значения характеристики
print_har:    sub     si, 9
               mov     al, [si]
               ;----- вывод первой цифры характеристики
               and     al, 11110000b
               shr     al, 4
               add     al, 30h
               mov     char, al
               call    print_char
               call    next_cursor_column
               ;----- вывод второй цифры характеристики
               mov     al, [si]
               and     al, 00001111b
               add     al, 30h
               mov     char, al
               call    print_char
               call    next_cursor_column
               mov     flag, 0
               popa
               ret
fprint32     endp
main         proc
             ....
             call    fprint32
             ....

```

Глава 7

Работа с файлами в программах на ассемблере

Очевидно, целесообразно рассматривать прикладную систему как некоторую совокупность данных и операций, определенных на этих данных, а не просто как набор программ, взаимодействующих между собой посредством обмена данными. Программы — это всего лишь рабочие инструменты, в то время как данные — это своего рода исходное сырье, из которого вырабатывается конечный продукт.

Набор данных вне среды, приспособленной для его использования, не представляет никакой ценности, однако в рамках такой среды именно набор данных приобретает основное значение.

Динар Нурмухамедович Бибишев

Язык ассемблера не содержит средств для работы с файлами. Если такая необходимость возникает, то программа должна включать в себя фрагменты кода, в которых производится обращение к средствам операционной системы, осуществляющим взаимодействие с файловой системой. Это лишний раз подтверждает тот факт, что в области контактов с внешним миром программа на ассемблере оказывается привязанной как к конкретной аппаратной, так и к конкретной операционной платформам. В сегодняшней ситуации программисту все еще приходится сталкиваться с необходимостью программирования для MS DOS. Поэтому изучение средств для работы с файлами этой операционной платформы не потеряло своей актуальности и эти средства в плане совместимости поддерживаются различными реализациями Windows. В редакции MS DOS 7.0 введена поддержка длинных имен файлов, используемых системой файлового ввода-вывода Win32. Таким образом, можно выделить четыре аспекта работы с файлами из программ на ассемблере:

- работа с системой файлового ввода-вывода MS DOS, использующей короткие имена (по схеме «8.3»);
- работа с системой файлового ввода-вывода MS DOS, расширенной длинными именами (длиной до 255 символов);
- работа с системой файлового ввода-вывода Win32;
- использование файлов особого вида, поддерживаемых Win32, — проецированных на память.

Цель данной главы — предоставление читателю фрагментов кода, реализующих наиболее часто востребуемые на практике файловые операции для различных операционных платформ. Соответственно, нам не обойтись без определенной систематизации, но это будет сделано лишь для того, чтобы создать у читателя общее представление о поднятом вопросе. Если у вас впоследствии возникнет потребность в реализации файловых функций, чье полное практическое описание отсутствует в материале данной главы, то более подробные сведения о них можно будет найти в других справочных руководствах, например библиотеке MSDN. Далее, опираясь на общие принципы организации ввода-вывода, рассмотренные ниже, вы сможете без труда решить неожиданную проблему. Материалы всех разделов подобраны так, чтобы читатель мог познакомиться с общими принципами организации ввода-вывода в каждом случае, начиная от самых простых.

Работа с файлами в MS DOS (имена 8.3)

В основе файловой системы MS DOS лежит древовидная структура каталогов. Корень этой структуры представляет собой совокупность ограниченного числа дескрипторов, описывающих файлы и каталоги (подкаталоги) следующего уровня. Подкаталог представляет собой структуру особого типа, которая содержит дескрипторы файлов и подкаталогов очередного нижележащего уровня. В отличие от корневого каталога, количество дескрипторов в подкаталоге не ограничено и определяется только размером диска. Дескриптор представляет собой экземпляр структуры размером 32 байта. Поля этой структуры содержат различную информацию о файле: идентификатор файла и его характеристики — дату и время создания (модификации), номер начального кластера, длину файла и его атрибуты.

Использование файла в программе основывается на следующих операциях:

- создание нового файла;
- открытие существующего файла;
- запись/чтение в/из файл(а);
- закрытие файла.

Операционная система MS DOS поддерживает эти операции с помощью набора функций прерывания 21h. Кроме них, данное прерывание содержит функции для работы с каталогами:

- создать каталог;
- удалить каталог;
- сменить каталог.

Существует также ряд других функций для работы с файловой системой, в том числе для поиска файлов и получения информации о них.

Создание, открытие, закрытие и удаление файла

Прежде чем использовать файл в программе, его необходимо открыть с помощью функции 3dh прерывания 21h. Если файл не существует, то перед открытием его нужно создать. Оба эти действия выполняются одной из следующих функций: 3ch, 5bh, 5ah, 6ch.

Создание файла с усечением существующего до нулевой длины

Вход: AH = 3Ch; CX = атрибуты файла (значения битов: 0 = 1 — только чтение; 1 = 1 — скрытый файл; 2 = 1 — системный файл; 3 = 0 — игнорируется; 4 = 0 — зарезервирован (каталог), должен быть равен 0; 5 — бит архивации; 6 = 0 — резерв; 7 = 1 — общий файл в системе Novell NetWare; 8–15 = 0 — резерв); DS:DX — ASCIIZ-имя файла.

Выход: CF = 0 — AX = дескриптор файла; CF = 1 — AX = код ошибки (3 — нет такого пути; 4 — нет свободного дескриптора файла; 5 — отказ в доступе).

```

-----+-----
:| Программа: prg07_01.asm. Создание в текущем каталоге функцией 3ch |
:| файла my_file.txt. |
:|-----+-----
.data
handle      dw      0          ; дескриптор файла
filename     db      'my_file.txt', 0
point_fname  dd      filename
.code
;....
xor          cx, cx          ; атрибуты файла — обычный файл
lds          dx, point_fname ; формируем указатель на имя файла
mov          ah, 3ch         ; номер функции DOS
int          21h             ; создаем файл
jc           exit            ; переход в случае ошибки
;----- действия при успешном создании файла
mov          handle, ax      ; сохраним дескриптор файла
;....

```

Функция 3ch обладает особенностью, заключающейся в том, что если файл уже существует, то он открывается с нулевой длиной, при этом его прежнее содержимое теряется. Это неудобно. Чтобы предотвратить эффект очистки содержимого, для открытия файла можно использовать функцию 5bh или более современную функцию 6ch.

Открытие существующего файла

Когда файл создан, его можно открыть функцией 3dh. При этом необходимо указать режим доступа к файлу.

Вход: AH = 3dh; AL = режимы доступа и разделения — определяются состоянием битов: 2 = 0 — режимы доступа (000 — только чтение; 001 — только запись; 010 — чтение/запись); 3 — зарезервирован (0); 4–6 — режим совместного использования (000 — режим совместимости; 001 — запрет чтения и записи другими программами; 010 — блокировка записи другими программами; 011 — запрет чтения другими программами; 100 — разрешение полного доступа другим программам; 111 — сетевой FCB (доступен только в течение серверного вызова)); 7 — наследование (если установлен, то файл принадлежит только текущему процессу и не наследуется дочерними процессами); DS:DX — адрес ASCIIZ-цепочки с именем файла.

Выход: CF = 0 — AX = дескриптор файла; CF = 1 — AX = код ошибки: 3 — нет такого пути; 4 — нет свободного дескриптора файла; 5 — доступ отказан; 12h — недействительный код доступа.

Функция 3dh возвращает дескриптор файла, указатель в файле устанавливает на начало файла.

```

-----+-----
:| Программа: prg07_02.asm. Открытие существующего файла my_file.txt |
:| в текущем каталоге функцией 3dh файла. |
:|-----+-----
.data
handle      dw      0          : дескриптор файла
filename     db      'my_file.txt'. 0
point_fname  dd      filename
             ....
.code
             ....
mov     al, 02h          : режим доступа
lds     dx, point_fname  : формируем указатель на имя файла
mov     ah, 3dh          : номер функции DOS
int     21h             : открываем файл
jc      exit            : переход в случае ошибки
:----- действия при успешном открытии файла
mov     handle, ax       : сохраним дескриптор файла
             ....

```

Создание нового файла с сохранением существующего

Еще одна функция создания файла 5bh, в отличие от функции 3ch, позволяет провести процесс открытия файла более мягко — без ущерба прежнему содержимому.

Вход: AH = 5bh; CX = атрибуты файла; DS:DX — ASCII-имя файла.

Выход: CF = 0 — AX = дескриптор файла; CF = 1 — AX = код ошибки: 3 — нет такого пути; 4 — нет свободного дескриптора файла; 5 — отказ в доступе; 50h — файл существует.

Если указанный файл существует, то функция 5bh завершается с кодом ошибки 50h (CF = 1). Поэтому после вызова данной функции необходимо анализировать флаг CF (командой JC или JNC), если CF = 1, то для открытия файла необходимо дополнительно вызвать функцию 3Dh.

```

-----+-----
:| Программа: prg07_03.asm. Создание нового файла с сохранением |
:| существующего в текущем каталоге функцией 5bh. |
:|-----+-----
.data
handle      dw      0          : дескриптор файла
filename     db      'my_file.txt'. 0
point_fname  dd      filename
             ....
.code
             ....
xor     cx, cx          : атрибуты файла — обычный файл
lds     dx, point_fname  : формируем указатель на имя файла
mov     ah, 5bh          : номер функции DOS
int     21h             : открываем файл
jnc     m1              : обойдем открытие файла
mov     al, 02h          : режим доступа
mov     ah, 3dh          : номер функции DOS
int     21h             : открываем файл
jc      exit            : переход в случае ошибки
:----- действия при успешном открытии файла
mov     handle, ax       : сохраним дескриптор файла
             ....
m1:
             ....

```


Открытие или создание файла с расширенными возможностями

Функция 6ch появилась в последних версиях MS DOS (DOS 4.0+). С ее появлением устраняется необходимость отслеживать существование создаваемого файла. Для корректной работы достаточно задать нужные значения в соответствующих регистрах. Анализ возможных значений показывает, что данная функция фактически заменяет существовавшие до этого функции создания и открытия файлов.

Вход: AX = 6C00h; BL = флаги — режим открытия (значения битов: 7 — наследование; 4–6 — режим разделения; 3 — резерв (0); 0–2 — режим доступа); BH = флаги (значения битов: 6 = 0 — использовать стандартную для MS DOS буферизацию; 6 = 1 — отменить стандартную для MS DOS буферизацию; 5 = 0 — использовать обычный обработчик ошибок (int 24h); 5 = 1 — не использовать обычный обработчик ошибок (int 24h), для выяснения причины ошибки вызвать функцию 59h int 21h); CX = атрибуты создаваемого (и только) файла; DL = действия, если файл существует или не существует (значения битов: 0–3 — действие, если файл существует (0000 — вернуть ошибку; 0001 — открыть файл; 0002 — открыть файл без сохранения существующего); 4–7 — действие, если файл не существует (0000 — вернуть ошибку; 0001 — открыть файл; 0002 — создать и открыть файл); DH = 00h — резерв; DS:SI — адрес строки с ASCII-именем файла.

Выход: CF = 0 — успешное выполнение функции: AX = дескриптор файла, CX = состояние (0 — файл открыт; 1 — файл создан и открыт; 2 — файл открыт без сохранения содержимого существующего файла); CF = 1 — AX = код ошибки.

Следующий фрагмент программы показывает вариант применения функции 6ch.

```

+-----+
: | Программа: prg07_04.asm. Демонстрация открытия или создания файла |
: | с расширенными возможностями в текущем каталоге функцией 6Ch. |
+-----+
.data
handle      dw      0           : дескриптор файла
filename     db      'my_file.txt', 0
point_fname  dd      filename
            :...

.code
:....
xor         cx, cx             : атрибуты файла — обычный файл
mov         bx, 2              : режим доступа обычный —
                                : доступ для чтения-записи
:----- если файл существует, то открыть его, в противном случае
: вернуть ошибку (для эксперимента)
mov         dx, 1
lds         si, point_fname    : формируем указатель на имя файла
mov         ah, 6ch            : номер функции DOS
int         21h                : открываем файл
jnc         ml                 : если файл существовал, то переход
mov         dx, 10h            : открыть файл
mov         ah, 6ch            : номер функции DOS
int         21h                : открываем файл
jc          exit               : переход в случае ошибки
: ----- действия при успешном открытии файла
ml:         mov     handle, ax  : сохраним дескриптор файла
            :...

```

Закрытие файла

В конце работы с файлом его нужно закрыть. Но это действие не является обязательным, так как функция 4ch, которая завершает выполнение программы, в числе прочих действий выполняет и закрытие всех файлов.

Вход: AH = 3eh; BX = дескриптор файла, полученный при его открытии.

Выход: CF = 0 — AX = не определен; CF = 1 — AX = код ошибки: 6 — недопустимый дескриптор.

Во время закрытия файла выполняются все незаконченные операции записи на диск в элементе каталога, соответствующего файлу, модифицируются различные поля, в том числе поля времени и даты устанавливаются в текущее время.

```

+-----+
+| Программа: prg07_05.asm. Закрытие файла функцией 3eh. |
+-----+
.data
handle      dw      0          : дескриптор файла
filename     db      'my_file.txt', 0
point_fname  dd      filename
            ....
.code
            ....
:----- открываем файл
xor          cx, cx           : атрибуты файла — обычный файл
lds          dx, point_fname  : формируем указатель на имя файла
mov          ah, 5bh          : номер функции DOS
int          21h
jnc          m1               : обойдем открытие файла
mov          al, 02h           : режим доступа
mov          ah, 3dh          : номер функции DOS
int          21h              : открываем файл
jc           exit             : переход в случае ошибки
:----- действия при успешном открытии файла
m1:          mov          handle, ax      : сохраним дескриптор файла
:----- закрываем файл
m2:          mov          ah, 3eh
mov          bx, handle
int          21h
jnc          exit             : переход, если нет ошибки
:----- обработка ошибки
            ....
jmp          m2               : повторяем операцию закрытия
            ....

```

Удаление файла

При необходимости файл может быть удален функцией 41h.

Вход: AH = 41h; DS:DX — ASCII-имя файла (в последних версиях MS DOS можно использовать символы группирования * и ?); CL = атрибуты удаляемого файла.

Выход: CF = 0 — AX = не определен; CF = 1 — AX = код ошибки: 2 — файл не найден; 3 — нет такого пути; 5 — в доступе отказано.

```

+-----+
+| Программа: prg07_06.asm. Удаление файла функцией 41h. |
+-----+
.data
handle      dw      0          : дескриптор файла
filename     db      'my_file.txt', 0
point_fname  dd      filename
            ....

```

```
.code
;....
;----- открываем файл
xor     cx, cx           ; атрибуты файла — обычный файл
lds     dx, point_fname  ; формируем указатель на имя файла
mov     ah, 5bh          ; номер функции DOS
int     21h
jnc     m1               ; обойдем открытие файла
mov     al, 02h          ; режим доступа
mov     ah, 3dh          ; номер функции DOS
int     21h              ; открываем файл
jc      m3               ; переход в случае ошибки
;----- действия при успешном открытии файла
m1:     mov     handle, ax ; сохраним дескриптор файла
;----- закрываем файл
m2:     mov     ah, 3eh
mov     bx, handle
int     21h
jnc     exit             ; переход, если нет ошибки
;----- обработка ошибки
;....
jmp     m2               ; повторяем операцию закрытия
;----- удаляем файл
exit:   mov     ah, 41h
lds     dx, point_fname  ; формируем указатель на имя файла
xor     cx, cx           ; атрибуты файла
int     21h
jnc     m3               ; переход, если нет ошибки
;----- обработка ошибки
;....
jmp     exit             ; повторяем операцию удаления
m3:     ;....           ; выход из программы
```

Функция 41h не позволяет удалять файлы с атрибутом «только для чтения». В этом случае необходимо изменить атрибуты удаляемого файла с помощью функции 43h.

Создание временного файла

Вход: AH = 5Ah; CX = атрибуты файла; DS:DX — указатель на ASCII-строку с путем, заканчивающимся символом \ и 13 дополнительными нулевыми байтами, которые в результате вызова функции будут заполнены символами сгенерированного имени.

Выход: CF = 0 — AX = дескриптор файла, открытого для чтения/записи в режиме совместимости; DS:DX — путь к файлу, дополненный сгенерированным именем временного файла; CF = 1 — AX = код ошибки: 3 — нет такого пути; 4 — нет свободного дескриптора файла; 5 — в доступе отказано; 50h — файл существует.

В результате вызова функции 5Ah создается файл с уникальным именем, который после своего закрытия должен быть явно удален создавшей его программой. Закрывание файла производится функцией 3Eh.

```
+-----+
;| Программа: prg07_07.asm. Создание временного файла функцией 5ah. |
+-----+
.data
handle      dw      0           ; дескриптор файла
filename    db      '\'. 13 dup(0). 0
point_fname dd      filename
;....
.code
;....
```

```

:----- открываем временный файл
xor     cx, cx           : атрибуты файла — обычный файл
lds     dx, point_fname : формируем указатель на имя файла
mov     ah, 5ah          : номер функции DOS
int     21h
jc      m3              : в случае ошибки — на конец

:----- действия при успешном открытии файла
mov     handle, ax       : сохраним дескриптор файла

:----- закрываем файл
m2:     mov     ah, 3eh
        mov     bx, handle
        int     21h
        jnc     exit      : переход, если нет ошибки

:----- обработка ошибки
:....
        jmp     m2        : повторяем операцию закрытия

:----- удаляем файл
exit:   mov     ah, 41h
        lds     dx, point_fname : формируем указатель на имя файла
        xor     cx, cx       : атрибуты файла
        int     21h
        jnc     m3          : переход, если нет ошибки

:----- обработка ошибки
:....
        jmp     exit      : повторяем операцию удаления

:----- выход из программы
m3:     :....

```

В случае задания имени, как в примере выше, файл будет создан в корневом каталоге текущего диска. Для того чтобы разместить файл в конкретном каталоге, необходимо указать полный путь к нему с завершающим символом \ и 13 нулевыми байтами на конце, например:

```
Filename    db      'e:\asm_on_a\' , 13 dup(0), 0
```

Чтение, запись, позиционирование в файле

При работе с материалом данного раздела помните, что функции чтения и записи можно использовать не только с дескрипторами заранее открытых файлов, но и с дескрипторами стандартных устройств. Эти дескрипторы имеют постоянное значение и доступны в любое время функционирования системы: 0 — клавиатура; 1 и 2 — экран; 3 — последовательный порт COM1; 4 — параллельный порт LPT1.

Установка текущей файловой позиции

Чтение-запись в файле производится с текущей файловой позиции, на которую указывает *указатель позиции*. Функция 42h MS DOS предоставляет гибкие возможности как для начального, так и для текущего позиционирования файлового указателя для последующей операции ввода-вывода.

Вход: AH = 42h; BX = дескриптор файла, полученный при его открытии; AL = начальное положение в файле, относительно которого производится операция чтения-записи (00h — смещение (беззнаковое значение в CX:DX) от начала файла; 01h — смещение (значение со знаком в CX:DX) от текущей позиции в файле; 02h — смещение (значение со знаком в CX:DX) от конца файла); CX:DX = смещение новой позиции в файле относительно начальной.

Выход: CF = 0 — DX:AX = значение новой позиции в байтах относительно начала файла; CF = 1 — AX = код ошибки: 1 — неверное значение в AL; 6 — недопустимый дескриптор файла.

Методы позиционирования, заданные величиной в AL, по-разному трактуют значение в паре регистров CX:DX. Метод AL = 00h понимает значение в CX:DX как абсолютное. Два других метода (AL = 01h и AL = 02h) подразумевают, что в CX:DX значение со знаком. Необходимо быть внимательным при выполнении операции позиционирования во избежание последующих ошибок при операции чтения-записи. Так, значение в CX:DX, позиционирующее указатель, может указывать за пределы файла. При этом выделяются два случая:

- значение в CX:DX указывает на позицию перед началом файла — в этом случае последующая операция чтения-записи будет выполнена с ошибкой;
- значение в CX:DX указывает на позицию за концом файла — в этом случае последующая операция записи приведет к расширению файла в соответствии со значением в CX:DX.

Примеры использования функции 42h приведем при рассмотрении функций чтения-записи.

Запись в файл или устройство

Запись в файл производится функцией 40h с текущей позиции указателя.

Вход: AH = 40h; BX = дескриптор файла; CX = количество байтов для записи; DS:DX — указатель на область, из которой записываются данные.

Выход: CF = 0 — AX = число действительно записанных байтов в файл или устройство; CF = 1 — AX = код ошибки: 5 — в доступе отказано; 6 — недопустимый дескриптор.

Если при вызове функции 40h регистр CX равен нулю, то данные в файл не записываются и он усекается или расширяется до текущей позиции указателя. Если CX не равен нулю, то данные в файл записываются начиная с текущей позиции. Операция записи также продвигает указатель смещения на число действительно записанных байтов.

Положение указателя можно изменять явно с помощью функции 42h.

Вначале рассмотрим пример программы, выводящей данные на экран.

```

+-----+
+| Программа: prg07_08.asm. Вывод на экран строки функцией 40h. |
+-----+
.data
string      db      'строка для вывода на экран функцией 40h'
             len_string = $ - string
point_fname dd      string
             ...
.code
             ...
mov         bx, 1           ; стандартный дескриптор — экран
mov         cx, len_string
lds         dx, point_fname ; формируем указатель на строку string
mov         ah, 40h         ; номер функции DOS
int         21h             ; выводим
jc          exit            ; переход в случае ошибки
nop         .               ; для тестирования
             ...

```

Далее поглядим на пример программы, которая заполняет файл my_file.txt данными в виде строк символов, вводимых с клавиатуры. Длина строк — не более 80 символов. Нажатие клавиши Enter после ввода каждой строки означает, что эта

строка символов должна являться отдельной строкой файла `my_file.txt`. Соответственно, перед выводом каждой строки в файл в ее конце необходимо вставлять символы `0d0ah`. При нажатии клавиши `Space` (Пробел) в начале ввода очередной строки (ASCII-код — 32 (десятичный) или 20 (шестнадцатеричный)) направление ввода данных в файл изменяется следующим образом: файл расширяется на величину, равную количеству уже введенных символов, и дальнейший ввод осуществляется с конца файла. Завершение работы программы определяется моментом, когда оба введенных потока в файле встречаются (не перекрываясь).

```

-----+
: | Программа: prg07_09.asm. Заполнение файла my_file.txt данными |
: | в виде строк символов, вводимыми с клавиатуры. |
-----+

buf_0ah      struc
len_buf      db      B3          ; длина buf_0ah
len_in       db      0          ; действительная длина введенного слова
                                ; (без учета 0dh)
buf_in       db      82 dup (20h) ; буфер для ввода (с учетом 0dh
                                ; и позднее добавляем 0ah)

ends
.data
handle       dw      0          ; дескриптор файла
filename     db      'my_file.txt'. 0
point_fname  dd      filename
buf          buf_0ah <>
prev_d       label  dword       ; для сохранения длины предыдущей
                                ; строки при выводе с конца файла
prev         dw      0
            dw      0
middle       dd      0          ; позиция в середине файла,
                                ; при достижении которой снизу
                                ; выходим из программы

:....

.code
:....
:----- открываем файл
xor     cx, cx          ; атрибуты файла — обычный файл
mov     bx, 2           ; режим доступа — доступ для чтения-
                        ; записи, режим буферизации MS DOS
mov     dx, 12h         ; если файл существует, то открыть его
                        ; без сохранения прежнего содержимого,
                        ; иначе — создать его
lds     si, point_fname ; формируем указатель на имя файла
mov     ah, 6ch         ; номер функции DOS
int     21h             ; открываем (создаем) файл
jc      exit            ; если ошибка, то переход на конец
:----- действия при успешном открытии файла
mov     handle, ax      ; сохраним дескриптор файла
:----- позиционируем файловый указатель на начало файла
mov     ah, 42h
xor     al, al
xor     cx, cx
xor     dx, dx
mov     bx, handle
int     21h
:----- вводим очередную строку с клавиатуры
cycl:   lea     dx, buf
mov     ah, 0ah
int     21h
:----- для красоты выводим на экран символ 0ah
mov     dl, 0ah
mov     ah, 2

```

```

int      21h
cmp      buf.buf_in, 20h ; первый символ введенной строки
                        ; сравниваем с пробелом
je       revers         ; переход на изменение ввода – добавляем
                        ; 0ah в конец введенной строки

lea      si, buf.buf_in
mov      al, buf.len_in
cbw
push     si
add      si, ax
inc      si              ; учитываем неучтенный в len_in
                        ; символ 0dh

mov      byte ptr [si], 0ah
:----- вывод в файл
pop      dx              ; указатель на область, откуда
                        ; будем выводить строку

mov      bx, handle
add      ax, 2           ; учет в len_in символа 0dh
mov      cx, ax          ; длина выводимых данных
mov      ah, 40h
int      21h
jmp      cys1

:----- записываем файл с конца, предварительно расширив его.
: узнаем, сколько было уже записано ранее:
: для этого вначале сбрасываем буферы на диск
revers:   mov      bx, handle
mov      ah, 68h
int      21h

:----- теперь можно и узнать – определение длины файла
mov      al, 2
xor      cx, cx
xor      dx, dx          ; CX:DX = 0 – нулевое смещение
mov      ah, 42h
int      21h            ; в DX:AX -> длина файла в байтах
jc       exit           ; если ошибка
:----- формируем полную длину в edx
shl      eax, 16
shld     edx, eax, 16
mov      middle, edx     ; сохраним как условие выхода
                        ; при достижении снизу
:----- расширение файла с помощью функции 42h int 21h
: и последующей записи; умножаем длину на 2.
: при первой операции записи файл расширится
shl      edx, 1
shld     ecx, edx, 16
mov      al, 0
xor      cx, cx
mov      ah, 42h
int      21h            ; расширяем файл, устанавливая указатель
jc       exit           ; если ошибка
:----- расширим файл, выведя последнюю введенную строку с пробелом
cys12:   lea      si, buf.buf_in
mov      al, buf.len_in
cbw
push     si
add      si, ax
inc      si              ; учитываем в len_in символ 0dh
:----- добавляем 0ah в конец введенной строки
mov      byte ptr [si], 0ah
:----- выводим в файл
pop      dx              ; указатель на область, откуда
                        ; будем выводить строку
add      ax, 2           ; учитываем в len_in символ 0dh
mov      cx, ax          ; длина выводимых данных
mov      prev, ax        ; сохраним длину для корректировки

```

```

                                : при выводе следующей строки
mov     bx, handle
mov     ah, 40h
int     21h
jc      exit
:----- сбрасываем буфер, чтобы смотреть изменения в файле
:       при работе в отладчике — легче запретить (см. ниже)
mov     bx, handle
mov     ah, 68h
int     21h
:----- вводим очередную строку с клавиатуры
lea     dx, buf
mov     ah, 0ah
int     21h
:----- для лоска выводим на экран символ 0ah
mov     dl, 0ah
mov     ah, 2
int     21h
:----- использование 42h с отрицательным смещением относительно
:       текущего значения указателя позиции:
:       устанавливаем указатель в позицию вывода следующей строки
:       с учетом того, что выводим с конца (текущей позиции) файла
xor     ecx, ecx
mov     al, buf.len_in
cbw
add     prev, ax
add     prev, 2           : учитываем наличие 0d0ah
sub     ecx, prev_d       : получаем отрицательное смещение —
                        : сформируем его в паре CX:DX
shrd    edx, ecx, 16
shr     edx, 16           : "довернем" edx
shr     ecx, 16           : и ecx
:----- устанавливаем файловую позицию для записи очередной строки
mov     bx, handle
mov     ah, 42h
mov     al, 1             : смещение от текущей позиции
int     21h
:----- сравним текущую позицию с middle
shl     eax, 16
shld    edx, eax, 16
cmp     edx, middle
jl      exit
jmp     cyc12
exit:
:....

```

Программа выглядит не очень эстетично, но главная ее цель достигнута — показать работу с файловым указателем при записи в файл. Мы попробовали разные варианты: позиционирование на конец файла (при этом можно узнать длину файла); использование отрицательного смещения (задавая нулевое значение в CX:DX при AL = 1, можно получить в DX:AX текущую позицию в файле); расширение файла путем задания в CX:DX заведомо большего значения, чем длина файла. Как видно из программы выше, все эти и другие эффекты достигаются за счет манипулирования значениями в парах CX:DX и DX:AX, а также в регистре AL, где указывается начальное смещение в файле, относительно которого производится операция чтения-записи.

В заключение рассмотрения функции 40h записи в файл отметим то, для чего мы использовали функцию сброса буферов на диск 68h. Для этого коротко необходимо коснуться проблемы буферизации ввода-вывода в MS DOS. Эта ОС использует буферизацию ввода-вывода для ускорения работы с диском. В частности, дан-

ные, записываемые на диск, не попадают на него сразу, а помещаются вначале в буфер. Запись буфера на диск производится при его заполнении. Буферизация актуальна при интенсивной работе с одними и теми же данными. Тогда при необходимости чтения данных с диска они будут читаться из буфера. В нашей программе буферизация только мешала, так как при работе в отладчике мы не могли своевременно наблюдать за изменениями выходного файла `my_file.txt`. Для этого нам приходилось использовать функцию `68h` принудительного сохранения буферов на диск.

Вход: `AH = 68h`; `BX` = дескриптор файла.

Выход: `CF = 0` в случае успеха; `CF = 1` — `AX` = код ошибки.

В результате работы функции все данные из буферов дисков DOS немедленно записываются на диск, при этом модифицируется соответствующий файлу элемент каталога.

Для нашей задачи буферизацию лучше вообще запретить, тогда отпадет необходимость в принудительном сохранении строк в файле для того, чтобы в динамике отслеживать его изменения. Для этого при вызове функции `6ch` в регистре `BH` требуется установить бит 6 следующим образом: `6 = 0` — использовать стандартную для MS DOS буферизацию; `6 = 1` — отменить стандартную для MS DOS буферизацию. В нашем примере это может выглядеть так:

```

:----- открываем файл
xor     cx, cx           : атрибуты файла — обычный файл
mov     bx, 4002h        : режим доступа — доступ для
                           : чтения-записи, запрет буферизации
:----- если файл существует, то открыть его без сохранения прежнего
:      содержимого, в противном случае создать файл
mov     dx, 12h
lds     si, point_fname  : формируем указатель на имя файла
mov     ah, 6ch          : номер функции DOS
int     21h              : открываем (создаем) файл
jc      exit              : если ошибка, то переход на конец

```

Все вызовы функции `68h` в приведенной выше программе можно закомментировать.

Чтение из файла или устройства

Чтение из файла в область памяти осуществляется функцией `3Fh`.

Вход: `AH = 3Fh`; `BX` = дескриптор файла; `CX` = количество байтов для чтения; `DS:DX` — указатель на область памяти, в которую помещаются прочитанные байты.

Выход: `CF = 0` — `AX` = число действительно прочитанных байтов из файла; `CF = 1` — `AX` = код ошибки: 5 — в доступе отказано; 6 — недопустимый дескриптор.

Чтение данных производится начиная с текущей позиции в файле, которая после успешного чтения смещается на значение, равное количеству прочитанных байтов. Если в качестве файла используется стандартная консоль (клавиатура), то чтение выполняется до первого символа `CR` (carriage return) с кодом `0Dh`, соответствующего нажатию клавиши `Enter`. Это, кстати, еще один способ ввода данных с клавиатуры в программу. Кроме символов введенной строки в ее конец помещаются символы `0dh` и `0ah`. Это необходимо учитывать при задании размера буфера для ввода. Способ ввода данных с экрана с помощью функции `3Fh` иллюстрирует представленный ниже пример программы.

```

+-----+
+| Программа: prg07_10.asm. Ввод данных с экрана с помощью функции 3Fh. |
+-----+
.data
string          db      80 dup (" ")
len_string = $ - string
point_fname     dd      string
:....
.code
:....
:----- вводим с клавиатуры
mov     bx, 0           ; стандартный дескриптор — клавиатура
mov     cx, len_string
lds     dx, point_fname ; формируем указатель на строку
mov     ah, 3fh         ; номер функции DOS
int     21h
jc      exit           ; переход в случае ошибки
:----- выводим на экран
: (две строки ниже в данном случае можно опустить)
mov     bx, 1           ; стандартный дескриптор — экран
mov     cx, len_string
lds     dx, point_fname ; формируем указатель на строку string
mov     ah, 40h         ; номер функции DOS
int     21h             ; открываем файл
jc      exit           ; переход в случае ошибки
:....

```

Для демонстрации работы функции с дисковым файлом приведем программу чтения и вывода на экран содержимого файла, имя которого указывается в командной строке. Побочная цель этой программы — научиться обрабатывать в программе командную строку DOS. Поясним последний момент. Содержимое командной строки, следующее за именем программы при ее вызове, помещается в префикс программного сегмента (PSP) со смещением 80h от его начала и максимально имеет размер 128 байтов. Первый байт этой области содержит длину параметров команды, а первый символ параметров, при его наличии, располагается со смещением 81h от начала PSP. Последний символ параметров команды — всегда 0dh. Начало PSP найти очень легко — когда программа загружается в память для исполнения, то загрузчик устанавливает регистры ES и DS равными адресу PSP.

```

+-----+
+| Программа: prg07_11.asm. Чтение и вывод на экран содержимого файла, |
+| имя которого вводится в командной строке. |
+-----+
.data
file_name       db      128 dup (" ") ; буфер для пути к файлу
point_fname     dd      file_name
string          db      80 dup (" ")
len_string = $ - string
point_string    dd      string
handle          dw      0             ; дескриптор файла
size_f          dd      0             ; размер файла
:....
.code
main:
:----- копируем командную строку в file_name:
: вначале уберем (установкой указателя) ведущие пробелы
: в командной строке перед путем к файлу
mov     di, 81h
mov     al, " "
mov     cx, 128
repe    scasb

```

```

        dec     di
        push   di
        pop    si
        mov     ax, @data      ; адрес сегмента данных – в регистр AX
        mov     es, ax         ; ax в es
        mov     cl, ds:[80h]
        dec     cl
        lea     di, file_name
        rep     movsb
        push   es
        pop    ds
:----- открываем файл
        mov     al, 00h        ; режим доступа – только чтение
        lds     dx, point_fname ; формируем указатель на имя файла
        mov     ah, 3dh        ; номер функции DOS
        int     21h
        jc      exit           ; переход в случае ошибки
        mov     handle, ax
:----- определяем размер файла
        mov     bx, ax         ; дескриптор файла – в bx
        mov     al, 2
        xor     cx, cx
        xor     dx, dx         ; CX:DX = 0 – нулевое смещение
        mov     ah, 42h
        int     21h            ; в DX:AX -> длина файла в байтах
        jc      exit           ; если ошибка
:----- формируем полную длину в edx
        shl     eax, 16
        shld    edx, eax, 16
        mov     size_f, edx    ; сохраним как условие выхода
                                ; при достижении снизу
:----- устанавливаем указатель на начало файла
        mov     bx, handle     ; дескриптор файла – в bx
        mov     al, 0
        xor     cx, cx
        xor     dx, dx         ; CX:DX = 0 – нулевое смещение
        mov     ah, 42h
        int     21h            ; текущий указатель в начале файла
        jc      exit           ; если ошибка
:----- читаем файл по len_string байтов
cyc1:   mov     bx, handle     ; дескриптор файла в bx
        mov     cx, len_string
        lds     dx, point_string ; формируем указатель на строку
        mov     ah, 3fh        ; номер функции DOS
        int     21h            ; открываем файл
        jc      exit           ; переход в случае ошибки
:----- выводим на экран целиком
        mov     bx, 1          ; стандартный дескриптор – экран
        mov     cx, len_string
        lds     dx, point_string ; формируем указатель на строку
        mov     ah, 40h        ; номер функции DOS
        int     21h            ; открываем файл
        jc      exit           ; переход в случае ошибки
        cwde                ; расширяем количество выводимых байтов
        sub     size_f, eax
        cmp     size_f, 0
        jle     exit           ; достигли конца файла?
        jmp     cyc1
:----- выход из программы
exit:   mov     al, 1
        int     21h
        ....

```

Не забывайте после определения размера файла возвращать файловый указатель в нужное место файла.

Получение и изменение атрибутов файла

MS DOS позволяет получить для анализа и при необходимости изменить имя файла, байт атрибутов файла, время и дату его последней модификации в соответствующем элементе каталога. Для этого предназначены функции 43h, 56h, 57h. Подфункция 00h функции 43h прерывания 21h предназначена для получения слова атрибутов файла.

Получить атрибуты файла

Вход: AX = 4300h; DS:DX — ASCIIZ-строка с именем (путем) файла.

Выход: CF = 0 — CX = слово атрибутов файла; CF = 1 — AX = код ошибки: 1 — неверное значение в AL; 2 — файл не найден; 3 — несуществующий путь; 5 — доступ запрещен.

```

+-----+
+| Программа: prg07_12.asm. Получение атрибутов файла. |
+-----+
.data
fname      db      "maket.asm"
point_fname dd      fname
; ...
.code
; ...
;----- получим атрибуты файла
lds        dx, point_fname : формируем указатель на строку
mov        ax, 4300h       : номер функции DOS
int        21h
jc         exit             : переход в случае ошибки
; ...                       : в cx атрибуты (см. ниже)

```

Напомним формат байта атрибутов:

Биты	Описание
7	Разделяемый в Novell NetWare
6	Не используется
5	Архивный
4	Каталог
3	Метка тома (только исполнение Novell NetWare)
2	Системный
1	Скрытый
0	Только чтение

Установить атрибуты файла

Подфункция 01h функции 43h прерывания 21h предназначена для установки значений слова атрибутов файла.

Вход: AX = 4301h; CX = новое слово атрибутов файла; DS:DX — ASCIIZ-строка с именем (путем) файла.

Выход: CF = 0 — AX = не определен; CF = 1 — AX = код ошибки: 1 — неверное значение в AL; 2 — файл не найден; 3 — несуществующий путь; 5 — доступ запрещен.

Переименовать файл

Для переименования файла используется функция 56h.

Вход: AH = 56h; DS:DX — ASCIIZ-имя существующего файла; ES:DI — ASCIIZ-имя нового файла; CL = маска атрибутов.

Выход: CF = 0 — при успешном переименовании; CF = 1 — AX = код ошибки: 2 — файл не найден; 3 — несуществующий путь; 5 — доступ запрещен; 11h — устройства для старого и нового файлов не совпадают.

Функция 56h позволяет произвести перемещение между каталогами, не изменяя устройства.

```

;-----+
; | Программа: prg07_13.asm. Перемещение между каталогами      |
; | без смены устройства функцией 56h.                          |
;-----+
.data
fname_s      db      "maket.asm". 0
point_fname_s dd      fname_s
fname_d      db      "e:\maket.asm". 0
point_fname_d dd      fname_d
;...
.code
;...
;----- переместим файл из текущего в корневой каталог
lds     dx, point_fname_s ; указатель на fname_s (исх. файл)
di,     point_fname_d ; указатель на fname_d (целев. файл)
mov     ah, 56h ; номер функции DOS
int     21h
jc      exit ; переход в случае ошибки
;... ; в сч атрибуты (см. ниже)

```

Получить дату и время создания или последней модификации файла

Получить/изменить дату и время создания или модификации файла можно с помощью подфункций функции 57h.

Вход: AX = 5700h; BX = дескриптор файла.

Выход: если CF = 0 — CX = время, DX = дата. Если CF = 1 — AX = код ошибки (CF = 1): 1 — недопустимый номер подфункции в AL; 6 — недопустимый дескриптор.

Время и дата файла получают в следующих форматах.

Время		Дата	
Биты	Описание	Биты	Описание
15–11	Часы (0–23)	15–9	Год
10–5	Минуты	8–5	Месяц
4–0	Секунды	4–0	День

Установить дату и время создания или последней модификации файла

Вход: AX = 5701h; BX = дескриптор файла; CX = новое время, DX = новая дата.

Выход: если CF = 0 — CX = время, DX = дата. Если CF = 1 — AX = код ошибки: 1 — недопустимый номер подфункции в AL; 6 — недопустимый дескриптор.

Работа с дисками, каталогами и организация поиска файлов

Задача поиска традиционно является актуальной. При рассмотрении вопроса работы с файлами ее также не обойти. Мы рассмотрим номенклатуру средств, предлагаемых MS DOS для поиска файла и определения его местоположения в древовидной структуре каталогов текущего диска.

Знакомясь с предыдущими программами, вы должны были заметить, что при задании имен файлов мы практически не указывали имен дисководов и путей к этим файлам. MS DOS имеет средства для установки текущего диска и каталога, в котором выполняются все текущие операции с файлами. При необходимости текущий диск и каталог можно изменить. Ниже приведено несколько функций для работы с текущими диском и каталогом — определение, смена, получение информации.

Получить номер заданного по умолчанию дисковода

Вход: AH = 19h.

Выход: AL — номер дисковода (00h — A:, 01h — B: и т. д.).

```

:-----+-----
:] Программа: prg07_14.asm. Получение номера текущего (по умолчанию)
:] дисковода функцией 19h.
:-----+-----
.code
:....

:----- получить номер текущего (по умолчанию) дисковода
mov     ah, 19h          ; номер функции DOS
int     21h
jc      exit             ; переход в случае ошибки
:....                   ; в al номер текущего диска

```

Выбрать заданный по умолчанию диск

Вход: AH = 0Eh; DL = номер нового диска по умолчанию (00h — A:, 01h — B: и т. д.).

Выход: AL = максимально возможный в данной системе номер дисковода (00h — A:, 01h — B: и т. д.) определяется на основе параметра LASTDRIVE в файле CONFIG.SYS.

Получить информацию о свободном дисковом пространстве

Вход: AH = 36h; DL = номер диска (00h — текущий, 01h — A: и т. д.).

Выход: AX = FFFFh — неправильный номер устройства в DL; иначе: AX = число секторов в одном кластере; BX = количество свободных кластеров; CX = размер сектора (в байтах); DX = общее число кластеров на диске.

Используя информацию, возвращаемую функцией 36h, можно подсчитать как свободное пространство на диске — произведение $AX \times BX \times CX$, так и полный объем диска — произведение $AX \times CX \times DX$.

MS DOS предоставляет следующие возможности для манипулирования каталогами: создание и удаление каталога, получение информации о текущем каталоге и его смена.

Создание каталога

Вход: AH = 39h; DS:DX — ASCIIZ-строка пути к создаваемому каталогу.

Выход: AX = не определен (CF = 0); AX = код ошибки (CF = 1): 3 — несуществующий путь; 5 — доступ запрещен.

Путь к каталогу должен содержать перечисление всех каталогов начиная от корневого на пути к создаваемому каталогу; при этом они, естественно, должны существовать. Последнее имя пути — имя создаваемого каталога.

```

;+-----+
;| Программа: prg07_15.asm. Создание каталога функцией 39h. |
;+-----+
.data
dname      db      "c:\windows\my_dir", 0
point_dname dd      dname
;...

.code
;...
;----- создадим каталог в каталоге c:\windows
lds        dx, point_dname ; формируем указатель на строку
; с именем нового каталога
mov        ah, 39h         ; номер функции DOS
int        21h
jc         exit            ; переход в случае ошибки
;...

```

Удаление каталога

Вход: AH = 3Ah; DS:DX — ASCIIZ-строка пути к удаляемому каталогу.

Выход: CF = 0 — AX = не определен; AX = код ошибки (CF = 1): 3 — несуществующий путь; 5 — доступ запрещен; 10h — попытка удаления текущего каталога.

Удаляемый каталог должен быть пустым.

```

;+-----+
;| Программа: prg07_16.asm. Удаление каталога функцией 3Ah. |
;+-----+
.data
dname      db      "c:\windows\my_dir", 0
point_dname dd      dname
;...

.code
;...
;----- удалим каталог my_dir в каталоге c:\windows
lds        dx, point_dname ; формируем указатель на строку
; с именем нового каталога
mov        ah, 3ah         ; номер функции DOS
int        21h
jc         exit            ; переход в случае ошибки
;...

```

Изменить текущий каталог

MS DOS позволяет установить текущий каталог для того, чтобы не указывать полный путь для последующих операций с файлами. При необходимости можно получить полный путь к текущему каталогу в виде ASCIIZ-строки.

Вход: AH = 3Bh; DS:DX — указатель на буфер, содержащий полный путь от корневого каталога в виде ASCIIZ-строки (до 64 байтов).

Выход: CF = 0 — AX = не определен; CF = 1 — AX = код ошибки: 03h — путь не найден.

```

:-----+
:| Программа: prg07_17.asm. Изменение текущего каталога функцией 3Bh. |
:-----+
.data
dname      db      "c:\windows", 0
point_dname dd      dname
           :....
.code
           :....
:----- изменить текущий каталог на каталог c:\windows
           lds      dx, point_dname : формируем указатель на строку
                                           : с именем нового каталога
           mov      ah, 3bh          : номер функции DOS
           int      21h
           jc        exit            : переход в случае ошибки
           :....

```

Получение текущего каталога

Вход: 0Ah = 47h; DL = номер устройства (00h = текущее (по умолчанию), 01h — A: и т. д.); DS:SI — указатель на 64-байтовый буфер для записи полного пути от корневого каталога (ASCIIZ-строка).

Выход: AX = не определен или 0100h (CF = 0);

AX = код ошибки (CF = 1): 0Fh — недопустимый дисковод.

```

:-----+
:| Программа: prg07_18.asm. Получение текущего каталога функцией 47h. |
:-----+
.data
dname      db      "e:\tools", 0
point_dname dd      dname
d_cur_name db      64 dup (20h), 0
point_d_cur_name dd  d_cur_name
           :....
.code
           :....
:----- изменить текущий каталог на каталог \tools
           lds      dx, point_dname : формируем указатель на строку
                                           : с именем нового каталога
           mov      ah, 3bh          : номер функции DOS
           int      21h
           jc        exit            : переход в случае ошибки
:----- получим текущий каталог
           lds      si, point_d_cur_name : формируем указатель на строку
                                           : с именем нового каталога
           mov      ah, 47h          : номер функции DOS
           int      21h
           jc        exit            : переход в случае ошибки
           :....

```

Возвращаемый путь не содержит имени диска и первого символа \.

Последняя проблема, на которой мы остановимся в этом разделе, — проблема поиска файлов. Для поиска в каталогах используется пара функций — 4eh и 4fh. В имени искомого файла можно указывать символы подстановки * и ?. Совместное использование функций 4eh и 4fh подчинено следующему алгоритму. Первой вызывается функция 4eh. В качестве параметров ей передаются адрес ASCIIZ-строки с путем к искомому файлу и комбинация его атрибутов. Имя файла может быть

задано в виде шаблона. В случае успеха ($CF = 0$), то есть при обнаружении первого подходящего шаблону файла данная функция помещает его имя и расширение в область DTA со смещением $1Eh$ от ее начала (см. таблицу ниже). Далее можно либо открыть файл, либо продолжить поиск, но уже функцией $4Fh$. При работе с шаблоном функцию $4Fh$ разумно вызывать циклически, до тех пор пока в процессе перебора не будут просмотрены имена всех подходящих файлов. Об этом можно узнать по состоянию флага CF , которое должно стать равным 1 в случае, когда файлов, удовлетворяющих шаблону, в данном каталоге больше нет.

Поиск первого удовлетворяющего шаблону файла

Вход: $AH = 4Eh$; CX = атрибуты файла (биты 0 и 5 игнорируются); $DS:DX$ — ASCII-имя файла (возможно, с путем к нему и символами шаблона * и ?).

Выход: если $CF = 0$, то в DTA возвращается блок данных для первого найденного файла (см. ниже). Если $CF = 1$, то в AX = код ошибки: 2 — файл не найден; 3 — несуществующий путь; $12h$ — больше файлов в каталоге нет.

Область DTA (Data Transfer Area) располагается в префиксе программного сегмента со смещением $80h$ от его начала и занимает 128 байтов. При успешном окончании поиска функция $4Eh$ (и $4Fh$ тоже) формирует блок данных, имеющий приведенный ниже формат.

Смещение	Размер в байтах	Описание
00h	1	Буква логического диска, если бит 7 = 0, то удаленный диск
01h	11	Поисковый шаблон
0Ch	1	Атрибуты поиска
0Dh	2	Порядковый номер файла в каталоге
0Fh	2	Номер кластера начала каталога предыдущего уровня
11h	4	Резерв
15h	1	Атрибуты найденного файла
16h	2	Время создания (модификации) файла
18h	2	Дата создания файла
1Ah	4	Размер файла
1Eh	13	ASCII-имя файла с расширением

После анализа данной области в программе принимается решение об окончании или продолжении поиска.

В качестве шаблона можно задать символы *.* , тогда мы сможем получить имена и проанализировать все файлы в текущем каталоге. Это бывает полезным при программировании операции перемещения или копирования каталога.

Найти следующий соответствующий шаблону файл

Вход: $AH = 4Fh$; в области DTA должен содержаться блок данных, заполненный единственным вызовом $4Eh$ в начале поиска.

Выход: если CF = 0 — успех; если CF = 1 — в AX = код ошибки: 12h — больше файлов в каталоге нет.

Для работы с DTA в системе MS DOS существуют две функции: 1ah и 2fh. После старта программы текущей DTA является область по адресу PSP:0080h. Мы уже имели с ней дело, когда рассматривали проблему обработки содержимого командной строки.

Получить адрес области DTA

Для выполнения работы, связанной с файлами, MS DOS предоставляет возможность установить собственную область DTA.

Вход: AH = 2Fh.

Выход: ES:BX — адрес области, которую впоследствии функцией 1ah можно сделать текущей областью DTA для последующих операций ввода-вывода.

Установить текущую область DTA

Вход: AH = 1Ah; DS:DX — адрес области, которая будет областью DTA для последующих файловых операций.

Понятно, что даже если мы устанавливаем свою область DTA, все смещения и данные, формируемые функциями 4Eh и 4Fh, остаются актуальными.

Работа с файлами в MS DOS (длинные имена)

Перечисленные выше функции работают в различных версиях «чистой» системы MS DOS, вплоть до версии 6.22 включительно. Операционные системы Windows 95/98/Me также поддерживают свою версию MS DOS, которая имеет номер 7.0. Операционные системы Windows 95/98/Me организуют для программ MS DOS специальное окружение, называемое *сеансом DOS*. Система MS DOS 7.0, будучи созданной для работы в среде Windows 95, имеет в своем составе средства для работы с файловой системой Windows. Эта файловая система, как известно, отличается тем, что полное имя файла может достигать длины до 255 символов. MS DOS 7.0 также умеет работать с длинными именами файлов. В данном разделе мы рассмотрим предназначенные для работы с файловой системой Windows средства среды MS DOS, работающей под управлением Windows.

Определить факт того, в какой системе работает программа, можно по результатам работы функций 30h — получить версию DOS (прерывания 21h) и 4a33h (прерывания 2fh).

Использование функции 30h предполагает следующие входные и выходные значения:

Вход: AH = 30h; AL — определяет значение, возвращаемое в BH: 00h — OEM-номер (как для DOS 2.0–4.0x); 01h — номер версии.

Выход: AL = старший номер версии; AH = младший номер версии; BL:CH = 24-битовый серийный номер пользователя (необязательно).

```
+-----+
+| Программа: prg07_19.asm. Определение, в какой системе работает программа. |
+-----+
```

```
.code
:-----
:....
:определим номер версии ОС MS DOS
mov     al, 00
mov     ah, 30h           : номер функции DOS
int     21h              : AL = главный номер версии.
                        : AH = младший номер версии
jc      exit              : переход в случае ошибки
:....
```

В регистрах AL и AH возвращаются главный и вторичный номера версии MS DOS. При функционировании под Windows эти номера равны 07h и 0ah соответственно. Задание значения AL = 1 дает такой же эффект.

```
+-----+
+| Программа: prg07_20.asm. Определение факта работы в среде MS DOS 7.0. |
+-----+
```

```
.code
:-----
:....
:определить факт работы в среде MS DOS 7.0
mov     ax, 4a33h         : номер функции DOS
int     2fh
cmp     ax, 0
jne     exit              : переход, если не MS DOS 7.0
:....
```

Последняя функция возвращает 0 для MS DOS версии 7.0 и выше.

Установить факт того, что система поддерживает длинные имена файлов, можно вызовом функции 71a0h прерывания 21h — *получить информацию о томе*. Если она возвращает ошибку (CF = 1), то текущая файловая система не поддерживает длинных имен файлов. Для вызова этой функции необходимо указать корневой каталог интересующего тома.

Вход: AX = 71A0h; DS:SI — ASCIIZ-имя корневого каталога тома, о котором нужно получить информацию (например, C:\); ES:DI — буфер для имени файловой системы; CX = размер буфера, адрес которого задан в ES:DI (32 байта).

Выход: CF = 0 в случае успеха, следующие регистры установлены: BX = флаги файловой системы: 0 — при поиске учитывать регистр букв в именах файлов; 1 — сохранять регистр букв в элементах каталога; 2 — использование символов Unicode в именах каталогов и файлов; 3–13 — резерв (0); 14 — поддержка DOS-функций для длинных имен файлов; 15 — сжатый том; CX = максимальная длина файловых имен (обычно 255); DX = максимальная длина пути (обычно 260); ES:DI — в буфере по этому адресу находится ASCIIZ-имя файловой системы, например FAT, FAT32, NTFS, CDFS; CF = 1 в случае неудачи, при этом AX = код ошибки или AX = 7100h, если функция не поддерживается.

```
+-----+
+| Программа: prg07_21.asm. Определение факта поддержки длинных имен файлов. |
+-----+
```

```
.data
dname      db      "c:\". 0
point_dname dd      dname
f_name      db      64 dup (20h). 0
point_f_name dd      f_name
.code
:-----
:....
:определить факт поддержки длинных имен файлов
lds     si, point_dname
les     di, point_f_name
```

```

mov     cx, 32
mov     ax, 71a0h      ; номер функции DOS
int     21h
jc       exit          ; переход, если текущая файловая система
                        ; не поддерживает длинных имен файлов
; ...

```

В Windows 95/98 появились дополнительные возможности как самой файловой системы, так и средств по ее управлению. Основное нововведение — поддержка длинных имен файлов. Основа файловой системы та же — таблица размещения файлов FAT, но любой файл в этой системе имеет два имени — длинное имя и его псевдоним, который соответствует формату 8.3. Данный псевдоним создается системой Windows 95/98 автоматически.

Нужно правильно понимать различие в способах использования длинных имен файлов в приложениях MS DOS и Win32. Приложения MS DOS получают доступ к длинным именам файлов с помощью дополнительных функций прерывания 21h. Приложения Windows используют для этого соответствующие функции API.

Длинное имя файла представляет собой ASCIIZ-строку длиной до 255 символов. Система формирует псевдоним для этого имени форматом 8.3 в соответствии со следующим правилом: берутся первые 6 символов длинного имени, после них добавляется символ тильды (~), за тильдой ставится некий порядковый номер. Для первого имени формата 8.3 это 1. Если такой псевдоним уже существует, то порядковый номер очередного псевдонима будет на единицу больше. Расширение псевдонима формируется из первых трех символов расширения длинного имени (если оно существует). Если похожих имен много, то номер в псевдониме может быть двузначным, при этом первая (символьная) часть псевдонима сокращается до 5 символов и т. д.

Рассмотренные выше функции MS DOS для работы с файлами и каталогами не поддерживают длинных имен. Для этого система Windows 95/98 предоставляет приложениям MS DOS аналогичные функции, но имеющие другие номера. Впрочем, при внимательном рассмотрении большинства из этих номеров видно, какой из старых функций они соответствуют. Новые номера состоят из четырех цифр: первые две — 71h, последние две — номер старой функции. Для некоторых функций существуют особенности в их работе. Так, для поиска файлов по-прежнему используются две функции (по новой нумерации — 714eh и 714fh) прерывания 21h. Новые функции теперь помещают информацию о файлах в специальную структуру WIN32_FIND_DATA, адрес которой возвращается в качестве результата их работы.

При работе с функциями, поддерживающими длинные имена файлов, используются еще две структуры: BY_HANDLE_FILE_INFORMATION и FILETIME. Назначение структуры BY_HANDLE_FILE_INFORMATION и работу с ней мы рассмотрим при обсуждении функции MS DOS 71a6h. Структура FILETIME содержит 64-разрядное значение, которое определяет число станаосекундных интервалов, прошедших с 12:00 утра 1 января 1901 года.

```

FILETIME struct
DwLowDateTime dd ?      ; младшие 32 бита значения времени
DwHighDateTime dd ?     ; старшие 32 бита значения времени
FILETIME ends

```

Теперь приведем перечень функций прерывания 21h, работающих с файлами, которые имеют длинные имена. Для удобства дальнейшего рассмотрения в следу-

ющей таблице представлены соответствующие функции API Win32 и «старые» функции прерывания 21h.

Новая функция int 21h	Старая функция int 21h	Назначение	Функция API Win32
5704h		Получить дату и время последнего доступа	GetFileTime
5705h		Установить дату и время последнего доступа	SetFileTime
5706h		Получить дату и время создания	GetFileTime
5707h		Установить дату и время создания	SetFileTime
7139h	39h	Создать каталог	CreateDirectory
713Ah	3Ah	Удалить каталог	RemoveDirectory
713Bh	3Bh	Изменить текущий каталог	SetCurrentDirectory
7141h	41h	Удалить файл	DeleteFile
7143h	43h	Получить или установить атрибуты файла	GetFileAttributes, SetFileAttributes
7147h	47h	Получить текущий каталог	GetCurrentDirectory
714Eh	4Eh	Найти первый файл	FindFirstFile
714Fh	4Fh	Найти следующий файл	FindNextFile
7156h	56h	Переименовать файл	MoveFile
7160h		Получить полный путь	GetFullPathName
7160h		Получить полный путь с краткими именами	GetShortPathName
7160h		Получить полный путь с длинными именами	Отсутствует
716Ch	3Ch, 3Dh, 5Bh	Создать или открыть файл	CreateFile, OpenFile
71A0h		Получить информацию о томе	GetVolumeInformation
71A1h		Завершить поиск	FindClose
71A6h		Получить информацию о файле по описателю	GetFileInformationByHandle
71A7h		Преобразовать время файла в DOS-время	FileTimeToDosDateTime
71A7h		Преобразовать из DOS-времени	DosDateTimeToFileTime
71A8h		Создать псевдоним	Отсутствует
71A9h		Создать или открыть файл на сервере	Отсутствует
71AAh		Провести подмену	Отсутствует
71AAh		Отменить подмену	Отсутствует
71AAh		Получить информацию о подмене	Отсутствует

Остановимся на наиболее интересных в контексте нашего изложения функциях этой таблицы. Информацию по остальным функциям можно получить, в частности, из библиотеки MSDN. Порядок рассмотрения проведем от простых функций к сложным так, как это было в предыдущем разделе, посвященном функциям работы с файлами MS DOS, имеющими короткие имена.

Создание, открытие, закрытие и удаление файла

Функции MS DOS, поддерживающие длинные имена файлов, имеют номера из четырех цифр — первые две равны 71h, последние две соответствуют номеру аналогичной старой функции MS DOS. В программах старые и новые функции применяются вместе по принципу: там, где функция должна работать непосредственно с длинными именами файлов и каталогов, срабатывают новые функции; там, где функции нужен дескриптор файла, используются старые функции. Новые функции также предназначены для реализации новых возможностей по работе с файловой системой.

Открытие или создание файла

Для создания или открытия файла с длинным именем используется функция 716Ch (создать или открыть файл). Эта функция аналогична функции 6ch, которая появилась в последних версиях MS DOS (DOS 4.0+). Мы уже обсуждали ее в разделе, посвященном функциям работы с файлами с короткими именами.

Вход:

- AX = 716Ch;
- BX = режимы доступа и флаги:
 - режим доступа: 0000h — файл только для чтения; 0001h — файл только для записи; 0002h — файл для чтения и записи; 0003h — резерв; 0004h — открыть файл для чтения без изменения даты последнего доступа к файлу;
 - режим разделения: 0000h — режим эмуляции — файл можно открывать любой программой любое количество раз; 0010h — файл открыт в монопольном режиме доступа; 0020h — файл открыт в монопольном режиме доступа по записи; 0030h — файл открыт в монопольном режиме доступа по чтению; 0040h — открыть файл, разрешая другим процессам доступ по чтению-записи, но с запретом режима эмуляции;
 - флаги: 0080h — дочерний процесс не наследует дескриптор файла, его при необходимости нужно передавать явно; 0100h — не использовать буферизацию или кэширование средствами ОС, операции чтения-записи выполняются напрямую с диском в соответствии с текущим положением указателя позиции; 0200h — файл нельзя сжимать; 0400h — содержимое регистра DI следует трактовать как порядковый номер в псевдониме файла; 2000h — не вызывать обработчик критической ошибки (int 24h), MS DOS вернет программе код ошибки; 4000h — после каждой операции записи MS DOS будет отправлять данные на диск без их кэширования;
- CX = атрибуты создаваемого (и только) файла: 0000h — файл доступен по записи и чтению; 0001h — файл доступен по чтению; 0002h — скрытый файл; 0004h — системный файл; 0008h — метка тома; 0020h — архивный файл;

■ **DX** = действия, если файл существует или не существует, значения битов: 0010h — вернуть ошибку, если файл существует, иначе создать файл; 0001h — открыть файл, если он существует, иначе вернуть ошибку; 0002h — открыть файл без сохранения существующего, в противном случае вернуть ошибку (если файл не существует);

■ **DS:SI** — ASCII-имя файла;

■ **DI** — порядковый номер, который добавляется к концу имени в псевдониме файла (для этого должен быть задан флаг 0400h в регистре **BX**). Номер будет десятичным письмом, то есть если **DI** = 0010h, то конец псевдонима — ~16.

Выход: **CF** = 0 — успешное выполнение функции; **AX** = дескриптор файла, **CX** = состояние: 1 — файл открыт; 2 — файл создан и открыт; 2 — файл открыт без сохранения содержимого существующего файла; **CF** = 1 — **AX** = код ошибки.

После того как файл открыт или создан функцией 716ch, с ним можно работать, используя старые функции чтения-записи и позиционирования. Следующий фрагмент программы показывает вариант применения функции 716Ch.

```

+-----+
| Программа: prg07_22.asm. Применение функции 716Ch для создания |
| или открытия файла с длинным именем.                             |
+-----+
.data
handle      dw      0          ; дескриптор файла
filename     db      'my_file with long name.txt', 0
point_fname  dd      filename
            ....
.code
            ....
;----- открываем файл
mov     bx, 0100h+0400h      ; не использовать буферизацию +
                               ; содержимое DI в псевдоним
mov     dx, 1                ; открыть файл, если он существует,
                               ; иначе вернуть ошибку
lds     si, point_fname      ; формируем указатель на имя файла
di, 7                          ; добавить в конец псевдонима символ 7
repeat:  mov     ax, 716ch     ; номер функции DOS
xor     cx, cx                ; атрибуты файла — обычный файл —
                               ; доступ для чтения-записи

int     21h                  ; если файл существовал, то переход
jnc     m1                    ; создать файл
mov     dx, 10h               ; переход — повторим открытие файла
jmp     repeat
;----- действия при успешном открытии файла
m1:      mov     handle, ax    ; сохраним дескриптор файла
            ....

```

Закрытие файла производится функцией 3Eh, которая использовалась для файловых функций MS DOS с короткими именами.

Удаление файла

Удаление файлов, имеющих длинные имена, производится функцией 7141h прерывания 21h. Имя файла может быть задано при помощи символов шаблона * и ?, тогда в результате работы функции будут удалены все файлы, чье имя удовлетворяет заданному шаблону.

Вход: **AX** = 7141h; **SI** = ограничения поиска: 0 — символы шаблона в имени файла и атрибуты в **CX** для поиска не разрешены; 1 — можно задавать символы шаблона

в имени файла и атрибуты в CX для поиска; CX — обязательные (CH) и необязательные (CL) атрибуты для поиска удаляемых файлов: 00h — файл для чтения-записи; 01h — файл только для чтения; 02h — скрытый файл; 04h — системный файл; 08h — метка тома; 10h — каталог; 20h — архивный файл; DS:DX — ASCII-имя файла (символы шаблона разрешены, если SI = 1 (см. выше)).

Выход: CF = 0 — успешное выполнение функции (AX = не определен); CF = 1: AX = код ошибки: 2 — файл не найден; 3 — нет такого пути; 5 — в доступе отказано; AX = 7100h — функция не поддерживается.

```

-----+
:| Программа: prg07_23.asm. Применение функции 7141h прерывания 21h
:| для удаления файлов, имеющих длинные имена.
:|-----+
.data
handle          dw      0              : дескриптор файла
filename         db      'my_file with long name.txt', 0
point_fname     dd      filename
fname_del       db      'my_file with long name.*', 0
point_fname_del dd      fname_del
               ....

.code
               ....
:----- открываем файл
mov     bx, 0100h+0400h : не использовать буферизацию +
                        : содержимое DI в псевдоним
mov     dx, 1           : открыть файл, если он существует.
                        : иначе вернуть ошибку
lds     si, point_fname : формируем указатель на имя файла
mov     di, 7           : добавить в конец псевдонима символ 7
repeat: mov     ax, 716ch : номер функции DOS
xor     cx, cx          : атрибуты файла — обычный файл —
                        : доступ для чтения-записи

int     21h
jnc     m1              : если файл существовал, то переход
mov     dx, 10h        : создать файл
jmp     repeat          : переход — повторим открытие файла
:----- действия при успешном открытии файла
m1:     mov     handle, ax : сохраним дескриптор файла
               ....
:----- удалим файл
xor     cx, cx          : атрибуты файла
mov     si, 1           : будем использовать шаблон для поиска
                        : и последующего удаления
lds     dx, point_fname_del : формируем указатель на имя файла
mov     ax, 7141h
int     21h
               ....

```

В результате работы этой программы будет создан файл my_file with long name.txt, который затем будет удален в соответствии с шаблоном my_file with long name.*.

Чтение, запись, позиционирование в файле, имеющем длинное имя, производится старыми функциями MS DOS 3fh и 40h.

Получение и изменение атрибутов файла

В «старой» MS DOS существовал ряд функций для получения и установки таких характеристик файла, как его атрибуты, время создания или последней его модификации. В версии MS DOS, используемой в Windows, также имеются подобные

функции, но у них есть свои особенности. Эти особенности объясняются тем, что в Win32 несколько расширен набор характеристик, связанных с файлом. Поэтому для получения и изменения атрибутов файла требуется функция 7143h, которая по сравнению с аналогичной функцией 43h работает с большим объемом информации. Так, помимо информации об атрибутах файла, функция 7143h формирует время и дату создания и модификации файла. Следует отметить, что в Win32 с файлом связаны три значения времени: время создания файла, время последнего доступа к файлу и время последней записи в файл. Ниже приведены дополнительные функции для работы с этими характеристиками файла в Windows-версии MS DOS.

Получить дату последней модификации файла

Вход: AX = 5704h; BX = дескриптор файла.

Выход: CF = 0 — успешное выполнение функции (CX = 0000h), DX = биты установлены следующим образом: 0–4 = день месяца в диапазоне 1–31; 5–8 = месяц в диапазоне 1–12; 9–15 = число лет начиная с 1980 года. CF = 1 — AX = код ошибки.

Получение времени последней модификации не реализовано, в чем можно убедиться, вставив в свою программу следующий фрагмент:

```

:....
mov     bx, handle
mov     ax, 5704h
int     21h
jc      exit           : переход, если ошибка
:....

```

Получить дату и время создания файла

Вход: AX = 5706h; BX = дескриптор файла.

Выход: CF = 0 — успешное выполнение функции; CX = биты установлены следующим образом: 0–4 = секунды, деленные на 2; 5–10 = минуты (0–59); 11–15 = часы; DX = биты установлены следующим образом: 0–4 = день месяца в диапазоне 1–31; 5–8 = месяц в диапазоне 1–12; 9–15 = число лет начиная с 1980 года; SI = двоичное значение количества десятизмиллисекундных интервалов, добавляемых ко времени MS DOS в диапазоне 0–199; иначе CF = 1 — AX = код ошибки.

Данная функция реализована в полном объеме.

Установить дату последней модификации файла

Вход: AX = 5705h; BX = дескриптор файла; CX = 0000h; DX = биты установлены следующим образом: 0–4 = день месяца в диапазоне 1–31; 5–8 = месяц в диапазоне 1–12; 9–15 = число лет начиная с 1980 года.

Выход: CF = 0 — успешное выполнение функции; CF = 1 — AX = код ошибки.

Аналогично функции 5704h, данная функция позволяет установить только дату создания файла.

Установить дату и время создания или последней модификации файла

Вход: AX = 5707h; BX = дескриптор файла; CX = биты установлены следующим образом: 0–4 = секунды, деленные на 2; 5–10 = минуты (0–59); 11–15 = часы; DX = биты установлены следующим образом: 0–4 = день месяца в диапазоне 1–31; 5–8 = ме-

сяц в диапазоне 1–12; 9–15 = число лет начиная с 1980 года; SI = двоичное значение количества десятимиллисекундных интервалов, добавляемых к времени MS DOS в диапазоне 0–199.

Выход: CF = 0 — успешное выполнение функции; CF = 1 — AX = код ошибки.

Данная функция реализована в полном объеме.

Кроме дополнительных функций, для работы с различными временными характеристиками файла Windows-версия MS DOS содержит две функции для преобразования форматов времени. Дело в том, что Windows работает со временем в 64-разрядном формате. При этом точкой отсчета является 00 часов 00 минут 1 января 1901 года. Значение времени содержит число стонаносекундных интервалов, прошедших с этой даты. По расчетам разработчиков, этого значения должно хватить на 400 лет. Для того чтобы представить его в виде, воспринимаемом человеком (DOS-время), введена функция 71a7h.

Вход: AX = 71a7h; BL = 0 — преобразовать 64-разрядное время в DOS-время; DS:SI = указатель на экземпляр структуры FILETIME, содержащей 64-битовое значение времени.

Выход: CF = 0 — успешное выполнение функции, при этом регистры устанавливаются следующим образом: BH = число десятимиллисекундных интервалов, добавляемых к времени MS DOS (значение в диапазоне 0–199); CX = время в упакованном формате со значением битов: 0–4 — секунды, деленные на 2; 5–10 — минуты в диапазоне 0–59; 0–4 — часы в диапазоне 0–23; DX = дата в упакованном формате со значением битов: 0–4 — день месяца в диапазоне 1–31; 5–8 — месяц в диапазоне 1–12; 9–15 — число лет начиная с 1980 года (для получения истинного значения прибавьте 1980); иначе CF = 1 — AX = код ошибки.

Структура FILETIME описывается в программе следующим образом:

FILETIME	struc		
DwLowDateTime	dd	?	: младшие 32 бита значения времени
DwHighDateTime	dd	?	: старшие 32 бита значения времени
FILETIME	ends		

Вход: AX = 71a7h; BL = 1 — преобразовать DOS-время в 64-разрядное время; BH = число десятимиллисекундных интервалов, добавляемых ко времени MS DOS (значение в диапазоне 0–199); CX = время в упакованном формате со значением битов: 0–4 — секунды, деленные на 2; 5–10 — минуты в диапазоне 0–59; 0–4 — часы в диапазоне 0–23; DX = дата в упакованном формате со значением битов: 0–4 — день месяца в диапазоне 1–31; 5–8 — месяц в диапазоне 1–12; 9–15 — число лет начиная с 1980 года (для получения истинного значения прибавьте 1980); DS:SI = указатель на экземпляр структуры FILETIME, в которой вернется 64-битовое значение времени.

Выход: CF = 0 — успешное выполнение функции, при этом в области памяти, адресуемой DS:SI, возвращается 64-битовое значение времени; CF = 1 — AX = код ошибки.

Получить атрибуты файла

Вход:

- AX = 7143h;
- BX = действие:

- 0 — получить атрибуты, на выходе CX = атрибуты файла: 0000h — файл доступен по записи и чтению; 0001h — файл доступен по чтению; 0002h — скрытый файл; 0004h — системный файл; 0008h — метка тома; 0010h — каталог; 0020h — архивный файл;
- 2 — получить размер сжатого файла — на выходе DX:AX = размер сжатого файла в байтах на диске;
- 4 — получить дату и время последней записи — на выходе: CX = время в формате: 0–4 = секунды, деленные на 2; 5–10 = минуты (0–59); 11–15 = часы (0–23); DI = дата в формате: 0–4 = день месяца (1–31); 5–8 = месяц (1–12); 9–15 = число лет с 1980 года;
- 6 — получить дату последнего доступа — на выходе: DI = дата (см. BX = 4);
- 8 — получить дату и время создания — на выходе: CX = время в формате, DI = форматированная дата (см. BX = 4), SI = двоичное значение количества десятизмиллисекундных интервалов, добавляемых ко времени MS DOS в диапазоне 0–199;

■ DS:DX — ASCIIZ-строка с именем (путем) файла.

Выход: CF = 0 в случае успеха, информация в регистрах определяется значением BX на входе (см. выше): AX = код ошибки (CF = 1): 1 — неверное значение в AL; 2 — файл не найден; 3 — несуществующий путь; 5 — доступ запрещен.

Установить атрибуты файла

Вход:

■ AX = 7143h;

■ BX = действие:

- 1 — установить атрибуты, на входе: CX = атрибуты файла: 0000h — файл доступен по записи и чтению; 0001h — файл доступен по чтению; 0002h — скрытый файл; 0004h — системный файл; 0020h — архивный файл;
- 3 — установить дату и время последней записи: CX = время в формате: 0–4 = секунды, деленные на 2; 5–10 = минуты (0–59); 11–15 = часы (0–23); DI = дата в формате: 0–4 = день месяца (1–31); 5–8 = месяц (1–12); 9–15 = число лет с 1980 года;
- 5 — установить дату последнего доступа (см. BX = 3);
- 7 — установить дату и время создания: CX = время (см. BX = 3), DI = дата (см. BX = 3), SI = двоичное значение количества десятизмиллисекундных интервалов, добавляемых ко времени MS DOS в диапазоне 0–199;

■ DS:DX — ASCIIZ-строка с именем (путем) файла.

Выход: CF = 0 — CX = слово атрибутов файла; CF = 1 — AX = код ошибки: 1 — неверное значение в AL; 2 — файл не найден; 3 — несуществующий путь; 5 — доступ запрещен.

Переименовать файл

Вход: AH = 7156h; DS:DX — ASCIIZ-имя существующего файла; ES:DI — ASCIIZ-имя нового файла; CL = маска атрибутов.

Выход: CF = 0 — при успешном переименовании; CF = 1 — AX = код ошибки; 2 — файл не найден; 3 — несуществующий путь; 5 — доступ запрещен; 11h — устройства для старого и нового файлов не совпадают.

Работа с дисками, каталогами и организация поиска файлов

Получить информацию о томе

Вход: AH = 71A0h; DS:DX — адрес ASCIIZ-строки с именем корневого каталога диска, о котором необходимо получить информацию (C:\); ES:DI — адрес буфера, в который будет помещена ASCIIZ-строка с именем файловой системы; CX = размер буфера, в который будет помещена ASCIIZ-строка с именем файловой системы.

Выход: CF = 0 — успешное выполнение, при этом в буфер по адресу в ES:DI помещается ASCIIZ-строка с именем файловой системы и устанавливаются следующие регистры: BX = флаги файловой системы (комбинация значений: 0001h — при поиске учитывается регистр букв в именах файлов; 0002h — файловая система сохраняет регистр букв в элементах каталога; 0004h — использование символов Unicode в именах каталогов и файлов; 4000h — файловая система поддерживает длинные имена файлов и функции для работы с ними; 8000h — том сжат); CX = максимально допустимая длина имени файла на данном томе без последнего нулевого символа (до 255); DX = максимально допустимая длина пути для данного тома, включая последний нулевой символ (до 260); иначе CF = 1 — AX = код ошибки.

Создание каталога

Вход: AH = 7139h; DS:DX — адрес строки с ASCIIZ-именем существующего файла.

Выход: CF = 0 — при успешном переименовании; CF = 1 — AX = код ошибки; 3 — несуществующий путь; 5 — доступ запрещен.

Удаление каталога

Удаляемый каталог должен быть пуст.

Вход: AH = 713Ah; DS:DX — ASCIIZ-строка пути к удаляемому каталогу.

Выход: CF = 0 — AX = не определен; CF = 1 — AX = код ошибки; 3 — несуществующий путь; 5 — доступ запрещен; 10h — попытка удаления текущего каталога.

Изменить текущий каталог

Понятие текущего каталога аналогично тому, что приводилось выше при рассмотрении функций для работы с файлами, имеющими короткие имена.

Вход: AH = 713Bh; DS:DX — указатель на буфер, содержащий полный путь от корневого каталога в виде ASCIIZ-строки и в качестве последнего элемента включающий имя нового текущего каталога (естественно, что допустимы длинные имена с ограничениями по максимальной длине (см. функцию 71a0h)).

Выход: CF = 0 — AX не определен; CF = 0 — AX = код ошибки; 03h — путь не найден.

Получение текущего каталога

Вход: AH = 7147h; DL = номер устройства (00h — текущее (заданное по умолчанию), 01h — A: и т. д.); DS:SI — указатель на буфер для записи полного пути от корневого

к текущему каталогу (длина буфера должна быть не менее длины, возвращаемой в регистре DX функцией 71a0h).

Выход: CF = 0 — успешное выполнение функции, в результате чего полный путь от корневого каталога в виде ASCIIZ-строки без имени диска и символа \ записывается в буфер, адрес которого указан в DS:SI; либо AX = код ошибки (CF = 1): 0Fh — недопустимый дисковод.

Среди новых функций, работающих, в том числе, с длинными именами файлов, существует функция 7160h, позволяющая получить полные пути для указанных файлов или относительных путей: получить полный путь (CX = 0), получить полный путь с краткими именами (CX = 1), получить полный путь с длинными именами (CX = 2).

Получить полный путь

Вход: AH = 7160h; CL = 0; CH — содержимое результата (CH = 80h — получить имя диска; CH = 0 — получить полный путь); DS:SI — адрес ASCIIZ-строки с именем файла или каталога, для которых запрашивается полный путь. Допускаются оба типа имен — длинные и короткие; ES:DI — адрес строки, в которую нужно записать полный путь. Размер буфера должен быть достаточным для размещения пути максимальной длины (функция 71a0h).

Выход: CF = 0 — успешное выполнение функции, в результате чего полный путь от корневого каталога в виде ASCIIZ-строки сохраняется в буфере, адрес которого указан в ES:DI; или же CF = 1 — AX = код ошибки.

```

:-----+-----
: | Программа: prg07_24.asm. Применение функции 7160h (CL=0)
: | прерывания 21h для получения полного пути.
:-----+-----
.data
filename      db      'my file with long name.txt'. 0
point_fname   dd      filename
PathFull      db      260 dup (0)
point_Path    dd      PathFull
              ....
.code
              ....
lds          si, point_fname : формируем указатель на имя файла
les          di, point_Path  : формируем указатель на буфер
                                для полного пути
mov          ax, 7160h       : номер функции DOS
mov          ch, 80h         : CH = 80h — получить имя диска
mov          cl, 0           : получить полный путь
int          21h
jc           exit
              ....

```

Данная функция работает очень примитивно — при указании имени файла или относительного пути (с символами «.» и «..») она не проверяет его существование, а лишь добавляет к нему имя текущего диска и каталога. Поэтому при использовании этой функции требуются другие средства, позволяющие контролировать реальное наличие файла или пути на диске.

Получить полный путь с краткими именами (в формате 8.3)

Вход: AH = 7160h; CL = 1; CH — содержимое результата (CH = 80h — получить имя диска; CH = 0 — получить полный путь); DS:SI — адрес ASCIIZ-строки с именем

файла или каталога, для которых запрашивается путь в короткой форме. Допускаются оба типа имен — длинные и короткие; ES:DI — адрес строки, в которую требуется записать полный путь. Размер буфера должен быть достаточным для размещения пути максимальной длины (функция 71a0h).

Выход: CF = 0 — успешное выполнение функции, в результате чего полный путь от корневого каталога в виде ASCIIZ-строки оказывается в буфере, адрес которого указан в ES:DI; CF = 1 — AX = код ошибки.

```

+-----+
:| Программа: prg07_25.asm. Применение функции 7160h (CL=1) прерывания 21h |
:| для получения полного пути с краткими именами (в формате 8.3). |
+-----+
.data
filename      db      'my_file with long name.txt'. 0
point_fname   dd      filename
PathFull      db      260 dup (0)
point_Path    dd      PathFull
:....

.code
:....
lds      si, point_fname : формируем указатель на имя файла
les      di, point_Path  : формируем указатель на буфер
                        : для полного пути
mov      ax, 7160h        : номер функции DOS
mov      ch, 80h          : CH = 80h — получить имя диска
mov      cl, 1            : полный путь с краткими именами
int      21h
jc       exit
:....

```

На выходе функция формирует строку, содержащую полный путь, причем все длинные компоненты этого пути заменяются их краткими псевдонимами, удовлетворяющими схеме «8.3». Данный вариант функции (при CL = 1), в отличие от ее предыдущего варианта, производит проверку наличия файла или пути.

Получить полный путь с длинными именами

Вход: AH = 7160h; CL = 2; CH — содержимое результата (CH = 80h — получить имя диска; CH = 0 — получить полный путь); DS:SI — адрес ASCIIZ-строки с именем файла или каталога, для которых запрашивается путь в длинной форме. Допускаются оба типа имен — длинные и короткие; ES:DI — адрес строки, в которую следует записать полный путь. Размер буфера должен быть достаточным для помещения пути максимальной длины (функция 71a0h).

Выход: CF = 0 — успешное выполнение функции, в результате чего полный путь от корневого каталога в виде ASCIIZ-строки попадает в буфер, адрес которого указан в ES:DI; CF = 1 — AX = код ошибки.

Получить информацию о файле по описателю

Вход: AH = 71A6h; флаг CF = 1; BX = дескриптор файла; DS:DX — адрес структуры BY_HANDLE_FILE_INFORMATION.

Выход: CF = 0 — успешное выполнение функции; CF = 1 — AX = код ошибки.

Формат структуры BY_HANDLE_FILE_INFORMATION приведен ниже.

```

BY_HANDLE_FILE_INFORMATION struc
dwFileAttributes      dd      ?
ftCreationTime         dd      2 dup(?)

```

```

ftLastAccessTime    dd    2 dup(?)
ftLastWriteTime     dd    2 dup(?)
dwVolumeSerialNumber dd    ?
nfileSizeHigh       dd    ?
nfileSizeLow        dd    ?
nnumberOfLinks      dd    ?
nfileIndexHigh      dd    ?
nfileIndexLow       dd    ?
BY_HANDLE_FILE_INFORMATION ends

```

Поля этой структуры описаны в следующей таблице.

Поле	Описание
dwFileAttributes	Атрибуты файла. Этот элемент может быть комбинацией следующих значений: ■ FILE_ATTRIBUTE_NORMAL (00000000h) — файл доступен по чтению и записи; этот атрибут нельзя комбинировать с другими; ■ FILE_ATTRIBUTE_READONLY (00000001h) — файл только для чтения; ■ FILE_ATTRIBUTE_HIDDEN (00000002h) — скрытый файл; ■ FILE_ATTRIBUTE_SYSTEM (00000004h) — системный файл; ■ FILE_ATTRIBUTE_DIRECTORY (00000010h) — каталог; ■ FILE_ATTRIBUTE_ARCHIVE (00000020h) — архивный файл
FtCreationTime	Время создания файла в 64-разрядном формате
ftLastAccessTime	Время последнего доступа к файлу в 64-разрядном формате
ftLastWriteTime	Время последней записи в файл в 64-разрядном формате
dwVolumeSerialNumber	Серийный номер тома, на котором находится файл
NFileSizeHigh	Старшее слово значения, определяющего размер файла
NFileSizeLow	Младшее слово значения, определяющего размер файла
NNumberOfLinks	Число связей с данным файлом. В файловых системах FAT и HPFS этот элемент всегда равен 1. В файловой системе NTFS число связей может превышать 1
NFileIndexHigh	Старшее слово уникального дескриптора, связанного с файлом
NFileIndexLow	Младшее слово уникального дескриптора, связанного с файлом. Файл однозначно определяется дескриптором и серийным номером тома

```

+-----+
| Программа: prg07_26.asm. Применение функции 71A6h прерывания 21h |
| для получения информации о файле по описателю.                  |
+-----+
BY_HANDLE_FILE_INFORMATION struc
DwFileAttributes    dd    ?
FtCreationTime      dd    2 dup(?)
FtLastAccessTime    dd    2 dup(?)

```

```

FtLastWriteTime      dd    2 dup(?)
DwVolumeSerialNumber dd    ?
NfileSizeHigh        dd    ?
NfileSizeLow         dd    ?
NnumberOfLinks       dd    ?
NfileIndexHigh       dd    ?
NfileIndexLow        dd    ?
BY_HANDLE_FILE_INFORMATION ends
.data
file_info_BY_HANDLE_FILE_INFORMATION <>
point_find_          dd    file_info
filename              db    'my_file with long name.txt'. 0 ; файл, о котором
                                                              : будем получать информацию

point_fname          dd    filename
handle               dw    0
                    :....

.code
                    :....
                    :----- открываем файл
mov     bx, 0100h+0400h ; не использовать буферизацию +
                    : содержимое DI в псевдоним
mov     dx, 1           ; открыть файл, если он существует,
                    : иначе вернуть ошибку
lds     si, point_fname ; формируем указатель на имя файла
mov     di, 7           ; добавить в конец псевдонима символ 7
repeat: mov     ax, 716ch ; номер функции DDS
xor     cx, cx          ; атрибуты файла — обычный файл —
                    : доступ для чтения-записи

int     21h
jnc     ml              ; если файл существовал, то переход
mov     dx, 10h         ; создать файл
jmp     repeat          ; переход — повторим открытие файла
ml:     ;----- действия при успешном открытии файла
mov     handle, ax      ; сохраним дескриптор файла
                    :....
                    :----- получаем информацию о файле
mov     bx, handle
stc     ; это обязательно
lds     dx, point_find_ ; формируем указатель на структуру
                    : BY_HANDLE_FILE_INFORMATION

mov     ax, 71a6h
int     21h
;----- обрабатываем полученную информацию
                    :....

```

Создать псевдоним

Функция 71A8h предназначена для генерации короткого (в формате 8.3) имени для заданного файла с длинным именем.

Вход: AH = 71A8h; DS:SI — адрес строки (с нулевым символом в конце), содержащей длинное имя нужного файла без указания пути; ES:DI — адрес буфера, в котором возвращается псевдоним; DH — формат псевдонима (0 — 11-символьное имя элемента каталога; 1 — имя файла в формате 8.3); DL — набор символов для длинного имени и псевдонима. Это значение — упакованная величина в формате: биты 0–3 — набор символов в исходном имени файла (0 — Windows ANSI; 1 — OEM; 2 — Unicode); биты 4–7 — набор символов в создаваемом коротком имени (0 — Windows ANSI; 1 — OEM; 2 — Unicode).

Выход: CF = 0 — успешное выполнение функции; CF = 1 — AX = код ошибки.


```

+-----+
: | Программа: prg07_27.asm. Применение функции 71A8h прерывания 21h |
: | для создания псевдонима. |
+-----+
.data
filename_long db 'my_file with long name.txt', 0
point_fname_long dd filename_long
filename_short db 11 dup (20h)
point_fname_short dd filename_short
handle dw 0
:....

.code
:....
:----- открываем файл
mov bx, 0100h+0400h ; не использовать буферизацию +
; содержимое DI в псевдоним
mov dx, 1 ; открыть файл, если он существует,
; иначе вернуть ошибку
lds si, point_fname_long ; формируем указатель на имя файла
mov di, 7 ; добавить в конец псевдонима символ 7
repeat: mov ax, 716ch ; номер функции DOS
xor cx, cx ; атрибуты файла — обычный файл —
; доступ для чтения-записи

int 21h
jnc m1 ; если файл существовал, то переход
mov dx, 10h ; создать файл
jmp repeat ; переход — повторим открытие файла
:----- действия при успешном открытии файла
m1: mov handle, ax ; сохраним дескриптор файла
:----- создадим псевдоним
lds si, point_fname_long
les di, point_fname_short
mov dh, 1
mov dl, 0
mov ax, 71a8h
int 21h
:....

```

Действие данной функции несколько отличается от процесса формирования псевдонима файла операционной системой и заключается в том, что длинное имя попросту обрезается по границам «8.3». В этом несложно убедиться, проанализировав работу приведенной выше программы в отладчике.

Поиск файлов и каталогов

В Windows-версии MS DOS процесс поиска немного отличается от рассмотренного выше. Для этого используются три функции и структура **WIN32_FIND_DATA** в памяти, в которой возвращается информация о файле. Для запуска процесса поиска вызывается функция **714eh** — найти первый файл.

Вход: **AH = 714eh**; **CL** — атрибуты искомых файлов (0000h — файл доступен по записи и чтению; 0001h — файл доступен по чтению; 0002h — скрытый файл; 0004h — системный файл; 0008h — метка тома; 0010h — каталог; 0020h — архивный файл); **CH** — дополнительные атрибуты искомых файлов (0000h — файл доступен по записи и чтению; 0001h — файл доступен по чтению; 0002h — скрытый файл; 0004h — системный файл; 0008h — метка тома; 0010h — каталог; 0020h — архивный файл); **DS:DX** — адрес ASCII-строки с именем искомого файла или каталога. Допускаются оба типа имен — длинные и короткие. В именах допустимы символы шаблона * и ?; **ES:DI** — адрес структуры **WIN32_FIND_DATA**, в которой будет возвращена инфор-

мация о файле; SI — формат, в котором возвращается дата и время (0 — дата и время в 64-разрядном формате; 1 — дата и время в формате MS DOS).

Выход: CF = 0 — успешное выполнение функции, в результате в регистрах AX и CX оказывается следующая информация: AX = дескриптор, использующийся далее для процесса поиска; CX = возможные значения: 0000h — все символы структуры WIN32_FIND_DATA, составляющие основное и альтернативное имя файла, успешно преобразованы из Unicode; 0001h — основное имя, возвращенное в структуре WIN32_FIND_DATA, содержит знаки подчеркивания на месте символов, не преобразованных из Unicode; 0002h — альтернативное имя, возвращенное в структуре WIN32_FIND_DATA, со знаками подчеркивания на месте символов, не преобразованных из Unicode; или CF = 1 — AX = код ошибки при неудачном выполнении функции.

Вызов функции 714eh приводит к заполнению полей структуры WIN32_FIND_DATA, после чего можно проанализировать ее поля. Основным интерес представляют поля основного (CfileName) и альтернативного (CaltenateFileName) имен. Их можно анализировать на предмет удовлетворения условиям поиска. Если необходимо продолжить поиск, вызывается функция 714fh — найти следующий файл. Если же поиск считается удачным либо его необходимо прекратить, то вызывается функция 71a1h — прекратить поиск. Ниже описаны порядок вызова функций 714fh и 71a1h и формат структуры WIN32_FIND_DATA.

Вход: AH = 714fh; BX = дескриптор, полученный функцией 714eh; ES:DI — адрес структуры WIN32_FIND_DATA, в которой будет возвращена информация о файле; SI — формат, в котором возвращается дата и время (0 — дата и время в 64-разрядном формате; 1 — дата и время в формате MS DOS).

Выход: CF = 0 — успешное выполнение функции, в результате в регистры AX и CX заносится следующая информация: CX = возможные значения: 0000h — все символы структуры WIN32_FIND_DATA, составляющие основное и альтернативное имя файла, успешно преобразованы из Unicode; 0001h — основное имя, возвращенное в структуре WIN32_FIND_DATA, содержит знаки подчеркивания на месте символов, не преобразованных из Unicode; 0002h — альтернативное имя, возвращенное в структуре WIN32_FIND_DATA, со знаками подчеркивания на месте символов, не преобразованных из Unicode; иначе CF = 1 — AX = код ошибки при неудачном выполнении функции.

Функция 714eh, в отличие от аналогичных функций «старой» MS DOS, использует не область DTA, а некоторый блок в памяти. Этот блок важно своевременно освобождать, для этого и предназначена функция 71a1h.

Вход: AH = 71a1h; BX = дескриптор, полученный функцией 714eh.

Выход: CF = 0 — успешное выполнение функции; CF = 1 — AX = код ошибки при неудачном выполнении функции.

Ниже приведена структура WIN32_FIND_DATA, в которую в процессе поиска записывается информация о файлах.

```
WIN32_FIND_DATA struc
dwFileAttributes    dd ?
ftCreationTime       dd 2 dup(?)
ftLastAccessTime     dd 2 dup(?)
ftLastWriteTime      dd 2 dup(?)
nfileSizeHigh        dd ?
```

: размер файла в байтах (старшее слово)

```

NfileSizeLow      dd  ?           : размер файла в байтах (младшее слово)
dwReserved0       dd  0           : резерв
dwReserved1       dd  0           : резерв
cFileName         db  MAX_PATH dup(?)
cAlternateFileName db  14 dup(?)
WIN32_FIND_DATA ends

```

Поля этой структуры описаны в следующей таблице.

Поле	Описание
dwFileAttributes	Атрибуты найденного файла (см. описание аналогичного элемента структуры BY_HANDLE_FILE_INFORMATION)
ftCreationTime	Время создания файла в одном из двух форматов: MS DOS или 64-разрядном, в зависимости от параметров, указанных при вызове функций 714eh (найти первый файл) и 714fh (найти следующий файл)
ftLastAccessTime	Время последнего доступа к файлу в одном из двух форматов: MS DOS или 64-разрядном, в зависимости от параметров, указанных при вызове функций 714eh и 714fh
ftLastWriteTime	Время последней записи в файл в одном из двух форматов: MS DOS или 64-разрядном, в зависимости от параметров, указанных при вызове функций 714eh и 714fh
cFileName	ASCIIZ-строка, содержащая имя файла. Размер строки должен быть не менее 256 символов
cAlternateFileName	ASCIIZ-строка, содержащая альтернативное имя файла в стандартном формате 8.3. Если элемент cFileName содержит имя в формате 8.3 или файловая система не поддерживает альтернативные имена в формате 8.3, то элемент cAlternateFileName равен нулю

Рассмотрим пример поиска файла по шаблону. Для этого предварительно создадим несколько файлов в соответствии с шаблоном file_*.*. Среди этих файлов должен быть файл file_05.txt. В отладчике проследим за тем, как изменяется содержимое области памяти, отведенное для экземпляра структуры WIN32_FIND_DATA. Выход из программы — при обнаружении файла file_5.txt.

```

:-----+
:| Программа: prg07_28.asm. Поиск файла по шаблону. |
:-----+
WIN32_FIND_DATA struc
dwFileAttributes dd  ?
ftCreationTime   dd  2 dup(?)
ftLastAccessTime dd  2 dup(?)
ftLastWriteTime  dd  2 dup(?)
NfileSizeHigh    dd  ?           : размер файла в байтах (старшее слово)
NfileSizeLow     dd  ?           : размер файла в байтах (младшее слово)
dwReserved0      dd  0           : резерв
dwReserved1      dd  0           : резерв
cFileName        db  MAX_PATH dup(?)
cAlternateFileName db  14 dup(?)
WIN32_FIND_DATA ends
.data
find WIN32_FIND_DATA <>
point_find_ dd find_

```

```

f_name_pattern db 'file *.*'.0
point_f_name_pattern dd f_name_pattern
filename db 'file 05.txt'.0 ; искомый файл
len_filename = $ - filename
handle dw 0
;....

.code
;....
mov cl, 0 ; атрибуты искомого файла
mov ch, 0 ; дополнительные атрибуты для поиска
lds dx, point_f_name_pattern ; формируем указатель
; на строку с шаблоном
les di, point_find_ ; формируем указатель на экземпляр
; структуры WIN32_FIND_DATA
mov ax, 714eh ; номер функции DOS
int 21h
jc exit
;----- в ax был возвращен дескриптор — если нужно, то сохраним
mov handle, ax
;----- проверяем, тот ли это файл
ml: mov cx, len_filename
lea di, find_.CfileName
lea si, filename
repe cmpsb
jz exit
;----- продолжаем поиск — в bx дескриптор, полученный от 714eh
mov bx, handle
les di, point_find_ ; формируем указатель на экземпляр
; структуры WIN32_FIND_DATA
mov ax, 714fh ; номер функции DOS
xor si, si ; формат даты
int 21h
jnc ml
;----- завершить поиск
exit: mov ax, 7141h
mov bx, handle
int 21h
;....

```

В отладчике хорошо видно, что выход из данной программы происходит в двух случаях:

- когда файл найден, выход из программы производится в результате сравнения командой **CMP SB** (флаг **ZF** устанавливается в 1);
- когда файлов, удовлетворяющих шаблону, нет, функция поиска 714eh или 714fh завершается неудачей (флаг **CF** устанавливается в 1).

В качестве шаблона можно задать символы ***.***, тогда мы сумеем получить имена и проанализировать все файлы в текущем каталоге. Это бывает полезным при программировании операции перемещения или копирования каталога.

Остальные функции работы с файлами, предназначенные для записи/чтения/позиционирования, остались прежними.

Файловый ввод-вывод в Win32

В этом разделе будут представлены минимальные сведения, необходимые для выполнения простых операций с файлами. В отличие от MS DOS, среда Win32 способна поддерживать несколько файловых систем. Главные требования к этим

системам — иерархичность и соблюдение определенных правил присвоения имен каталогам и файлам.

Перечислим функции API Win32, имеющие отношение к работе с файловой системой. Полное их описание можно найти в библиотеке MSDN.

Функция	Назначение
AreFileApisANSI	Определение набора символов файла — ANSI или OEM
CancelIo	Отменить все ждущие обработки операции ввода и вывода (I/O)
CloseHandle	Закрыть открытый дескриптор файла
CopyFile, CopyFileEx	Копирование существующего файла в новый файл
CopyProgressRoutine	Определенная приложением функция обратного вызова, используемая с функциями CopyFileEx и MoveFileWithProgress. Она вызывается, когда завершается часть операции копирования или пересылки
CreateDirectory, CreateDirectoryEx	Создать каталог
CreateFile	Создать файл или объект специального типа
DefineDosDevice	Определить, переопределить или удалить имена устройства MS DOS
DeleteFile	Удалить файл
FindClose	Закрыть указанный поисковый дескриптор (см. функции FindFirstFile и FindNextFile)
FindCloseChangeNotification	Закрыть объект-уведомление об изменении файла
FindFirstChangeNotification	Создать объект-уведомление об изменении файла
FindFirstFile, FindFirstFileEx, FindNextFile	Поиск файлов
FindNextChangeNotification	Сброс объекта-уведомления в занятое состояние
FlushFileBuffers	Очистка буфера для указанного файла и запись всех буферизированных данных в файл
GetBinaryType	Определить, является ли файл исполняемым, и если это так, то для какой подсистемы — Win32, MS DOS, OS/2, UNIX (POSIX) и т. д.
GetCurrentDirectory	Получить текущий каталог
GetDiskFreeSpace, GetDiskFreeSpaceEx	Информация относительно указанного диска, включая количество свободного пространства на нем
GetDriveType	Определить тип диска — съемный, фиксированный, CD-ROM, электронный или сетевой
GetFileAttributes, GetFileAttributesEx	Получить атрибуты файла или каталога
GetFileInformationByHandle	Найти информацию относительно указанного файла

Функция	Назначение
GetFileSize, GetFileSizeEx	Получить размер указанного файла
GetFileType	Получить тип указанного файла
GetFullPathName	Получить полный путь и имя для указанного файла
GetLogicalDrives, GetLogicalDriveStrings	Определить доступные в настоящее время дисководы
GetLongPathName	Преобразовать указанный путь к его длинной форме
GetShortPathName	Получить псевдоним файла
GetTempFileName	Создать имя для временного файла
GetTempPath	Получить путь каталога для временных файлов
LockFile, LockFileEx	Блокировка файла
MoveFile, MoveFileEx, MoveFileWithProgress	Переименование файла
QueryDosDevice	Получить информацию об именах устройства MS DOS
ReadDirectoryChangesW	Получить информацию об изменениях в пределах каталога
ReadFile, ReadFileEx	Чтение файла с текущей позиции
ReadFileScatter	Чтение данных из файла с сохранением их в наборе буферов
RemoveDirectory	Удалить существующий пустой каталог
SearchPath	Поиск указанного файла
SetCurrentDirectory	Установить текущий каталог
SetEndOfFile	Поместить символ конца файла (EOF) в текущую позицию указателя
SetFileApisToANSI	Использовать кодовую страницу набора символов ANSI при работе функций файлового ввода-вывода
SetFileApisToOEM	Использовать кодовую страницу набора символов OEM при работе функций файлового ввода-вывода
SetFileAttributes	Установить атрибуты файла
SetFilePointer, SetFilePointerEx	Переместить указатель в определенную позицию файла
SetVolumeLabel	Задать метку тома
UnlockFile, UnlockFileEx	Снять блокировку файла
WriteFile, WriteFileEx	Запись в файл
WriteFileGather	Запись данных в файл из набора буферов

Далее на примерах конкретных программ разберемся с тем, как использовать в программах на ассемблере наиболее интересные и часто применяемые функции из перечисленных выше для работы с файлами API Win32. В целях экономии места все примеры реализованы в виде консольных приложений. Основное внима-

ние уделено не полноте описания параметров для вызова той или иной функции и результатов ее работы (эту информацию можно найти в справочниках по функциям API), а деталям практической реализации файловых операций в программах на языке ассемблера. Для изучения подробностей работы функций API Win32 нам понадобится какой-либо отладчик для Windows, например TD32.EXE.

Обработка ошибок

Прежде чем рассматривать функции API Win32, относящиеся к файловому вводу-выводу, отметим, как можно выяснить причину их ошибочного завершения. Для этого Windows предоставляет функцию `GetLastError`:

```
DWORD GetLastError(void);
```

Внутри функции `GetLastError` не нужно передавать никаких параметров. Эту функцию следует вызывать сразу после вызова API Win32, успешность работы которого мы проверяем.

```
:...
push    offset info
push    hFile
call    GetFileInformationByHandle
call    GetLastError    ; в регистре EAX — код ошибки
:...
```

В регистре **EAX** возвращается код ошибки. Расшифровать его можно с помощью файла `Winerror.h`, где вместе с кодами ошибок приведены короткие сообщения о причине их возникновения.

Создание, открытие, закрытие и удаление файла

Создание и открытие файла в Win32 производится одной функцией `CreateFile`.

```
HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess,
    DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);
```

Параметры данной функции имеют размер двойного слова (4 байта). Их назначение следующее (параметры описаны в порядке, обратном их заталкиванию в стек):

- **lpFileName** — указатель на ASCIIZ-строку с именем (путем) открываемого или создаваемого файла;
- **dwDesiredAccess** — тип доступа к файлу:
 - **GENERIC_READ** = 80000000h — доступ по чтению;
 - **GENERIC_WRITE** = 40000000h — доступ по записи;
 - **GENERIC_READ + GENERIC_WRITE** = 0C0000000h — доступ по чтению-записи;
- **dwShareMode** — режим разделения файлов между разными процессами, данный параметр может принимать значения:
 - **0** — монополизация доступа к файлу;
 - **FILE_SHARE_READ** = 00000001h — другие процессы могут открыть файл, но только по чтению, запись в файл монополизирована процессом, открывшим файл;
 - **FILE_SHARE_WRITE** = 00000002h — другие процессы могут открыть файл, но только по записи, чтение в файл монополизировано процессом, открывшим файл;

- `FILE_SHARE_READ + FILE_SHARE_WRITE = 00000003h` — другие процессы могут открывать файл по чтению-записи;
- `lpSecurityAttributes` — указатель на структуру `Security_Attributes` (файл `winbase.h`), определяющую защиту связанного с файлом объекта ядра, при отсутствии защиты заносится `NULL`;
- `dwCreationDistribution` — определяет действия для случаев, когда файл существует или не существует (аналог этого параметра используется при вызове описанных выше функций `MS DOS 6ch` и `716ch`), данный параметр принимает значения:
 - `CREATE_NEW = 1` — создать новый файл, если файл не существует; если файл существует, то функция завершается формированием ошибки;
 - `CREATE_ALWAYS = 2` — создать новый файл, если файл не существует; если он существует, то заместить новым;
 - `OPEN_EXISTING = 3` — открыть файл, если он существует; если файл не существует, формируется ошибка;
 - `OPEN_ALWAYS = 4` — открыть файл при его существовании и создать его, если файла нет;
 - `TRUNCATE_EXISTING = 5` — открыть файл с усечением его до нулевой длины; если файл не существует, то вырабатывается ошибка;
- `dwFlagsAndAttributes` — флаги и атрибуты; этот параметр используется для задания характеристик создаваемого файла:
 - `FILE_ATTRIBUTE_READONLY = 00000001h` — файл только для чтения;
 - `FILE_ATTRIBUTE_HIDDEN = 00000002h` — скрытый файл;
 - `FILE_ATTRIBUTE_SYSTEM = 00000004h` — системный файл;
 - `FILE_ATTRIBUTE_DIRECTORY = 00000010h` — каталог;
 - `FILE_ATTRIBUTE_ARCHIVE = 00000020h` — архивный файл;
 - `FILE_ATTRIBUTE_NORMAL = 00000080h` — обычный файл для чтения-записи (этот атрибут нельзя комбинировать с другими);
 - `FILE_ATTRIBUTE_TEMPORARY = 00000100h` — создается временный файл (преимущество которого в том, что система стремится не записывать этот файл на диск, а работает с ним в памяти; данный атрибут выгодно комбинировать с флагом `FILE_FLAG_DELETE_ON_CLOSE`, тогда после закрытия файла в программе он будет удален, не оставив следов на диске, иначе, как и в `MS DOS`, программе придется «подчищать» за собой содержимое диска);
 - `FILE_FLAG_WRITE_THROUGH = 80000000h` — не использовать промежуточное кэширование при записи на диск, а все изменения сбрасывать прямо на диск;
 - `FILE_FLAG_NO_BUFFERING = 20000000h` — не использовать средства буферизации операционной системы;
 - `FILE_FLAG_RANDOM_ACCESS = 10000000h` — прямой доступ к файлу (установка этого флага или флага `FILE_FLAG_SEQUENTIAL_SCAN` позволяет оптимизировать системой процесс кэширования);

- `FILE_FLAG_SEQUENTIAL_SCAN = 08000000h` — последовательный доступ к файлу;
- `FILE_FLAG_DELETE_ON_CLOSE = 04000000h` — удалить файл после его закрытия (см. описание атрибута `FILE_ATTRIBUTE_TEMPORARY`);
- `FILE_FLAG_OVERLAPPED = 40000000h` — асинхронный доступ к файлу (синхронность означает то, что программа, вызвавшая функцию для доступа к файлу, приостанавливается до тех пор, пока не закончит работу функция ввода-вывода);
- `hTemplateFile` — параметр используется только при создании нового файла, его значением является дескриптор другого существующего и предварительно открытого файла, а новый файл создается с теми же значениями атрибутов и флагов, что и у файла, дескриптор которого указан в параметре `hTemplateFile`.

При успешном завершении функция возвращает в регистре `EAX` дескриптор нового файла. В случае неудачи функция заносит в регистр `EAX` значение `NULL`.

Закрытие файла

Закрытие файла производится функцией `CloseHandle`:

```
BOOL CloseHandle( HANDLE hObject );
```

Функция имеет один параметр размером в двойное слово — дескриптор, полученный при открытии файла функцией `CreateFile`.

При удачном завершении функция возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

Win32 поддерживает несколько функций для часто используемых операций над файлами: копирование, перемещение и переименование файлов.

Копирование файла

Для копирования файлов в Win32 используется функция `CopyFile`:

```
BOOL CopyFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, BOOL bFailIfExists);
```

Параметрами этой функции являются:

- `lpExistingFileName` — указатель на ASCIIZ-строку с именем файла-источника;
- `lpNewFileName` — указатель на ASCIIZ-строку с именем файла-приемника, который может и не существовать;
- `bFailIfExists` — параметр, задаваемый равным 0 или 1, в зависимости от условий копирования:
 - 0 — при наличии файла он удаляется и создается новый с содержимым файла-источника;
 - 1 — при наличии файла копирование не производится, а функция `CopyFile` возвращает ошибку.

При удачном завершении функция возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

```

+-----+
: | Программа: prg07_29.asm. Win32-программа консольного приложения для      |
: | исследования работы функции CopyFile API Win32.                          |
+-----+
.data
TitleText      db      'Копирование файлов в Win32'. 0

```

```

s_file      db      "p". 0      : имя входного файла
d_file      db      "pl". 0     : имя выходного файла
.code

:....
mov     eax, 1
push    eax
push    offset d_file
push    offset s_file
call    CopyFileA
cmp     eax, 0
jz      exit      : выход в случае неудачи
:....

```

Перемещение файла

Для перемещения файла Win32 предусматривает две функции — **MoveFile** и **MoveFileEx**:

BOOL moveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);

BOOL moveFileEx(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, DWORD dwFlags);

Параметрами функции **MoveFile** являются указатели на ASCII-строки с именами файла-источника и файла назначения. Функция **MoveFileEx** обладает дополнительными свойствами благодаря наличию третьего параметра, который определяет особенности перемещения:

- **MOVEFILE_REPLACE_EXISTING** = 00000001h — при существовании целевого файла он замещается содержимым файла-источника;
- **MOVEFILE_COPY_ALLOWED** = 00000002h — если не указывать специально, то функция **MoveFileEx** не перемещает файлы на другой диск, а если перемещение требуется, необходимо устанавливать этот флаг;
- **MOVEFILE_DELAY_UNTIL_REBOOT** = 00000004h (только для Windows NT и выше) — файл-источник не удаляется до перезагрузки системы;
- **MOVEFILE_WRITE_THROUGH** = 00000008h — установка флага гарантирует, что возврат из функции не произойдет до фактического перемещения и удаления файла.

Кроме этого, функция **MoveFileEx** допускает указание на месте второго параметра значения **NULL**, тем самым моделируя вызов функции **DeleteFile**.

При успешном завершении функции **MoveFile** и **MoveFileEx** возвращают ненулевое значение в регистре **EAX**. В случае неудачи функции помещают в регистр **EAX** значение **NULL**.

```

:-----+-----+
:| Программа: prg07_30.asm. Win32-программа консольного приложения для      |
:| исследования работы функции moveFile(Ex) API Win32.                      |
:-----+-----+
.data
TitleText      db      'Перемещение файлов в Win32'. 0
s_file         db      "p". 0      : имя входного файла
d_file         db      "pl". 0     : имя выходного файла
.code

:....
push    offset d_file
push    offset s_file
call    moveFileA
cmp     eax, 0
jz      exit      : выход в случае неудачи
:....

```

Переименование файла

Специальной функции для переименования файла нет, так как она и не нужна — перемещение файла в пределах одного каталога по сути и является его переименованием.

Удаление файла

Для удаления файла применяется функция `DeleteFile`:

```
BOOL DeleteFile(LPCTSTR lpFileName);
```

У нее единственный параметр — указатель на ASCII-строку с именем (путем) удаляемого файла. Перед удалением файл необходимо закрыть, хотя в некоторых версиях Windows это не является обязательным.

При успешном завершении функция возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция заносит в регистр `EAX` значение `NULL`.

Чтение, запись, позиционирование в файле

Необходимо сразу отметить, что Win32 допускает два режима доступа к файлу — синхронный и асинхронный. Необходимость введения этих двух режимов в архитектуру Win32 обусловлена тем, что файловый ввод-вывод относится к наиболее медленным операциям и способен значительно ухудшить впечатление от компьютера с хорошей центральной частью (процессором и материнской платой) и плохой дисковой подсистемой. Поэтому разработчики Win32 уделяют много внимания моделированию файловых операций с использованием памяти компьютера. Это и упомянутые выше временные файлы и гибкая система кэширования вводимых и выводимых данных, а также файлы, отображаемые в память, работу с которыми мы рассмотрим в конце данного раздела. Наше дальнейшее изложение будет посвящено организации синхронного ввода-вывода. Для обычных несложных приложений Win32 его вполне достаточно.

Установка текущей файловой позиции

Доступ к содержимому файла может быть произвольным (прямым) и последовательным. Как обычно, функции ввода-вывода работают с файловым указателем. Но необходимо иметь в виду, что файловый указатель связан только с дескриптором файла. Его значение равно текущему номеру позиции в файле, с которой будет производиться чтение-запись данных при очередном вызове функции ввода-вывода. В первый момент после открытия значение указателя равно 0, то есть он установлен на начало файла. Функции, производящие чтение-запись в файле, меняют значение указателя на количество прочитанных или записанных байтов. При необходимости — а при организации прямого доступа к файлу без этого не обойтись — значение файлового указателя можно изменять с помощью функции `SetFilePointer`:

```
DWORD SetFilePointer( HANDLE hFile, LONG lDistanceToMove, PLONG lpDistanceToMoveHigh,
    DWORD dwMoveMethod );
```

Параметры этой функции имеют размер двойного слова и следующее назначение:

- `hFile` — дескриптор файла, в котором производится позиционирование указателя позиции (предполагается, что этот дескриптор был заранее получен функцией `CreateFile`);

- **lDistanceToMove** — расстояние в байтах, на которое необходимо переместить указатель; это число может трактоваться как знаковое и беззнаковое (см. **dwMoveMethod**) — его отрицательное значение соответствует случаю, когда указатель требуется перемещать к началу файла;
- **lpDistanceToMoveHigh** — если 32-битового значения, позволяющего определить знаковую величину 2 Гбайт, в **lDistanceToMove** недостаточно, то для создания 64-битового знакового значения указателя позиции используют поле **lpDistanceToMoveHigh**, представляющее собой указатель на 32-битовую переменную, значение которой будет являться значением старших 32 битов 64-разрядного указателя позиции;
- **dwMoveMethod** — параметр определяет то, каким образом правильно трактовать значения в паре, определяемой значениями **lpDistanceToMoveHigh** и **lDistanceToMove**:
 - **FILE_BEGIN** = 0 — указатель позиции — значение без знака, заданное содержимым полей **lpDistanceToMoveHigh** и **lDistanceToMove**;
 - **FILE_CURRENT** = 1 — текущее значение указателя позиции складывается со знаковым значением, заданным содержимым полей **lpDistanceToMoveHigh** и **lDistanceToMove**;
 - **FILE_END** = 2 — в этом случае значение, определяемое содержимым полей **lpDistanceToMoveHigh** и **lDistanceToMove**, должно представлять собой отрицательное значение, и смещение в файле вычисляется как сумма, в которой первое слагаемое определяется **lDistanceToMove** или **lpDistanceToMoveHigh** и **lDistanceToMove**, а второе является размером файла.

Функция **SetFilePointer** возвращает значение новой позиции указателя. Но при этом необходимо учитывать значение **lpDistanceToMoveHigh** на входе: если оно нулевое, то **SetFilePointer** завершилась успешно; если **lpDistanceToMoveHigh** было ненулевым, то считается, что функция возвращает в регистре **EAX** младшие 32 бита нового значения указателя позиции, а его старшие 32 бита необходимо извлечь по адресу, заданному параметром **lpDistanceToMoveHigh**. В случае ошибки и при условии **lpDistanceToMoveHigh** = 0 функция **SetFilePointer** возвращает **EAX** = 0xffffffffh.

Если вы внимательно изучали материал раздела, посвященный функциям **MS DOS**, то наверняка заметили много общего, за исключением, пожалуй, одной детали. Функция позиционирования **MS DOS 42h** позволяла устанавливать указатель позиции за концом файла. Последующая операция записи в файл приводила к его расширению. В **Win32** для расширения (и, наоборот, уменьшения) файла существует отдельная функция **SetEndOfFile**:

```
BOOL SetEndOfFile( HANDLE hFile );
```

Алгоритм ее применения следующий.

1. С помощью функции **SetFilePointer** позиционировать указатель на необходимую длину файла.
2. Вызвать функцию **SetEndOfFile**, передав ей дескриптор файла, размер которого изменяется.
3. Закрыть файл функцией **CloseHandle**.

При успешном завершении функция `SetEndOfFile` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция заносит в регистр `EAX` значение `NULL`.

Чтение и запись в файл

Чтение и запись в файл производятся двумя функциями, `ReadFile` и `WriteFile`:

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead, LPDWORD lpOverlapped );
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
              LPDWORD lpNumberOfBytesWritten, LPDWORD lpOverlapped );
```

Параметры этих функций означают следующее:

- `hFile` — дескриптор файла, с которым производится операция чтения-записи;
- `lpBuffer` — указатель на буфер, где содержатся данные для операции чтения-записи;
- `nNumberOfBytesToRead` и `nNumberOfBytesToWrite` — число данных для чтения-записи;
- `lpNumberOfBytesRead` и `lpNumberOfBytesWritten` — указатель на буфер, в который записывается число действительно считанных или записанных байтов;
- `lpOverlapped` — указатель на структуру `OVERLAPPED`, используемую в процессе асинхронного ввода вывода (для синхронного режима — `NULL`).

При успешном завершении функции `ReadFile` и `WriteFile` возвращают ненулевое значение в регистре `EAX`. В случае неудачи функции заносят в регистр `EAX` значение `NULL`.

Выше не раз акцентировались возможности по кэшированию данных в процессе текущего файлового ввода-вывода. Отметим в связи с функцией записи данных в файл еще одну полезную функцию — `FlushFileBuffers`, которая позволяет принудительно сбросить все буферизованные данные, соотнесенные с файлом, дескриптор которого указан в качестве входного параметра данной функции:

```
BOOL WINAPI FlushFileBuffers (HANDLE hFile);
```

При успешном завершении функция `FlushFileBuffers` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция заносит в регистр `EAX` значение `NULL`.

Для иллюстрации вышесказанного разработаем программу чтения содержимого файла и его построчного вывода на экран. Этот тренировочный пример позволит нам освоить основные операции для работы с файлом и его содержимым. Суть действий программы состоит в следующем. Имеется файл с кодом на языке ассемблера — для большего интереса пусть это будет тот же самый файл, который содержит эту программу. Необходимо разработать консольное приложение, которое выводит построчное содержимое этого файла на экран.

```
+-----+
| prg07_31.asm — Win32-программа консольного приложения для исследования |
|                  работы с файлами на ассемблере — построчное чтение      |
|                  и вывод на экран содержимого файла.                      |
+-----+
.data
lpBuf      db 260 dup (20h)
lpBufSize  = $-lpBuf
lpNameFileInBuf dd 0
sAsm       db "asm"
hFile      dd 0
```

```

FileSize    dd    0
con         Coord <>
TitleText   db    'Работа с файлами в Win32'. 0
dOut        dd    0 ; дескриптор вывода консоли
dIn         dd    0 ; дескриптор ввода консоли
HandHead    dd    0 ; адрес кучи
p_start     dd    0 ; адрес блока для текста программы из общей кучи процесса
Numwiri     dd    0 ; количество символов очередной строки, выведенной на экран
.code
;....
;----- текст окна заголовка
push        offset TitleText
call        SetConsoleTitleA
;----- вывод строки текста:
;         вначале получим дескрипторы ввода и вывода консоли
push        STD_OUTPUT_HANDLE
call        GetStdHandle
mov         dOut, eax          ; dOut-дескриптор вывода консоли
push        STD_INPUT_HANDLE
call        GetStdHandle
mov         dIn, eax          ; dIn-дескриптор ввода консоли
;----- установим курсор в позицию (2, 5)
;....
call        SetConsoleCursorPosition
cmp         eax, 0
jz          exit              ; если неуспех
;----- имя исходного файла программы получаем исходя
;         из двух предположений:
;         1. имя исходного файла совпадает с именем его
;            исполняемого модуля
;         2. оба файла расположены в одном каталоге
;         HMODULE GetModuleHandle(LPCTSTR lpModuleName);
push        0
call        GetModuleHandleA
;----- DWORD GetModuleFileName(HMODULE hModule, LPTSTR lpFileName,
;         DWORD nSize);
push        lpBufSize
push        offset lpBuf
push        eax                ; дескриптор этого .exe-файла,
                                ; полученный GetModuleHandle
call        GetModuleFileNameA
;----- в eax длина полного пути к исполняемому файлу
;         (с расширением .exe)
mov         lpNameFileInBuf, eax
;----- для получения имени исходного файла заменим
;         расширение .exe на .asm
std
lea         esi, sAsm+2
lea         edi, lpBuf
dec         eax
add         edi, eax
push        ds
pop         es
mov         ecx, 3
rep         movsb
;----- открываем файл:
;         HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess,
;         ;         DWORD dwShareMode, LPSECURITY_ATTRIBUTES
;         ;         lpSecurityAttributes, DWORD dwCreationDisposition,
;         ;         DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);
push        0
push        0                  ; атрибуты (они игнорируются)
push        OPEN_EXISTING      ; открыть существующий файл.

```

```

                                : если его нет — ошибка
push    0                      : защита файла не требуется
push    FILE_SHARE_READ       : разрешено совместное использование
                                : файла (по чтению)
                                : разрешено чтение из файла
push    GENERIC_READ
push    offset TpBuf
call    CreateFileA
cmp     eax, 0xffffffff
je      exit                   : если неуспех
mov     hFile, eax             : дескриптор файла

;----- определим размер файла:
:      DWORD GetFileSize(HANDLE hFile, LPDWORD lpFileSizeHigh):
push    0
push    hFile
call    GetFileSize
cmp     eax, 0
jz      exit                   : если неуспех
mov     fileSize, eax          : сохраним размер файла

;----- выделяем блок в памяти, в который будем читать файл
:      HandHead — адрес с описателем кучи
:      используем кучу, выделяемую процессу по умолчанию
:      HANDLE GetProcessHeap (VDID):
call    GetProcessHeap
mov     HandHead, eax

;----- запрашиваем блок памяти из кучи:
:      LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes):
push    fileSize
push    0                      : флаги не задаем
push    HandHead               : дескриптор кучи
call    HeapAlloc
mov     p_start, eax           : адрес блока с текстом программы
                                : из общей кучи процесса

;----- читаем файл:
:      BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer,
:      :      DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
:      :      LPOVERLAPPED lpOverlapped):
push    0
push    offset fileSize        : сколько байтов прочитано реально
push    fileSize
push    p_start                : буфер, в который читаем файл
push    hFile
call    ReadFile
cmp     eax, 0
jz      exit                   : если неуспех

;----- установим курсор в позицию (1, 5)
:      ...
call    SetConsoleCursorPosition
cmp     eax, 0
jz      exit                   : если неуспех

;----- теперь можно построчно и на экран
cld
mov     edi, p_start
mov     esi, p_start
mov     ecx, fileSize
mov     al, 0dh
repne   scasb                  : в esi — адрес строки для вывода
cmp     byte ptr [edi], 0ah
jne     $-5
inc     edi
dec     ecx
jz      exit                   : весь файл прочитан
mov     eax, edi
sub     eax, esi               : в eax — длина строки для вывода

;----- вывести очередную строку
:      ...

```

```

call    WriteConsoleA
cmp     eax, 0
jz      exit          ; если неуспех
;....
add     esi, eax
jmp     cyscl
:----- закрываем файлы
exit:    ;....

```

При небольшой модификации программы можно построчно выводить содержимое любого текстового файла. Для небольших файлов совсем необязательно отслеживать конец каждой строки — проще выводить из буфера сразу весь файл одним вызовом функции `WriteConsoleA`.

Получение и изменение атрибутов файла

Аналогично группе функций MS DOS для работы с файловой системой, файловая подсистема Win32 содержит ряд функций, с помощью которых она позволяет определить характеристики конкретного файла.

Начальные значения атрибутов файла назначаются при создании файла. Впоследствии их можно изменить вызовом функции `SetFileAttributes`:

```
BOOL SetFileAttributes(LPCTSTR lpFileName, DWORD dwFileAttributes);
```

Параметры этой функции означают следующее:

- `lpFileName` — указатель на ASCIIZ-строку, содержащую имя файла;
- `dwFileAttributes` — двойное слово, определяющее, какие атрибуты файла могут быть установлены.

Планируя использование функции, необходимо иметь в виду, что не все возможные атрибуты файлов могут быть установлены с ее помощью. Перечислим те атрибуты, комбинацию которых допустимо задавать для изменения атрибутов файла, специфицированного параметром `lpFileName`: `FILE_ATTRIBUTE_ARCHIVE`, `FILE_ATTRIBUTE_HIDDEN`, `FILE_ATTRIBUTE_NORMAL`, `FILE_ATTRIBUTE_READONLY`, `FILE_ATTRIBUTE_SYSTEM`, `FILE_ATTRIBUTE_TEMPORARY`.

При успешном завершении функция `SetFileAttributes` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция заносит в регистр `EAX` значение `NULL`.

Для получения атрибутов файла используется функция `GetFileAttributes`:

```
DWORD GetFileAttributes(LPCTSTR lpFileName);
```

Функция имеет один параметр `lpFileName`, который является указателем на ASCIIZ-строку, содержащую имя файла.

При успешном завершении функция `GetFileAttributes` возвращает значение в регистре `EAX`, которое является комбинацией атрибутов файла, специфицированного параметром `lpFileName`. Выделить эти атрибуты можно, используя логические команды ассемблера или команды обработки битов. В случае неудачи функция заносит в регистр `EAX` значение `NULL`.

В приложениях очень часто требуется определить размер файла. Для этого в Win32 имеется отдельная функция `GetFileSize`:

```
DWORD GetFileSize( HANDLE hFile, LPDWORD lpFileSizeHigh );
```

Параметры функции означают следующее:

- `hFile` — дескриптор файла, размер которого требуется определить;

- **lpFileSizeHigh** — адрес области памяти, куда помещаются старшие 32 бита значения размера файла, младшие 32 бита возвращаются функцией в регистре **EAX**.

При успешном завершении функция **GetFileSize** возвращает значение младших 32 битов размера файла в регистре **EAX**. В случае неудачи функция заносит в регистр **EAX** значение **0xffffffffh**.

Особого разговора заслуживают возможности получения информации о временных характеристиках файлов. По сравнению с аналогичными средствами **MS DOS**, в **Win32** этот вопрос проработан значительно глубже. Хотя если посмотреть номенклатуру и описание функций **MS DOS** для работы с длинными именами файлов, то видно, что у них уже есть общие идеи, реализованные рассматриваемыми ниже функциями **Win32**.

Как уже отмечалось выше, в **Win32** с файлом связаны три значения времени: время создания, время последнего доступа и время последней модификации. Получить эти значения можно с помощью функции **GetFileTime**:

```
BOOL GetFileTime( HANDLE hFile, LPFILETIME lpCreationTime, LPFILETIME lpLastAccessTime,
  LPFILETIME lpLastWriteTime);
```

Перед вызовом данной функции необходимо открыть файл, о значениях временных параметров которого мы хотим получить информацию. Функции **GetFileTime** передается дескриптор этого файла и указатели на три экземпляра структуры **FILETIME**, в которые будут записаны время создания (**lpCreationTime**), время последнего доступа (**lpLastAccessTime**) и время последней записи (**lpLastWriteTime**).

Аналогично функции **MS DOS 71a7h**, **Win32** предоставляет две функции для взаимного преобразования **DOS**-времени файла в 64-битовое представление времени:

```
BOOL FileTimeToDosDateTime(CONST FILETIME *lpFileTime, LPWORD lpFatDate,
  LPWORD lpFatTime);
BOOL DosDateTimeToFileTime(WORD wFatDate, WORD wFatTime, LPFILETIME lpFileTime);
```

Функция **FileTimeToDosDateTime** в качестве входного параметра принимает указатель ***lpFileTime** на экземпляр структуры **FILETIME**. Этот указатель содержит представление времени в виде 64-битового значения. На выходе данная функция формирует два значения в переменных размером в слово, адреса которых указаны параметрами **lpFatDate** и **lpFatTime**. Формат этих слов совпадает с форматом соответствующих параметров, которыми манипулирует функция **71a7h**.

Функция **DosDateTimeToFileTime**, наоборот, преобразует время в формате **DOS**, представленное в виде двух слов — **wFatDate**, **wFatTime** (для времени и даты соответственно), в 64-битовое значение **lpFileTime**.

Установить время создания, последнего доступа или модификации файлов можно с помощью функции **SetFileTime**:

```
BOOL SetFileTime( HANDLE hFile, const FILETIME *lpCreationTime,
  const FILETIME *lpLastAccessTime, const FILETIME *lpLastWriteTime);
```

В качестве входных параметров функция **SetFileTime** принимает дескриптор файла и указатели на три экземпляра структуры **FILETIME**. Экземпляры структур уже заполнены необходимыми временными величинами. Если какое-либо из значений устанавливать не нужно, то вместо указателя на соответствующую структуру передается **NULL**. В случае успешного завершения функция возвращает ненулевое значение в регистре **EAX**.

Из вышеизложенного видно, что для получения различных характеристик файла используется множество различных функций. Работа с ними может утомить кого угодно. Нельзя ли чего-нибудь попроще? Можно. Win32 предоставляет функцию **GetFileInformationByHandle**:

```
BOOL GetFileInformationByHandle( HANDLE hFile,
    LPBY_HANDLE_FILE_INFORMATION lpFileInformation);
```

На вход данной функции передается дескриптор файла, о котором необходимо получить информацию, и указатель на экземпляр структуры **BY_HANDLE_FILE_INFORMATION**, который заполняется этой функцией. Как видно из названия полей (см. код ниже), в упомянутой структуре сосредоточена вся информация о файле. Ниже приведен пример кода, использующего данную структуру.

```
+-----+
| Программа: prg07_32.asm – win32-программа консольного приложения для Win32 |
| для исследования работы функции GetFileInformationByHandle API Win32.      |
+-----+

;----- описание структур
FILETIME struc
dwLowDateTime dd ?           ; младшие 32 бита значения времени
dwHighDateTime dd ?          ; старшие 32 бита значения времени
FILETIME ends
BY_HANDLE_FILE_INFORMATION struc
dwFileAttributes dd 0         ; атрибуты файла
struc
ftCreationTime_DwLowDateTime dd ? ; младшие 32 бита времени создания файла
ftCreationTime_DwHighDateTime dd ? ; старшие 32 бита времени создания файла
ends
struc
ftLastAccessTime_DwLowDateTime dd ? ; младшие 32 бита времени последнего доступа
ftLastAccessTime_DwHighDateTime dd ? ; старшие 32 бита времени последнего доступа
ends
struc
ftLastWriteTime_DwLowDateTime dd ? ; младшие 32 бита времени последней записи
ftLastWriteTime_DwHighDateTime dd ? ; старшие 32 бита времени последней записи
ends
dwVolumeSerialNumber dd 0      ; серийный номер тома, на котором
                                ; находится файл
nFileSizeHigh dd 0             ; старшие 32 бита размера файла
nFileSizeLow dd 0              ; младшие 32 бита размера файла
nNumberOfLinks dd 0            ; число ссылок на файл
nFileIndexHigh dd 0            ; старшие 32 бита идентификатора файла
nFileIndexLow dd 0             ; младшие 32 бита идентификатора файла
ends
.data
info
TitleText db "Получение информации о файле в Win32", 0
lpBuf db "p", 0
hFile dd 0
.code

;....
;----- CreateFile
;----- открываем файл
push 0
push 0 ; атрибуты (они игнорируются)
push OPEN_EXISTING ; открыть существующий файл,
; если его нет – ошибка
push 0 ; защита файла не требуется
push FILE_SHARE_READ ; разрешено совместное использование
; файла (по чтению)
push GENERIC_READ ; разрешено чтение из файла
```

```

push    offset lpBuf
call    CreateFileA
cmp     eax, 0xffffffffh
je      exit          ; если неуспех
mov     hFile, eax     ; дескриптор файла
:----- GetFileInformationByHandle
push    offset info
push    hFile
call    GetFileInformationByHandle
cmp     eax, 0
jz      exit          ; выход в случае неудачи
:----- результат смотрим в отладчике TD32.exe
:....

```

Результат работы данной программы можно посмотреть и проанализировать в отладчике.

Работа с дисками, каталогами и организация поиска файлов

Win32 располагает большим набором функций для получения информации о структуре файловой системы конкретного компьютера. Часть этих функций развивает идеи работы с файловой подсистемой, появившиеся в последних версиях MS DOS. Другие функции являются уникальными для платформы Win32. Рассмотрим наиболее интересные из них.

Программа, предназначенная для исследования файловой системы, прежде всего должна знать, какие логические диски присутствуют в системе. Среди нескольких функций, выполняющих эту работу, наиболее удобной для процесса обработки является `GetLogicalDrivesString`:

```
DWORD GetLogicalDriveStrings(DWORD nBufferLength, LPTSTR lpBuffer);
```

Данной функции передаются два параметра: `lpBuffer` — адрес буфера, в который помещаются имена корневых каталогов логических дисков, установленных в системе; `nBufferLength` — длина буфера, заданного указателем `lpBuffer`. В качестве возвращаемого значения функция формирует длину буфера, действительно необходимую для размещения строки с именами корневых каталогов логических дисков. Например, при наличии трех логических дисков структура заполненного буфера будет следующей: `A:\0B:\0C:\0`. Заметьте, что имена корневых каталогов разделены нулевыми символами (ANSI-ASCII или Unicode). Более эффективно вызывать эту функцию два раза: первый раз с нулевым значением первого параметра, при этом функция вернет требуемое количество байтов для размещения буфера; второй раз функцию уже можно вызывать, подставив на место первого параметра значение, возвращенное при первом вызове.

```

:-----+
:| Программа: prg07_33.asm. Win32-консольное приложение для Win32 для |
:| исследования работы функции GetLogicalDriveStrings API Win32.   |
:-----+
.data
TitleText    db      'Получение информации о дисках в Win32'. 0
info_buf     db      10 dup (0)
.code
:....
:----- GetLogicalDriveStrings
push    offset info_buf
push    0
call    GetLogicalDriveStringsA

```

```

cmp     eax, 0
jz      exit      ; выход в случае неудачи
;----- вызываем функцию второй раз, когда известно количество байтов
; требуемое для записи списка корневых каталогов
push    offset info_buf
push    eax
call    GetLogicalDriveStringsA
cmp     eax, 0
jz      exit      ; выход в случае неудачи
;----- результат смотрим в отладчике TD32.exe
;....

```

Недостаток функции **GetLogicalDriveStrings** состоит в том, что она работает не во всех версиях Windows. Альтернативным вариантом получения информации о наличии дисков в системе является функция **GetLogicalDrives**.

DWORD GetLogicalDrives(VOID);

Эта функция возвращает в регистре **EAX** битовую маску, в которой установленные биты указывают на существование логического диска: бит 0 — A, бит 1 — B, бит 2 — C... Таким образом, с помощью функции **GetLogicalDrives** можно достичь того же самого результата, что и с помощью функции **GetLogicalDriveStrings**, но несколько большими усилиями.

```

;+-----+
;| Программа: prg07_34.asm. Win32-консольное приложение для исследования |
;| работы функции GetLogicalDrives API Win32.                             |
;+-----+
.data
TitleText      db      'Получение информации о дисках в Win32'. 0
info_buf       db      10 dup (0)
.code
;....
call    GetLogicalDrives
cmp     eax, 0
jz      exit      ; выход в случае неудачи
;----- результат смотрим в отладчике TD32.exe
;....

```

После того как информация о номенклатуре логических дисков в системе известна, можно получить информацию о каждом из них. Для этого используется функция **GetVolumeInformation**:

BOOL GetVolumeInformation(LPCTSTR lpRootPathName, LPTSTR lpVolumeNameBuffer, DWORD nVolumeNameSize, LPDWORD lpVolumeSerialNumber, LPDWORD lpMaximumComponentLength, LPDWORD lpFileSystemFlags, LPTSTR lpFileSystemNameBuffer, DWORD nFileSystemNameSize);

На вход функции **GetVolumeInformation** подаются следующие параметры:

- **lpRootPathName** — указатель на строку с именем корневого каталога диска, информация о котором запрашивается (если параметр равен **NULL**, функция формирует информацию о текущем диске). Формат задания имени корневого каталога диска — имя_диска:\. Это единственный параметр, значение которого нужно задавать, остальные параметры — адреса областей памяти, в которые будут помещены значения, формируемые функцией;
- **lpVolumeNameBuffer** и **nVolumeNameSize** — указатель на буфер и размер буфера, в который будет записано имя диска;
- **lpVolumeSerialNumber** — адрес двойного слова, куда функция поместит серийный номер. Если информация о серийном номере диска не нужна, то при вызове функции значение этого параметра необходимо сделать равным **NULL**;

- `lpMaximumComponentLength` — адрес целевого двойного слова, предназначенного для значения максимальной длины пути, возможного в данной файловой системе;
- `lpFileSystemFlags` — флаги с дополнительной информацией о файловой системе:
 - `FS_CASE_SENSITIVE = FILE_CASE_SENSITIVE_SEARCH = 00000001h` — поддержка со стороны файловой системы поиска с сохранением регистра букв;
 - `FS_CASE_IS_PRESERVED = FILE_CASE_PRESERVED_NAMES = 00000002h` — при записи на диск сохранить регистр букв в имени файла;
 - `FS_UNICODE_STORED_ON_DISK = FILE_UNICODE_ON_DISK = 00000004h` — файловая система поддерживает хранение имен файлов в кодировке Unicode;
 - `FS_PERSISTENT_ACLS = FILE_PERSISTENT_ACLS = 00000008h` — файловая система способна оперировать со списками контроля доступа (ACL) — только для NTFS;
 - `FS_FILE_COMPRESSION = FILE_FILE_COMPRESSION = 00000010h` — файловая система поддерживает сжатие файлов;
 - `FS_VOL_IS_COMPRESSED = FILE_VOLUME_IS_COMPRESSED = 00008000h` — том, о котором запрашивается информация, был сжат;
- `lpFileSystemNameBuffer` и `nFileSystemNameSize` — указатель и размер буфера, в который будет помещено имя файловой системы. Если `lpFileSystemNameBuffer = NULL`, то в эти параметры ничего не записывается.

Изменить метку диска может вызов функции `SetVolumeLabel`:

```
BOOL SetVolumeLabel(LPCTSTR lpRootPathName, LPTSTR lpVolumeName);
```

Параметр `lpRootPathName` задает адрес строки с именем корневого каталога диска, метку которого меняем. Второй параметр — `lpVolumeName` — строка с меткой тома. Для удаления метки тома с диска параметр `lpVolumeName` нужно задать равным `NULL`.

Получить информацию о свободном дисковом пространстве

Информацию о свободном дисковом пространстве, а заодно и о разбиении диска на секторы и кластеры позволяет получить функция `GetDiskFreeSpace`:

```
BOOL GetDiskFreeSpace(LPCTSTR lpRootPathName, LPDWORD lpSectorsPerCluster,
    LPDWORD lpBytesPerSector, LPDWORD lpNumberOfFreeClusters,
    LPDWORD lpTotalNumberOfClusters);
```

На вход функции нужно подать строку с именем корневого каталога интересующего диска (`NULL` для текущего диска) и адреса буферов, куда будет помещена следующая информация: общее количество кластеров на диске (`lpTotalNumberOfClusters`), общее количество свободных кластеров (`lpNumberOfFreeClusters`), количество байтов в секторе (`lpBytesPerSector`), количество секторов в кластере (`lpSectorsPerCluster`).

Создание и удаление каталога

Создание каталога выполняет функция `CreateDirectory`:

```
BOOL CreateDirectory(LPCTSTR lpPathName, LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Первый параметр этой функции `lpPathName` — указатель на ASCIIZ-строку с путем, последний элемент которого является именем нового каталога. Параметр `lpSecurityAttributes` — указатель на экземпляр структуры `Security_Attributes`.

```
SECURITY_ATTRIBUTES struc
nLength          dd 0
lpSecurityDescriptor dd 0
bInheritHandle    dd 0
SECURITY_ATTRIBUTES ends
```

С помощью структуры `Security_Attributes` можно ограничить доступ пользователя к каталогу. Параметр `lpSecurityAttributes` обычно задается равным `NULL`. Более подробная информация о параметрах структуры имеется в документации MSDN.

За удаление каталога отвечает функция `RemoveDirectory`:

```
BOOL RemoveDirectory(LPCTSTR lpPathName);
```

Единственный параметр этой функции `lpPathName` — указатель на ASCIIZ-строку с путем, последний элемент которого является именем удаляемого каталога. Удаляемый каталог не должен быть пустым.

Определение и изменение текущего каталога

Аналогично принципам организации файловой системы MS DOS, в Win32 также существует понятие текущего каталога, то есть каталога, в котором выполняются текущие операции по работе с файлами. В отличие от MS DOS, понятие текущего каталога относится к текущему процессу. При запуске процесса текущим будет являться каталог, из которого этот процесс был запущен. Определяет текущий каталог процесса функция `GetCurrentDirectory`:

```
DWORD GetCurrentDirectory(DWORD nBufferLength, LPTSTR lpBuffer);
```

Параметры `lpBuffer` и `BufferLength` устанавливают соответственно адрес и длину буфера, в который помещается путь с текущим каталогом (строка с завершающим нулем). Функция возвращает `NULL` в случае ошибки и число байтов, необходимое для записи данных в буфер, в случае удачного срабатывания. Завершающий нуль в возвращаемом функцией числе не учитывается. Если буфер мал, то с помощью возвращаемого значения можно изменить его размер.

Изменить текущий каталог процесса можно посредством функции `SetCurrentDirectory`:

```
BOOL SetCurrentDirectory(LPCTSTR lpzPathName);
```

Параметр `lpzPathName` — адрес ASCIIZ-строки с путем, последний элемент которого — новый текущий каталог данного процесса.

Платформа Win32 также поддерживает понятие системного и основного каталога Windows. Для определения системного каталога существует специальная функция `GetSystemDirectory`:

```
UINT GetSystemDirectory(LPTSTR lpzBuffer, UINT uSize);
```

Два параметра этой функции — это адрес и размер буфера, в который записывается путь к системному каталогу Windows. Возвращаемое значение — количество реально записанных в буфер байтов. Его можно использовать для корректировки параметра `uSize`, если тот был задан слишком маленьким, и повторного вызова функции `GetSystemDirectory`. Проверить, является ли каталог системным, можно,

попытавшись создать в нем файл. Если файл системный, ваша попытка будет неудачной.

Для определения основного каталога Windows существует специальная функция **GetWindowsDirectory**:

```
UINT GetWindowsDirectory(LPTSTR lpBuffer, UINT uSize);
```

Два параметра этой функции определяют адрес и размер буфера, в который записывается путь к основному каталогу Windows. Возвращаемое значение — количество реально записанных в буфер байтов. Оно применяется для корректировки параметра **uSize**, если последний был задан слишком маленьким, и повторного вызова функции **GetWindowsDirectory**.

Для демонстрации применения представленных выше функций рассмотрим комплексный пример, в ходе которого продемонстрируем порядок вызова и анализа возвращаемых значений функциями API Win32 для работы с каталогами. То есть: определим текущий каталог процесса, создадим новый каталог, сделаем его текущим, удалим новый каталог, определим системный каталог и каталог Windows.

```
+-----+
:| Программа: prg07_35.asm. Win32-консольное приложение для исследования
:| функций для работы с каталогами API Win32.
+-----+
.data
TitleText      db      "Работа с каталогами в Win32". 0
NewDir         db      "Новый каталог". 0
dir_buf        db      50 dup ("?")
size_dir_buf   equ     $ - dir_buf
Parent         db      "..". 0
.code

:....
:----- определим текущий каталог
push offset dir_buf
push size_dir_buf
call GetCurrentDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи

:----- создадим каталог
push 0
push offset NewDir
call CreateDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи

:----- сделаем новый каталог текущим
push offset NewDir
call SetCurrentDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи

:----- проверим новый текущий каталог
push offset dir_buf
push size_dir_buf
call GetCurrentDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи

:----- SetCurrentDirectory
:----- вернемся в родительский каталог
push offset Parent
call SetCurrentDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи

:----- проверим новый текущий каталог
push offset dir_buf
```

```

push    size_dir_buf
call    GetCurrentDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи
:----- удалим новый текущий каталог
push    offset NewDir
call    RemoveDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи
:----- определим системный каталог
mov     eax, size_dir_buf
push    eax
push    offset dir_buf
call    GetSystemDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи
:----- определим основной каталог Windows
mov     eax, size_dir_buf
push    eax
push    offset dir_buf
call    GetWindowsDirectoryA
cmp     eax, 0
jz      exit          ; выход в случае неудачи
:----- результат смотрим в отладчике TD32.exe
:....

```

Среди функций Win32, работающих с текущим каталогом, присутствует функция `GetFullPathName`, которая по имени файла формирует его полное имя, состоящее из пути от корневого каталога к текущему. Последний элемент этого имени — имя входного файла.

`DWORD GetFullPathName(LPCTSTR lpFileName, DWORD nBufferLength, LPTSTR lpBuffer, LPTSTR *lpFilePart);`

На входе функция принимает имя файла в виде ASCIIZ-строки. Три других параметра:

- `lpBuffer` — адрес буфера, в который помещается полный путь с именем файла;
- `nBufferLength` — длина буфера, на который указывает параметр `lpBuffer`, в символах;
- `lpFilePart` — адрес ячейки размером с двойное слово, в которую помещается указатель на позицию внутри буфера, идентифицированную параметром `lpBuffer` и соответствующую первому символу имени файла после имен всех каталогов.

Самое интересное в этой функции — механизм ее работы. Суть его в том, что реально функция `GetFullPathName` не ищет файл, на имя которого ссылается параметр `lpBuffer`. Результат своей деятельности — полный путь — она формирует из двух компонентов: полного пути к текущему каталогу данного процесса и имени файла, наличие которого на диске функция `GetFullPathName` даже не смотрит. Для подобной работы ей вовсе не нужно обращаться к диску. С ее аналогом мы уже сталкивались, когда рассматривали функции MS DOS для работы с файлами, имеющими длинные имена.

Поиск файлов

При последовательном изучении материала данного раздела читатель, кроме знакомства со средствами по работе с файлами операционных систем фирмы Microsoft, поневоле должен был оценить процесс эволюции этих средств. Особенно очевиден этот прогресс в отношении к поиску файлов.

Платформа Win32 предлагает два способа поиска файлов:

- при помощи функции `SearchPath`;
- посредством функций `FindFirstFile`, `FindNextFile` и структуры `WIN32_FIND_DATA`.

Поиск файлов с помощью функции `SearchPath`

Функция `SearchPath` ищет файлы в указанном при ее вызове списке каталогов.

```
DWORD SearchPath(LPCTSTR lpPath, LPCTSTR lpFileName, LPCTSTR lpExtension,
                DWORD nBufferLength, LPTSTR lpBuffer, LPTSTR *lpFilePart);
```

Первый параметр `lpPath` определяет список каталогов, в которых будет осуществляться поиск файла. Параметры `lpFileName` и `lpExtension` указывают на ASCII-строки с именем и расширением искомого файла. Наличие пары этих параметров позволяет задавать имя и расширение файла двумя способами:

- одной ASCII-строкой — на нее указывает параметр `lpFileName`, при этом параметр `lpExtension` равен `NULL`;
- отдельными ASCII-строками — в этом случае параметр `lpFileName` содержит указатель на ASCII-строку с именем файла, а второй параметр — `lpExtension` — содержит указатель на ASCII-строку с расширением файла; строка с расширением должна начинаться с символа `.` (точка).

Параметр `lpBuffer` указывает на буфер, куда записывается ASCII-строка с путем к искомому файлу. Длина буфера определяется параметром `nBufferLength`. Если эта длина слишком мала, то ее можно подкорректировать значением, возвращаемым функцией в регистре `EAX`. Данное значение является количеством символов, действительно необходимых для записи полного имени найденного файла в буфер. Если в `EAX` возвращается `NULL`, то это говорит об ошибке вызова функции.

Последний параметр `lpFilePart` является указателем на символ в буфере, с которого начинается собственно имя файла.

При вызове функции `SearchPath` параметр `lpPath` можно задать равным `NULL`. В этом случае поиск файла будет осуществляться в следующих каталогах (порядок перечисления соответствует очередности просмотра при поиске):

- каталог, из которого запущено приложение;
- текущий каталог;
- системный каталог;
- основной каталог Windows;
- каталоги, перечисленные в переменной окружения `PATH`.

Поиск файлов с помощью функций `FindFirstFile` и `FindNextFile`

Предыдущий способ поиска обладает существенным недостатком — ограниченным числом каталогов диска, подвергающихся просмотру в процессе поиска. По этой причине он не применим для поиска в пределах всего диска. Означенный недостаток устраняется при втором способе поиска — с использованием функций `FindFirstFile`, `FindNextFile` и структуры `WIN32_FIND_DATA`. Этот способ реализует определенный алгоритм поиска. Вначале вызывается функция `FindFirstFile`, которая имеет два параметра: `lpFileName` — указатель на ASCII-строку с именем файла; `lpFindFileData` — указатель на экземпляр структуры `WIN32_FIND_DATA`.

```
HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);
```

Имя файла может содержать символы шаблона * и ?. Кроме того, имя может содержать путь, с которого нужно начинать поиск. Выше, при знакомстве с функциями MS DOS для работы файлами, имеющими длинные имена, приводилось описание структуры WIN32_FIND_DATA и ее полей.

В случае успеха функция FindFirstFile заполняет поля структуры WIN32_FIND_DATA и возвращает значение дескриптора внутренней структуры в памяти, который впоследствии может быть использован функцией FindNextFile или FindClose. В случае неудачи функция не изменяет содержимое структуры WIN32_FIND_DATA и возвращает значение INVALID_HANDLE_VALUE (EAX = -110 (десятичное) = 0xffffffffh).

Проанализировав результаты поиска, программа может продолжить или прекратить его. Для продолжения поиска необходимо вызвать функцию FindNextFile:

```
BOOL FindNextFile( HANDLE hFindFile, LPWIN32_FIND_DATA lpFindFileData );
```

В качестве параметров используются дескриптор, полученный в регистре EAX в результате поиска функцией FindFirstFile, и указатель на экземпляр структуры WIN32_FIND_DATA. В случае успеха функция FindNextFile возвращает ненулевое значение в регистре EAX и заполняет структуру WIN32_FIND_DATA. При неудаче — EAX = 0.

Для продолжения поиска при неизменных исходных параметрах поиска функция FindNextFile вызывается циклически.

Для окончания процесса поиска необходимо вызвать функцию FindClose:

```
BOOL FindClose( HANDLE hFindFile );
```

Функция FindClose имеет один параметр — дескриптор, полученный функцией FindFirstFile в начале поиска. В случае успеха функция FindClose возвращает ненулевое значение в регистре EAX, при неудаче — EAX = 0.

Файлы, отображаемые в память

Платформа Win32 позволяет организовать работу с содержимым файла как с областью оперативной памяти, без обращения к операциям файлового ввода-вывода. Этот механизм отображает (проецирует) содержимое файла на область оперативной памяти. Программе передается адрес этой области, после чего работа с содержимым файла осуществляется командами работы с памятью.

Для «проецирования» файла необходимо выполнить следующие действия.

1. Требуется создать (для несуществующего файла) или открыть (для существующего файла) объект ядра *файл*. Цель этого шага — сообщить системе, где находится физическое представление файла. Создание или открытие объекта ядра *файл* выполняется с помощью функции CreateFile (см. выше). Все параметры этой функции задаются обычным образом. На выходе в случае успеха функция формирует дескриптор (в регистре EAX), в обратном случае — значение INVALID_HANDLE_VALUE (EAX = -1 (десятичное) = 0xffffffffh).
2. Требуется создать объект ядра *проекция файла*. Цель этого шага — сообщить системе размер проецируемого файла. Для этого предназначена функция CreateFileMapping:

```
HANDLE CreateFileMapping(HANDLE hFile, LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow,
    LPCTSTR lpName);
```

Параметр `hFile` является дескриптором файла, полученным функцией `CreateFile`. Параметр `lpFileMappingAttributes` — указатель на экземпляр структуры `SECURITY_ATTRIBUTES`, которая служит для установки защиты. Присвойте параметру `lpFileMappingAttributes` значение `NULL`. Параметр `flProtect` предназначен для установки атрибутов защиты страниц физической памяти в адресном пространстве процесса, на которые отображается файл. Используют один из следующих атрибутов:

- `PAGE_READONLY = 02` — доступ к файлу разрешен только по чтению (при использовании этого параметра вызов `CreateFile` должен производиться с флагом `GENERIC_READ`);
- `PAGE_READWRITE = 04` — доступ к файлу только по записи (вызов `CreateFile` должен производиться с флагом `GENERIC_READ | GENERIC_WRITE`);
- `PAGE_WRITECOPY = 08` — доступ к файлу в режиме чтения-записи с созданием копии данных из файла; при этом исходный файл не изменяется, изменения касаются лишь модифицированных страниц копии в страничном файле (вызов `CreateFile` должен производиться с флагом `GENERIC_READ` или `GENERIC_READ | GENERIC_WRITE`).

Параметры `dwMaximumSizeHigh` и `dwMaximumSizeLow` предназначены для того, чтобы сообщить системе максимальный размер файла в байтах. При этом в `dwMaximumSizeLow` указываются младшие 32 бита этого значения, а в `dwMaximumSizeHigh` — старшие 32 бита. Если предполагается размер файла, равный текущей его длине, то следует при вызове функции передать `dwMaximumSizeLow = dwMaximumSizeHigh = NULL`.

Последний параметр `lpName` — указатель на ASCII-строку с именем объекта «проецируемый файл» для обеспечения доступа к нему других процессов. Обычно задают равным `NULL`.

3. Требуется выполнить проецирование файла на адресное пространство процесса. Здесь преследуются две цели. Первая — сообщить системе порядок отображения (проецирования) файла на адресное пространство процесса — полный или частичный. Вторая цель — получить адрес этого проецирования в памяти. Реализация означенных целей достигается функцией `MapViewOfFile`:

```
LPVOID MapViewOfFile(HANDLE hFileMappingObject, DWORD dwDesiredAccess,
                    DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, DWORD dwNumberOfBytesToMap);
```

Параметр `hFileMappingObject` — дескриптор, возвращенный функцией `CreateFileMapping` на предыдущем шаге. Параметр `dwDesiredAccess` определяет вид доступа к данным:

- `FILE_MAP_COPY = 1` — данные в файле доступны по чтению, хотя отображенные данные доступны по чтению и по записи; операция записи приводит к созданию копии страницы в страничном файле, в которую производится запись, поэтому после первой операции записи теряется соответствие между реальными данными на диске и данными, с которыми работает приложение (при использовании этого значения параметра `dwDesiredAccess` функция `CreateFileMapping` должна быть вызвана с одним из атрибутов: `PAGE_READONLY`, `PAGE_READWRITE` или `PAGE_WRITECOPY`);

- `FILE_MAP_WRITE = 2` — данные в файле доступны по чтению-записи (тогда функция `CreateFileMapping` должна быть вызвана с атрибутом `PAGE_READWRITE`);
- `FILE_MAP_READ = 4` — данные в файле доступны по чтению (функция `CreateFileMapping` должна быть вызвана с одним из атрибутов: `PAGE_READONLY`, `PAGE_READWRITE` или `PAGE_WRITECOPY`);
- `FILE_MAP_ALL_ACCESS = 000F0000h | 00000001h | 00000002h | 00000004h | 00000008h | 00000010h` — данные в файле доступны по чтению-записи (соответственно функция `CreateFileMapping` должна быть вызвана с атрибутом `PAGE_READWRITE`).

Параметры `dwFileOffsetHigh`, `dwFileOffsetLow` и `dwNumberOfBytesToMap` предназначены для указания позиции в файле, с которой начинать отображение, и количества отображаемых байтов (`dwNumberOfBytesToMap`). Параметр `dwFileOffsetHigh` — старшие 32 бита этого смещения, а параметр `dwFileOffsetLow` — младшие 32 бита этого смещения. Таким образом, с файлом допустимо работать не целиком, а по частям, эффективно расходуя при этом оперативную память. Заметим, что если задать параметр `dwNumberOfBytesToMap` равным `NULL`, то система будет пытаться отобразить содержимое файла с места, указанного парой `dwFileOffsetHigh:dwFileOffsetLow`, и до конца файла.

В случае успеха функция формирует адрес проекции в памяти (регистр `EAX`), в противном случае `EAX = 0`. После получения в `EAX` адреса начала отображения в памяти, приложение может работать с данными файла обычными командами работы с памятью. При необходимости функция `MapViewOfFile` может быть вызвана повторно с другими параметрами — `dwDesiredAccess`, `dwFileOffsetHigh`, `dwFileOffsetLow` и `dwNumberOfBytesToMap`. При этом (запомните!) резервируется новый регион в памяти.

После выполнения необходимых действий приложение должно корректно завершить работу с отображением файла.

4. Требуется выполнить разрыв связи данных в файле и соответствующими данными, отображенными на адресное пространство процесса. За это действие отвечает функция `UnmapViewOfFile`:

```
BOOL UnmapViewOfFile( LPCVOID lpBaseAddress);
```

Эта функция имеет единственный параметр — `lpBaseAddress`, который является значением, возвращенным функцией `MapViewOfFile`. С помощью функции `UnmapViewOfFile` необходимо разрывать каждое из отображений, созданных последовательностью вызовов `MapViewOfFile`, сохраняя при этом их соответствия. Также имейте в виду, что если функция `MapViewOfFile` была вызвана с параметром `FILE_MAP_COPY`, то после вызова `UnmapViewOfFile` теряются все внесенные в отображенные данные изменения.

5. Далее нужно закрыть объект ядра «проекция файла». В принципе, этот и следующий шаг не являются обязательными, так как система в процессе завершения работы приложения самостоятельно открепит все ресурсы. Освобождение объекта ядра «проекция файла» производится функцией `CloseHandle`.

```
BOOL CloseHandle( HANDLE hObject );
```

Функции `CloseHandle` передается единственный параметр `hObject` — дескриптор, сформированный как результат вызова функции `CreateFileMapping`.

6. Требуется закрыть объект ядра «файл». Освобождение объекта ядра «файл» также производится функцией `CloseHandle`:

```
BOOL CloseHandle( HANDLE hObject );
```

Функции `CloseHandle` передается единственный параметр `hObject` — дескриптор, полученный как результат вызова функции `CreateFile`.

Пример программы (`prg07_36.asm`), демонстрирующей порядок использования файлов, отображаемых в адресное пространство процесса, достаточно велик и по этой причине вынесен отдельно, его можно найти среди материалов, прилагаемых к книге. Работа программы проста и заключается в следующем: поставлена задача вывести содержимое некоторого файла на экран — в окно консоли. Имя исходного файла вводится с клавиатуры.

Глава 8

Оптимизация программного кода. Профайлер

Конструктор знает, что он достиг совершенства не тогда, когда нечего больше добавить, а тогда, когда нечего больше убрать.

Антуан де Сент-Экзюпери

Часто задают вопрос о времени выполнения той или иной команды. Возникновению этого вопроса есть две причины. Первая — *практическая*. Всем хочется, чтобы его программа работала быстро и была не очень требовательна к ресурсам. Учитывая то, что современные компьютеры имеют объемы памяти и процессоры с быстродействием, достаточным для сглаживания недостатков в подготовке программиста в плане написания оптимальных программ, существует вторая причина — *спортивный интерес*. Здесь ситуация как в спорте — получить максимальный результат и моральное удовлетворение. Популярные сегодня различные соревнования по программированию подтверждают это. На них часто встречаются задачи, требующие понимания задачи оптимизации вообще и наличия определенных навыков оптимизации в частности. В любом случае, задуматься о том, насколько оптимален создаваемый код — признак мастерства программиста. В большинстве случаев понятия «оптимизация» и «скорость работы» — синонимы.

Задача оптимизации исходного кода на любом языке программирования многогранна и предполагает многоуровневую схему своей реализации. Во-первых, оптимальным должен быть алгоритм верхнего уровня, то есть основной алгоритм, реализующий постановку задачи. Во-вторых, дополнительной оптимизации могут быть подвергнуты отдельные участки алгоритма верхнего уровня. Как правило, для большой программы всегда можно выбрать несколько вариантов реализации ее отдельных фрагментов. И, в-третьих, оптимальный алгоритм должен учитывать особенности аппаратуры, на которой будет выполняться реализованная на его основе программа. Как правило, реализация этого уровня в программе производится с использованием ассемблерных вставок или внешних процедур, написанных на ассемблере. Исходя из этого, в данной главе мы и рассмотрим вопросы, связанные с этим уровнем, хотя некоторые моменты могут быть приложены и ко второму уровню.

Из материала учебника [39] следует, что код, написанный на любом языке программирования, в конечном итоге оказывается переведенным компилятором на машинный язык и исполнен вполне конкретным процессором. Современные компиляторы достаточно «умны» и «знакомы» с особенностями многих современных процессоров. Большинство типовых алгоритмических конструкций программы они переводят на машинный язык наиболее оптимальным способом и зачастую далеко не так, как это себе представляет программист. Поэтому, с одной стороны, анализируя, а тем более реассемблируя код, программист должен понимать разницу между исходным и внутренним представлением программы. С другой стороны, необходимо осознавать то, что, оптимизируя код, компилятор работает на основе некоторых шаблонов и типовых схем и не способен предусмотреть все возможные случаи. Поэтому программист должен всегда иметь возможность внести свой вклад в оптимизацию программы, исходя из особенностей конкретного алгоритма.

Целью этой главы не является составление законченного рецепта, с помощью которого все ваши программы вдруг станут оптимальными. У нее другая задача — сформировать правильное понимание проблемы оптимизации низкого уровня, показать источники, где искать эти рецепты, и, в конечном итоге, побудить читателя к тому, чтобы задуматься об этой проблеме и осмысленно подходить к выбору средств для ее решения.

В самом общем виде план низкоуровневой оптимизации программы можно представить в виде следующих этапов:

1. Уточнение типов процессоров, на которых планируется исполнение программы.
2. Выяснение и выделение критичных, с точки зрения быстродействия, участков программы.
3. Выбор приемов оптимизации.
4. Оценка качества оптимизации.

Выбор методов оптимизации и ее результаты прямо зависят от типа процессора, на котором предполагается функционирование программы. Программа, предназначенная для работы на разных процессорах, должна первым делом определять текущий процессор. Нет проблемы написания оптимальной программы на языке ассемблера вообще, есть проблема написания программы, оптимально исполняемой на конкретном типе процессора. Чтобы оптимизировать программу, программист должен представлять себе особенности функционирования процессора, для которого она пишется. По этой причине при обсуждении архитектурных элементов семейств процессоров Pentium будем указывать, о каком конкретно процессоре или семействе процессоров идет речь — семейство P6 (Pentium II/III — PII/III) и Net Burst (Pentium 4). При необходимости программа должна иметь несколько ветвей, содержащих разные реализации одного и того же участка кода, отличия которых заключаются в использовании приемов оптимизации, ориентированных на конкретный тип процессора или семейство процессоров.

Определение типа процессора

Начиная с систем команд последних процессоров i80486 и первых Pentium, сведения о типе и особенностях текущего процессора можно получить с помощью команды CPUID. Подробную информацию об этой команде вы найдете в приложе-

нии А учебника [39], пример использования приведен там же в главе 18. Если вы хотите написать универсальную программу, пригодную для работы на всех типах Intel-совместимых процессоров, то команду CPUID лучше использовать в соответствии со следующими правилами:

1. Выяснить, поддерживает ли используемый транслятор ассемблера команду CPUID. Этот факт выявится после первого запуска вашей программы на трансляцию. В листинге строка с командой CPUID будет помечена как ошибочная. Из данной ситуации два выхода: первый — обновить версию пакета ассемблера; второй — моделировать команду CPUID последовательностью байтов db0fh, 0a2h.
2. Выяснить, поддерживает ли текущий процессор команду CPUID. Для этого нужно проанализировать бит 21 регистра EFLAGS. Он введен в архитектуру процессора специально с целью поддержки программной идентификации текущим процессором команды CPUID:

```

pushfd                ; регистр EFLAGS
pop    eax             ; EFLAGS > EAX
mov    ebx, eax        ; сохранили для дальнейшего сравнения
xor    eax, 200000h    ; изменили 21-й бит EAX
;----- восстанавливаем EFLAGS и проверяем факт изменения 21-го бита
push   eax
popfd
pushfd
pop    eax
xor    eax, ebx
test   eax, 200000h    ; проверяем 21 бит
jz     not_support     ; CPUID не поддерживается

```

3. Выяснить количество функций, поддерживаемых командой CPUID текущего процессора. Команда CPUID не является застывшим образованием. Она развивается вместе с развитием семейств процессоров Intel и изначально спроектирована так, чтобы разработчик программы мог динамически узнавать возможности процессора и при необходимости корректировать ход выполнения программы. В Pentium 4 доступно четыре функции, их номера (от 0 до 3) предварительно помещаются в регистр EAX. В ответ процессор возвращает определенную информацию о текущем процессоре [39].

Приемы оптимизации

В этом разделе мы рассмотрим элементы программы на ассемблере, на которые следует обращать внимание при проведении оптимизации, сами приемы оптимизации, а также разъяснение причин архитектурного уровня процессора, благодаря которым тот или иной прием оптимизации работает. Интересно то, что после станет видна неоптимальность большинства представленных в учебнике и данном практикуме программ. Не нужно спешить с критикой, главное помнить правило — то, что оптимально для одного процессора, совсем не обязательно будет оптимальным для другого процессора. Целями и задачами учебника и практикума является обучение использованию всего множества команд Intel-совместимых процессоров для программирования типовых алгоритмов. На этапе изучения языка ассемблера задачу оптимизации логично вынести на последний этап этого процесса.

Перечислим элементы программы на ассемблере, на которые необходимо обращать внимание при оптимизации программы. В первую очередь это:

- описание данных и операций доступа к памяти;
- циклические конструкции и переходы;
- оптимизация, где возможно, последовательности команд. Замена одних команд другими, эквивалентными по принципу работы, но более оптимальными для текущего процессора;
- планирование исполнения команд с учетом особенностей текущего процессора.

Архитектурные особенности процессоров Pentium

Ниже приведено более подробное обсуждение приемов оптимизации вышеперечисленных программных элементов. Целевыми процессорами обсуждения выбраны процессоры Pentium. Вначале рассмотрим подсистемы процессоров Pentium, особенности работы которых имеют значение для эффективного решения проблемы автоматизации. Это кэш-память и конвейер.

Кэш

Идея кэширования данных получила свое воплощение в процессорах Intel, начиная с i80486, и оказалась эффективным средством повышения их производительности. С тех пор архитектура подсистемы кэширования вызревала параллельно с развитием архитектуры процессора и все более в нее интегрировалась. Понимание особенностей работы кэш-памяти играет важную роль в планировании и проведении оптимизации программы.

Основная цель наличия кэш-памяти — хранить последние, недавно использовавшиеся данные и команды «поближе» к процессору. Расчет простой — с определенной вероятностью можно предположить, что обращение к этим данным и командам будет скоро произведено снова. Современные процессоры обычно поддерживают до трех уровней кэш-памяти, они обозначаются L1, L2, L3 соответственно. Кэш-память *первого* уровня, по преимуществу, очень маленькая и находится на одном кристалле с процессором. Как правило, она является разделенной — по 8 или 16 Кбайт для кода и для данных. Так, в процессорах семейства P6 кэш-памяти команд и данных первого уровня (L1) имеют размеры по 8 или 16 Кбайт. Кэш-память *второго* уровня (L2) является смешанной, имеет много больший размер (до 1 Мбайт), чем кэш-память первого уровня, и содержит все ее данные и команды. Эта память уже не находится на одном кристалле с процессором, но конструктивно объединена с ним посредством высокоскоростной шины.

Существуют несколько архитектур кэш-памяти. В основе их работы лежат следующие принципы. Вся оперативная память делится на блоки фиксированного размера, называемые *строками кэш-памяти*. Длина строки до 64 байтов. Все обращения процессора к памяти, за исключением некоторых случаев (см. ниже), производятся через кэш-память. Первым его действием является попытка обнаружить ячейку памяти или команду в кэш-памяти первого уровня. Основой для этого яв-

ляется целевой адрес в оперативной памяти. Если попытка была неудачной (*кэш-промах*), то производится генерация запроса на системную шину для доступа к нужной ячейке памяти и одновременно делается попытка обнаружить нужные данные или команду в кэше более высокого уровня (L2), если он есть. Таким образом, кэш-память всегда чем-то заполнена.

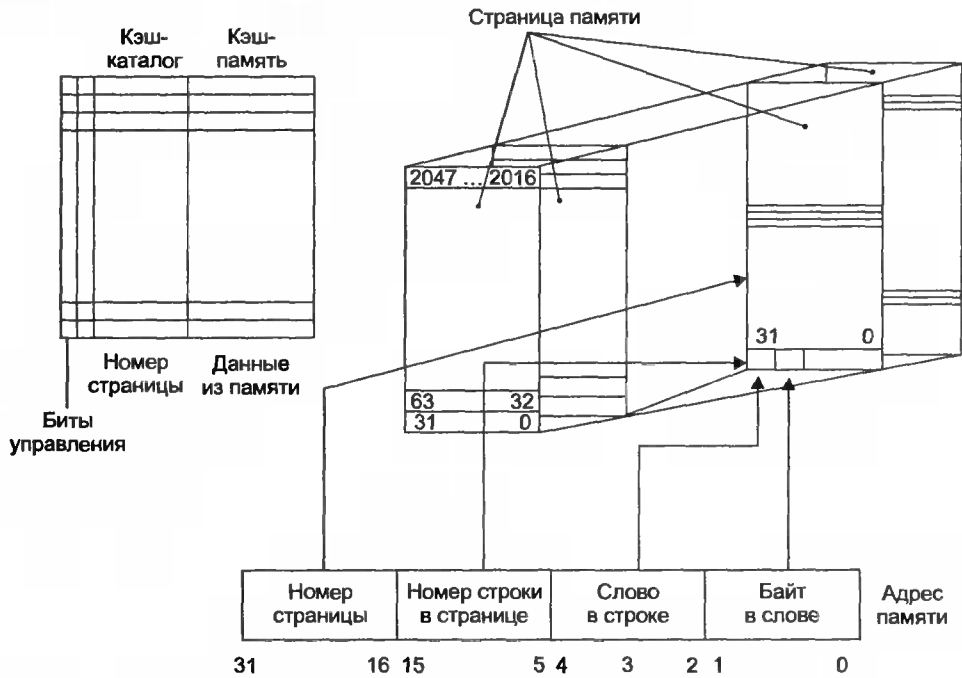


Рис. 8.1. Архитектура кэш-памяти прямого отображения

Принцип работы кэш-памяти поясним на примере ее самой простой организации — кэш-памяти прямого отображения. Вся оперативная память компьютера представляется контроллером кэш-памяти разбитой на страницы размером 65 536 байтов (рис. 8.1). К какой странице принадлежит конкретный 32-битовый адрес памяти, можно вычислить, если обнулить его младшие 16 битов. Далее, с точки зрения контроллера кэш-памяти, каждая страница представляется разбитой на строки по 32 байта (или 64 байта в семействе Net Burst). В примере на рис. 8.1 каждая страница поделена на 2048 (0–2047) строк по 32 байта каждая. Таким образом, емкость страницы — $2048 \cdot 32 = 65\,536$ байтов. Количество страниц также вычислить легко — $0\text{FFFF}0000\text{h} = 4\,294\,901\,760$. Максимальный объем памяти, с которым может работать кэш-память такой организации, — $281\,470\,681\,743\,360$ байтов. Это намного превышает предельный объем адресуемой памяти в IA-32 — $4\,294\,967\,295$ байтов (32-битовый адрес) или, при некоторых условиях, $68\,719\,476\,735$ байтов (36-битовый адрес). Поэтому реально используются лишь несколько младших битов из поля «номер страницы». Логически все строки разных страниц с одинаковым значением поля адреса «номер страницы» можно объединить в *линейку*. Сама кэш-память, показанная на рисунке слева, логически пред-

ставляет собой таблицу 32-байтовых элементов. Каждой строке таблицы приписан дескриптор, в котором выделяются два поля — тег и бит достоверности. Поле «тег» содержит номер страницы оперативной памяти, строка которой находится в данной строке кэш-памяти. Бит достоверности показывает актуальность строки кэш-памяти. С помощью этого бита в начале работы процессора все строки обозначаются недостоверными («грязными»). Важно понимать то, что в каждой строке кэш-памяти могут содержаться строки только из одной линейки строк из разных страниц оперативной памяти. Действия процессора после формирования адреса для чтения из памяти:

1. Выделение из адреса поля «номер строки».
2. Обращение к строке таблицы кэш-памяти со значением номера, равным значению поля адреса «номер строки».
3. Проверка бита достоверности — если элемент достоверен, то переход к пункту 4. Если бит помечен как «грязный», то считается, что нужной строки памяти в кэше нет, и запускается процесс чтения ее из оперативной памяти и замещение ею данной строки кэш-памяти.
4. Сравниваются значения полей «номер страницы» таблицы кэш-памяти и адреса. Если они равны, то нужное слово или байт в строке кэш-памяти есть и производится его выборка. Если эти значения не совпадают, то считается, что нужная строка отсутствует (*промах кэш-памяти*) и, как в п. 3, производится запуск процесса чтения нужной строки из памяти и замещение ею данной строки кэш-таблицы. Предварительно требуется записать вытесняемую строку из кэш-памяти на свое место в основную память.

Хранение данных в кэш-памяти значительно ускоряет доступ к данным в основной памяти. Фактически в кэше можно хранить одновременно до 65 536 смежных байтов оперативной памяти (64 Кбайт). Но если представить ситуацию, когда в программе есть близко расположенные команды, работающие с ячейками памяти, отстоящими друг от друга на расстояние большее, чем 65 536, то мы получим значительное снижение производительности. Причина в том, что процессору придется постоянно обновлять содержимое кэш-памяти и делать это не одним байтом или словом, а, как минимум, порциями по 32 байта. Чтобы снизить вероятность таких ситуаций (*коллизий*), кэш-память первого уровня, во-первых, разделяют для кода и данных и, во-вторых, применяют наборно-ассоциативную архитектуру кэш-памяти.

Основная идея наборно-ассоциативной архитектуры — поддержка нескольких связанных кэш-таблиц, подобных применяемым в архитектуре кэш-памяти прямого отображения (рис. 8.2). В этой архитектуре становится возможным хранить строки байтов из различных страниц памяти с одинаковыми номерами строк. Такая архитектура кэш-памяти усложняет алгоритм управления ею. При необходимости чтения из памяти контроллер кэша разбирает адрес на составляющие, так как это было в кэше прямого отображения, а затем производит поиск в наборе таблиц кэш-памяти элемента, номер страницы которого совпадает с нужным. Если такового нет, то нужно решать, какой элемент подлежит замещению новыми данными. Обычно контроллер реализует политику удаления наиболее давно неисполь-

зовавшихся элементов (LRU — Least Recently Used). Все элементы одного уровня (с одним номером строки) связываются в список, каждый элемент которого имеет поле, с помощью которого реализуется принцип удаления по давности использования. В процессорах Pentium применяется двух- или четырехвходовая наборно-ассоциативная архитектура кэш-памяти с размером строки 32 байта (в Net Burst — 64 байта). Полную информацию о характеристиках кэш-памяти конкретного процессора можно получить из документации на него.

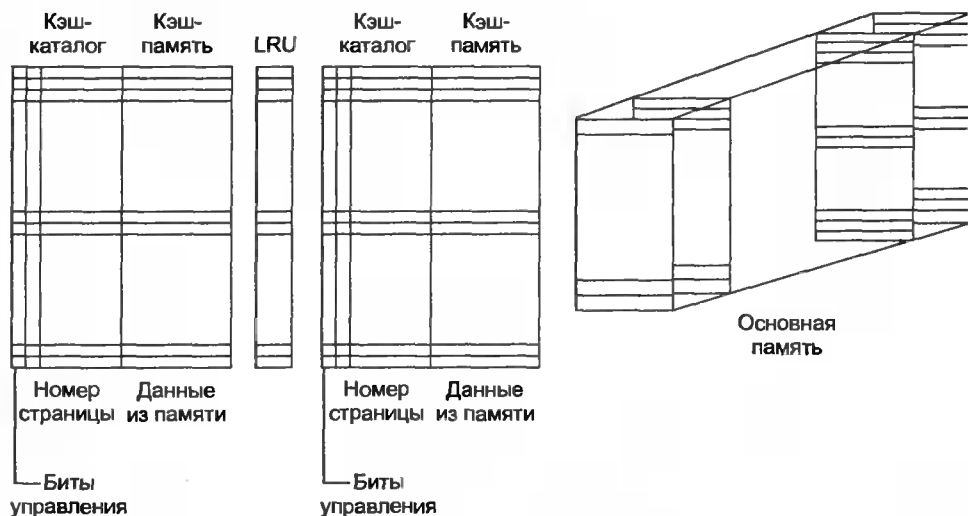


Рис. 8.2. Архитектура наборно-ассоциативной кэш-памяти (2 входа)

К чему весь этот разговор о кэш-памяти? Дело в том, что, зная особенности ее функционирования и предоставляемые конкретным процессором средства, можно значительно повысить скорость работы программ определенного класса. Данные кэша первого уровня доступны для чтения и перезаписи за один такт работы процессора. Работа с памятью обходится значительно «дороже». Поэтому важно представлять себе потенциал конкретного процессора и его системы команд относительно возможностей влияния на процесс кэширования.

Начиная с i80486, фирма Intel вводит средства (в виде команд **INVD**, **INVLPG** и **WBIND**), с помощью которых можно влиять на процесс кэширования. Далее, особенно по мере развития мультимедийных возможностей компьютера, архитектура подсистемы кэширования процессоров Intel и номенклатура средств, доступных программисту для воздействия на процесс кэширования, только расширялась и совершенствовалась. На рис. 8.3 и 8.4 показаны архитектуры подсистем кэширования процессоров семейств P6 и Net Burst.

Основное отличие подсистемы кэширования процессора семейства Net Burst от процессоров семейства P6 состоит в отсутствии кэша команд 1 уровня (L1). В семействе Net Burst он заменен на кэш трасс (см. ниже раздел «Оптимизация для процессоров семейств P6 и Net Burst (Pentium 4)»).

Перечислим особенности работы кэш-памяти, которые можно использовать для увеличения скорости работы программ.

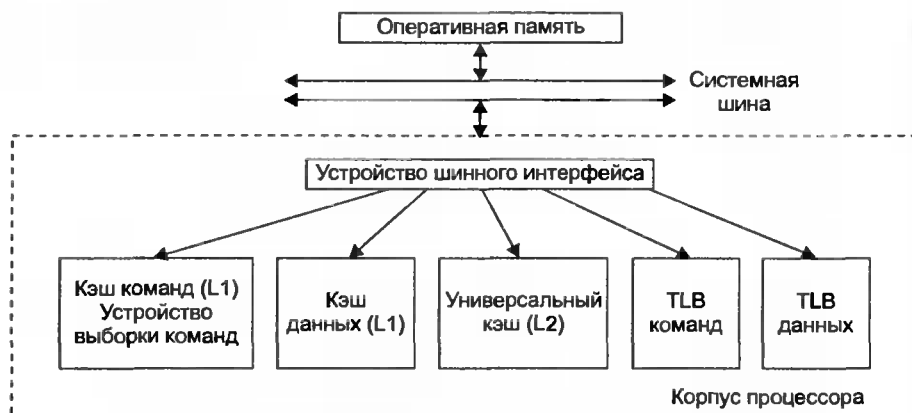


Рис. 8.3. Архитектура подсистемы кэширования процессоров семейства R6

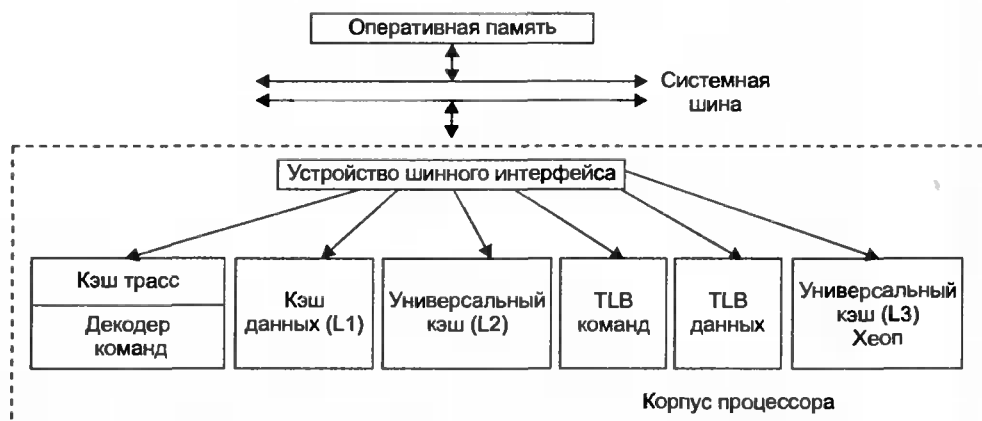


Рис. 8.4. Архитектура подсистемы кэширования процессоров семейства Net Burst

- Чтение даже одного байта из памяти приводит к чтению всего кэш-ряда (32 или 64 байта). Кэш-ряд всегда выровнен на величину, кратную 32 или 64. Получается, что чтение остальных 31 или 63 байтов не стоило нам ничего. Отсюда вывод: данные, обрабатываемые вместе, нужно располагать рядом и выравнивать на границу, кратную 32 или 64.
- Необходимо учитывать закономерности заполнения наборно-ассоциативной памяти кэш-памяти данными. Каждый кэш-ряд этой памяти ассоциирован с вполне определенными адресами оперативной памяти. На заполнение конкретного кэш-ряда могут претендовать только данные, имеющие одинаковые биты «номер строки» адреса (см. рис. 8.1). Поэтому для наборно-ассоциативной памяти с двумя входами в кэше могут присутствовать два блока данных (32 или 64 байта) с одинаковыми значениями битов адреса «номер строки». Для наборно-ассоциативной памяти с четырьмя входами, соответственно, в кэше

могут присутствовать до четырех блоков. Если, к примеру, при наличии наборно-ассоциативной кэш-памяти с двумя входами ведется одновременная работа с тремя блоками памяти, имеющими одинаковое значение поля адреса «номер строки», то это будет приводить к постоянной перезаписи обоих кэш-рядов, соответствующих данному адресу. Поэтому при планировании программы, нужно следить за тем, чтобы адреса массивов данных, с которыми ведется работа на достаточно локальном участке программы, имели разные значения битов адреса «номер строки». Это сэкономит десятки тактов работы процессора. Чтобы не очень сильно задумываться о значении адресов, в качестве самой простой рекомендации можно посоветовать держать данные оптимизируемого участка программы вместе, в пределах одного блока, размером не более общего размера кэш-памяти. Если все же данных много, то нужно собрать вместе наиболее часто используемые из них и разместить их в пределах одного блока.

- Для процессоров Pentium, вплоть до PIII, существует отдельный кэш кода первого уровня. Для его эффективной работы также по возможности нужно располагать критические и часто востребуемые фрагменты кода так, чтобы они могли разместиться в его границах.
- Использовать команды процессора, прямо учитывающие особенности работы с кэш-памятью. Существует несколько команд предварительной записи в кэш-память данных из оперативной памяти — `PREFETCHQ`, `PREFETCH1`, `PREFETCH2`, `PREFETCHA`. В MMX- и XMM-расширениях процессоров Pentium предусмотрено несколько команд, которые позволяют процессору работать с данными минуя кэш, тем самым исключая загрязнение кэша разовыми данными. Пример таких команд — `MASKMOVQ`, `MOVNTQ` и т. д.
- Критичные циклические участки кода есть смысл делать как можно меньшими с тем, чтобы они полностью размещались в кэше.

Конвейер процессоров семейства P6 (Pentium II/III)

При оптимизации программы для процессоров семейства P6 можно опереться на особенности конвейера для этих процессоров. Его схематическое представление приведено на рис. 8.5.

Процессор с помощью *устройства выборки команд* ищет очередные команды для выборки в кэше команд первого уровня. Если его нет, то одновременно организуется поиск в кэше второго уровня и формирование адреса очередной инструкции на шине адреса. Если требуемый код присутствует в кэше второго уровня, то он помещается в кэш первого уровня, а процесс выборки команд из памяти прекращается. После выборки команд из кэша первого уровня вычисляется их длина и осуществляется их *декодирование*. Суть декодирования — перевод каждой команды в последовательность более элементарных действий, называемых *микрооперациями*. Помните, сколько было разговоров о преимуществах RISC- и CISC-архитектур процессоров лет десять назад. Потом все это поутихло. Причина — начиная с процессоров семейства P6, Intel реализовала основные идеи RISC-архитектуры внутри CISC-архитектуры. Перед декодированием определяется длина

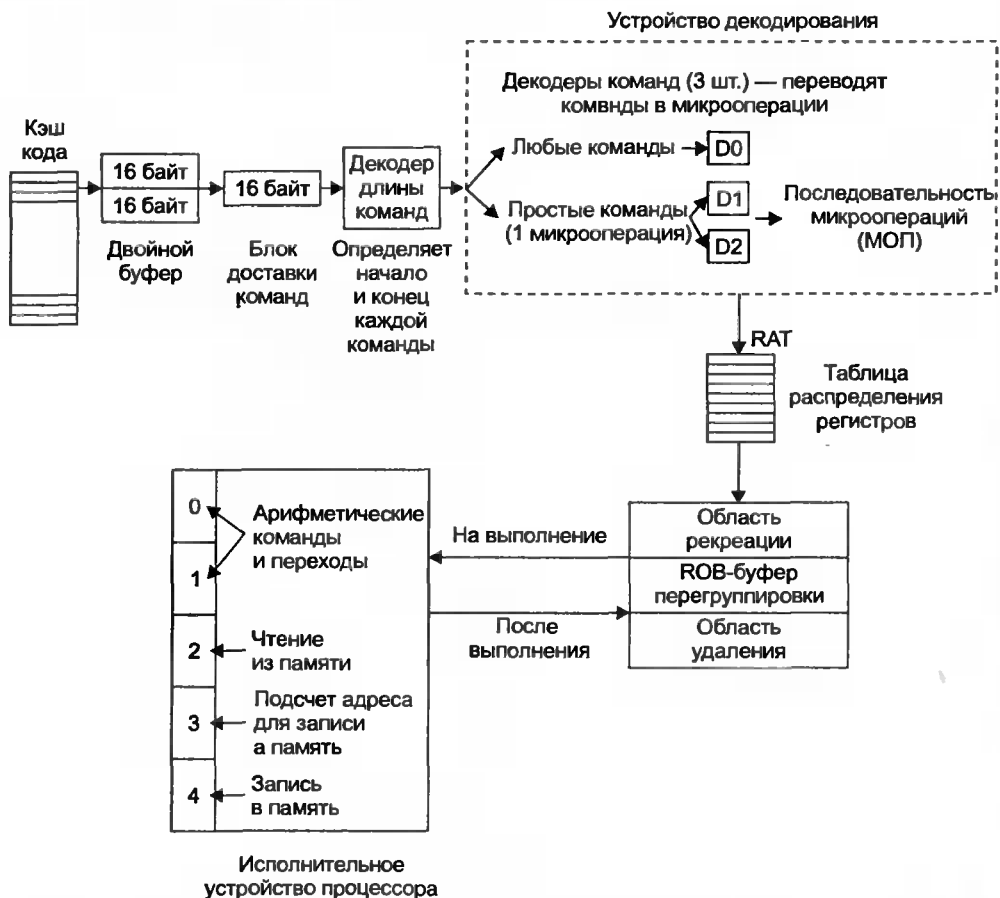


Рис. 8.5. Архитектура конвейера процессоров семейства P6

команд и производится их идентификация. Декодированием команд одновременно занимаются три декодера. Эти декодеры неравнозначны. Только один из них — D0 может обрабатывать любые команды. Остальные два декодера — D1 и D2, способны обрабатывать только команды, генерирующие одну микрооперацию. Декодер D0 преобразует исходную команду в последовательность из двух и более (оптимально — четырех) микроопераций. Информация о последовательности микроопераций для реализации конкретной машинной команды содержится в блоке *микропрограммного управления*. Кроме команд, декодеры обрабатывают также префиксы команд.

Декодер команд в состоянии формировать до шести микроопераций за такт — по одной от простых декодеров и до четырех от сложного декодера. Для достижения наибольшей производительности работы декодеров необходимо, чтобы на их входы поступали команды, декодируемые шестью микрооперациями в последовательности $4 + 1 + 1$. Поэтому, в целях оптимизации, есть смысл переупорядочи-

вать исходный набор команд таким образом, чтобы группы команд формировали последовательности микроопераций по схеме $4 + 1 + 1$. Тогда не будет простоев блока декодирования. Количество микроопераций в той или иной команде можно узнать из таблицы, приведенной в файле среди материалов, прилагаемых к книге. В ней приняты следующие обозначения:

- цифра от 1 до 4 — количество микроопераций, в которые декодируется команда;
- меньше 4 (<) — в зависимости от операндов команда декодируется в 1–4 микрооперации;
- больше 4 (>) — команда декодируется в более чем 4 микрооперации.

Команды MMX- и XMM-расширений, которые не обращаются к памяти, декодируются в одну микрооперацию. Команды MMX- и XMM-расширений, осуществляющие доступ к памяти, декодируются в две микрооперации.

Микрооперации выполняются пятью работающими параллельно исполнительными устройствами процессора: два из них обрабатывают все арифметические операции, третье производит операцию чтения из памяти, четвертое выполняет вычисление адреса для чтения из оперативной памяти, пятое записывает в память.

Планирование и учет особенностей исполнения команд

Еще и еще раз нужно подчеркнуть, что универсальных (для всех процессоров) рецептов оптимизации достаточно мало. Если планировать глубокую оптимизацию кода, то предварительно нужно изучить руководство по оптимизации от фирмы-производителя, которым сопровождается любой процессор. Например, для процессора Pentium 4 это [49], доступное в электронном виде на сайте <http://developer.intel.com>. В нем можно найти рекомендации по использованию (или неиспользованию) тех или иных команд и особенности их выполнения. В этом разделе приведем некоторые советы, которые целесообразно учитывать в своей работе.

Циклические конструкции и переходы

При оптимизации программы особое внимание следует уделять циклам и переходам. Причина этому — конвейеризация вычислений. Условие перехода всегда становится известно, когда на конвейер уже подана определенная последовательность команд программы. Современные процессоры имеют средства для предугадывания переходов для того, чтобы среди этой последовательности команд оказалась команда, соответствующая адресу перехода. Практикуются различные подходы к принятию решения о предстоящем переходе. Формированием содержимого конвейера занимается устройство выборки/декодирования. Как только очередная команда распознана как команда перехода, должно быть принято решение о месте программы, с которого следует начинать выборку очередных команд на конвейер. Самый простой вариант действий — в расчете на то, что переход планируется назад, он считается наиболее вероятным, и устройство выборки выбирает команды,

начиная с этого адреса. И наоборот, переход вперед считается наименее вероятным, и устройство выборки продолжает обработку команд, следующих за данной командой перехода. Если предсказание выполнено неправильно, то цена этой ошибки велика — процессору приходится сбрасывать весь конвейер и начинать выборку команд с действительного адреса перехода. Поэтому в современных процессорах технике прогнозирования переходов уделяется большое внимание.

Вид предсказания, разобранный выше, называется *статическим*. Используется также и другой подход, называемый *динамическим* предсказанием. Его суть в поддержании процессором специальной *таблицы переходов*. Эта таблица организована подобно кэш-таблице. Каждая строка таблицы содержит адрес одной из ранее встретившихся команд перехода и поле *признаков*, на основе которого делается прогноз о предстоящем переходе. В простейшем случае это поле состоит из одного бита, содержимое которого показывает, был ли в действительности переход в последний раз, когда выполнялась данная команда. Для более сложных алгоритмов поле признаков может состоять из нескольких битов.

Зная особенности, заложенные в подсистеме предсказания переходов процессора, можно подыграть им при написании программы и в определенной степени оптимизировать выполнение циклов и переходов. Ниже перечислены некоторые рекомендации:

- По возможности нужно вообще избегать переходов, в том числе вызовов процедур. Небольшие по размеру процедуры лучше вообще заменять макросами. При этом гарантированно экономятся такты на вызов процедуры и возврат из нее. В отдельных случаях имеет смысл даже дублировать код, на который совершается переход.
- Для перемещения данных по условию можно использовать команды `CMOVcc` и `FCMOVcc`.
- Команда `LOOP` требует много тактов процессора для своего выполнения, поэтому ее следует заменять на сочетание команд `DEC ECX` и `JNZ` в конце цикла.
- Предпочтительны циклы с проверкой условия выхода в конце тела цикла. При большом количестве повторений сглаживается свойство таких циклов — выполнение его тела хотя бы один раз.
- Для небольших циклов с ограниченным, тоже небольшим, числом повторений часто бывает полезным разворачивание цикла, то есть просто повтор тела цикла нужное (заранее известное) количество раз.

Выравнивание данных и кода

Для облегчения процессору работы с данными в памяти их необходимо выравнивать. Выравниванию подлежат все виды данных: одиночные переменные простых типов, поля структур, переменные в стеке. Невыравненные переменные заставляют процессор совершать дополнительные циклы для доступа к ним. Данные поступают в процессор через кэш данных. При выравнивании необходимо учитывать размер операнда и тип процессора.

Размер операнда, битов	Семейство процессоров		
	P5	P6	Net Burst
8	Любой адрес	Любой адрес	Любой адрес
16	Кратный 2	Кратный 2	В пределах выравненных 4 байтов
32	Кратный 4	Кратный 4	Кратный 4
64	Кратный 8	Кратный 8	Кратный 8
80	Кратный 8	Кратный 16	Кратный 16
128	—	Кратный 16	Кратный 16

При размещении локальных переменных процедуры в стеке возможны потери производительности из-за проблем выравнивания данных в стеке. Поэтому важно соответствующий кадр стека выравнивать на определенную границу (8/16/32/64 бита). Для этого в вызываемой процедуре можно использовать следующие коды пролога и эпилога:

```

my_proc    align    16          : выравнием точку входа в процедуру
proc
local_frame equ    20          : к примеру, зарезервируем 20 байтов
                                : для локальных переменных
:----- пролог
push       ebp                : сохраним предыдущий кадр
                                : (для вложенных процедур)

mov        ebp, esp
and        ebp, 0fffffff0h    : выравнивание на 64-битовую границу
mov        [ebp], esp         : сохраним старый указатель на стек
sub        esp, local_frame   : место для локальных переменных
:----- доступ к локальным переменным относительно ebp
:....
:----- эпилог
mov        esp, [ebp]         : восстановили старый указатель на стек
pop        ebp                : восстановили указатель на предыдущий
                                : кадр стека

ret
my_proc    proc

```

Оптимизация для процессоров семейств P6 и Net Burst (Pentium 4)

Системы команд процессоров семейства P6 и процессора Pentium 4 содержат ряд команд, которые следует исключить из оптимизируемой программы по причине их медлительности:

- команды **LEAVE** и **ENTER** изначально были предназначены для поддержки блочной структуры программы (работы с кадром стека) для использования в языках верхнего уровня;
- цепочечные команды без префикса повторения выполняются медленнее команд загрузки и сохранения и по возможности их следует заменять эквивалентными фрагментами кода. Из всех цепочечных команд наиболее быстрыми являются

те, у которых достаточно большое число повторений и которые оперируют двойными словами (то есть оптимально работают с кэшем);

- команда **LOOP** не рекомендуется для организации циклов;
- команды работы с битами **BT**, **BTC**, **BTR**, **BTS**, а также команды сдвига **RCR** и **RCL** со значением счетчика сдвига выполняются более медленно, чем обычные логические команды и команды сдвига на единицу, и их следует избегать;
- команду целочисленного умножения **IMUL** также желательно заменять на эквивалентный код (**SHL**, **ADD**, **SUB**, **LEA** или масштабирование);
- использовать, где возможно, команды, одним из операндов которых является регистр **EAX** (аккумулятор) или регистр в принципе. Эти команды, как правило, короче и выполняются быстрее;
- команда **LEA** хорошо подходит для вычисления сумм, разностей и произведений (например, вместо **IMUL**) для трех операндов:
 - **lea eax, [eax+ebx]** — складываются **EAX** и **EBX**;
 - **lea eax, [eax+eax*2]** — вычисляется утроенное значение **EAX**;
 - **lea eax, [eax-ebx+2]** — к разности **EAX** и **EBX** прибавляется 2;
 - **lea ebx, [edx+1]** — производится присвоение значения одного регистра другому с одновременным инкрементом.

Следует заметить, что **LEA** с масштабным множителем на Pentium 4 выполняется медленнее, чем на процессорах семейства P6. Поэтому для масштабирования рекомендуется использовать последовательность команд **ADD**.

Отметим основные отличия Pentium 4 от Pentium III:

- Введение *кэша трасс* (trace cache) в Pentium 4 снизило важность выстраивания команд по схеме $4 + 1 + 1$.
- Предпочтительными являются переходы вперед и циклы.
- Уменьшено влияние зависимостей между регистрами; желательно использовать полные 32-разрядные регистры и команды для работы с ними; 8- и 16-битовые регистры фактически противопоказаны. Чтобы поместить 8- и 16-битовые операнды в 32-разрядный регистр, можно использовать команду **MOVZX**.
- Изменились задержки времени выполнения некоторых команд. В Pentium 4 в два раза быстрее стали выполняться следующие арифметические команды: **ADD**, **SUB**, **CMR**, **TEST**, **AND**, **OR**, **XOR**, **NEG**, **NOT**, **SARHF**, **MOV**. И наоборот, увеличилось время выполнения команд сдвига, циклического сдвига и традиционно медленного целочисленного умножения. Определенной осторожности в своем применении требует команда **LEA**.
- Для Pentium 4 следует избегать применения команд **INC** и **DEC**. Вместо них нужно использовать команды **ADD** и **SUB** соответственно. Дело в том, что команды **INC** и **DEC** модифицируют только подмножество флагов в регистре **EFLAGS**, поэтому может возникнуть нежелательная зависимость последующих перед ними команд от предыдущих. Команды **ADD** и **SUB** перезаписывают все флаги, что исключает появление подобных зависимостей.
- Не стоит без особой надобности употреблять сложные команды типа **ENTER**, **LEAVE** или **LOOP**, то есть те команды, которые требуют более четырех микроопераций для своего выполнения.

- Команды сдвига — традиционно медленные, и с ними лучше не связываться. На Pentium 4 они выполняются еще медленнее. Если нет возможности обойтись без них, то лучше использовать самые быстрые из них — сдвиги на 1 (например, разбивать сдвиг на два разряда на два одиночных сдвига).
- Целочисленное умножение выполняется сопроцессором, поэтому это очень медленная операция и, как уже говорилось, ее желательно заменять сложениями (в разумных пределах) или командой LEA (см. выше).
- Очистку регистра следует производить командами XOR, SUB и PXOR. Процессор Pentium 4 при обнаружении в качестве операндов этих команд одинаковых регистров понимает, что от него хотят и, разрывая цепочку зависимостей, очищает нужный регистр.
- Для сравнения некоторой величины с нулем или константой рекомендуется использовать команду TEST. Команда TEST выполняет логическую операцию «И» над операндами. Ее отличие от команды AND в том, что результат никуда не записывается, производится только установка флагов (см. описание команд в учебнике). Сравнивать операнды можно и командой CMP, но TEST имеет меньший размер и хорошо спаривается. В то же время, не нужно забывать о том, что многие команды при формировании своего результата анализируют значения операндов и устанавливают флаги, поэтому в ряде случаев об их операндах и результате можно судить по состоянию этих флагов и применять команды условного перехода Jcc. Например, при использовании MOV и LEA для анализа операндов предпочтительна команда TEST.
- Для Pentium 4, в котором имеется кэш трасс, можно улучшить производительность при работе с операндами в памяти. Это касается случаев, когда схема команды следующая: коп рег, mem. В Pentium 4 (как, впрочем, и в семействе P6) существует пул рабочих регистров (ROB), доступных только процессору. Всего имеется порядка 40 регистров, в которых и хранятся операнды всех команд, загруженных в процессор. Поэтому, вопреки мнению, что команда быстрее всего выполняется над операндами в регистрах, для данной схемы команд нет смысла производить дополнительную пересылку второго операнда в регистр.
- В соответствии с общими рекомендациями следует использовать команды, требующие минимального количества микроопераций и работающие с регистровыми операндами.
- Если программа активно ведет работу с памятью, то желательно сгруппировать эти команды в блоки операций чтения и операций записи.
- Для ускорения выборки операндов из памяти хорошо подходят команды предварительной загрузки PREFETCH и mem. С помощью данной команды можно заранее произвести заполнение строки кэш-памяти данными из памяти. Поэтому, если есть возможность, то необходимо вычислять адреса данных в памяти как можно раньше и загружать их в кэш одного или нескольких уровней этой командой.
- Некоторые команды поддерживают *сериализацию* (строго последовательное выполнение). Они начинают свою работу только после завершения всех микроопераций, инициализированных к данному моменту времени. Это команды

работы с дескрипторными таблицами GDT, LDT, IDT, загрузки в системные регистры MOV, работы с кэшем INVD, INVLPG, WBIND и некоторые другие.

- При планировании размещения данных необходимо следить за тем, чтобы поле данных не оказалось разделено по двум строкам кэш-памяти.
- Помните о командах MMX- и XMM-расширений при необходимости выполнения одних и тех же операций над массивами данных. В особенности это касается арифметических, логических операций, операций манипулирования данными. Есть смысл внимательно отнестись к этим командам с тем, чтобы использовать их не только в коде ассемблера, но и при программировании ассемблерных вставок в программах на языках высокого уровня.
- Цепочки зависимости. В процессорах семейства P6 и Pentium 4 реализовано выполнение кода не по порядку его следования в исходной последовательности команд. Попросту говоря, в процессоре практически одновременно (разными исполнительными устройствами) «по предположению» выполняются несколько команд. Упреждающее выполнение блока команд тем быстрее, чем меньше команды зависят друг от друга. Если команды фрагмента программы зависят друг от друга по данным, то говорят, что между ними образовалась цепочка зависимости:

```
mov    eax, op1
sub    eax, op2
add    eax, op3
```

Это типичная цепочка зависимых команд при арифметических вычислениях. В случае команд SUB и ADD это не так страшно, так как они требуют мало микроопераций. Что же касается команд умножения и деления, то здесь ситуация усугубляется, так как количество микроопераций для их выполнения исчисляется десятками, и лучше выполнить три сложения, чем умножение на три. Умножения и деления следует по возможности заменять более «дешевыми» эквивалентами — сдвигами, сложениями, командой LEA.

Большинство приведенных выше положений достаточно относительно. Относительно относительно стиля программирования, целевого процессора, целей разработки и т. п. Объективную оценку качества оптимизации кода может дать только эксперимент. Существуют специальные средства измерения производительности, но во многих случаях достаточно собственных «доморощенных» средств. Одним из таких инструментов является пример программы-профайлера, приведенный в следующем разделе. Подробное ее обсуждение вы найдете в учебнике. Приступая к оценке кода, необходимо помнить об эффекте «выполнения кода в первый раз». Он имеет место при выполнении циклов или наличии в программе повторяющихся участков памяти. Исполнение кода первый раз занимает гораздо больше времени, чем его последующие исполнения. Объяснение этому простое — необходимость первичного заполнения кэш-памяти. При повторном исполнении кода он уже находится в кэш-памяти первого или второго уровня, и его выборка в процессор производится намного быстрее. Более того, при первом проходе кода информации о переходах нет в буфере предсказания переходов, и вероятность правильного предсказания переходов очень мала.

Профайлер

Функционирование профайлера (иначе профилировщика) основано на работе со счетчиком меток реального времени TSC (Time Stamp Counter), представляющего собой регистр, содержимое которого инкрементируется с каждым тактом процессорного ядра. Всякий раз при аппаратном сбросе (сигналом RESET) отсчет в этом счетчике сбрасывается в ноль. Разрядность регистра обеспечивает счет без переполнения в течение сотен лет. Счетчик продолжает отсчет как при исполнении инструкции HLT, так и при остановке процессора по сигналу STPCLK# (для энергосбережения). Чтение счетчика обеспечивает команда RD TSC, которую можно сделать привилегированной (доступной лишь при CPL = 0) установкой бита CR4.TSD. Присутствие счетчика TSC определяется по инструкции CPUID (EAX = 1). Если в результате ее вызова бит 4 регистра EDX равен 1, то процессор поддерживает счетчик меток реального времени TSC.

Команда RD TSC (Read from Time Stamp Counter — чтение 64-разрядного счетчика меток реального времени TSC (Time Stamp Counter)) не имеет операндов. Машинный код этой команды — 0F 31. Команда проверяет состояние второго бита регистра CR4.TSD (Time Stamp Disable — отключить счетчик меток реального времени):

- если CR4.TSD = 0, то выполнение команды RD TSC разрешается на любом уровне привилегий;
- если CR4.TSD = 1, то выполнение команды RD TSC разрешается только на нулевом уровне привилегий.

Когда выполнение команды разрешено на текущем уровне привилегий, производится сохранение значения 64-битового MSR-счетчика TSC в паре 32-битовых регистров EDI:EAX. Если выполнение команды запрещено, то работа команды заканчивается.

Напишем две макрокоманды, использование которых позволит нам получать количественную оценку работы кода, начиная от полной программы и заканчивая отдельной командой. Эти макрокоманды не будут отличаться компактностью, но этого от них и не требуется, так как они являются лишь средством оценки эффективности оптимизации кода, востребуемым на этапе отладки, и в конечном коде их наличие не требуется.

Применение наших макрокоманд можно подчинить следующему алгоритму. Вызов первой макрокоманды, назовем ее profiler_in, должен зафиксировать момент, относительно которого будет производиться отсчет тактов процессора, то есть в начале профилируемого участка программы. Вызов второй макрокоманды profiler_out должен зафиксировать момент окончания работы на этом участке программы. Необходимо иметь в виду, что это «грязное» время работы программы, по которому можно производить только приблизительную оценку ее скорости работы. Для этого есть внутренние и внешние причины. Внутренняя причина заключается в том, что полученная величина включает в себя время, затраченное на работу некоторых команд, составляющих тело самой макрокоманды. Этот недостаток исправить легко. Что касается внешних причин, то они объективны по отношению к программе пользователя и мешают получению истинного времени профилиро-

вания. Такими внешними причинами являются программы операционной системы, которые могут приостанавливать на время программу пользователя.

Ниже приведены макрокоманды `profiler_in` и `profiler_out` с тестовым примером для проверки их работы. Данные макрокоманды производят корректировку результата своей деятельности, с тем чтобы исключить обсужденные выше внутренние причины «грязного» времени работы программы. Заметим, что не всякий транслятор ассемблера «знает» о новых командах процессора, в том числе и о команде `RDTSCL`. По этой причине мы ее моделируем, инициализируя в сегменте кода два байта значениями машинного кода этой команды — `db 0fh,31h`.

```

:-----+-----
: | Программа: prg08_01.asm. Демонстрация использования макрокоманд |
: | profiler_in и profiler_out для оценки производительности фрагментов кода. |
:-----+-----
:
: ...
:-----+-----
: | Макрокоманда: rdtsc. Эмуляция rdtsc. |
:-----+-----
rdtsc      macro
           db      0fh, 31h
endm

:-----+-----
: | Макрокоманда: bin_dec_fpu. Вывод на консоль десятичного числа. |
:-----+-----
: | Вход: string_bin_qword – адрес 64-битовой ячейки (описанной dt) |
: |       с преобразуемым двоичным целым числом. |
: |       string_pack – адрес 80-битовой ячейки (описанной dt). |
: |       в которую сохраняется упакованное 10-е значение. |
: |       adr_string_pack – ячейка с адресом string_pack (описанная dd). |
: |       len_string_pack – длина string_pack. |
: |       adr_string – ячейка с адресом string (описанная dd). |
: |       В string помещаются символы десятичных цифр для вывода. |
: |       len_string – размер string (18 байтов). |
:-----+-----
bin_dec_fpu macro string_bin_qword:REQ, string_pack:REQ,\
                  adr_string_pack:REQ, len_string_pack:REQ,\
                  adr_string:REQ, len_string:REQ
           local cycl
:----- преобразуем bin->dec
           finit
           fild string_bin_qword : заносим в сопроцессор
                                           : двоичное целое число
           fbstp string_pack      : извлекаем упакованное десятичное
:----- распакуем
           lds si, adr_string_pack
           add si, len_string_pack - 2 : на конец string_pack
                                           : (18 упакованных десятичных цифр)
           les di, adr_string
           mov cx, 9 : 9 пар упакованных десятичных цифр
cycl:      xor ax, ax
           std      : string_pack обрабатываем с конца
           lodsb    : в al очередные 2 упакованные десятичные цифры
:----- распаковываем – ah = младшая, al = старшая
           shl ax, 4
           rol al, 4
           or ax, 3030h : преобразуем в символическое представление
           xchg ah, al : ah = младшая, al = старшая
           cld : в string записываем с начала
           stosw
           dec cx
           jz cycl

```



```

        popad
;----- выводим
        bin_dec_fpu val_2_q, string_pack, adr_string_pack, \
                    len_string_pack, adr_string, len_string
endm
.data
val_2      label dword
val_2_q    dq      0
val_1      label dword
val_1_q    dq      0
string_pack dt      0          ; исходное значение из val_2_q
                                ; в упакованном десятичном формате
        len_string_pack = $ - string_pack
adr_string_pack dd     string_pack
string      db      18 dup(0)  ; максимальный результат состоит
                                ; из 18 десятичных цифр
        len_string = $ - string
adr_string  dd      string
.code
;....
;----- профилируем выполнение команд работы со стеком
        profiler_in val_1
        push     eax
        pop      eax
        profiler_out val_1, val_2
exit:    ;.... ; выход из программы

```

Составьте тестовые примеры и «поиграйтесь» с данной программой. Обратите внимание, что при задании пустой последовательности команд между парой макросов `profiler_in` и `profiler_out` все равно получается некоторая величина профилирования. Она постоянна, ее источник — сами команды `RDTSC`, которые требуют тактов процессора для своего исполнения. Эту величину можно скорректировать разными способами, но можно и не трогать, а учитывать при подведении окончательных результатов тестирования нужного вам фрагмента кода. На компьютере автора эта величина равна 2.

В представленной программе для визуализации результатов профилирования разработана макрокоманда `bin_dec_fpu`, производящая перевод значения из двоичной системы в десятичную.

В качестве вывода к данной главе хотелось бы еще раз подчеркнуть, что это далеко не полный разговор об оптимизации, а лишь фрагменты верхушки айсберга этой безбрежной темы. Впрочем, их достаточно для того, чтобы задуматься о необходимости и способах помощи процессору в исполнении написанного вами кода.

Глава 9

Вычисление CRC

Где начало того конца, которым оканчивается начало?

Козьма Прутков

В своей практической работе каждый пользователь наверняка сталкивался с ситуацией, когда неблагоприятные условия перемещения файлов (любым способом) приводили к порче последних. Типичное проявление этой ситуации — сообщение об ошибке при попытке чтения некоторого файла. Причина — внесенная извне техническая ошибка, приведшая к нарушению целостности исходной информации. Существует много методов для исправления подобных ошибок, но прежде чем исправлять, необходимо эти ошибки обнаружить. Для этого также существуют определенные методы [44], основанные на избыточности передаваемой информации, что позволяет не только выявлять наличие факта искажения информации, но и в ряде случаев устранять такие искажения. Перечислим наиболее известные из методов обнаружения ошибок передачи данных.

- *Посимвольный контроль четности*, называемый также *поперечным* (рис. 9.1), подразумевает передачу с каждым байтом дополнительного бита, принимающего единичное значение по четному или нечетному количеству единичных битов в контролируемом байте. Может использоваться два типа контроля четности — на четность и нечетность. Алгоритм вычисления контрольного бита при контроле на четность предполагает его установку таким образом, чтобы общее количество битов в битовой последовательности (включая и сам бит четности) было четным. И наоборот, контроль на нечетность подразумевает установку контрольного бита так, чтобы общее количество битов в битовой последовательности (включая и сам бит четности) являлось нечетным. Посимвольный контроль четности прост как в программной, так и в аппаратной реализации, но его вряд ли можно назвать эффективным методом обнаружения ошибок, так как искажение более одного бита исходной последовательности резко снижает вероятность обнаружения ошибки передачи. Этот вид контроля обычно реализуется аппаратно в устройствах связи.
- *Поблочный контроль четности*, называемый *продольным*. Схема данного контроля (рис. 9.2) подразумевает, что для источника и приемника информации за-

ранее известно, какое число передаваемых символов будет рассматриваться ими как единый блок данных. При этом для каждой позиции разрядов в символах блока (поперек блока) рассчитываются свои биты четности, которые добавляются в виде обычного символа в конец блока. Схема выполнения продольного контроля по-прежнему предполагает наличие у каждого из этих символов дополнительного бита четности (см. выше). По сравнению с посимвольным контролем четности поблочный контроль четности обладает большими возможностями по обнаружению и даже корректировке ошибок передачи, но все равно ему не удастся обнаруживать определенные типы ошибок. Кроме того, этот вид не реализуется эффективно в аппаратных решениях, в силу чего редко используется. Главное его достоинство в том, что с него можно начинать рассмотрение идеи обнаружения ошибок на уровне блоков передаваемой информации.

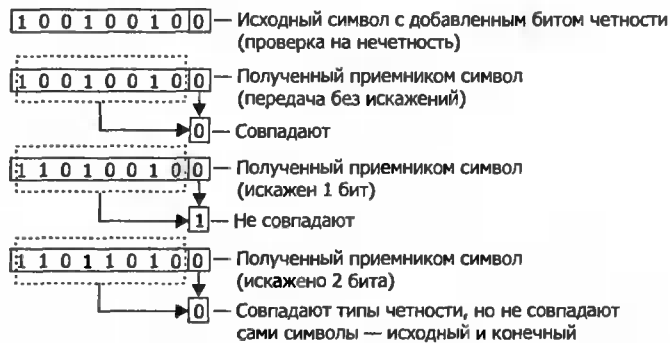


Рис. 9.1. Схема выполнения посимвольного контроля четности при передаче информации

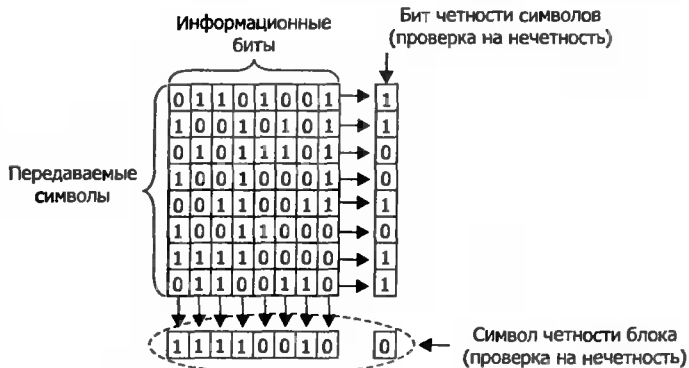


Рис. 9.2. Схема выполнения поблочного контроля четности при передаче информации

■ **Вычисление контрольных сумм.** В отличие от рассмотренных выше методов, для метода контрольных сумм нет четкого определения алгоритма. Каждый разработчик трактует понятие *контрольной суммы* по-своему. В простейшем виде контрольная сумма — это арифметическая сумма двоичных значений контролируемого блока символов. Но такой метод обладает практически теми же недостатками, что и предыдущие, самый главный из которых — нечувствитель-

ность контрольной суммы к четному числу ошибок в одной колонке и самому порядку следования символов в блоке.

- *Контроль циклически избыточным кодом — CRC (Cyclical Redundancy Check).* Это гораздо более мощный и широко распространенный метод обнаружения ошибок передачи информации. Он обеспечивает обнаружение ошибок с вероятностью до 99%. Кроме того, данный метод обладает рядом других полезных моментов, которые могут найти свое воплощение в практических задачах. Рассмотрению этого метода и будет посвящено дальнейшее изложение.

Сама по себе аббревиатура «CRC» знакома многим пользователям компьютера, особенно тем, кому приходится часто переносить свои архивные файлы посредством гибкого диска. В один прекрасный день попытка распаковки архивного файла наверняка приводила к выводу на экран сообщения о том, что у этого файла какое-то неправильное значение CRC. Что же представляет собой это загадочное значение CRC, какую пользу можно извлечь из знаний о нем? В данном разделе мы постараемся с этим разобраться, тем более, что поставленная задача является хорошей возможностью очередной раз продемонстрировать возможности и преимущества ассемблера по обработке данных на уровне битов, строк битов, последовательности байтов (в том числе и неопределенной длины).

CRC (Cyclic Redundancy Code) — последовательность битов, полученная по определенному алгоритму на основании другой (исходной) битовой последовательности. Главная особенность (и практическая значимость) значения CRC состоит в том, что оно однозначно идентифицирует исходную битовую последовательность и поэтому используется в различных протоколах связи, таких как HDLC и ZMODEM, а также для проверки целостности блоков данных, передаваемых различными устройствами. В силу этих свойств алгоритм вычисления CRC часто реализуется на аппаратном уровне. Если взять пример с архиватором, то его работа в общем случае заключается в следующем: архиватор упаковывает файлы в соответствии с некоторым алгоритмом архивации, вычисляя для каждого обрабатываемого файла значение CRC. После этого сжатые файлы могут множество раз копироваться, пересылаться по сети, в том числе с использованием электронной почты, и т. д. В процессе своих путешествий файл может столкнуться с различными неприятными воздействиями внешней среды, например с неисправным диском, искажением его внутреннего содержимого во время передачи по сети и т. п. Эти изменения не обязательно должны быть глобальными, они могут касаться всего одного бита. Когда приходит время, пользователь распаковывает архив, при этом архиватор в первую очередь проверяет целостность файлов в нем. Архиватор опять по содержимому файла вычисляет его CRC и сравнивает полученное значение с тем значением CRC, которое было вычислено при упаковке файла. Если они равны, то считается, что целостность файла не была нарушена, и он распаковывается, в противном случае, если новое и старое значения CRC не совпадают, то считается, что архивный файл поврежден, и процесс его распаковки завершается. Необходимо отметить, что CRC не обязательно рассчитывать для больших массивов данных, каким является файл. Его можно вычислять и для отдельных строк текста и даже слов с целью организации простейшего контроля целостности и отождествления символьных (числовых) последовательностей. Более того, из рассмотре-

ния ниже вы увидите, что алгоритм вычисления CRC, по сути, является еще одним методом хэширования, которые мы подробно рассматривали в разделе «Таблицы с вычисляемыми входами» главы 2. Таким образом, алгоритм вычисления CRC имеет много достоинств, которые могут найти применение в самых различных практических задачах.

Основная идея вычисления CRC заключается в следующем. Исходная последовательность байтов, которой могут быть и огромный файл, и текст размером в несколько слов и даже символов, представляется единой последовательностью битов. Эта последовательность делится на некоторое фиксированное двоичное число. Интерес представляет остаток от деления, который и является значением CRC. Все, что теперь требуется, — это некоторым образом запомнить его и передать вместе с исходной последовательностью. Приемник данной информации всегда может таким же образом выполнить деление и сравнить его остаток с исходным значением CRC. Если они равны, то считается, что исходное сообщение не повреждено, и т. д. Но этот лишь общая схема. Реальный алгоритм вычисления CRC использует особые правила арифметики, в соответствии с которыми производятся все вычисления, назовем их *правилами CRC-арифметики*. В силу принципиальной важности этих правил для нашего изложения коротко рассмотрим их отличия от правил обычной двоичной арифметики.

CRC-арифметика

Расчеты CRC ведутся в двоичной системе счисления. При проведении CRC-вычислений используется специальная *CRC-арифметика*, которая, по сути, является полиномиальной арифметикой по модулю 2. Полиномиальная арифметика по модулю 2 — это еще один из видов арифметик, востребованных для решения задач в определенной предметной области и отличающихся от привычной двоичной арифметики с циклическим переносом отсутствием переносов и вычислением всех коэффициентов по модулю 2 [5]. В главе 10 «Расширение традиционной архитектуры Intel» этого практикума при рассмотрении MMX-команд вы познакомитесь с одной из таких альтернативных арифметик — *арифметикой с насыщением*. Пока же остановимся на особенностях CRC-арифметики.

Итак, как отмечено выше, в основе CRC-арифметики лежит полиномиальная арифметика [5, 44]. По определению, *полином* — линейная комбинация (сумма) произведений целых степеней заданного набора переменных с постоянными коэффициентами. Частный случай — полином, содержащий одну переменную:

$$u(x) = u_n \cdot x_n + \dots u_1 \cdot x_1 + u_0 \cdot x_0.$$

Здесь u_n, u_1, u_0 — элементы некоторой алгебраической системы S , называемые *коэффициентами*; x — переменная полинома, которую можно рассматривать как формальный символ без определенного значения. Алгебраическая система S обычно представляет собой множество целых или рациональных чисел в диапазоне $0 \dots m - 1$ со сложением, вычитанием и умножением, выполняемыми по модулю m . Для нашего рассмотрения особенно важна полиномиальная арифметика по модулю 2, в которой каждый коэффициент полинома равен одному из двух значений — 0

или 1. Например, шестнадцатеричное значение 0e3h может быть представлено следующим полиномом:

$$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0.$$

Если ввести в качестве переменной $x = 2$, то получим следующий двоичный полином:

$$1 \times x^7 + 1 \times x^6 + 1 \times x^5 + 0 \times x^4 + 0 \times x^3 + 0 \times x^2 + 1 \times x^1 + 1 \times x^0.$$

В этом полиноме, строго говоря, значение x не играет особой роли, так как данное двоичное число можно представить полиномом в другой системе счисления, например шестнадцатеричной: $E \times x^1 + 2 \times x^0$, где $x = 16$. Заметим, что в том и другом случае цифры 0, 1, E, 2 — это просто цифры двоичной и шестнадцатеричной систем счисления.

Так как слагаемые двоичного полинома с нулевыми коэффициентами не дают никакого вклада в конечный результат, то их можно попросту отбросить, оставив только слагаемые, переменные которых имеют единичные коэффициенты:

$$\begin{aligned} &1 \times x^7 + 1 \times x^6 + 1 \times x^5 + 0 \times x^4 + 0 \times x^3 + 0 \times x^2 + 1 \times x^1 + 1 \times x^0 = \\ &= 1 \times x^7 + 1 \times x^6 + 1 \times x^5 + 1 \times x^1 + 1 \times x^0 = \\ &= x^7 + x^6 + x^5 + x^1 + x^0. \end{aligned}$$

Здесь $x = 2$. Над полиномами можно производить арифметические операции: сложение, умножение и вычитание. Процессы выполнения этих операций для полиномиальной арифметики и обычной арифметики многократной точности (см. главу 1) похожи. Главное отличие в том, что из-за отсутствия связи между коэффициентами полинома понятие переноса в полиномиальной арифметике отсутствует.

Например, для умножения 7h (0111b) на 5h (0101b) выполняются действия:

$$\begin{aligned} (x^2 + x^1 + x^0) \times (x^2 + x^0) &= (x^4 + x^3 + x^2 + x^2 + x^1 + x^0) = \\ &= x^4 + x^3 + x^2 + x^2 + x^1 + x^0 + x^0 = x^4 + x^3 + 2 \cdot x^2 + x^1 + 2 \cdot x^0. \end{aligned}$$

Для того чтобы получить в ответе 35, мы должны x взять равным 2 и привести коэффициенты у членов полинома $2 \cdot x^0$ и $2 \cdot x^2$ к двоичным, сделав перенос соответствующих единиц в старшие разряды:

$$\begin{array}{rcl} 01010 & \text{переносы} & \\ + & & \\ 11111 & \text{результат} & \\ \hline 100011 & = 35_{10} & \end{array}$$

В результате получим двоичный полином $x^5 + x^1 + x^0$.

Эти рассуждения призваны сформулировать очередной тезис о том, что переносы, как и в обычной арифметике, можно выполнять в случае, когда известно основание системы счисления. То есть до тех пор, пока мы не знаем x , мы не можем производить и переносы. В приведенном выше примере мы не знаем, что $2 \cdot x^2$ и $2 \cdot x^0$ на самом деле являются x^3 и x^1 до того момента, пока не известно, что $x = 2$. То есть в полиномиальной арифметике коэффициенты при разных степенях изолированы друг от друга и отношения между ними не определены. Из-за этого возникает первая странность полиномиальной арифметики — операции сложения и вычитания в ней абсолютно идентичны, и вместо них можно смело оставлять одну.

Например, сложение по правилам полиномиальной арифметики по модулю 2, будем ее далее называть CRC-арифметикой, будет выполнено так:

```

11111011
+
11001010
-----
00110001

```

Из этого примера видны правила сложения двоичных разрядов в арифметике с отсутствием переносов:

$0 + 0 = 0$; $0 + 1 = 1$; $1 + 0 = 1$; $1 + 1 = 0$.

Операцию вычитания демонстрирует следующий пример:

```

11111011
-
11001010
-----
00110001

```

Правила выполнения вычитания в арифметике с отсутствием переносов:

$0 - 0 = 0$; $0 - 1 = 1$; $1 - 0 = 1$; $1 - 1 = 0$.

Сравнение примеров для сложения и вычитания полиномов по модулю 2, а также правил, по которым они выполняются, показывает, что эти две операции CRC-арифметики идентичны и по принципу формирования результата они аналогичны команде ассемблера XOR. Цель, которой достигают всеми этими условностями, — исключить из поля внимания все величины (путем заемов/переносов), лежащие за границей старшего разряда.

Умножение в арифметике с отсутствием переносов также выполняется с учетом особенностей CRC-сложения:

```

1101
×
1011
-----
1101
1101
0000
1101
-----
1111111

```

Видно, что в самом умножении специфики нет, а вот сложение промежуточных результатов производится по правилам CRC-сложения.

Для нашего рассмотрения особый интерес представляет операция деления, так как в основе любого алгоритма вычисления CRC лежит представление исходного сообщения в виде огромного двоичного числа, делении его на другое двоичное число и использовании остатка от этого деления в качестве значения CRC. Деление — самая сложная из операций CRC-арифметики. Здесь необходимо ввести так называемое *слабое понятие размерности*: число X больше или равно числу Y , если оба

значения имеют одинаковую размерность и позиции их старших битов единичны, то есть соотношение остальных битов X и Y для операции сравнения не имеет значения. Рассмотрим примеры операции деления, причем для лучшего понимания сделаем это в двух вариантах: вначале вспомним, как выполняется обычное деление (по правилам двоичной арифметики с циклическим переносом), а затем выполним собственно CRC-деление.

Возьмем произвольную двоичную последовательность и разделим ее на некоторое число по правилам двоичной арифметики с циклическим переносом (рис. 9.3).

По правилам CRC-арифметики деление для приведенных выше исходных данных даст следующий результат (рис. 9.4).

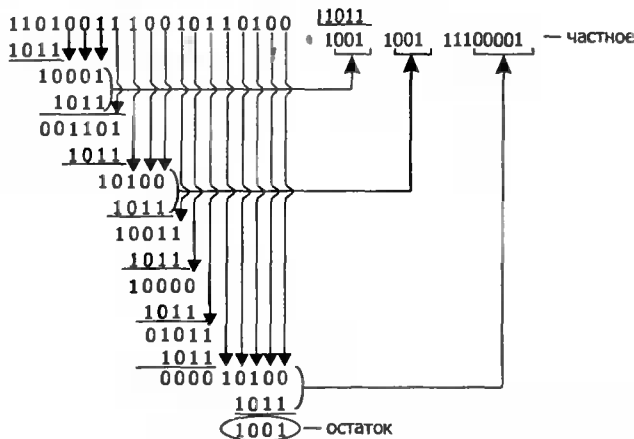


Рис. 9.3. Схема вычисления двоичного деления (с циклическим переносом)

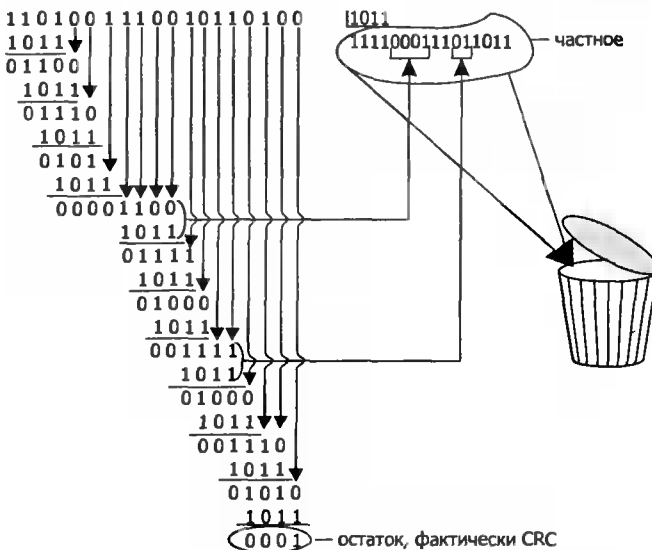


Рис. 9.4. Схема вычисления CRC-деления

Как видите, результаты обычного и CRC-делений отличаются. Главное, чтобы вы подметили особенность выполнения CRC-деления. Особое внимание обратите на порядок формирования промежуточных результатов в процессе деления. Снос двоичных разрядов из делимого продолжается до тех пор, пока размерности промежуточного битового значения и делителя не сравняются, а их старшие разряды не станут равными 1. После этого над двоичными разрядами выполняется побитовое CRC-вычитание. Соотношение разрядов не важно, главное — это одинаковая размерность и единичные старшие биты. В этом случае считается, что уменьшаемое больше вычитаемого, что влечет за собой включение 1 в частное и выполнение операции CRC-вычитания. Это как раз и есть проявление *слабого понятия размерности*. Корзина для мусора, показанная на рис. 9.4, означает тот факт, что частное для процесса вычисления CRC-битовой последовательности не играет никакой роли.

Реальные двоичные последовательности являются результатом сцепления порой огромного количества отдельных байтов (символов), образуя одно большое двоичное число, для представления которого нужно использовать двоичные полиномы огромных степеней. При этом каждый бит в подобной последовательности произвольной длины представляется в виде коэффициента длинного полинома. Например, для представления 128-разрядного блока необходим полином, состоящий из 1024 слагаемых, а для 1024-битового блока требуется полином уже с 8192 слагаемыми. В терминах полиномиальной арифметики двоичное число, сформированное в результате подобной сцепки составляющих блока данных, называется *полиномом данных (сообщения)* и обозначается как $D(x)$ [5, 44]. В алгоритме вычисления CRC вводится еще несколько полиномов и соотношений между ними:

- *порождающий полином* $G(x)$ — предварительно особым образом выбранный полином, на который делится исходный полином сообщения;
- *полином-частное* $Q(x)$ — полином, получившийся в качестве частного от деления полиномов $D(x)/G(x)$;
- *полином-остаток* $R(x)$ — полином, получившийся в качестве остатка от деления полиномов $D(x)/G(x)$.

Между перечисленными полиномами существуют следующие отношения:

$$\begin{aligned} D(x) &= Q(x) \times G(x) + R(x), \\ Q(x) &= (D(x) - R(x))/G(x). \end{aligned}$$

Эти соотношения приводят к следующим основополагающим для дальнейшего рассмотрения тезисам:

- операция деления двух двоичных полиномов $D(x)/G(x)$, где $G(x) \neq 0$, дает в качестве результата полином-частное $Q(x)$ и полином-остаток $R(x)$, удовлетворяющие условию: $D(x) = Q(x) \times G(x) + R(x)$;
- остаток от деления двух полиномов $R(x)$ является двоичным числом, которое после вычитания из $D(x)$ дает в результате еще один полином, делящийся без остатка на $G(x)$; частное $Q(x)$, которое образуется в результате этого деления, отбрасывается за ненадобностью, а полином-остаток $R(x)$ называют CRC (*Cyclic Redundancy Code*).

Из приведенного выше описания общей схемы вычисления CRC возникает ряд вопросов: что представляет собой этот магический делитель $G(x)$, каков его размер? Выбор порождающего полинома $G(x)$ — достаточно сложная задача. Перечислим некоторые важные свойства, которые должны учитываться при выборе.

- Число разрядов (количество членов) в полиноме-остатке $R(x)$ непосредственно определяется длиной самого порождающего полинома $G(x)$. Выбор $G(x)$ длиной n гарантирует, что полином-остаток от деления $R(x)$ будет иметь разрядность не более, чем $n - 1$. Это следует из общего свойства операции деления, которое предполагает, что остаток от деления должен быть меньше делителя.
- Порождающий полином $G(x)$ должен быть полиномиально простым. Это означает его неделимость нацело на полиномы со значением в диапазоне от 2 до самого себя.
- Способность порождающего полинома $G(x)$ к выявлению ошибок, специфичных для передачи данных по каналам связи. Это такие ошибки, как ошибки в одном, двух битах, нечетном количестве битов, а также ошибки блока битов. Выше было отмечено, что более простые методы обнаружения ошибок передачи не способны обнаружить с достаточной степенью вероятности большинство таких типов ошибок.

Для большей части алгоритмов вычисления CRC значение порождающего полинома известно заранее и, сверх того, утверждено соответствующими стандартами. Поэтому программисту без особой надобности не стоит терять времени и силы на изобретение нового значения порождающего полинома $G(x)$. Приведем значения некоторых широко известных полиномов, используемых на практике [44].

- $x^{16} + x^{12} + x^5 + x^0$ — полином 1021h, встроенный в протокол XMODEM и производных от него протоколах передачи данных. Он стандартизован в спецификации V.41 МККТТ «Кодонезависимая система контроля ошибок».
- $x^{16} + x^{15} + x^2 + x^0$ — полином 8005h, используемый в протоколе двоичной синхронной передачи фирмы IBM. Он также стандартизован в приложении «Процедура коррекции ошибок для оконечного оборудования линии с использованием асинхронно-синхронного преобразования» к стандарту v.42 МККТТ. Этот же полином широко известен как полином, реализуемый в алгоритме вычисления CRC — CRC16.
- $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^6 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0$ — полином 04c11db7h, используемый в алгоритме вычисления CRC — CRC32 и также специфицированный в стандарте v.42 МККТТ. Этот полином, в частности, находит применение в технологии локальных вычислительных сетей Ethernet. Необходимо отметить, что вычисление по алгоритму CRC32 зачастую проводят и с другим полиномом: 0edb88320h: $x^{32} + x^{31} + x^{30} + x^{29} + x^{27} + x^{26} + x^{24} + x^{23} + x^{21} + x^{20} + x^{19} + x^{15} + x^9 + x^8 + x^5$. К последнему полиному прибегают различные архиваторы. Необходимо заметить, что полином 0edb88320h — это просто зеркальное отражение полинома 04c11db7h.

В заключение привлеку внимание читателя к тому, почему выгодно увеличивать число разрядов CRC. Выше уже говорилось, что алгоритм вычисления CRC, по сути своей, является одним из возможных (и неплохих) алгоритмов хэширования. Разрядность порождающего полинома $G(x)$ 16 битов обеспечивает до 65 535 значений хэш-функции. Увеличение разрядности полинома $G(x)$ до 32 битов приводит к расширению набора значений хэш-функции уже до 4 294 967 295.

С полиномом связано еще одно понятие — *степени полинома*, которое по определению является номером позиции его старшего единичного бита (считая с нуля). Например, для полинома 1011 из приведенного выше примера (см. рис. 9.4) степень равна 3.

В качестве выводов следует сказать, что CRC-арифметика отличается от двоичной отсутствием переносов/заемов, а CRC-вычитание и сложение выполняются по тем же правилам, как и команда ассемблера XOR, что и обуславливает ее важную роль при вычислении значений CRC.

Прямой алгоритм вычисления CRC

После всех этих рассуждений мы готовы к тому, чтобы осмысленно воспринять общую схему реального алгоритма расчета CRC — алгоритма прямого (поразрядного) вычисления CRC. При этом удобно CRC-алгоритм рассматривать с точки зрения двух сторон-участников процесса: источника — объекта, формирующего сообщение для передачи, и приемника — объекта, который принимает сообщение и проверяет его целостность.

Действия источника следующие.

1. Выбрать полином P , в результате автоматически становится известной его степень N .
2. Добавить к исходной двоичной последовательности N нулевых битов. Это добавление делается для гарантированной обработки всех битов исходной последовательности.
3. Выполнить деление дополненной N нулями исходной строки S на полином P по правилам CRC-арифметики. Запомнить остаток, который и будет являться CRC.
4. Сформировать окончательное сообщение, которое будет состоять из двух частей: собственно сообщения и добавленного в его конец значения CRC.

К примеру, вычисление по этому алгоритму CRC для исходной последовательности 1101001110010110100 (см. рис. 9.4) и сама окончательная последовательность на стороне источника должны выглядеть так, как показано на рис. 9.5.

Из рисунка видно, что в начале вычисления исходная последовательность 1101001110010110100 дополняется нулями в количестве, равном степени полинома ($P = 1011$ — степень полинома $N = 3$): 1101001110010110100 + 000. При выполнении CRC-деления эти дополнительные биты гарантируют, что все биты исходной последовательности примут участие в процессе формирования значения CRC. Результирующая последовательность получается равной исходной последовательности, дополненной значением CRC: 1101001110010110100+011. Заметим, что длина

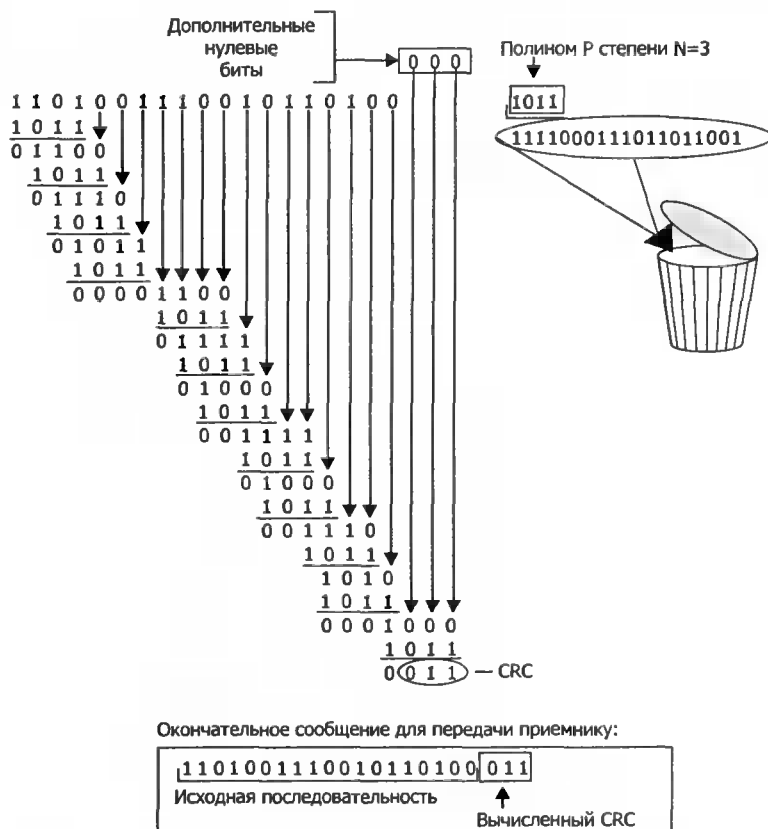


Рис. 9.5. Схем формирования выходного сообщения из исходного с использованием CRC-алгоритма

присоединяемого к исходной последовательности значения CRC должна быть равна степени полинома, даже если CRC, как в нашем случае, имеет ведущие нули. Это очень важный момент, понимание которого является ключом к проникновению в суть процессов, происходящих на стороне приемника при получении и определении целостности исходного сообщения.

Действия алгоритма для приемника просты — выполнить деление полученной последовательности на полином. При делении нет необходимости дополнять исходную последовательность нулями, тем более что на практике соблюдение этого условия крайне неудобно. Приемник попросту осуществляет CRC-деление полученной исходной строки (дополненной в конце исходным значением CRC) на полином и анализирует остаток. Если остаток от этого деления нулевой, то исходная последовательность не была повреждена во время передачи. В противном случае существует очень большая вероятность нарушения целостности исходной последовательности, и нужно принимать дополнительные меры по выяснению и исправлению ситуации. Одной из таких мер может быть попытка восстановления нужного значения CRC.

Описанный выше алгоритм вычисления значения CRC называется прямым и чаще всего реализуется аппаратно. Но тем не менее, для совершенствования навыков программирования на ассемблере составим реализующий его пример программы. Хотя эффективность этой программы не слишком высока, у нее есть две учебные цели:

- показать в виде программной реализации суть алгоритма вычисления CRC и самого CRC-деления;
- подготовить себя к пониманию более совершенных алгоритмов расчета CRC, к которым относится, в частности, рассматриваемый ниже табличный алгоритм.

Для компьютерной реализации алгоритмов вычисления CRC удобно выбирать полиномы со степенями, кратными 8 (то есть размерности регистров) — 8, 16, 24, 32 или даже 64. В этом случае можно подобрать команды из системы команд процессора, наиболее оптимально реализующие алгоритмы вычисления CRC. В качестве полинома выберем один из рекомендуемых полиномов (см. ниже) — 4003. И еще одно важное замечание — степень полинома определяет размерность регистра, используемого в алгоритме, при этом считается, что старший (всегда единичный) бит полинома находится сразу за левой границей регистра. В этих условиях программа реализации *прямого алгоритма вычисления CRC* функционирует следующим образом (для лучшего понимания в процессе разбора алгоритма см. рис. 9.6). В регистр побитово вдвигаются биты исходной строки. Это происходит до тех пор, пока при очередном сдвиге слева появится единичный бит. Тогда все содержимое регистра подвергается операции XOR со значением полинома без старшего бита. Далее процесс сдвига и анализа выдвигаемого бита продолжается до того момента, пока не будет выдвинут очередной единичный бит, в результате чего опять между регистром и полиномом выполняется операция XOR, и т. д. После того как последний бит вдвинут в регистр, в него вдвигается количество нулевых битов, равное степени полинома. Этим, как мы не раз уже отмечали, достигается участие всех битов исходной битовой строки в формировании значения CRC. В итоге в регистре остается значение CRC, которое необходимо добавить к исходной строке и передать приемнику.

```

:-----+
:| Программа: prg09_01.asm. Демонстрация прямого алгоритма вычисления CRC |
:|                               (сторона-источник).                         |
:-----+
.data
bit_string      db      "6476c8"          ; исходная битовая последовательность
                                           ; в символах
len_bit_string = $ - bit_string
adr_bit_string  dd      bit_string
polinom         dw      4003h
.code
        ;...
        lds     si, adr_bit_string
        mov     cx, len_bit_string
        mov     bx, polinom
        shl     ebx, 16                    ; подготовим полином к XOR с EAX
ml:      ;----- вложенные циклы
        push    cx
        mov     cx, 8
        lodsb

```

```

m2:      shl     eax, 1
        jnc     m3          : старшие разряды не равны – выполняем
                             : сдвиг (частное нас не интересует)
:----- старшие разряды равны – выполняем XOR
        xor     eax, ebx    : eax(31..16) XOR polinom
m3:      loop   m2
        pop     cx
        loop   m1
:----- вдвигаем нулевые биты числом N
        mov     cl, 24 + 1  : N=16 (степень полинома) + 8 (так как
                             : работаем в eax)+1 (для loop)
m4:      shl     eax, 1
        jnc     m5          : старшие разряды не равны – выполняем
                             : сдвиг (частное нас не интересует)
:----- старшие разряды равны – выполняем XOR
        xor     eax, ebx    : eax(31..16) XOR polinom
m5:      loop   m4
        :...
    
```

В результате вычисления CRC символьной последовательности "6476c8" получим CRC = 35dah.

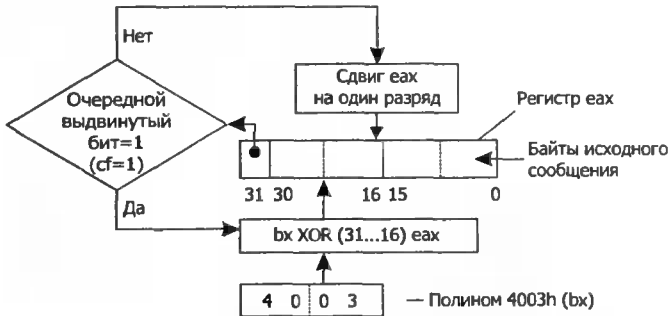


Рис. 9.6. Схема вычисления значения CRC прямым алгоритмом

Для того чтобы смоделировать действия стороны приемника, подходит та же самая программа со слегка измененными исходными данными — к строке `bit_string` добавляем вычисленное значение CRC. После этого под отладчиком наблюдаем за процессом CRC-деления, причем контролируем получаемый остаток. В определенный момент увидим, что он стал нулевым — это свидетельствует о том, что исходная последовательность не была изменена. Для эксперимента можно изменить значения одного или более битов исходной последовательности и посмотреть, что получится.

```

;-----+
;| Программа: prg09_02.asm. Демонстрация прямого алгоритма вычисления CRC |
;| (сторона-приемник). |
;-----+
.data
bit_string db "6476c8"          : исходная битовая последовательность
                                     : в символах
len_bit_string = $ - bit_string
adr_bit_string dd bit_string
polinom dw 4003h
.code
main:
:.... : см. предыдущую программу
exit:  :.... : выход из программы
    
```

Табличные алгоритмы вычисления CRC

Очевидный недостаток прямого метода — большое количество операций сдвига, исключаяющих операций «ИЛИ» (XOR) и операций условного перехода, которые выполняются для каждого бита исходного сообщения. Поэтому на практике используется другой способ расчета CRC, называемый *табличным*.

Основы

Для того чтобы лучше понять суть табличного алгоритма вычисления CRC, обратимся опять к прямому методу, точнее к той схеме его вычисления (см. рис. 9.6), которая была реализована в приведенной выше программе.

Из нее видно, что для текущего содержимого старшей половины регистра EAX можно прогнозировать, как будет изменяться содержимое его битов по мере их сдвига. Для этого достаточно подвергнуть анализу биты EAX начиная с самого старшего. Пронумеруем старшие 8 битов EAX как $a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0$. При следующем сдвиге (см. рис. 9.6) прямой алгоритм определяет, будет ли произведена операция XOR операнда с полиномом $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ в ВХ ($a_7 = 1$) или нет ($a_7 = 0$). Если выдвинутый бит был равен 1, то прежнее содержимое старшей половины регистра EAX будет подвергнуто операции XOR с соответствующими битами полинома. В противном случае, если выдвинутый бит был равен 0, значения битов будут не изменены, а просто сдвинуты влево на один разряд. В принципе, имея большое желание, можно рассчитать заранее, каким будет содержимое k -го бита в k -й итерации сдвига. К примеру, значение нового старшего бита, определяющего действия алгоритма в следующей итерации, можно рассчитать по содержимому двух старших битов старшего байта исходного операнда — $a_6 \text{ XOR } a_7 \text{ AND } b_7$, где b_7 — старший бит полинома (всегда равный единице).

Теперь остановимся для того, чтобы рассмотреть и обсудить очередную схему (рис. 9.7).

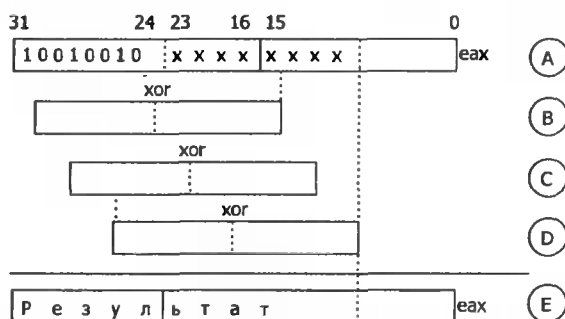


Рис. 9.7. Влияние на регистр EAX серии операций XOR при вычислении CRC

Из рассуждений выше следует, что если взять для рассмотрения старший байт операнда, то по его содержимому можно однозначно предположить, сколько операций XOR и когда будет выполнено (см. рисунок). Обозначим старшую половину регистра EAX как переменную A , а операнды со значениями полинома, объединяемые с A при единичном состоянии очередного выдвигаемого слева бита, обозна-

чим соответственно как B , C , D (помним, что $B = C = D$). Тогда формирование результата E можно представить формулой:

$$E = ((A \ll \text{сдвиги}) \text{ XOR } B) \ll \text{сдвиги} \text{ XOR } C) \ll \text{сдвиги} \text{ XOR } D$$

Здесь $\ll \text{сдвиги}$ представляют собой значение от 0 до 7 и определяются текущим содержимым старшего байта операнда (регистра EAX). Благодаря ассоциативному свойству операции XOR тот же самый результат можно получить, если предварительно выполнить операцию XOR над полиномами B , C , D с соответствующими значениями сдвигов, а затем результат объединить по XOR с A :

$$F = \ll \text{сдвиги} \text{ XOR } (B \ll \text{сдвиги} \text{ XOR } C \ll \text{сдвиги} \text{ XOR } D)$$

$$E := A \text{ XOR } F$$

Отсюда следуют важные выводы:

- величина F является совершенно точно предсказуемой по содержимому старшего байта операнда;
- если величина F определяется содержимым старшего байта операнда и собственно значением полинома, то существует всего 256 возможных значений этой величины (по количеству значений, представимых беззнаковым байтом);
- исходя из первых двух положений, величина F не зависит от значения операнда и может быть рассчитана заранее, при этом результаты ее расчетов поддаются сведению в таблицу (!).

Вот мы и выяснили, на чем основано название табличного алгоритма расчета CRC. Теперь со знанием сути дела приведем его общее описание (рис. 9.8). В качестве основы для рассуждений по-прежнему используем программу прямого вычисления значения CRC и соответствующую схему (см. рис. 9.6).

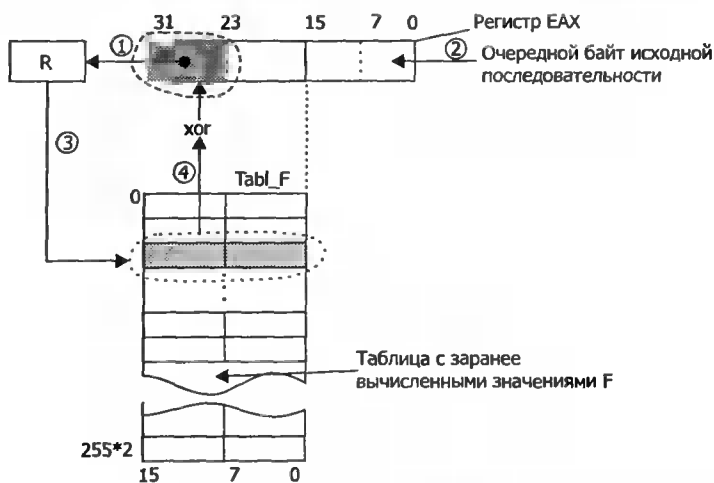


Рис. 9.8. Общая схема табличного алгоритма

На схеме, показанной на рисунке, цифрами обозначена последовательность шагов табличного алгоритма. Шаги 1 и 2 выполняются одновременно и означают, что старший байт из регистра EAX выдвигается в переменную R , а младший байт этого регистра заполняется очередным байтом исходной последовательности. Зна-

чение переменной *R* используется на шаге 3 в качестве индекса в таблице *TABL_F* для извлечения 16-битового значения, которое на шаге 4 будет объединено операцией XOR с содержимым старших 16 битов регистра *EAX*. Таким образом, в отличие от прямого алгоритма, процесс преобразования вырастает до уровня байтов и содержит три операции: сдвига, доступа к таблице для извлечения нужного значения и операции XOR извлеченного значения с содержимым старшей половины *EAX*. По окончании процесса в старшей половине *EAX* будет находиться значение CRC. Сообщение по-прежнему должно быть выровненным, то есть дополненным количеством битов, равным степени полинома, или для данного случая — 16. Для практических приложений это крайне неудобно, и решение проблемы будет показано чуть ниже. Пока же разработаем программу вычисления содержимого таблицы на основе полинома 1021h степени 16.

```

+-----+
:| Программа: prg09_03.asm. Вычисление содержимого таблицы
:| на основе полинома 1021h степени 16.
+-----+
.data
tabl_16      dw      256 dup (0)      ; CRC-таблица
len_tabl_16 = $ - tabl_16
adr_tabl_16  dd      tabl_16
polinom      dw      1021h
.code
;...
les         di, adr_tabl_16
add         di, len_tabl_16 - 2
std                     ; идем назад по таблице
mov         cx, 255
mov         bx, polinom
m1:         xor         ax, ax
mov         ah, cl      ; индекс в таблице для вычисления CRC
push        cx          ; вложенные циклы
mov         cx, 8
m2:         shl         ax, 1
jnc         m3          ; старшие разряды не равны — выполняем
; сдвиг (частное нас не интересует)
;----- старшие разряды равны — выполняем XOR
xor         ax, bx      ; ax XOR polinom
m3:         loop        m2
pop         cx
stosw
loop        m1
;...

```

В результате работы этой программы область памяти *tabl_16* будет инициализирована таблицей значений, которые могут быть задействованы в схеме вычисления значения CRC исходной последовательности (см. рис. 9.8).

Прямой табличный алгоритм CRC16

Необходимость дополнения исходной строки завершающими нулевыми байтами представляет большое неудобство при реализации общей схемы табличного алгоритма. Поэтому на практике выбирают другую схему вычисления CRC, не требующую завершения исходного сообщения нулевыми байтами. В этой схеме очередной байт исходной строки объединяется операцией XOR со старшим байтом регистра, выдвинутым с левой стороны этого регистра. Полученное в результате объединения значение используется в качестве индекса для доступа к CRC-табли-

це. Извлекаемое из CRC-таблицы значение объединяется операцией XOR с содержимым регистра. Описанный алгоритм называют прямым табличным алгоритмом. Ниже приведены его схема (рис. 9.9) и демонстрационная программа.



Рис. 9.9. Схема вычислений CRC с использованием прямого табличного алгоритма

На схеме вычислений CRC с использованием прямого табличного алгоритма цифрами обозначена последовательность шагов вычисления CRC.

1. Выдвижение старшего байта регистра AX в байтовую ячейку.
2. Выполнение операции XOR над выдвинутым на шаге 1 в байтовую ячейку старшим байтом регистра AX и очередным байтом исходной строки.
3. Полученное на шаге 2 значение используется в качестве индекса для доступа к элементу CRC-таблицы.
4. Извлеченное из CRC-таблицы значение объединяется по XOR со значением в регистре AX.
5. Результат выполнения на шаге 4 операции XOR помещается обратно в регистр AX.

После обработки последнего байта исходной строки регистр AX содержит значение CRC. Программа вычисления CRC посредством прямого табличного алгоритма приведена ниже.

```

+-----+
: | Программа: prg09_04.asm. Вычисление CRC с использованием
: | прямого табличного алгоритма. |
+-----+
.data
bit_string      db      "6476c8"          ; исходная битовая последовательность
                                           ; в символах
len_bit_string  = $ - bit_string
adr_bit_string  dd      bit_string
tabl_16         dw      256 dup (0)        ; CRC-таблица
len_tabl_16     = $ - tabl_16
adr_tabl_16     dd      tabl_16
polynom        dw      1021h
.code
:....
:----- расчитываем CRC-таблицу
les     di, adr_tabl_16

```

```

        add    di, len_tabl_16 - 2
        std    : идем назад по таблице
        mov    cx, 255
        mov    bx, polinom
m1:     xor     ax, ax
        mov    ah, cl      : индекс в таблице для вычисления CRC
        :----- вложенные циклы
        push   cx
        mov    cx, 8
m2:     shl    ax, 1
        jnc    m3          : старшие разряды не равны — выполняем
                           : сдвиг (частное нас не интересует)
        :----- старшие разряды равны — выполняем XOR
        xor    ax, bx      : ax XOR polinom
m3:     loop   m2
        pop    cx
        stosw
        loop   m1
        :----- закончили расчет CRC-таблицы
        xor    ax, ax
        xor    bx, bx
        lds    si, adr_bit_string
        mov    cx, len_bit_string
m4:     mov    bl, ah
        shl    ax, 8
        xor    bl, [si]
        shl    bx, 1
        xor    ax, tab1_16[bx]
        xor    bh, bh
        inc    si
        loop   m4
        :...

```

Рассмотрением этого алгоритма введение в проблему вычисления CRC можно было бы и закончить. Все существующие модели вычисления CRC являются, по сути, различными модификациями описанного выше табличного алгоритма. Эти модификации преследуют разные цели, перечислим некоторые из них:

- переход от цикла по всем битам к циклу по большим порциям данных — байтам, словам и т. д.;
- повышение разрядности порождающего полинома;
- отражение особенностей функционирования аппаратуры передачи данных (наличие этой цели обусловлено тем, что микросхемы, поддерживающие работу последовательного порта компьютера, передачу байта начинают с его младших битов, поэтому описанные ниже алгоритмы расчета CRC, учитывающие эту особенность, называются зеркальными).

Алгоритмы вычисления CRC получили свое закрепление в некоторых стандартах. Перечислим отличительные особенности основных алгоритмов расчета CRC. Итак, основные алгоритмы вычисления CRC различаются:

- по значению и степени порождающего полинома, при этом значение полинома обычно указывается без ведущей единицы в старшем разряде;
- по начальному содержимому регистра, в котором формируется значение CRC;
- по тому, производится ли перестановка в обратном порядке битов каждого байта перед его обработкой;
- по способу прохода байтов через регистр — способ может быть косвенным или прямым;

- по тому, осуществляется ли перестановка в обратном порядке конечного результата;
- по конечному значению, с которым производится объединение по XOR результата вычисления CRC.

После появления 32-разрядных процессоров наибольшей популярностью стали пользоваться 32-разрядные алгоритмы вычисления CRC. В различных источниках рассматривают два типа таких алгоритмов — *прямой* и *зеркальный*. Рассмотрим их конкретные реализации, рекомендуемые стандартами.

Прямой табличный алгоритм CRC32

Как и любой табличный алгоритм, табличный алгоритм вычисления CRC32 требует задания CRC-таблицы. Ее можно задать в программе статически, явно прописав значения элементов таблицы в сегменте кода, или динамически, вычислив значения элементов таблицы перед началом расчета CRC. Ниже приведена программа вычисления CRC-таблицы для полинома 04c11db7.

```

:-----+
:| Программа: prg09_05.asm. Вычисление CRC-таблицы для полинома 04c11db7. |
:-----+
.data
:----- CRC32-таблица для прямого табличного алгоритма
:      вычисления CRC32
tabl_32_direct dd 256 dup (0)
len_tabl_32_direct = $ - tabl_32_direct
adr_tabl_32_direct dd tabl_32_direct
polinom dd 04c11db7h
.code
:....
les di, adr_tabl_32_direct
add di, len_tabl_32_direct - 4
std ; идем назад по таблице
mov cx, 255
mov ebx, polinom
m1: xor eax, eax
shrd eax, ecx, 8
:----- вложенные циклы
push cx
mov cx, 8
m2: shl eax, 1
jnc m3 ; старшие разряды не равны — выполняем
; сдвиг (частное нас не интересует)
:----- старшие разряды равны — выполняем XOR
xor eax, ebx ; ax XOR polinom
loop m2
pop cx
stosd
loop m1
:....

```

Прямой табличный алгоритм CRC32 удобно рассматривать со стороны участников процесса, как это мы делали выше. Источник выполняет следующие действия.

1. Делает начальную установку регистра, в котором будет производиться формирование CRC, значением 0FFFFFFFFh.
2. Вычисляет значение CRC для каждого байта исходной последовательности, принцип которой показан на схеме (см. рис. 9.9). Читатель понимает, что хотя

эта схема иллюстрирует принцип вычисления CRC16, но для 32-разрядного алгоритма нужно только увеличить размер элементов таблицы до 32 битов и задействовать весь регистр EAX.

3. Объединяет по XOR итоговое значение в EAX со значением 0FFFFFFFFh.
4. Добавляет вычисленное значение в конец двоичной исходной последовательности. После этого его можно передать по назначению.

Ниже приведена программа вычисления CRC32 по прямому табличному алгоритму.

```

;+-----+
;| Программа: prg09_06.asm. Вычисление CRC32 по прямому табличному алгоритму. |
;+-----+

bit_string      db      "123456789"          : исходная битовая последовательность
                                                    : в символах
len_bit_string  dd      0                    : сюда поместим значение CRC32
adr_bit_string  dd      bit_string
tabl_32_direct  dd      256 dup (0)          : CRC32-таблица для прямого табличного
                                                    : алгоритма вычисления CRC32
len_tabl_32_direct = $ - tabl_32_direct
adr_tabl_32_direct dd tabl_32_direct
polinom        dd      04c11db7h
.code
;....
;----- рассчитываем CRC32-таблицу
les     di, adr_tabl_32_direct
add     di, len_tabl_32_direct-4
std     : идем назад по таблице
mov     cx, 255
mov     ebx, polinom
m1:     xor     eax, eax
shrd    eax, ecx, 8
;----- вложенные циклы
push    cx
mov     cx, 8
m2:     shl     eax, 1
jnc     m3
;----- старшие разряды не равны – выполняем
; сдвиг (частное нас не интересует)
xor     eax, ebx
m3:     loop    m2
pop     cx
stosd
loop    m1
;----- закончили расчет CRC32-таблицы
mov     eax, 0ffffffffh
lds     si, adr_bit_string
mov     cx, len_bit_string
xor     ebx, ebx
m4:     shld    ebx, eax, 8
shl     eax, 8
xor     bl, [si]
xor     eax, tabl_32_direct[ebx]
inc     si
loop    m4
;----- запишем crc-32 в конец последовательности
; (или начало, см. обсуждение ниже)
xor     eax, 0xffffffffh
mov     crc_32, eax
; добавляем в конец исходной
; последовательности
;....

```

Значение CRC32 для строки "123456789" равно 9c970409h.

Если для источника стандарт определяет практически однозначную последовательность действий, то для приемника возможны несколько вариантов поведения. Отметим два из них.

В первом варианте критерием для вывода о целостности полученного приемником сообщения является результат сравнения или нулевой результат.

1. Выполнить начальную установку регистра, в котором будет производиться формирование значения CRC.
2. Вычислить значение CRC для каждого байта полученной последовательности, принцип которой показан на схеме (см. рис. 9.9 и замечания выше о различиях CRC16 и CRC32).
3. Объединить по XOR итоговое значение в EAX со значением 0FFFFFFFh.
4. Далее можно сделать одно из двух действий:

- сравнить значение в EAX со значением CRC, полученным вместе с сообщением, — если они равны, то поступившее сообщение идентично исходному;
- пропустить байты со значением CRC через процесс получения CRC наравне с другими байтами — об успехе будет свидетельствовать нулевая величина (этот вариант предпочтительнее, так как не предполагает получение предварительных сведений о длине начальной последовательности без учета исходного значения CRC).

Во втором варианте критерием для вывода о целостности полученного приемником сообщения является фиксированная двоичная константа 6b202ca2h.

1. Выполнить начальную установку регистра, в котором будет производиться формирование CRC, в значение 0FFFFFFFh;
2. Вычислить значение CRC32 для каждого байта полученной последовательности (вместе со значением CRC32 в конце), принцип которой показан на схеме (см. рис. 9.9 и замечания выше о различиях CRC16 и CRC32). Должно получиться фиксированное двоичное значение — 6b202ca2h. Необходимо заметить, что в литературе можно встретить другое число — 0debb20e3h, но у автора получилось первое.

Основное отличие этих двух вариантов в том, что нужно каким-то образом отделять контролируемую строку и ее значение CRC32. Избежать прямого отслеживания длины принимаемой строки на стороне приемника можно путем формирования источником значения CRC32 в начале исходной строки либо вообще вне строки. Последний случай, например, характерен для программы, которая отслеживает целостность файлов в некотором каталоге. Тогда результаты подсчета CRC для каждого файла хранятся в некотором служебном файле. Если такой вариант вас не устраивает, следует написать код приемника для прямого табличного алгоритма так, как это сделано для приемника в зеркальном табличном алгоритме. Попробуйте самостоятельно определить константу, соответствующую успешному принятию приемником исходной строки.

Учитывая практическую важность обоих вариантов и для демонстрации разнообразия подходов к проблеме вычисления CRC32, работу приемника для пря-

мого табличного алгоритма продемонстрируем на примере первого варианта, приемник «зеркального» табличного алгоритма CRC32 разработаем исходя из положений второго варианта. Но вы должны понимать, что правильное понимание принципов расчета CRC позволяет вам при соответствующей доработке кода менять варианты работы приемников. В поле `crc_32` должно быть значение CRC32, рассчитанное предыдущей программой. Автор специально не стал объединять эти программы. Пусть это сделает читатель в нужном для его текущей работы контексте.

```

;-----+
;| Программа: prg09_07.asm. Демонстрация действий приемника |
;| табличного алгоритма CRC32 при анализе поступившего сообщения. |
;-----+
.data
bit_string db "123456789" ; исходная битовая последовательность
; в символах
len_bit_string = $ - bit_string
crc_32 dd 9c970409h ; значение CRC32, рассчитанное источником
; для данной исходной последовательности
adr_bit_string dd bit_string
tabl_32_direct dd 256 dup (0) ; CRC32-таблица для прямого табличного
; алгоритма вычисления CRC32
len_tabl_32_direct = $ - tabl_32_direct
adr_tabl_32_direct dd tabl_32_direct
polinom dd 04c11db7h
.code
;...
;----- рассчитываем CRC32-таблицу
les di, adr_tabl_32_direct
add di, len_tabl_32_direct - 4
std ; идем назад по таблице
mov cx, 255
mov ebx, polinom
m1: xor eax, eax
shrd eax, ecx, 8
push cx ; вложенные циклы
mov cx, 8
m2: shl eax, 1
jnc m3 ; старшие разряды не равны - выполняем
; сдвиг (частное нас не интересует)
;----- старшие разряды равны - выполняем XOR
xor eax, ebx ; ax XOR polinom
m3: loop m2
pop cx
stosd
loop m1
;----- закончили расчет CRC32-таблицы
mov eax, 0ffffffffh
lds si, adr_bit_string
mov cx, len_bit_string
m4: xor ebx, ebx
shld ebx, eax, 8
shl eax, 8
xor bl, [si]
xor eax, tabl_32_direct[ebx]
inc si
loop m4
xor eax, 0ffffffffh
;----- если исходная последовательность цела, то должно получиться
; значение crc32=9c970409h его можно сравнить с исходным
; и на этом закончить работу или продолжить до победного.
; то есть до 0
mov cx, 4 ; 4 (длина crc_32), в si адрес crc_32

```

```

mov     ebx, crc_32
bswap   ebx
mov     crc_32, ebx
m5:     xor     ebx, ebx
        shld    ebx, eax, 8
        shl     eax, 8
        xor     bl, [si]
        xor     eax, tbl_32_direct[bx]
        inc     si
        loop    m5
        ;...           : должен быть 0

```

Заметьте, что для получения нулевого результата нам пришлось использовать команду `BSWAP`, чтобы «перевернуть» значение в поле `crc_32`.

«Зеркальный» табличный алгоритм CRC32

В заключение данного раздела обсудим «зеркальный» вариант табличного алгоритма — алгоритм CRC32 (v.42 МККТТ). Этот вариант вычисления CRC обязан своим возникновением существованию последовательного интерфейса, который посылает биты, начиная с наименее значимого (бит 0) и заканчивая самым старшим (бит 7), то есть в обратном порядке. В «зеркальном» регистре все биты отражены относительно центра. Например, 10111011001 есть отражение значения 10011011101.

Вместо того чтобы менять местами биты перед их обработкой, можно зеркально отразить все значения и действия, участвующие в прямом алгоритме. При этом направление расчетов поменяется — байты теперь будут сдвигаться вправо, полином 04c11db7h зеркально отразится относительно его центра, в результате получится значение 0edb88320h. CRC32-таблица будет зеркальным отражением аналогичной таблицы для прямого алгоритма (рис. 9.10).

Для зеркального алгоритма вычисления CRC32 процесс вычисления такой же, за исключением порядка — сдвиги и заполнение регистра осуществляются вправо. Ниже приведена программа вычисления таблицы для зеркального алгоритма CRC32 и полинома 0EDB88320h.

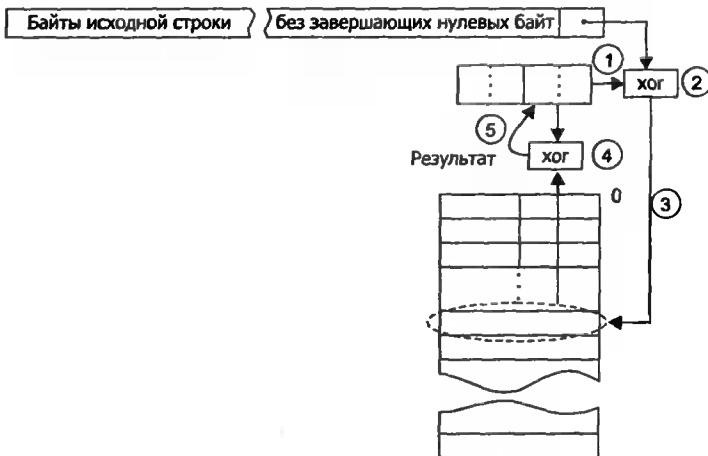


Рис. 9.10. Схема зеркального табличного алгоритма


```

+-----+
| Программа: prg09_08.asm. Вычисление таблицы для зеркального |
| алгоритма CRC32 и полинома 0EDB88320h.                       |
+-----+
.data
tabl_32_mirror dd 256 dup (0) ; CRC32-таблица для зеркального
; табличного алгоритма расчета CRC32
len tabl_32_mirror = $ - tabl_32_mirror
adr tabl_32_mirror dd tabl_32_mirror
polinom dd 0EDB88320h
.code
;...
les di, adr_tabl_32_mirror
add di, len_tabl_32_mirror - 4
std ; идем назад по таблице
mov cx, 255
mov ebx, polinom
m1: xor eax, eax
mov al, cl ; индекс в таблице для вычисления CRC
push cx ; вложенные циклы
mov cx, 8
m2: shr eax, 1
jnc m3 ; старшие разряды не равны – выполняем
; сдвиг (частное нас не интересует)
; ----- старшие разряды равны – выполняем XOR
xor eax, ebx ; ax XOR polinom
m3: loop m2
pop cx
stosd
loop m1
;...

```

Для того, кто обладает солидным багажом знаний по проблеме CRC-вычислений, написание программы, реализующей зеркальный алгоритм, — дело техники. На стороне источника код будет выглядеть так:

```

+-----+
| Программа: prg09_09.asm. Вычисление кода CRC32 на стороне |
| источника для зеркального алгоритма CRC32 и полинома |
| 0EDB88320h.                                             |
+-----+
.data
bit_string db "123456789" ; исходная битовая последовательность
; в символах
len_bit_string = $ - bit_string
crc_32 dd 0 ; сюда поместим значение CRC32
adr_bit_string dd bit_string
tabl_32_mirror dd 256 dup (0) ; CRC32-таблица для зеркального
; табличного алгоритма вычисления CRC32
len tabl_32_mirror = $ - tabl_32_mirror
adr tabl_32_mirror dd tabl_32_mirror
polinom dd 0EDB88320h
.code
;...
; ----- рассчитываем зеркальную CRC32-таблицу
les di, adr_tabl_32_mirror
add di, len_tabl_32_mirror - 4
std ; идем назад по таблице
mov cx, 255
mov ebx, polinom
m1: xor eax, eax
mov al, cl ; индекс в таблице для вычисления CRC
; ----- вложенные циклы
push cx
mov cx, 8

```



```

:----- старшие разряды равны – выполняем XOR
m3:   xor     eax, ebx           ; ax XOR полином
      loop   m2
      pop     cx
      stosd
      loop   m1                ; закончили расчет CRC32-таблицы
      xor     bx, bx
      mov     eax, 0FFFFFFFFh
      lds     si, adr_bit_string
      mov     cx, len_bit_string + 4 ; 4 – длина crc_32
m4:   mov     bl, al
      shr     eax, 8
      xor     bl, [si]
      xor     eax, tabl_32_mirror[bx]
      inc     si
      loop    m4
:----- сравнить – результат должен быть константой 6b202ca2h
      ....

```

Этот вариант работы алгоритма вычисления CRC32 удобен тем, что не нужно знать длину собственно исходной последовательности (без значения CRC). Достаточно просто обработать весь входной поток, не различая в строке завершающую ее подстроку с CRC. Далее необходимо сравнить содержимое регистра EAX с 6b202ca2h. Если эти значения равны, значит, исходная последовательность нарушена не была. Для получения собственно строки достаточно отбросить последние 4 байта сообщения, поступившего к приемнику. И последнее замечание, которое говорит о том, что проблема вычисления CRC неоднозначна для понимания и предоставления большого поля для проведения различных экспериментов и совершенствования существующих алгоритмов. Небольшой поправкой в алгоритме работы источника можно сделать так, что успехом целостности при принятии приемником сообщения может быть не указанная выше константа, а нуль. Для этого источнику достаточно не объединять вычисленное значение CRC32 по XOR с 0ffffffffh, а просто добавить его к исходной последовательности. Оба варианта хороши тем, что не нужно заранее знать длину анализируемого сообщения.

Здесь, пожалуй, можно и закончить обсуждение проблемы вычисления CRC. На примере этой достаточно сложной задачи были в очередной раз продемонстрированы варианты использования средств ассемблера для обработки информации на различных уровнях, вплоть до битового. За дальнейшими подробностями по проблематике вычисления CRC обратитесь к соответствующим источникам [5, 38, 44].

Глава 10

Расширения традиционной архитектуры Intel

Писать музыку не так уж трудно, сложнее всего зачеркивать лишние ноты.

Иоганн Брамс

С появлением процессоров пятого поколения (Pentium, Pentium MMX...) в программной модели процессора Intel следует различать два слоя архитектуры: базовый и модельно-зависимый.

Слой *базовой архитектуры* включает в себя элементы архитектуры, поддержку и неизменность которых производитель (Intel) гарантирует. Это набор и структура основных устройств процессора, системных регистров и регистров общего назначения, архитектура памяти и т. д.

Модельно-зависимый слой архитектуры включает в себя средства, поддержка которых привязана к конкретной модели процессора (как правило, снизу вверх). Для того чтобы программа, исполняемая на конкретном процессоре, могла получить сведения о его возможностях, в систему команд процессоров, поддерживающих IA32 (Intel, AMD и т. д.), включена команда CPUID. Данная команда позволяет программе в любой момент времени запросить сведения о классе, модели и архитектурных особенностях текущего процессора. Подробное описание данной команды приведено в приложении А учебника, а пример ее использования вы найдете в материале данной главы.

Ниже мы рассмотрим следующие расширения базовой архитектуры процессора Intel:

- целочисленное MMX-расширение;
- XMM-расширение — потоковое MMX-расширение для работы с данными вещественного типа;
- поддержка модельно-зависимых особенностей процессоров IA32.

MMX-технология процессоров Intel

В 1997 году фирма Intel представила свой новый процессор — Pentium® MMX™. По сравнению с предыдущими моделями процессоров этой фирмы (i486 и Pentium®), в его архитектуру были добавлены новые наборы регистров и команд.

Pentium® MMX™ стал своеобразной вехой в длинном ряду моделей процессоров семейства i80x86. С его появлением процессоры этого семейства естественно разбиваются на три группы, отражающие три этапа в истории развития процессоров.

- 16-разрядные процессоры i8086 и i80286 заложили основу популярности процессоров фирмы Intel. Их система команд содержала около 170 машинных инструкций, включая инструкции сопроцессора.
- Программная модель 32-разрядных процессоров i386, i486, Pentium, Pentium Pro практически одинакова и включает в себя порядка 220 команд.
- 32-разрядные процессоры Pentium MMX, Pentium II, Pentium III, Pentium 4 поддерживают новые технологии обработки данных, которые обеспечиваются введением в их архитектуру дополнительных регистров и команд. Эти новые архитектурные элементы составляют так называемое *MMX-расширение*. Система команд процессоров Pentium MMX и Pentium II насчитывает около 300 команд, а процессора Pentium III(4) — еще на 144 команды больше.

В этой главе мы обсудим MMX-расширение системы команд процессоров Pentium MMX, Pentium II и Pentium III(4).

MMX-расширение процессора Pentium предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные программы. Именно по этой причине данное расширение архитектуры процессоров Intel и названо MMX (MultiMedia eXtensions — мультимедийные расширения).

Ввиду появления процессора Pentium III следует различать MMX-расширения двух типов — целочисленное и с плавающей запятой. Каждое из этих MMX-расширений имеет свою программную модель и не зависит от другого. Чтобы обособить эти типы расширений, введем следующие условные обозначения: целочисленное MMX-расширение будем называть *MMX-расширением*, а MMX-расширение с плавающей запятой — *XMM-расширением*.

MMX-расширение архитектуры процессора Pentium

Целочисленное MMX-расширение архитектуры процессора Pentium представляет собой программно-аппаратное решение, дополняющее архитектуру данного процессора новыми свойствами. Впервые это расширение было введено в процессоре Pentium MMX. В неизменном виде оно поддерживается в последующих версиях процессоров Pentium. В Pentium III целочисленное MMX-расширение было дополнено новыми командами. Знакомство с архитектурой целочисленного MMX-расширения удобно вести в рамках модели, основу которой составляют два компонента — *программный* и *аппаратный*.

Модель целочисленного MMX-расширения

Основа *программного* компонента — система команд MMX-расширения (57 команд) и четыре новых типа данных (рис. 10.1). MMX-команды являются естествен-

ным дополнением основной системы команд процессора. Главным принципом их работы является одновременная обработка нескольких единиц однотипных данных одной командой (Single Instruction Multiple Data, SIMD).

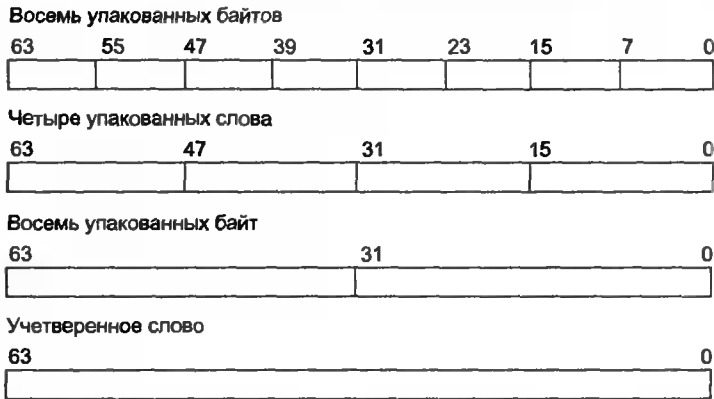


Рис. 10.1. Типы данных, поддерживаемые командами MMX-расширения

Из рисунка видно, что размер поля, занимаемого данными любого из этих типов, одинаков и составляет 64 бита. В них упаковываются и затем используются как отдельные объекты данные размером байт, слово и двойное слово (их мы будем называть *элементами* операндов). Именно с такими объектами работают MMX-команды. Такой подход приводит к существенному ускорению обработки данных, прежде всего, за счет экономии тактов процессора на передачу данных и выполнение самой команды. Кстати, здесь позитивно сказывается большая размерность шины данных процессора Pentium, которая равна 64 битам.

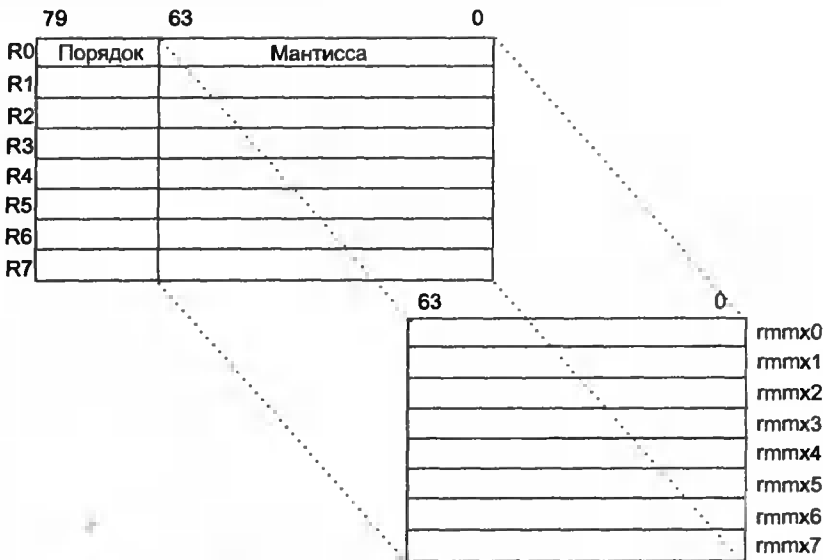


Рис. 10.2. Отображение MMX-регистров на регистры стека сопроцессора

Основа *аппаратного* компонента — восемь новых регистров. Слово «новые» не совсем корректно. На самом деле в MMX-расширении используются регистры сопроцессора. Как известно, регистры сопроцессорного стека имеют размерность 80 битов, что касается регистров MMX-расширения, то их размерность — 64 бита. Поэтому, когда регистры сопроцессора играют роль MMX-регистров, то доступными являются лишь их младшие 64 бита. К тому же, при работе стека сопроцессора в режиме MMX-расширения он рассматривается не как стек, а как обычный регистровый массив с произвольным доступом. Однако регистровый стек сопроцессора не может одновременно использоваться и по своему прямому назначению, и как MMX-расширение, поэтому забота о его разделении и корректной работе с ним ложится на программиста. Отображение MMX-регистров на регистры стека сопроцессора иллюстрирует рис. 10.2.

При выполнении MMX-команд сопроцессор переводится в состояние, которое характеризуется следующими признаками:

- регистр *тегов* сопроцессора обнуляется;
- все регистры стека сопроцессора в MMX-режиме адресуются физически вне зависимости от значений поля *TOS* регистра состояния *SWR* (поле *TOS*, кстати, тоже обнуляется);
- MMX-регистру *RMMX0* соответствует физический регистр сопроцессора *R0*, MMX-регистру *RMMX1* — *R1* и т. д., при этом логическая нумерация регистров сопроцессора не имеет никакого значения;
- содержимое других регистров сопроцессора не изменяется (за исключением случая применения команды *EMMS*);
- при записи в MMX-регистр данных в младшие 64 бита заносятся сами записываемые данные, а в биты 64–79 — единицы, чтобы при попытке случайного или преднамеренного использования командой сопроцессора MMX-данных не возникло какого-либо исключения сопроцессора;
- при чтении данных из MMX-регистров их содержимое не изменяется.

Особенности команд MMX-расширения

Важное отличие MMX-команд от обычных команд процессора — в том, как они реагируют на ситуации переполнения и заема. В разделе «Арифметические операции над целыми двоичными числами» урока 8 учебника нами разбирались ситуации, когда результат арифметической операции выходил за разрядную сетку исходных операндов. В этом случае производится усечение старших битов результата и возвращаются только те биты, которые умещаются в пределах исходного операнда. Этот принцип формирования результата называется *арифметикой с циклическим переносом* (*wraround arithmetic*). Некоторые MMX-команды в подобной ситуации действуют иначе. В случае выхода значения результата за пределы операнда в нем фиксируется максимальное или минимальное значение. Такой принцип формирования результата называется *арифметикой с насыщением* (*Saturation arithmetic*). MMX-команды выполняют арифметические операции с использованием обоих принципов. При этом среди них есть команды, учитывающие знаки (значения старших битов) элементов операндов. Принцип формирования резуль-

татов арифметических операций с циклическим переносом вы можете прочитать в материале главы 8 учебника, здесь же рассмотрим на примерах, как формируются результаты в MMX-командах сложения и вычитания, работающих по принципу насыщения.

Сложение чисел (беззнаковое насыщение):

```
254 = 11111110
+
5 = 00000101
=
259 <> 11111111
```

Результат MMX-сложения с беззнаковым насыщением равен 255. При сложении командами процессора ADD и ADC, использующими принцип циклического переноса, результат равен 00000011 = 3, а флаг CF устанавливается в 1. Это свидетельствует о факте переполнения.

Сложение чисел (знаковое насыщение):

```
+126 = 11111110 или -126 = 10000010
+
+5 = 00000101-5 = 11111011
==
131 <> 01111111-131 <> 10000000
```

Результат MMX-сложения со знаковым насыщением двух положительных чисел равен 127. При сложении командами процессора ADD и ADC, использующими принцип циклического переноса, результат равен 00000011 = 3, а флаг CF установлен в 1. Это свидетельствует о факте переполнения.

Вычитание чисел (беззнаковое насыщение):

```
05 = 00000101
-
10 = 00001010
=
-5 <> 00000000
```

Результат MMX-вычитания с беззнаковым насыщением двух чисел равен 00h. При вычитании командами процессора SUB и SBB, использующими принцип циклического переноса, результат равен 11111011 = -5 в дополнительном коде, а флаг CF устанавливается в 1. Это свидетельствует о факте воображаемого заема единицы из старшего разряда.

Вычитание чисел (знаковое насыщение):

```
+05 = 00000101или-5 = 11111011
-
-126 = 100000010+126 = 01111110
=
-131 <> 01111111-131 <> 10000000
```

Результат MMX-вычитания со знаковым насыщением двух чисел равен 80h. Это минимально возможное отрицательное число размер 1 байт. При вычитании командами процессора SUB и SBB, использующими принцип циклического пе-

реноса, результат равен $11111011 = -5$ в дополнительном коде, а флаг CF устанавливается в 1. Это свидетельствует о факте воображаемого заема единицы из старшего разряда.

Подобные рассуждения относятся и к некоторым другим MMX-командам, не являющимся арифметическими. В табл. 10.1 представлены граничные значения насыщения для всех четырех типов данных MMX-расширения.

Таблица 10.1. Граничные значения насыщения MMX-данных

Типы MMX-данных	Диапазон граничных значений (с насыщением)
Байт без знака	0...255 (00h...0ffh)
Слово без знака	0...65 535 (00h...0ffffh)
Двойное слово без знака	0...4 294 967 295 (00000000...0fffffffh)
Учетверенное слово без знака	(0000000000000000...0fffffffffffffffh)
Байт со знаком	-128...127 (80h...7fh)
Слово со знаком	-32 768...32 767 (8000h...7fffh)
Двойное слово со знаком	-2 147 483 648...2 147 483 647 (80000000...7fffffffh)
Учетверенное слово со знаком	(8000000000000000...7fffffffffffffffh)

Отметим некоторые особенности практического использования команд MMX-расширения.

Не все трансляторы языка ассемблера поддерживают MMX-команды. Это относится к трансляторам TASM версии 5.0 и MASM версии 6.11. В качестве транслятора, поддерживающего MMX-команды, можно рекомендовать транслятор NASM от Netwide, информацию о котором можно найти в Интернете по адресу <http://www.cryogen.com/nasm/>. А что же делать тем, кто располагает транслятором TASM или MASM и не имеет возможности воспользоваться услугами NASM? Выход здесь один — писать программу в машинных кодах, и дело это не такое уж безнадежное. Фирма Intel предвидела возникновение подобной проблемы и предложила вариант включаемого файла `iammx.inc` для транслятора ассемблера от Microsoft. Он содержит варианты макроопределений для всех мнемоник MMX-команд. Включив этот файл директивой `include` в начало вашей программы, вы можете использовать все MMX-команды в определенном формате. С транслятором TASM этот файл, увы, напрямую не работает. Его нужно предварительно немного скорректировать. Ситуация усложняется тем, что в пакет ассемблера TASM входят два транслятора: 16-разрядный (`tasm.exe`) и 32-разрядный (`tasm32.exe`). Разрабатываемые с их помощью программы имеют свои особенности. Этих особенностей достаточно много, поэтому здесь будет изложена методика, благодаря которой вы сумеете разрабатывать программы с MMX-командами для обоих типов трансляторов. Если же в вашем распоряжении есть другие средства для написания MMX-программ, к вашим услугам материал данного урока только в части, касающейся описания форматов MMX-команд и примеров их применения.

Перед началом работы доведем текст включаемого файла `iammx.inc` до рабочего состояния, пригодного для транслятора TASM 5.0. Для этого нам придется делать

два варианта этого файла — 16- и 32-разрядный. Назовем их, соответственно, `mtmx16.inc` и `mtmx32.inc`. Они будут использоваться при написании программ, обрабатываемых 16- и 32-разрядными трансляторами ассемблера (`tasm.exe` и `tasm32.exe`). Эти варианты включаемых файлов предназначены для работы в режиме MASM транслятора TASM 5.0. Объем исходных текстов включаемых файлов `mtmx16.inc` и `mtmx32.inc` достаточно велик, поэтому они находятся среди файлов, прилагаемых к книге. (Все имеющие отношение к книге файлы можно найти по адресу <http://www.piter.com/download/>.) Отличий в этих файлах два:

- в `mtmx16.inc` для моделирования MMX-команд используются 16-разрядные регистры общего назначения, а в `mtmx32.inc` — 32-разрядные регистры;
- в `mtmx.inc` для выяснения типа операнда задействуется оператор `.TYPE`. У автора при попытке указать эту директиву совместно с `tasm32.exe` возникала ошибка, которую удалось преодолеть только при применении аналогичного оператора для режима `Ideal` — `SYMTYPE`.

Можно попытаться объединить эти два файла и препроцессорными средствами ассемблера распознавать текущую вычислительную ситуацию, но большого смысла тут нет, так как обычно 16- и 32-разрядные коды не пересекаются. Проще разработать два файла для каждой из технологий и так избежать потенциальных ошибок. Поскольку содержательная часть этих файлов практически одинакова, есть смысл рассматривать только один из них, отмечая при необходимости имеющиеся отличия.

Рассмотрим структуру файла `mtmx16.inc`. Этот файл содержит макроопределения, каждое из которых моделирует определенную MMX-команду. В качестве основы для моделирования выступает команда основного процессора, которая должна удовлетворять определенным требованиям. Каковы они? В поисках ответа рассмотрим машинные коды MMX-команд. Видно, что общими у них являются два момента:

- поле кода операции MMX-команд состоит из двух байтов, первый из которых равен `0fh`;
- все MMX-команды, за исключением команды `emms`, используют форматы адресации с байтами `modR/M` и `sib` и, соответственно, допускают сочетание операндов как обычные двухоперандные команды целочисленного устройства — регистр-регистр или память-регистр.

Для моделирования MMX-команд нужно подобрать такую команду основного процессора, которая удовлетворяет обоим условиям. Проанализировав с позиций этих требований машинные коды команд основного процессора, выберем для моделирования команды `XADD` и `BTR`. В процессе моделирования на место второго байта кода операции этих команд помещается байт со значением кода операции MMX-команды. Когда процессор «видит», что очередная команда является MMX-командой, то он начинает трактовать коды регистров в машинной команде как коды MMX-регистров и ссылки на память размерностью, соответствующей данной команде. В машинном формате команды нет символических названий регистров, которыми мы пользуемся при написании исходного текста программы, например `AX` или `BX`. Во внутреннем представлении они определенным образом кодируются.

Например, регистр **AX** кодируется в поле **reg** машинной команды как **000** (см. главу 6 учебника). Если заменить код операции команды, в которой одним из операндов является регистр **AX**, кодом операции некоторой **MMX**-команды, то это же значение в поле **reg** процессор будет трактовать как регистр **RMMX0**. Таким образом, в **MMX**-командах коды регистров воспринимаются соответственно коду операции. В табл. 10.2 приведены коды регистров общего назначения и соответствующих им **MMX**-регистров. В правом столбце этой таблицы содержится условное обозначение **MMX**-регистров, принятое в файле **mmx16.inc**.

Таблица 10.2. Кодировка **MMX**-регистров в машинном коде команды

Код в поле reg	Регистр целочисленного устройства	MMX -регистр
000	AX/EAX	RMMX0
001	CX/ECX	RMMX1
010	DX/EDX	RMMX2
011	BX/EBX	RMMX3
100	SP/ESP	RMMX4
101	BP/EBP	RMMX5
110	SI/ESI	RMMX6
111	DI/EDI	RMMX7

Это же соответствие закреплено рядом следующих определений в этом файле:

```
rmmx0 equ <ax>
rmmx1 equ <cx>
rmmx2 equ <dx>
rmmx3 equ <bx>
rmmx4 equ <sp>
rmmx5 equ <bp>
rmmx6 equ <si>
rmmx7 equ <di>
```

В файле **mmx32.inc** эти же определения выглядят так:

```
rmmx0 equ <eax>
rmmx1 equ <ecx>
rmmx2 equ <edx>
rmmx3 equ <ebx>
rmmx4 equ <esp>
rmmx5 equ <ebp>
rmmx6 equ <esi>
rmmx7 equ <edi>
```

Теперь в исходном тексте программы можно использовать символические имена **MMX**-регистров в качестве аргументов макрокоманд, моделирующих **MMX**-команды.

Рассмотрим, как в файле **mmx16.inc** описано макроопределение для моделирования **MMX**-команды **PACKSSWB**, предназначенной для упаковки с насыщением слов в байты.

```
<1> packsswb macro dest:req, src:req
<2>             local pre, post
<3> pre:
<4>             xadd src, dest
```

```

<5> post:
<6>      org     pre + 1
<7>      db      _opc_packsswb
<8>      org     post
<9> endm

```

Понимание структуры приведенного макроопределения не должно вызывать у читателя трудностей. Центральное место здесь занимает команда целочисленного устройства (в данном случае **XADD**) и директива **ORG**. Директива **ORG** предназначена для изменения значения счетчика адреса (см. главу 10 учебника). В строке 6 директива **ORG** устанавливает значение счетчика адреса равным адресу **pre + 1**. Адрес метки **pre** является адресом первого байта машинного кода команды **XADD**. Соответственно, если этот адрес увеличить на 1, то получим адрес второго байта кода операции. Таким образом, значением текущего счетчика адреса в строке 7 будет адрес **pre + 1**. Директива **DB** в строке 7 размещает по этому адресу значение **opc_packsswb**, которое соответствует второму байту кода операции **MMX**-команды **PACKSSWB** (см. выше). Директива **ORG** в строке 8 устанавливает значение счетчика адреса равным значению адреса следующей после **XADD** команды. Для дотошных читателей замечу еще один характерный момент. Для полного понимания необходимо хорошо представлять себе формат машинной команды и назначение его полей. Достаточно полная информация об этом приведена в материалах главы 3 и приложения А учебника. Обратите внимание на порядок следования операндов в заголовке макрокоманды, который построен по обычной схеме: коп приемник, источник. В команде **XADD** порядок обратный, как требует синтаксис. Это хорошо поясняет назначение бита **d** во втором байте кода операции, который характеризует направление передачи данных: в процессор (в регистр) или в память из процессора (регистра). Вы можете провести эксперимент. Проанализируйте машинные коды команды **MOV**:

```

.data
p      dw      0
.code
      :...
      mov     p, bx      : машинный код: 89 1e 00 00. d=1. w=1
      mov     bx, p      : машинный код: 8b 1e 00 00. d=0. w=1

```

Из приведенного фрагмента видно, что изменение типов источника и приемника на обратный влияет только на поле кода операции, а именно на его второй бит. Это бит **d** (**Directory** — направление), который определяет направление передачи. Если приемник — регистр, то бит **d** = 1, а это означает передачу из памяти в процессор (регистр). И наоборот, если приемник — адрес памяти, то бит **d** = 0, то есть имеет место передача из процессора (регистра) в память. Таким образом, значения полей в остальных байтах машинного кода операции лишь определяют местоположение операндов и не зависят от реального направления передачи. Например, значение второго байта кода операции **opc_packsswb** равно 63h (01100011b). Он имеет значение бита **d** = 1, то есть данные передаются из регистра в память. Это нам и позволило в команде **XADD** изменить порядок следования операндов (иначе транслятор ассемблера эту команду не пропустит — подумайте, почему?).

В связи с обсуждаемым моментом интересно посмотреть макроопределение другой **MMX**-команды — **MOVD**:

```

<1> movd     macro  dest:req, src:req
<2>          local  pre, post
<3> if (.type(dest)) and 10h

```

```

<4> pre:
<5>      xadd    src, dest
<6> post:
<7>      org     pre + 1
<8>      db      _opc_movd_ld
<9>      org     post
<10> else
<11> pre:
<12>      xadd    dest, src
<13> post:
<14>      org     pre + 1
<15>      db      _opc_movd_st
<16>      org     post
<17> endif
<18> endm

```

Команда **MOVD** позволяет производить передачу в обоих направлениях: память-процессор и процессор-память. Для того чтобы правильно смоделировать машинное представление MMX-команды, необходимо определить, каким объектом является операнд **dest** (приемник): ячейкой памяти или регистром. После этого будет ясен тип второго операнда, так как реализованы могут быть лишь две схемы расположения операндов: регистр-регистр и память-регистр. Для выяснения типа операнда ассемблер предоставляет оператор **.type**, который имеет следующий синтаксис:

.type выражение

Оператор **.type** в зависимости от типа операнда **выражение** возвращает байтовые значения, которые представлены в табл. 10.3. В строке 3 листинга выше определяется тип операнда **dest** и, если это регистр, то формируется один код операции (строки 4–10), если нет, то другой (строки 11–17). Кстати, **MOVD** — это единственная MMX-команда, которая выпадает из общей схемы построения MMX-команд, так как только она позволяет производить обмен с 32-разрядным регистром общего назначения.

Таблица 10.3. Операнды и возвращаемые значения оператора **.type**

Значения битов	Тип объекта
0000 0000b	Указатель адреса памяти в сегменте кода
0000 0010b	Указатель адреса памяти в сегменте данных
0000 0100b	Константа
0000 1000b	В выражении используется режим прямой адресации
0001 0000b	Регистр
0010 0000b	Идентификатор
1000 0000b	Внешний идентификатор
???? ?00?b	В выражении используется режим косвенной адресации

Адреса операндов в памяти формируются таким же образом, как и для целочисленных команд, а их трактовка зависит от конкретной MMX-команды. Более детально с моделированием остальных MMX-команд вы можете познакомиться, изучив тексты включаемых файлов **mmx16.inc** и **mmx32.inc**, которые имеются среди файлов, прилагаемых к книге.

Далее мы рассмотрим MMX-команды, сопровождая описание характерными примерами, демонстрирующими механизм работы команд. В конце раздела будет приведен типовой пример использования MMX-команд.

Система команд

MMX-команды по функциональному признаку делятся на группы (рис. 10.3).

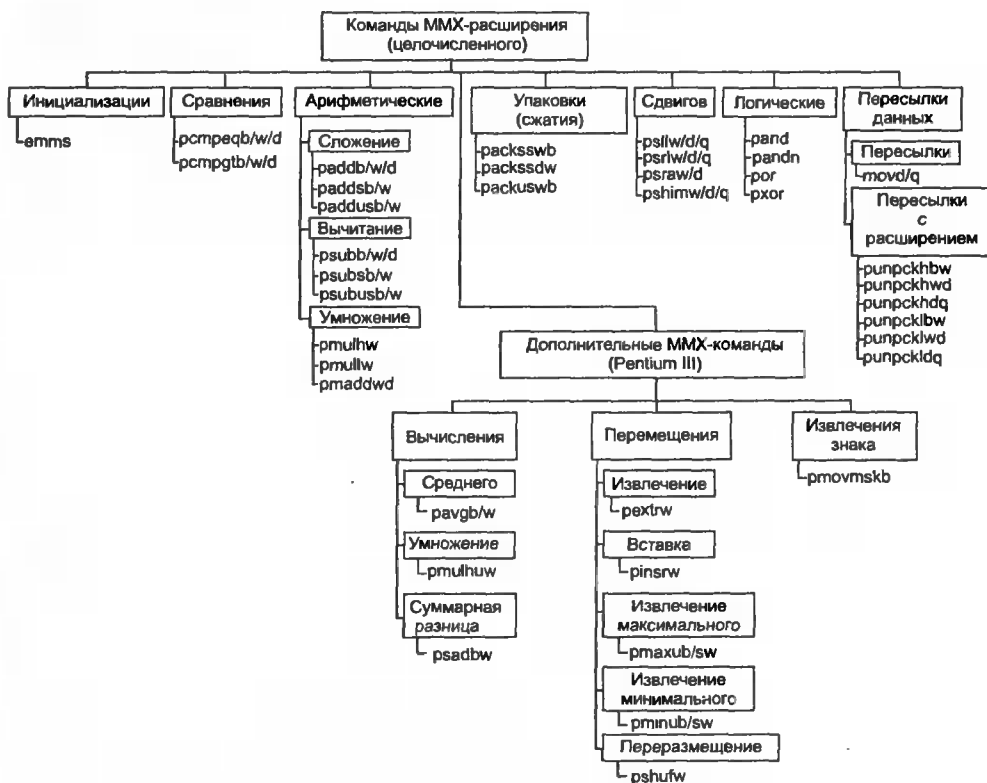


Рис. 10.3. Классификация целочисленных MMX-команд

Рассмотрим выделенные классы команд и укажем особенности их применения на примерах.

Команды передачи данных

MMX-команды пересылки, подобно их целочисленным аналогам, являются наиболее часто используемыми. Эти команды осуществляют передачу информации в MMX-регистры и из регистров. MMX-команды пересылки работают с 32- и 64-разрядными операндами. Ниже перечислены команды, входящие в данную группу.

- **MOVD** приемник, источник — пересылка 32 битов из источника в приемник. Один из операндов, источник или приемник, но не оба одновременно, должен быть MMX-регистром. Другой операнд должен быть 32-разрядным регистром или 32-разрядной ячейкой памяти.

- **MOVQ** приемник, источник — пересылка 64 битов из источника в приемник. В отличие от команды **MOVD**, оба операнда команды **MOVQ** могут быть **MMX**-регистрами. Если же операнды смешанные, то один из операндов, источник или приемник, должен быть **MMX**-регистром, а другой — адресом 64-разрядной ячейки памяти.

Команда **MOVD** работает только с младшей половиной **MMX**-регистра. Для доступа к старшей части **MMX**-регистра необходим либо сдвиг, либо команда **MOVQ**. Команда **MOVD** является единственной **MMX**-командой, допускающей использование в качестве операндов 32-разрядных регистров общего назначения. Это же обстоятельство является причиной того, что при применении в качестве приемника регистра общего назначения макрокоманда **MOVD** будет работать неправильно. Для того чтобы понять, в чем здесь дело, посмотрим еще раз на макроопределение, моделирующее эту **MMX**-команду:

```
movd      macro  dest:req, src:req
           local pre, post
           if (.type(dest)) and 10h
pre:       xadd   src, dest
post:      org    pre + 1
           db     _opc_movd_ld
           org    post
           else
pre:       xadd   dest, src
post:      org    pre + 1
           db     _opc_movd_st
           org    post
endif
endm
```

Допустимые сочетания операндов для команды **MOVD** следующие:

```
movd  mem32, rmmx
movd  rmmx, mem32
movd  rmmx, r32
movd  r32, rmmx
```

Приведенное макроопределение хорошо работает при первых трех сочетаниях операндов, а в последнем случае оно перестает понимать, что от него хотят, и дает сбой. Этому две причины. Во-первых, оператор **.type** (**SYMTYPE** в режиме **Ideal**) одинаково реагирует на сочетания **MOVD rmmx, r32** и **movd r32, rmmx**, так как в результате макроподстановки операнд **rmmx** заменяется одним из регистров **r32**. Поэтому в обоих случаях генерируется команда **XADD rmmx, r32** со вторым байтом кода операции, равным **6eh**. **MMX**-команда с этим кодом операции предполагает пересылку 32 битов из памяти или регистра общего назначения, но не наоборот. Для пересылки из регистра общего назначения (32-разрядной ячейки памяти) в **MMX**-регистр требуется машинная команда со вторым байтом кода операции **7eh**. Вторая причина является следствием первой. Так как **MMX**-регистры и регистры общего назначения кодируются в машинной команде одинаково, то для конкретной машинной команды по заложенной в ней схеме различают, какой из операндов является **MMX**-регистром, а какой — регистром общего назначения, и для указанного выше сочетания операндов это делается неправильно. Как выйти из положения? Можно, конечно, выполнить дополнительный анализ передаваемых в макрокоманду аргументов, но мы, учитывая единичность данного случая, пошли по самому легкому пути — включили в файл **mmx.inc** дополнительное макроопределение

MOVDR32. Это макроопределение специально предназначено для случая пересылки данных из MMX-регистра в 32-разрядный регистр общего назначения.

Следующий важный вопрос связан с тем, как описывать данные для работы с MMX-командами. Здесь также приходится учитывать то обстоятельство, что при использовании пакета TASM (как, впрочем, и MASM) MMX-команды приходится моделировать. Главное здесь — понять последовательность этого моделирования: на этапе трансляции моделируются операнды целочисленной команды (в нашем случае — XADD) и формируется поле кода операции в машинной команде (ее второй байт). Далее на этапе исполнения программы выполняется должная интерпретация операндов команды. Если посмотреть на описание команды XADD, то легко увидеть, что она может работать максимум с 32-разрядными операндами. А как же быть, если требуется моделировать MMX-команду, манипулирующую 64-разрядными операндами? В этом случае приходится в очередной раз заниматься обманом. Например, следующий фрагмент программы будет ошибочным:

```
include      mmx.inc
.data
mem64       dq      1111222233334444h
            :...
.code
            :...
            movq     rmmx0, mem64
            :...
```

Ошибка возникнет из-за того, что на этапе трансляции будет смоделирована команда XADD ax, mem64, которую транслятор не пропустит из-за несовпадения типов операндов. Суть «подлога» состоит в следующем описании данных:

```
include      mmx.inc
.data
mem64       dw      4444h
            df      111122223333h
            :...
.code
            :...
            movq     rmmx0, mem64
            :...
```

Помните, что при описании операнда в памяти продолжает действовать принцип размещения данных в памяти «младший байт по младшему адресу». При работе с 32-разрядным ассемблером tasm32.exe описание данных должно быть выполнено так:

```
include      mmx32.inc
.data
mem64       dd      33334444h
            dd      11112222h
            :...
.code
            :...
            movq     rmmx0, mem64
            :...
```

Ниже приведены примеры команд пересылки данных. Текст программы используйте в качестве основы для разработки других программ в этой главе. Далее для экономии места будут приводиться только фрагменты соответствующих программ.

```
<1> ; mmx16.asm
<2> .586p
```



```

<3> model use16 small ; use16 обязательно
<4> %NOINCL ; запретить вывод текста включаемых файлов
<5> include mmx16.inc
<6> .stack 100h
<7> .data ; сегмент данных
<8> mem dw 4444h
<9> df 111122223333h
<10> mem1 dw 3h
<11> df 0002cccc0004h
<12> mem2 dw 0h
<13> df 0001dddd0f03h
<14> mem3 dw 7fffh
<15> df 0001dddd0f03h
<16> .code
<17> main proc ; начало процедуры main
<18> mov ax, @data
<19> mov ds, ax
<20>
<21> movd rmmx0, mem ; rmmx0=0000 0000 3333 4444
<22> movdr32 ax, rmmx0 ; rmmx0=0000 0000 3333 4444, eax=3333 4444
<23> movq rmmx0, mem1 ; rmmx0=0002 cccc 0004 0003
<24> movd mem2, rmmx0 ; mem2=0300 0400 cccc 0200, rmmx0=0002
      cccc 0004 0003
<25> pxor rmmx0, rmmx0 ; rmmx0=0000 0000 0000 0000
<26> movd mem3, rmmx0 ; mem3=0000 0000 0000 0000
<27> emms
<28> mov ax, 4c00h ; пересылка 4c00h в регистр ax
<29> int 21h ; вызов прерывания с номером 21h
<30> main endp ; конец процедуры main
<31> end main ; конец программы с точкой входа main

```

Обратите внимание на строку 22, в которой команда **MOVD** (в учебнике мы обсуждали эту команду и соответствующую ей макрокоманду **MOVDR32**) пересылает 32 бита в регистр общего назначения, но при этом указывается 16-разрядный регистр **AX**. Указать напрямую **EAX** нельзя, так как для 16-разрядного ассемблера вместо **RMMX0** в ходе моделирования подставляется **AX** (см. определения в **mmx16.inc**). На этапе выполнения все расставляется на свои места, так как требуемую пересылку (из **RMMX0** в **EAX**) производит микропрограмма, реализующая **MMX**-команду **MOVD** по полю кода операции и коду регистра, который, как видно из табл. 10.3, одинаков для регистров **AX** и **EAX**.

Здесь уместно уделить внимание еще одной проблеме, которая неизбежно встает на определенном этапе разработки программы — какие отладочные средства можно использовать для просмотра результатов работы **MMX**-команд? Можно попробовать подыскать отладчик, поддерживающий работу с **MMX**-расширением, но это делать совсем не обязательно. Понимание того, что **MMX**-расширение архитектурно создано на базе сопроцессора, дает вам возможность использовать практически любой отладчик для работы с **MMX**-приложениями, в том числе **Turbo Debugger**. При этом, конечно, необходимо соблюдать определенные ограничения, источник возникновения которых в том, что **Turbo Debugger** ничего не знает о **MMX**-командах. Это обстоятельство может приводить к его неадекватной работе. Но если следовать некоторым рекомендациям, то можно пройти над «подводными камнями», не задев их.

В пакете **TASM** версии 5.0 есть два варианта отладчика **Turbo Debugger**:

- **td.exe** — для разработки 16-разрядных программ;
- **td32.exe** — для разработки 32-разрядных приложений.

В основном, все сказанное ниже касается любого из этих отладчиков, так как проблема у них общая — непонимание мнемоники MMX-команд.

Чтобы создать исполняемый файл, пригодный для отладки, задайте в качестве параметра транслятора (tasm.exe или tasm32.exe) ключ /zi, а для компоновщика (tlink.exe или tlink32.exe) — ключ /v. Загрузите исполняемый файл в отладчик (исходный текст появится в окне Module).

Теперь «в уме» разделите программу на фрагменты двух типов: содержащие MMX-команды и без них. С фрагментами, не содержащими MMX-команд, вы можете работать в обычном режиме. Для отладки программы на фрагментах, включающих в себя MMX-команды, перейдите в окно CPU (View ► CPU), а также откройте окно Numeric processor (View ► Numeric processor). После этого окно отладчика должно выглядеть так, как показано на рис. 10.4.

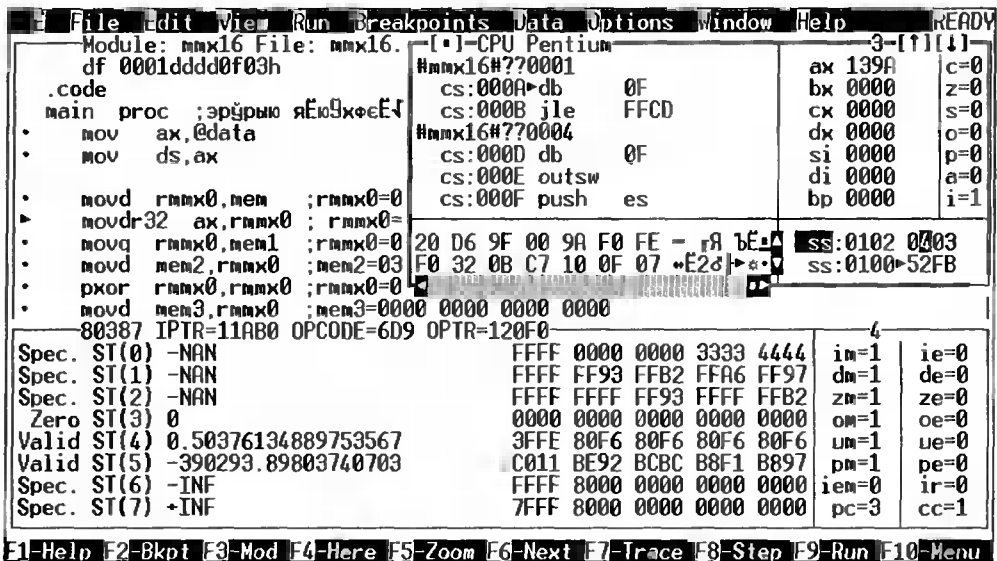


Рис. 10.4. Вид отладчика при работе с MMX-командами

Сам процесс отладки традиционен, и вы можете управлять им, используя привычные средства Turbo Debugger.

Арифметические команды

В группу арифметических входят команды для реализации трех основных арифметических операций: сложения, вычитания, умножения. При этом, как мы уже отметили выше, для операций сложения и вычитания предусмотрены команды, учитывающие знак операндов (значение старшего бита).

Сложение

Команды сложения делятся на две подгруппы, исходя из того, как формируется результат при возникновении переполнения — по принципу насыщения или по принципу циклического переноса.

- **PADDB | PADDW | PADDD** приемник, источник — сложение беззнаковых упакованных байтов, слов, двойных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При переполнении результат формируется по принципу циклического переноса (см. выше подраздел «Особенности команд MMX-расширения»). Перенос теряется и нигде не учитывается.

```
.data
mem      dw      4444h
          df      111122223333h
mem1     dw      0fffeh
          df      0fffcfffdffffh
.code
:....
movq     rmmx0, mem      ; rmmx0 = 11 11 22 22 33 33 44 44
: mem1 = ff fc ff fd ff ff ff fe
paddb    rmmx0, mem1     ; rmmx0 = 10 0d 21 1f 32 32 43 42
:....
```

- **PADDSB | PADDSW** приемник, источник — сложение упакованных байтов и слов со знаком. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу знакового насыщения (см. выше подраздел «Особенности команд MMX-расширения»).

```
.data
mem      dw      4444h
          df      111122223382h
mem1     dw      0157eh
          df      0717c3f7d7ffah
.code
:....
movq     rmmx0, mem      ; rmmx0 = 11 11 22 22 33 82 44 44
: mem1 = 71 7c 3f 7d 7f fa 15 7e
paddsb   rmmx0, mem1     ; rmmx0 = 7f 7f 61 7f 7f 80 59 7f
:....
```

Исходные данные этого примера подобраны так, чтобы было видно, как при переполнении формируется результат по принципу знакового насыщения. Сравните их с теми результатами, которые получаются в результате работы следующих команд.

- **PADDUSB | PADDUSW** приемник, источник — сложение беззнаковых упакованных байтов и слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении переполнения результат формируется по принципу беззнакового насыщения.

```
.data
mem      dw      4444h
          df      111122223382h
mem1     dw      0157eh
          df      0717c3f7d7ffah
.code
:....
movq     rmmx0, mem      ; rmmx0 = 11 11 22 22 33 82 44 44
: mem1 = 71 7c 3f 7d 7f fa 15 7e
paddusb  rmmx0, mem1     ; rmmx0 = 82 8d 61 9f b2 ff 59 c2
:....
```

Вычитание

Набор команд вычитания аналогичен командам сложения и содержит три группы команд: обработки беззнаковых операндов традиционным для целочисленного устройства способом — по принципу циклического переноса; обработки беззнаковых операндов по принципу беззнакового насыщения; обработки знаковых операндов по принципу знакового насыщения.

- **PSUBB | PSUBW | PSUBD** приемник, источник — вычитание беззнаковых упакованных байтов, слов, двойных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При переполнении результат формируется по принципу циклического переноса, то есть так, как это делается командами **SUB** и **SBB** процессора. Заем из старшего разряда, естественно, теряется и нигде не учитывается.

```
.data
mem      dw      4444h
          df      111122223382h
mem1     dw      157eh
          df      1123f7d7ffah
.code
: ...
movq     rmmx0, mem      ; rmmx0 = 1111 2222 3382 4444
                        ; mem1  = 1123 f7d7 ffah 157e
psubw    rmmx0, mem1     ; rmmx0 = 0fff e2a5 b388 2ec6
: ...
```

- **PSUBSB | PSUBSW** приемник, источник — вычитание упакованных байтов и слов со знаком. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении ситуации, когда результат вычитания получается меньше 80h (8000h), поле соответствующего байта (слова) формируется по принципу знакового насыщения (в нем остается значение 80h (8000h)). Если результат больше 7fh (7fffh), то результат насыщается до значения 7fh (7fffh).

```
.data
mem      dw      5h
          df      0fffb22223382h
mem1     dw      8003h
          df      7ffedf7d7ffah
.code
: ...
movq     rmmx0, mem      ; rmmx0 = fffb 2222 3382 0005
                        ; mem1  = 7ffe df7d 7ffa 8003
psubsw   rmmx0, mem1     ; rmmx0 = 8000 42a5 b388 7fff
: ...
```

Первая и последняя тетрады показывают результат, сформированный в соответствии с принципом знакового насыщения.

- **PSUBUSB | PSUBUSW** приемник, источник — вычитание беззнаковых упакованных байтов и слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. При возникновении ситуации, когда результат вычитания получается меньше 00h (0000h), поле соответствующего байта или слова формируется по принципу беззнакового насыщения (в нем остается значение 00h (0000h)).

```

.data
mem      dw      5h
         df      0ffffb22223382h
mem1     dw      8003h
         df      7ffe0f7d0ffah
.code
:....
movq     rmmx0, mem      ; rmmx0 = fffb 2222 3382 0005
                        ; mem1  = 7ffe 0f7d 0ffa 8003
psubusbw rmmx0, mem1     ; rmmx0 = 7ffd 12a5 2388 0000
:....

```

Обратите внимание на последнюю тетраду, которая соответствует результату вычитания (5 – 32 771).

Умножение

Команды MMX-умножения предназначены только для умножения 16-разрядных элементов, причем реализация этих команд сделана несколько непривычно. Команды умножения целочисленного устройства формируют результат, размер которого вдвое превышает размер исходных операндов. Команды умножения MMX-расширения реализуют это действие несколько иначе. Во-первых, умножению подвергаются одновременно 4 слова со знаком. Во-вторых, для получения полного результата умножения (размером в двойное слово) необходимо применение двух команд, PMULHW и PMULLW. С их помощью формируются соответственно старшая и младшая части произведения. Для объединения результата в единое двойное слово можно использовать команды расширения PUNPCKHWD или PUNPCKLWD.

- **PMULHW** приемник, источник — умножение четырех знаковых упакованных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. В приемник записываются не все 32 бита произведения, а только старшие 16 битов. Младшие 16 битов можно получить при помощи команды PMULLW.
- **PMULLW** приемник, источник — умножение знаковых упакованных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. В приемник записываются не все 32 бита произведения, а только младшие 16 битов. Старшие 16 битов можно получить, используя команду PMULHW.
- **PMADDWD** приемник, источник — умножение четырех знаковых упакованных слов. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Формирование результата осуществляется по схеме, представленной на рис. 10.5.

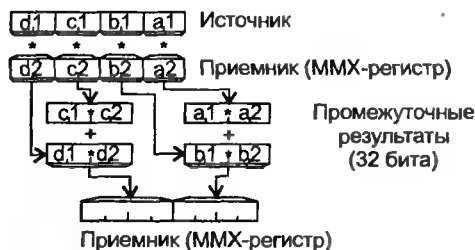


Рис. 10.5. Схема работы команды PMADDWD

Для иллюстрации работы команд **PMULHW** и **PMULLW** рассмотрим пример умножения четырех знаковых слов.

```
.data
mem      dw      5h
         df      7ffb22223382h
mem1     dw      8008h
         df      7ffe0f7d0ffaah
mem2     dw      0
         df      0
mem3     dw      0
         df      0
.code
;...
movq     rmmx0, mem      ; rmmx0 = 7ffb 2222 3382 0005
                        ; mem1  = 7ffe 0f7d 0ffa 8008
;----- получим младшие части произведений
pmullw   rmmx0, mem1     ; rmmx0 = 800a a89a eaf4 8028
movq     rmmx1, mem      ; rmmx1 = 7ffb 2222 3382 0005
                        ; mem1  = 7ffe 0f7d 0ffa 8003
;----- получим старшие части произведений
pmulhw   rmmx1, mem1     ; rmmx1 = 3ffc 0210 0336 fffd
;----- сохраним rmmx0 для последующих действий
movq     rmmx2, rmmx0    ; rmmx2 = 800a a89a eaf4 8028
;----- в поле mem2 содержится результат произведений операндов
;      младших половин исходных операндов
punpcklwd rmmx0, rmmx1   ; rmmx0=0336 eaf4 fffd 8028
movq     mem2, rmmx0     ; mem2=0336 eaf4 fffd 8028
;----- в поле mem3 полный результат произведений операндов
;      старших половин исходных операндов
punpckhwd rmmx2, rmmx1   ; rmmx2 = 3ffc 800a 0210 a89a
movq     mem3, rmmx2     ; mem3  = 3ffc 800a 0210 a89a
;...
```

Используя программу-калькулятор Windows, вы можете проверить полученные произведения. Вычислите с его помощью произведение $7ffb \times 7ffe = 3ffc800a$. Такое же значение мы получили в поле **mem3**. Аналогично можно проверить и другие результаты.

Результат работы команды **PMADDWD** представляет собой сумму произведений двух старших и двух младших слов операндов приемник и источник (см. рис. 10.5). Результат получается в виде двух двойных упакованных слов. Ниже показан пример использования команды **PMADDWD**:

```
.data
mem      dw      5h
         df      7ffb22223382h
mem1     dw      8008h
         df      7ffe0f7d0ffaah
mem2     dw      0
         df      0
mem3     dw      0
         df      0
.code
;...
movq     rmmx0, mem      ; rmmx0 = 7ffb 2222 3382 0005
                        ; mem1  = 7ffe 0f7d 0ffa 8008
pmaddwd  rmmx0, mem1     ; rmmx0 = 420d 28a4 033a 6b1c
;...
```

Команды сравнения

Группа команд сравнения MMX-расширения содержит команды двух типов. Команды простого сравнения (**PCMPQBW** | **PCMPQW** | **PCMPQD**) характеризуются тем,

что устанавливают только факт равенства операндов (равно — не равно). Команды сравнения по величине (CMPGTB | PCMPGTW | PCMPGTD) устанавливают соотношение операндов по величине.

■ **PCMPEQB | PCMPEQW | PCMPEQD** операнд_1, операнд_2 — сравнение упакованных байтов, слов или двойных слов. Результат формируется в первом операнде, который является одним из MMX-регистров. Второй операнд — либо MMX-регистр, либо 64-разрядная ячейка памяти. Элементы результата представляются в виде единичных или нулевых байтов, слов или двойных слов. Единичные байты, слова или двойные слова формируются, если соответствующие байты, слова или двойные слова исходных операндов равны. Нулевые байты, слова, двойные слова формируются, если соответствующие байты, слова или двойные слова исходных операндов не равны.

```
.data
mem          dw      5h
              df      7fff22223382h
mem1         dw      5h
              df      7ffe0f7d0ffah

.code
: ...
movq    rmmx0, mem      ; rmmx0 = 7fff 2222 3382 0005
: ...
pcmpeqw rmmx0, mem1     ; mem1 = 7ffe 0f7d 0ffa 0005
: ...
: rmmx0 = 0000 0000 0000 ffff
```

■ **PCMPGTB | PCMPGTW | PCMPGTD** операнд_1, операнд_2 — сравнение по величине упакованных байтов, слов или двойных слов. Результат формируется в первом операнде, который является одним из MMX-регистров. Второй операнд — либо MMX-регистр, либо 64-разрядная ячейка памяти. Элементы результата представляются в виде единичных или нулевых байтов, слов или двойных слов. Единичные байты, слова, двойные слова формируются в случае, если байты, слова или двойные слова исходного операнда операнд_1 были больше соответствующих байтов, слов или двойных слов операнда операнд_2. Иначе формируются нулевые байты, слова или двойные слова.

В примере ниже наглядно показано, что единичные элементы получаются только в случае, если элементы первого операнда (всегда находящегося в MMX-регистре) больше соответствующих элементов второго операнда.

```
.data
mem          dw      5h
              df      7fff22223382h
mem1         dw      5h
              df      7ffe0f7d0ffah

.code
: ...
movq    rmmx0, mem      ; rmmx0 = 7fff 2222 3382 0005
: ...
pcmpgtw rmmx0, mem1     ; mem1 = 7ffe 0f7d f0fa 0005
: ...
: rmmx0 = ffff ffff 0000 0000
```

Команды логических операций

Команды логических операций предназначены для поразрядной обработки содержимого двух MMX-регистров или MMX-регистра и 64-разрядного операнда в памяти. Эти команды реализуют логические операции И, И-НЕ, ИЛИ, исключая-

щее ИЛИ. Заметьте, что появилась новая логическая операция И-НЕ, которой нет среди логических команд основного процессора (чистой булевой логики). Другая особенность логических команд заключается в том, что они выполняются над всеми шестьюдесятью четырьмя разрядами MMX-операндов. Для структурирования результатов их работы нужно применять другие команды MMX-расширения.

- **PAND** приемник, источник — выполнение поразрядной операции И над операндами. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 10.4.

Таблица 10.4. Таблица истинности для операции И

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	0	0	1

Принцип работы команды **PAND** иллюстрирует пример:

```
.data
mem      dw      0505h
          df      7ff002203080h
mem1     dw      0005h
          df      7ff002000f80h

.code
; ...
movq     rmmx0, mem      ; rmmx0 = 7ff0 0220 3080 0505
; mem1   = 7ff0 0200 0f80 0005
pand     rmmx0, mem1     ; rmmx0 = 7ff0 0200 0080 0005
; ...
```

- **PANDN** приемник, источник — выполнение поразрядной операции И-НЕ над операндами. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 10.5. Из таблицы видно, что единичные биты в результате могут появиться только в одном случае, когда единичные биты источника совпадают с нулевыми битами приемника. Команда позволяет определить положение единичных битов источника, которым не соответствуют единичные биты приемника.

Таблица 10.5. Таблица истинности для операции И-НЕ

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	1	0	0

Принцип работы команды **PANDN** иллюстрирует пример:

```
.data
mem      dw      0505h
          df      7ff002203080h
mem1     dw      0005h
          df      7ff003000f80h

.code
; ...
```



```

movq    rmmx0, mem      ; rmmx0 = 7ff0 0220 3080 0505
                        ; mem1  = 7ff0 0300 0f80 0005
pandn   rmmx0, mem1     ; rmmx0 = 0000 0100 0f00 0000
:....

```

- **POR** приемник, источник — выполнение поразрядной операции **ИЛИ** над операндами. Результат помещается в приемник, который является одним из ММХ-регистров. Источник — либо ММХ-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 10.6.

Таблица 10.6. Таблица истинности для операции **ИЛИ**

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	1	1	1

Принцип работы команды **POR** иллюстрирует пример:

```

.data
mem      dw      0505h
         df      7ff002203080h
mem1     dw      0005h
         df      7ff003000f80h
.code
:....
movq     rmmx0, mem      ; rmmx0 = 7ff0 0220 3080 0505
                        ; mem1  = 7ff0 0300 0f80 0005
por      rmmx0, mem1     ; rmmx0 = 7ff0 0320 3f80 0505
:....

```

- **PXOR** приемник, источник — выполнение поразрядной операции «исключающее **ИЛИ**» над операндами. Результат помещается в приемник, который является одним из ММХ-регистров. Источник — либо ММХ-регистр, либо 64-разрядная ячейка памяти. Биты результата в приемнике формируются в соответствии с табл. 10.7.

Таблица 10.7. Таблица истинности для операции «исключающее **ИЛИ**»

Приемник	0	0	1	1
Источник	0	1	0	1
Результат	0	1	1	0

Принцип работы команды **PXOR** иллюстрирует пример:

```

.data
mem      dw      0505h
         df      7ff002203080h
mem1     dw      0005h
         df      7ff003000f80h
.code
:....
movq     rmmx0, mem      ; rmmx0=7ff0 0220 3080 0505
                        ; mem1 =7ff0 0300 0f80 0005
pxor     rmmx0, mem1     ; rmmx0=0000 0120 3f00 0500
:....

```

Команды сдвига

В главе 9 учебника мы познакомились с командами арифметического и логического сдвига основного процессора. Отличие этих двух типов команд сдвига — в способе интерпретации знакового бита операнда. Среди MMX-команд сдвига также существуют команды арифметического и логического сдвига. Обратите внимание на то обстоятельство, что MMX-команд сдвига упакованных байтов нет. Сдвигать можно только упакованные слова, двойные слова и учетверенные слова (целиком весь MMX-регистр).

- **PSLLW | PSLLD | PSLLQ** приемник, источник — команды логического сдвига влево упакованных слов, двойных слов или учетверенных слов в приемнике на количество разрядов, указанных значением источника. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига младшие биты упакованных элементов приемника заполняются нулями.
- **PSLLW | PSLLD | PSLLQ** приемник, количество_сдвигов — команды логического сдвига влево аналогичны рассмотренным выше командам за исключением того, что все упакованные слова, двойные слова и учетверенные слова в приемнике сдвигаются на количество разрядов, указанных значением непосредственного операнда количество_сдвигов. Освобождающиеся в результате сдвига младшие биты упакованных элементов приемника заполняются нулями.
- **PSRLW | PSRLD | PSRLQ** приемник, источник — команды логического сдвига вправо упакованных слов, двойных слов или учетверенных слов в приемнике на количество разрядов, указанных значением в источнике. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига старшие биты упакованных элементов приемника заполняются нулями.
- **PSRLW | PSRLD | PSRLQ** приемник, количество_сдвигов — команды логического сдвига вправо аналогичны рассмотренным выше командам за исключением того, что все упакованные слова, двойные слова или учетверенные слова в приемнике сдвигаются на количество разрядов, указанных значением непосредственного операнда количество_сдвигов. Освобождающиеся в результате сдвига старшие биты упакованных элементов приемника заполняются нулями.

```
.data
mem          dw      0ffffh
              df      0fffffffffffh
mem1         dw      4
              df      0

.code
;...
movq rmmx0, mem          ; rmmx0 = ffff ffff ffff ffff
                           ; mem1  = 0000 0000 0000 0004
psllw rmmx0, mem1        ; rmmx0 = fff0 fff0 fff0 fff0
psrlw rmmx0, 4           ; rmmx0 = 0fff 0fff 0fff 0fff
;...
```

Следующие команды являются командами арифметического сдвига. Эти команды сдвигают значение операнда вправо. Команд арифметического сдвига влево нет, так как они аналогичны командам логического сдвига, не сохраняющим значения знакового разряда.

- **PSRAW | PSRAD** приемник, источник — команды арифметического сдвига вправо упакованных слов или двойных слов в приемнике на количество разрядов, указанных в источнике. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти. Освобождающиеся в результате сдвига старшие биты упакованных элементов приемника заполняются значением знаковых (старших) разрядов этих элементов.

```
.data
mem      dw      0fff0h
          df      0fff0fff0fff0h
mem1     dw      4
          df      0

.code
: ...
movq     rmmx0, mem      ; rmmx0 = fff0 fff0 fff0 fff0
: ...
: mem1 = 0000 0000 0000 0004
psraw    rmmx0, 4        ; rmmx0 = ffff ffff ffff ffff
: ...
```

Команды упаковки и распаковки

Команды упаковки и распаковки предназначены для изменения размерности элементов операндов с учетом их значений. Команды упаковки позволяют уменьшить размерность элементов в два раза. При этом если значение сжимаемого элемента больше максимального допустимого значения, которое может содержаться в элементе меньшего размера, то результат формируется по принципу знакового насыщения.

- **PACKSSDW** приемник, источник — команда упаковки со знаковым насыщением двух двойных слов в приемнике и двух двойных слов в источнике в четыре слова в приемнике. Схема выполнения команды показана на рис. 10.6. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

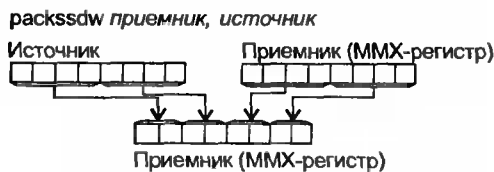


Рис. 10.6. Схема работы команды PACKSSDW

- **PACKSSWB** приемник, источник — команда упаковки со знаковым насыщением четырех слов в приемнике и четырех слов в источнике в четыре слова в приемнике. Схема выполнения команды показана на рис. 10.7. Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

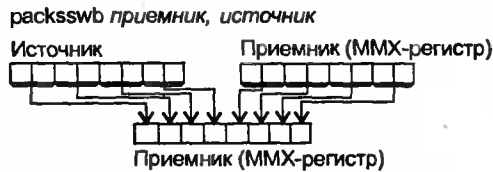


Рис. 10.7. Схема работы команды PACKSSWB

```
.data
mem      dw      0fe00h
          df      00457ffe00f0h

.code
:....
packsswb rmmx0, mem      ; rmmx0 = 45 7f 7f 80 00 00 00 00
:....
```

Пример показывает, как выполняется принцип знакового насыщения результата до 7fh и 80h. Подобная ситуация возникает каждый раз, когда значение в исходном слове превышает максимально возможное.

Следующая группа MMX-команд позволяет выполнить обратную операцию — расширить размер элементов операнда в два раза. При этом недостающая половина вновь формируемого элемента извлекается из второго операнда.

- **PUNPCKHBW** приемник, источник — команда распаковки байтов из старшей половины приемника в слова с использованием в качестве старшей половины этих слов байтов из источника. Формирование результата происходит посредством поочередной выборки байтов из приемника и источника (рис. 10.8). Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

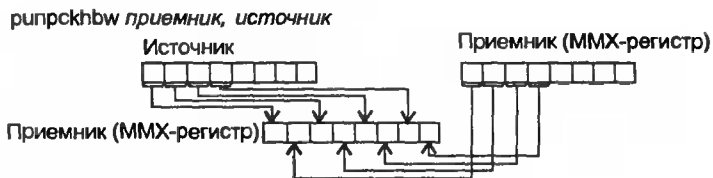


Рис. 10.8. Схема работы команды PUNPCKHBW

```
.data
mem      dw      0
          df      01020304ffffh
mem1     dw      0
          df      0f0f0f0feeeeh

.code
:....
movq     rmmx0, mem      ; rmmx0 = 01 02 03 04 ff ff 00 00
:....
punpckhbw rmmx0, mem1    ; mem1 = 0f 0f 0f 0f ee ee 00 00
:....
punpckhbw rmmx0, mem1    ; rmmx0 = 0f01 0f02 0f03 0f04
:....
```

- **PUNPCKHWD** приемник, источник — команда распаковки слов из старшей половины приемника в двойные слова с использованием слов из источника в качестве старшей половины этих двойных слов. Формирование результата происходит путем поочередной выборки слов из приемника и источника (рис. 10.9). Резуль-

тат формируется в приемнике, который является одним из MMX-регистров, в то же время источник может быть либо MMX-регистром, либо 64-разрядной ячейкой памяти.

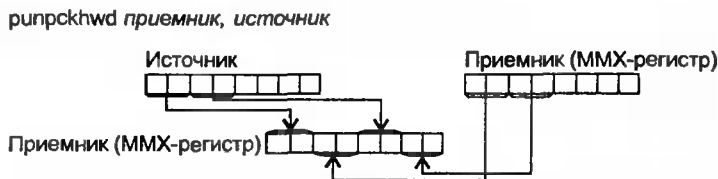


Рис. 10.9. Схема работы команды PUNPCKHWD

PUNPCKHDQ приемник, источник — команда распаковки двойных слов из старшей половины приемника в учетверенные слова с использованием в качестве старшей половины этих учетверенных слов двойных слов из источника. Формирование результата происходит путем поочередной выборки двойных слов из приемника и источника (рис. 10.10). Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

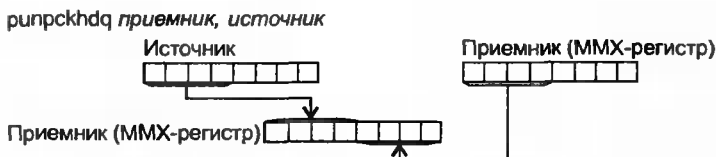


Рис. 10.10. Схема работы команды PUNPCKHDQ

Вы, наверное, обратили внимание, что предыдущие три команды работают со старшими половинами операндов. Следующие три команды, наоборот, работают с их младшими половинами.

PUNPCKLBW приемник, источник — команда распаковки байтов из младшей половины приемника в слова с использованием в качестве младшей половины этих слов байтов из источника. Формирование результата осуществляется путем поочередной выборки байтов из приемника и источника (рис. 10.11). Результат помещается в приемник, который является одним из MMX-регистров. Источник — MMX-регистр или 64-разрядная ячейка памяти.

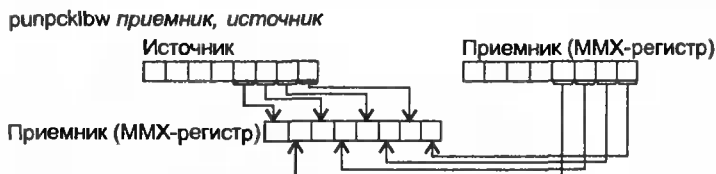


Рис. 10.11. Схема работы команды PUNPCKLBW

.data
mem

dw 0304h
df 0ffffeeeee0102h

```

mem1      dw      0f0fh
           df      0c0c0c0c0f0fh
.code
;....
movq      rmmx0, mem      ; rmmx0 = ff ff ee ee 01 02 03 04
; mem1 = 0c 0c 0c 0c 0f 0f 0f 0f
punpcklbw rmmx0, mem1     ; rmmx0 = 0f01 0f02 0f03 0f04
;....

```

- **PUNPCKLWD** приемник, источник — команда распаковки слов из младшей половины приемника в двойные слова с использованием в качестве их младшей половины слов из источника. Формирование результата осуществляется путем поочередной выборки слов из приемника и источника (рис. 10.12). Результат помещается в приемник, который является одним из MMX-регистров, в то же время источник может быть либо MMX-регистром, либо 64-разрядной ячейкой памяти.

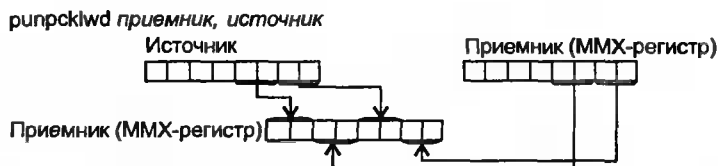


Рис. 10.12. Схема работы команды PUNPCKLWD

- **PUNPCKLDQ** приемник, источник — команда распаковки двойных слов из младшей половины приемника в учетверенные слова с использованием в качестве младшей половины этих учетверенных слов двойных слов из источника. Формирование результата осуществляется путем поочередной выборки двойных слов из приемника и источника (рис. 10.13). Результат помещается в приемник, который является одним из MMX-регистров. Источник — либо MMX-регистр, либо 64-разрядная ячейка памяти.

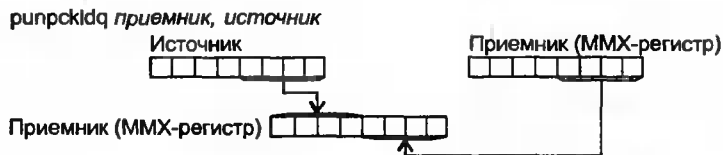


Рис. 10.13. Схема работы команды PUNPCKLDQ

Команда очистки стека регистров сопроцессора

В конце блока команд, содержащего MMX-инструкции, должна обязательно присутствовать команда **EMMS**, в функции которой входит очистка стека регистров и установка единичных значений в регистре тегов. Каждому регистру стека сопроцессора соответствует двухразрядное поле в этом регистре, единичное значение которого говорит о том, что регистр пуст.

Наличие команды **EMMS** обязательно, если MMX-команды комбинируются с командами сопроцессора.

В настоящее время пока еще достаточно большое количество компьютеров работает на процессорах, не поддерживающих MMX-технологии. Для того чтобы количество пользователей вашей программы было как можно большим, это обстоятельство необходимо учитывать. Соответственно при разработке программы должны предусматриваться альтернативные фрагменты кода, в которых функционирование MMX-команд будет эмулироваться с помощью целочисленных команд. Перед началом работы такая программа должна выполнять проверку текущего процессора на возможность поддержки им MMX-команд, по результатам которой следует передавать управление на соответствующий фрагмент кода. На основе чего выполняется подобная проверка?

В систему команд Intel-процессоров, начиная с последних 486 и Pentium, входит команда `CPUID` (см. приложение А учебника), с помощью которой можно получить информацию о текущем процессоре. Следующий листинг содержит пример программы, которая определяет факт поддержки процессором MMX-технологии.

```

+-----+
+ | Программа: test_cpuid.asm. Определение факта поддержки |
+ | процессором MMX-технологии. |
+-----+
.586p
model    small
.stack  100h
.data
mmx_mes    db    "Процессор поддерживает MMX-технологии". "$"
no_mmx_mes db    "Процессор не поддерживает MMX-технологии". "$"
.code
main       proc                ; начало процедуры main
mov        ax, @data
mov        ds, ax
xor        edx, edx
mov        eax, 1
cuidid
bt         edx, 23
jnc        no_mmx
mov        ah, 9
lea        dx, mmx_mes
int        21h
jmp        exit
no_mmx:    mov        ah, 9
lea        dx, no_mmx_mes
int        21h
exit:      mov        ax, 4c00h ; пересылка 4c00h в регистр ax
int        21h                ; вызов прерывания с номером 21h
main       endp              ; конец процедуры main
end        end                ; конец программы с точкой входа main

```

Пример применения MMX-технологии

В качестве примера применения MMX-технологии рассмотрим преобразование графического изображения. Как правило, большинство мультимедийных файлов, к которым можно отнести и графические файлы, представляют собой массивы однородных элементов. Такому массиву предшествует некоторая описательная информация (заголовок), в котором содержится общая информация о файле. Так как в подобных массивах размер элементов обычно одинаков, то их удобно обрабатывать группами. Наибольшая эффективность работы программы достигается, если использовать MMX-команды.

В качестве типового мультимедийного файла можно рассматривать файл растрового изображения. Среди типов файлов, предназначенных для хранения растровых изображений, наиболее простую структуру имеют BMP-файлы.

На практике часто возникает необходимость преобразования цветного изображения в полутоновое. Подобную возможность, в частности, поддерживают многие популярные редакторы растровой графики. Попробуем самостоятельно решить эту задачу, используя возможности команд ММХ-расширения.

Структурно файл растрового изображения в формате Windows BMP состоит из двух или трех блоков.

- **Обязательный заголовок** с информацией, характеризующий растровое изображение. По отношению к содержащимся в этом заголовке данным его можно разделить на две части:
 - первая часть, имеющая размер 14 байтов, предназначена для идентификации файла как растрового графического и хранения общей информации о нем;
 - вторая часть, имеющая размер 40 байтов, содержит характеристики самого растрового изображения.
- **Необязательный массив** с информацией о цвете. Элементы этого массива представляют собой структуру, состоящую из четырех полей. Размерность каждого поля составляет один байт. Три поля этой структуры содержат значения интенсивности красного, зеленого и синего цветов (RGB-модель цвета), четвертое поле всегда равно нулю и предназначено для выравнивания начала следующего элемента на четную границу. Для процессоров архитектуры Intel это означает существенное повышение производительности, так как выравненные данные выбираются из памяти быстрее. Более того, это обстоятельство дает хорошие предпосылки к эффективному использованию команд ММХ-расширения, так как данные во время обработки можно группировать. Массив с информацией о цвете отсутствует для растровых изображений с 24-разрядным представлением цвета (TrueColor). В случае с нашей программой мы имеем на входе именно такой файл. На выходе мы должны получить файл с полутоновым изображением, который уже будет содержать подобный массив оттенков.
- **Обязательный массив** с описанием пикселей растрового изображения. Формат элементов этого массива зависит от типа растрового изображения. В нашем случае мы имеем два типа растрового изображения: 24-разрядное (TrueColor) и 8-разрядное (полутоновое). Как отмечено выше, файл с описанием 24-разрядного изображения не содержит массива с информацией о цвете. В этом файле непосредственно за заголовком (54 байта) следует информация о цвете пикселей. Каждый пиксел описывается тремя байтами, которые содержат значения (из диапазона 0–255) интенсивностей красного, зеленого и синего цвета. Для других типов изображений информация о цвете каждого пиксела предоставляется по другому принципу. К примеру, для восьмиразрядного изображения (которое мы должны сформировать в нашей задаче) информация о цвете конкретного пиксела получается следующим образом. Каждый пиксел в массиве с описанием пикселей растрового изображения описывается одним байтом. Значение этого байта является индексом в массиве с информацией о цвете, рас-

смотренном в предыдущем пункте. Соответствующий элемент массива с информацией о цвете содержит значения красной, синей и зеленой составляющих цвета пиксела.

Рассмотрим суть алгоритма обработки, который реализует программа преобразования растрового изображения. Ниже приведены фрагменты исходного ВМР-файла (24-разрядное изображение) и соответствующего выходного ВМР-файла (8-разрядное полутоновое изображение), сформированного в результате программной обработки.

Для 24-разрядного цветного изображения с адреса 0000:0036 и до конца файла (адрес 0008:DCF5) располагается описание пикселей — три последовательных байта описывают красную, синюю и зеленую составляющие одного пиксела.

```
0000:0000 42 4D F6 DC 08 00 00 00 00 00 36 00 00 00 28 00 BM.....6...(.
0000:0010 00 00 B8 01 00 00 B8 01 00 00 01 00 18 00 00 00 .....
0000:0020 00 00 C0 DC 08 00 C2 1E 00 00 C2 1E 00 00 00 01 .....
0000:0030 00 00 00 00 00 00 49 7A A6 4A 7B A7 4D 7E AA 4F .....Iz.J{.M~.0
0000:0040 80 AC 50 81 AD 50 81 AD 4F 80 AC 4F 80 AC 4B 7C B.P..P..OБ.OБ.K|
...
0008:DCE0 A5 7A 93 A3 79 93 A3 79 92 A6 7D 99 B1 92 AD C8 .z..y..y..}.....
0008:DCF0 AD CA E5 C2 DE FC ...
```

Для 8-разрядного полутонового изображения с адреса 0000:0436 и до конца файла (адрес 0002:F875) находится описание пикселей. Каждый байт в этой части файла является индексом массива цветовых значений.

```
0000:0000 42 4D 76 F8 02 00 00 00 00 00 36 04 00 00 28 00 BM.....6...(.
0000:0010 00 00 B8 01 00 00 B8 01 00 00 01 00 08 00 00 00 .....
0000:0020 00 00 40 F4 02 00 C2 1E 00 00 C2 1E 00 00 00 01 ..@.....
0000:0030 00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02 .....
0000:0040 02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06 .....
0000:0050 06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A .....
0000:0060 0A 00 0B 0B 0B 00 0C 0C 0C 00 0D 0D 0D 00 0E 0E .....
0000:0070 0E 00 0F 0F 0F 00 10 10 10 00 11 11 11 00 12 12 .....
0000:0080 12 00 13 13 13 00 14 14 14 00 15 15 15 00 16 16 .....
0000:0090 16 00 17 17 17 00 18 18 18 00 19 19 19 00 1A 1A .....
0000:00A0 1A 00 1B 1B 1B 00 1C 1C 1C 00 1D 1D 1D 00 1E 1E .....
0000:00B0 1E 00 1F 1F 1F 00 20 20 20 00 21 21 21 00 22 22 .....!!!..""
0000:00C0 22 00 23 23 23 00 24 24 24 00 25 25 25 00 26 26 ".###.$$$.%%&&
```

... массив цветовых значений

```
0000:0400 F2 00 F3 F3 F3 00 F4 F4 F4 00 F5 F5 F5 00 F6 F6 .....
0000:0410 F6 00 F7 F7 F7 00 F8 F8 F8 00 F9 F9 F9 00 FA FA .....
0000:0420 FA 00 FB FB FB 00 FC FC FC 00 FD FD FD 00 FE FE .....
0000:0430 FE 00 FF FF FF 00 71 74 76 77 77 76 76 72 75 .....pgtvwvvru
0000:0440 78 77 72 69 5F 5A 61 62 63 66 69 6D 72 74 67 70 xwri_Zabcfimrtgp
0000:0450 77 77 74 72 70 6D 6E 6B 69 6A 6C 6C 68 64 63 6F wwtrpmkijllhdco
0000:0460 74 6F 6F 76 7A 75 68 73 75 6F 74 7E 78 69 76 71 toovzuhsuot-xivq
0000:0470 6A 6B 6C 74 7A 7E 86 7B 78 7C 78 6A 67 6E 69 79 jhlitz-.{x}xjgniy
...
0002:F860 73 71 6D 6C 70 74 77 75 75 79 7D 7E 7C 79 8B 8D sqmlptwuy~|y..
0002:F870 8C 8C 93 A7 C4 D8 ...
```

Программа преобразования 24-разрядного изображения в 8-разрядное выполнена в виде консольного приложения Windows. Таким образом, этим примером демонстрируется решение двух задач: во-первых, показана практика применения возможностей MMX-технологии обработки данных и, во-вторых, приведен еще один пример (см. главу 16 учебника) использования консольных Windows-приложений в программах на языке ассемблера.

Программа преобразования последовательно выполняет следующие действия.

1. Ввод имен исходного и выходного файлов. Следует отметить следующее требование к исходному файлу. Этот файл должен быть отсканирован с разрешением не менее 200 dpi, в противном случае возможны искажения. В нашем случае борьба за качество не является основной задачей, поэтому для исправления возможных искажений в программе не предпринимается никаких действий. Если сканера у вас нет, то следует подобрать файл соответствующего качества. Если у вас нет ни того ни другого, то среди файлов, прилагаемых к книге, есть файл, с которым вы можете проводить необходимые эксперименты.
2. Открытие исходного и создание выходного файла. Для работы с этими файлами используется еще один полезный механизм — *отображение файлов на память*. Этот механизм позволяет открыть файлы специальным образом и работать с ними далее как с обычными массивами. При рассмотрении программы обратите внимание на эту возможность и на то, как ею пользоваться в программах на языке ассемблера.
3. Определение корректности исходного файла. При этом проверяется два обстоятельства: первое — является ли он BMP-файлом, и второе — является ли он 24-разрядным файлом растрового изображения.
4. Формирование заголовка выходного файла на основании информации исходного файла. Основными при этом являются две задачи — определение размера выходного файла и заполнение полей, определяющих особенности данного растрового файла.
5. Формирование массива цветов. Массив содержит 256 элементов, что позволяет формировать 256-цветное растровое изображение. Файл с 8-разрядным полутоновым растровым изображением структурно ничем не отличается от 8-разрядного цветного файла. Единственное отличие можно увидеть при сравнении массивов цветов. Для массива 8-разрядного полутонового изображения значения красной, зеленой и синей составляющих одинаковы. Для цветных изображений подобной закономерности не существует, так как оттенки цветного изображения в модели RGB являются комбинациями отличающихся значений красной, зеленой и синей составляющих.
6. Формирование значений пикселей. Как уже упоминалось выше, каждый пиксел описывается одним байтом, который на самом деле является лишь индексом, обозначающим цвет в массиве цветов.
7. Закрытие файлов. В конце текста программы необходимо с помощью соответствующих функций Windows корректно закрыть файлы, в противном случае результаты работы программы будут потеряны.

Более детально алгоритм преобразования цветного изображения в полутоновое поясним по ходу рассмотрения текста программы, представленного ниже. При этом мы не будем обсуждать функции Windows, обеспечивающие функционирование программы. Такую информацию вы можете почерпнуть из многочисленных источников. В частности, программа разработана с использованием упоминавшегося выше механизма отображения файлов на память [10]. Основной акцент будет поставлен на MMX-команды.

```

: prg10.asm
+-----+
: | Программа: prg10.asm. Консольное приложение для Win32 (с использованием |
: | команд MMX-расширения и файлов, проецируемых в память). |
+-----+
.486
.model flat, STDCALL ; модель памяти flat
:STDCALL - передача параметров в стиле C (справа налево)
: вызываемая процедура чистит за собой стек
%NOINCL ; запрет вывода текста включаемых файлов
include mmx32.inc
include WindowConA.inc
:----- Объявление внешними используемых в данной программе
: функций Win32 (ASCII)
extrn AllocConsole:PRDC
:....
extrn WriteConsoleA:PROC
:----- структура BMP-файла
:....
:----- макроопределения типов
SSHORT equ <dw 0>
UINT equ <dw 0>
DWORD equ <dd 0>
LONG equ <dd 0>
WORD equ <dw 0>
BYTE equ <db 0>
:----- структура для установки положения курсора в консоли
Coord struc
xx SSHORT
yy SSHORT
Coord ends
:----- заголовок BMP-файла
BitMapFileHeader struc
bfType UINT ; символы "B" и "M"
bfSize DWORD ; размер файла в байтах
bfReserved1 UINT ; резерв
bfReserved2 UINT ; резерв
bfOffBits DWORD ; смещение в байтах к началу растрового
; изображения
; характеристики растрового изображения:
biSize DWORD ; размер данной структуры в байтах
; (должно быть равно 00000028h)
biWidth LONG ; ширина изображения в пикселах
biHeight LONG ; высота изображения в пикселах
biPlanes WORD ; число цветовых плоскостей (должно быть 1)
biBitCount WORD ; битов на пиксел (1, 4, 8, 24)
biCompression DWORD ; метод сжатия
biSizeImage DWORD ; размер собственно данных в байтах
biXPelsPerMeter LONG ; разрешение по горизонтали в пикселах/м
biYPelsPerMeter LONG ; разрешение по вертикали в пикселах/м
biClrUsed DWORD ; число цветов в изображении
biClrImportant DWORD ; число важных цветов изображения
BitMapFileHeader ends
.data
NumWri dd 0
inFile db 80 dup (20)
outFile db 80 dup (20)
con Coord <>
hinFile dd 0
houtFile dd 0
hMapinFile dd 0
hMapoutFile dd 0
TitleText db 'MMX-преобразование BMP-файла'. 0
mes1 db 'Введите путь к исходному файлу TrueColor: '
len_mes1 = $ - mes1

```

```

mes2      db      'Введите путь к выходному файлу: '
len_mes2  = $ - mes2
mesErr1   db      'Это не BMP-файл'
len_mesErr1 = $ - mesErr1
mesErr2   db      'Это не TrueColor-формат'
len_mesErr2 = $ - mesErr2
mesRet     db      'Нажмите любую клавишу для выхода'
len_mesRet = $ - mesRet
dOut      dd      0
dIn       dd      0
porog     label   dword
          dw      0000h
          dw      004dh
          dw      0097h
          dw      001ch      ; 00 77 151 28
RabOb1    dd      0
          dd      0
pixSizeLow dd      0
pixSizeHi dd      0
i8        dd      8
outFileSize dd     0
initRMMX0 label   dword
          db      00, 00, 00, 00, 01, 01, 01, 00
initRMMX1 label   dword
          db      02, 02, 02, 00, 02, 02, 02, 00
PointOutRegion dd     0
PointInRegion  dd     0
nWrByte       dd     0
temp          db     "B"
.code
start         proc   near      ; точка входа в программу
;-----      запрос консоли
              call   AllocConsole
              test   eax, eax
              jz     exit      ; неудача
;-----      текст окна заголовка
              push   offset TitleText
              call   SetConsoleTitleA
;-----      вывод строки текста:
;             вначале получим дескрипторы ввода и вывода консоли
              push   STD_OUTPUT_HANDLE
              call   GetStdHandle
              mov     dOut, eax      ; дескриптор ввода-вывода консоли
              push   STD_INPUT_HANDLE
              call   GetStdHandle
              mov     dIn, eax      ; дескриптор ввода-вывода консоли
;-----      установим курсор в позицию (2, 5)
              mov     con.xx, 2
              mov     con.yy, 5
              push    con
              push    dOut
              call    SetConsoleCursorPosition
              test    eax, eax
              jz     exit ; если неуспех
;-----      вывести приглашение на ввод имени исходного файла
              push    0
              push    offset NumWri
              push    len_mes1
              push    offset mes1
              push    dOut
              call    WriteConsoleA
;-----      установим курсор в позицию (2, 6)
              mov     con.xx, 2
              mov     con.yy, 6
              push    con

```

```

    push    dOut
    call    SetConsoleCursorPosition
    test    eax, eax
    jz      exit          ; если неудача
    push    0
    push    offset NumWri
    push    80
    push    offset inFile
    push    dIn
    call    ReadConsoleA
    lea     eax, inFile
    sub     NumWri, 2
    add     eax, NumWri
    mov     [eax].byte ptr 0
;----- установим курсор в позицию (2, 7)
    mov     con.xx, 2
    mov     con.yy, 7
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    test    eax, eax
    jz      exit          ; если неуспех
;----- вывести приглашение на ввод имени выходного файла
    push    0
    push    offset NumWri
    push    len_mes2
    push    offset mes2
    push    dOut
    call    WriteConsoleA
;----- установим курсор в позицию (2, 8)
    mov     con.xx, 2
    mov     con.yy, 8
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    test    eax, eax
    jz      exit          ; если неуспех

    push    0
    push    offset NumWri
    push    80
    push    offset outFile
    push    dIn
    call    ReadConsoleA
    lea     eax, outFile
    sub     NumWri, 2
    add     eax, NumWri
    mov     [eax].byte ptr 0
;----- открытие объекта ядра "файл" для исходного файла inFile
    push    NULL
    push    FILE_ATTRIBUTE_NORMAL
    push    OPEN_ALWAYS
    push    NULL
    push    0
    push    GENERIC_READ+GENERIC_WRITE ; разрешить чтение
                                         ; и запись в файл
    push    offset inFile
    call    CreateFileA
    cmp     eax, 0xffffffff
    je      exit          ; неудача
    mov     hInFile, eax
;----- создание объекта ядра "проецируемый файл"
; для исходного файла inFile
    push    NULL
    push    0 ; размер файла текущий

```

```

push    0
push    PAGE_READWRITE
push    NULL
push    hinFile
call    CreateFileMappingA
test    eax, eax
jz      exit      : неудача
mov     hMapInFile, eax
:----- проецирование файловых данных для исходного файла
: inFile на адресное пространство процесса
push    NULL
push    0          : сдвиг первого считываемого
                    : байта в файле
push    0
push    FILE_MAP_WRITE
push    hMapInFile
call    MapViewOfFile
test    eax, eax
jz      exit      : неудача
mov     PointInRegion, eax
:----- то ли мы прочитали?
mov     ebx, eax    : адрес начала исходного файла в памяти
cmp     [ebx].biSize, 2Bh
jne     exit_err1   : это не BMP-файл
cmp     [ebx].biBitCount, 1Bh
jne     exit_err2   : это не TrueColor-формат
:----- преобразование True -> Gray
:----- открытие объекта ядра "файл" для выходного файла outFile
push    NULL
push    FILE_ATTRIBUTE_NORMAL
push    OPEN_ALWAYS
push    NULL
push    0
push    GENERIC_READ+GENERIC_WRITE : разрешить чтение
                                       : и запись в файл
push    offset outFile
call    CreateFileA
cmp     eax, 0ffffffh
je      exit      : неудача
mov     houtFile, eax
:----- создание объекта ядра "проецируемый файл"
: для выходного файла outFile.
:----- вычисляем размер выходного файла TrueColor в пикселах
: (умножаем длину изображения на его высоту (в пикселах))
mov     eax, [ebx].biWidth
mul     dword ptr [ebx].biHeight
mov     pixSizeLow, eax : размер раstra в пикселах (младший)
mov     pixSizeHi, edx : размер раstra в пикселах (старший)
:----- в eax размер собственно растрового изображения в байтах
add     eax, 54 + 4 * 256 : 54 - размер заголовка
                    : 4 * 256 - массив цветовых значений
mov     outFileSize, eax
push    NULL
push    eax          : размер выходного файла в байтах
push    NULL
push    PAGE_READWRITE : PAGE_WRITECOPY
push    NULL
push    houtFile
call    CreateFileMappingA
test    eax, eax
jz      exit      : неудача
mov     hMapoutFile, eax
:----- проецирование файловых данных для выходного файла
: outFile на адресное пространство процесса
push    outFileSize

```

```

push    0                ; смещение первого считываемого байта
push    0
push    FILE_MAP_WRITE   ; FileMapAllAccess
push    hMapoutFile
call    MapViewOfFile
test    eax, eax
jz      exit             ; неудача
mov     PointOutRegion, eax
mov     edi, eax         ; адрес начала выходного файла в памяти
;...

```

Представленными в этом листинге действиями мы подготовили приложение к выполнению собственно преобразования. Для этого средствами Windows для работы с консолью были введены имена файлов. Затем эти файлы были открыты как файлы, отображаемые на память. Адреса файлов в памяти были помещены в ячейки `PointInRegion` и `PointOutRegion`. Для удобства работы эти адреса были продублированы также в регистрах `EBX` и `EDI` соответственно. Для того чтобы иметь возможность просматривать выходной файл стандартными средствами (графическим редактором `Paint`), необходимо сформировать корректный заголовок для выходного файла. При этом следует обратить внимание на формирование полей, характеризующих общий размер файла и размер собственно растрового изображения. Методика здесь простая. Из заголовка исходного файла извлекаются значения высоты и ширины растра в пикселах. Их произведение и дает размер всего растра в пикселах. Для 8-разрядного изображения это значение представляет собой длину растра в байтах. Для получения размера всего файла остается прибавить размер массива цветов (256×4 байта) и размер заголовка BMP-файла (54 байта). Эти действия мы выполняли, когда создавали выходной файл и определяли его размер. Их результат мы будем использовать ниже для формирования соответствующих полей заголовка. Рассмотрим продолжение листинга.

```

;----- формируем заголовок выходного файла
mov     ax, [ebx].bfType
mov     [edi].bfType, ax
mov     eax, outFileSize
mov     [edi].bfSize, eax ; общий размер файла
mov     [edi].bfReserved1, 0
mov     [edi].bfReserved2, 0
mov     [edi].bfOffBits, 54 + 4 * 256
mov     [edi].biSize, 28h
mov     eax, [ebx].biWidth
mov     [edi].biWidth, eax ; ширина растра в пикселах
mov     eax, [ebx].biHeight
mov     [edi].biHeight, eax ; высота растра в пикселах
mov     [edi].biPlanes, 1 ; число цветовых плоскостей
mov     [edi].biBitCount, 8 ; число битов на пиксел
mov     [edi].biCompression, 0 ; метод сжатия
mov     eax, outFileSize
sub     eax, 54 + 4 * 256
mov     [edi].biSizeImage, eax ; размер собственно
                                ; растрового изображения
mov     eax, [ebx].biXPelsPerMeter
mov     [edi].biXPelsPerMeter, eax ; ширина растра в пикселах
mov     eax, [ebx].biYPelsPerMeter
mov     [edi].biYPelsPerMeter, eax ; высота растра в пикселах
mov     [edi].biClrUsed, 100h ; число цветов в изображении
mov     [edi].biClrImportant, 0 ;

```

Далее формируем массив цветов. Особенности его содержимого мы обсуждали выше. Теперь нас интересует специфика процесса его формирования. Каждый эле-

мент этого массива занимает 4 байта, поэтому для его заполнения удобно использовать MMX-команды. Суть исполняемых действий очень проста, поэтому попробуйте разобраться в них самостоятельно, обращаясь для справки к описанию команд MMX-расширения, приведенных в тексте этой главы (см. выше подраздел «Система команд») и в приложении А учебника. Рассмотрим продолжение листинга.

```

:----- формируем массив цветов
add     edi, 54
mov     ecx, 128
movq    rmmx0, initRMMX0
movq    rmmx1, initRMMX1
xor     esi, esi
m3:     movq    [edi+esi*8], rmmx0
paddusb rmmx0, rmmx1
inc     esi
dec     ecx
jnz     m3

```

После формирования массива цветов все готово для выполнения собственно преобразования растрового изображения. Оно выполняется по определенному алгоритму, суть которого заключается в следующем. Каждому пикселу цветного изображения сопоставляется эквивалентный оттенок серого цвета. Для получения такого оттенка необходимо красную, синюю и зеленую составляющие цвета исходного пиксела умножить на определенные коэффициенты, после чего сложить. Полученный результат делится на 256 (это удобно делать сдвигом на 8 разрядов вправо). Частное от деления (размером в один байт) и является нужной градацией серого цвета, полученным из RGB-составляющих исходного цветного пиксела. Схематично процесс преобразования цвета пиксела показан на рис. 10.14.

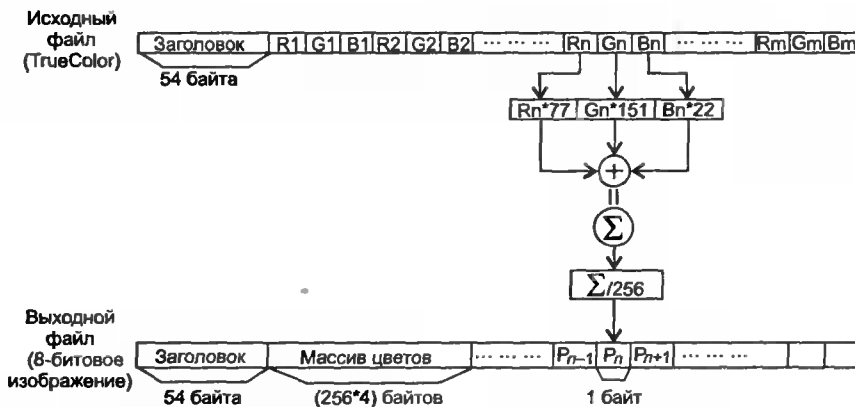


Рис. 10.14. Схема преобразования цветного пиксела (TrueColor) в эквивалентный серый

```

:----- позиционирование в выходном файле

mov     ecx, pixSizeLow
add     edi, 4 * 256      ; [edi] - на начало собственно
                        ; растрового изображения
                        ; в выходном файле

add     ebx, [ebx + 0ah]
dec     ebx              ; адрес в файле начала изображения - 1
xor     esi, esi

:----- загрузка в регистры цветовых значений для преобразования

```



```

m2:      punpcklbw rmmx0, [ebx][esi*4]
        psrlw    rmmx0, 8
        pmullw   rmmx0, porog
        movq     rmmx1, rmmx0
        psllq    rmmx1, 16
        paddusw  rmmx0, rmmx1
        psllq    rmmx1, 16
        paddusw  rmmx0, rmmx1
        psrlw    rmmx0, 8          : делим на 256
        movq     Rab0b1, rmmx0
        mov      al, byte ptr Rab0b1 + 6
        mov      [edi], al
        add      ebx, 3
        inc      edi
        dec      ecx
        jnz      m2
        jmp      exit

```

В данном варианте программы обрабатываются только две ошибочные ситуации: ввод некорректного имени файла (или пути к нему) и выбор в качестве исходного файла с типом, отличным от TrueColor. В случае возникновения других ошибок производится переход на метку `exit` с последующим выходом из программы.

```

:----- установим курсор в позицию (5, 10)
exit_err1:  mov     con.xx, 5
           mov     con.yy, 10
           push    con
           push    dOut
           call    SetConsoleCursorPosition
           test    eax, eax
           jz      exit          : если неуспех
:----- вывести сообщение об ошибке
           push    0
           push    offset NumWri
           push    len_mes2
           push    offset mesErr1
           push    dOut
           call    WriteConsoleA
           jmp     exit          : return
:----- установим курсор в позицию (5, 10)
exit_err2:  mov     con.xx, 5
           mov     con.yy, 10
           push    con
           push    dOut
           call    SetConsoleCursorPosition
           test    eax, eax
           jz      exit          : если неуспех
:----- вывести сообщение об ошибке
           push    0
           push    offset NumWri
           push    len_mes2
           push    offset mesErr2
           push    dOut
           call    WriteConsoleA
           jmp     return
:----- закрываем файлы
exit:      emms
:----- установим курсор в позицию (5, 12)
           mov     con.xx, 5
           mov     con.yy, 10
           push    con
           push    dOut
           call    SetConsoleCursorPosition

```

```

;----- вывести сообщение о нажатии любой клавиши
push 0
push offset NumWri
push len mesRet
push offset mesRet
push dOut
call WriteConsoleA
push 0
push offset NumWri
push 80
push offset inFile
push dIn
call ReadConsoleA

```

Перед выходом из программы необходимо подобающим образом закрыть файлы. Если этого не сделать, то их содержимое не изменится. Для корректного закрытия файла, спроецированного на память, следует вызвать функции `FlushViewOfFile` и `CloseHandle`. С помощью функции `FlushViewOfFile` измененные данные из памяти копируются на диск. Функция `CloseHandle` вызывается дважды для закрытия объектов ядра «файл» и «проецируемый файл».

```

push NULL
push PointInRegion
call FlushViewOfFile
push hInFile
call CloseHandle
push hMapInFile
call CloseHandle
push outFileSize
push PointOutRegion
call FlushViewOfFile
push hOutFile
call CloseHandle
push hMapOutFile
call CloseHandle
;----- выход из приложения
;----- готовим вызов VOID ExitProcess(UINT uExitCode)
return: push 0
call ExitProcess
start endp
end start

```

Дополнительные целочисленные MMX-команды

Представленные здесь команды нужно рассматривать как дополнение к системе целочисленных MMX-команд. Впервые они появились в архитектуре процессора Pentium III. Общее их количество — 12.

■ **PAVGB | PAVGW** приемник, источник — команды вычисления среднего двух значений. Значения представляют собой беззнаковые целочисленные упакованные элементы операндов приемник и источник размером байт/слово. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти, и его содержимое не изменяется. Изменяется только значение приемника, который должен быть MMX-регистром. В нем формируется результат вычисления среднего арифметического. Сама операция выполняется над парными элементами в обоих операндах следующим образом:

- 1) выполнить беззнаковое сложение парных элементов операндов приемника и источника;

- 2) запомнить перенос в старшие разряды;
- 3) сдвинуть вправо на один разряд результат сложения (без учета бита переноса), то есть разделить на 2;
- 4) результат сдвига сложить со значением переноса.

Дополнительные MMX-команды придают MMX-расширению ряд новых возможностей для организации более эффективной обработки данных. Так, следующие команды позволяют организовать доступ к отдельным элементам MMX-операнда.

- **PEXTRW** приемник, источник, маска — команда извлечения одного из четырех упакованных слов операнда источник. Источник должен быть MMX-регистром. Каждое выбираемое слово локализуется с помощью значения, задаваемого маской. Актуальность в ней имеют два младших бита, которые численно определяют номер слова (от 0 до 3), извлекаемого из источника. Это слово помещается в младшее слово приемника, являющегося одним из 32-разрядных регистров общего назначения. Старшее слово приемника обнуляется.
 - **PINSRW** приемник, источник, маска — команда вставки слова в одно из четырех упакованных слов операнда приемник. Приемник должен быть MMX-регистром. Источник может быть 32-разрядным регистром общего назначения или словом в памяти. Место вставки локализуется в приемнике с помощью значения, задаваемого маской. В ней значимы два младших бита, которые численно определяют номер слова (от 0 до 3) в приемнике, замещаемого новым значением из источника. Если источник — 32-разрядный регистр, то вставляемое из него слово должно быть младшим.
 - **PMAXUB | PMAXSW** приемник, источник — команды извлечения максимального значения из каждой пары упакованных элементов в операндах. Элементы представляют собой беззнаковые байты (для команды **PMAXUB**) или знаковые слова (для команды **PMAXSW**). Приемник должен быть MMX-регистром. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из максимальных элементов каждой пары формируется в операнде приемник.
 - **PMINUB | PMINSW** приемник, источник — команды для извлечения минимального значения из каждой пары упакованных элементов в операндах. Элементы представляют собой беззнаковые байты (для команды **PMINUB**) или знаковые слова (для команды **PMINSW**). Приемник должен быть MMX-регистром. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из минимальных элементов каждой пары формируется в операнде приемник.
- Следующая команда необычная — она позволяет выделить значения знака (старшего бита) упакованных байтовых элементов, после чего эти биты собираются в один байт и помещаются в младшие разряды регистра общего назначения. Подобная операция позволяет выполнять анализ знаков упакованных байтов в MMX-регистре и организовывать ветвление программы.
- **PMOVBK** приемник, источник — команда для формирования байтового значения, биты которого являются знаковыми для всех восьми байтов, упакованных в MMX-регистре. Источник должен быть MMX-регистром. Результат из ми-

нимальных элементов каждой пары формируется в операнде **приемник**. Приемник является 32-разрядным регистром общего назначения (формируемый из знаковых разрядов байт будет являться младшим).

- **PMULHUW** приемник, источник — команда умножения упакованных беззнаковых слов с выдачей в качестве результата старших слов произведения. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из старших слов произведения каждой пары формируется в операнде **приемник**. Приемник является MMX-регистром. Ранее мы уже рассматривали подобную команду — **PMULHW**, которая также работала со словами, но с учетом знака.
- **PSADBW** приемник, источник — команда вычисления суммарной разницы значений каждой пары беззнаковых байтов упакованных байтовых значений в приемнике и источнике. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат из старших слов произведения каждой пары формируется в операнде **приемник**. Приемник является MMX-регистром. Понимание алгоритма работы этой команды вызывает определенные трудности. Поэтому рассмотрим его более детально:
 - 1) для каждой пары байтовых элементов операндов приемник и источник вычислить модуль их разности;
 - 2) сложить модули разности всех пар байтовых элементов и записать полученный результат в младшее слово приемника;
 - 3) старшие три слова приемника обнулить.
- **PSHUFW** приемник, источник, маска — команда перераспределяет упакованные слова из операнда **источник** в операнд **приемник** в соответствии с маской, заданной операндом **маска**. Источник может быть MMX-регистром или 64-разрядной ячейкой памяти. Результат перераспределения слов второго операнда помещается в приемник, который является MMX-регистром. Исходное содержимое приемника не принципиально. Маска представляет собой байт, в котором численные значения номеров битов (0–7) определяют, какое из четырех слов источника необходимо разместить в соответствующих байтах (0–7) приемника (0 и 0, 1 и 1 и т. д.).

Например, если источник имеет значение 0012 950b 0054 0fd5 и задана маска 01101101b, то после применения команды **PSHUFW** в приемнике окажется значение 0054 950b 0012 0054.

ХММ-расширение архитектуры процессора Pentium

В 1999 году семейство процессоров Pentium фирмы Intel пополнилось новой моделью — процессором Pentium III. Основу его архитектуры составляет ядро процессора Pentium II, дополненное модулем SSE (Streaming SIMD Extensions — потоковое SIMD-расширение). Потоковое SIMD-расширение дополняет MMX-технологии средствами обработки данных с плавающей запятой. Выше мы условно называли этот тип MMX-расширения *ХММ-расширением*.

ХММ-расширение является продолжением политики Intel на ввод в архитектуру процессоров средств, повышающих эффективность вычислений на больших массивах однотипных данных. Программно-аппаратная модель ХММ-расширения имеет простую и гибкую структуру. В нее входит ряд новых регистров, новые команды и форматы данных. Программы, разработанные для ранних моделей процессоров, будут также выполняться на компьютерах, оснащенных процессором Pentium III, но они не смогут задействовать новшества, введенные ХММ-расширением. Для этого программа должна быть изначально написана с учетом логики работы программно-аппаратных средств, входящих в ХММ-расширение.

Модель ХММ-расширения

ХММ-расширение добавляет следующие новые элементы в архитектуру процессора:

- восемь 128-разрядных регистров с плавающей запятой — `xmm0 ...xmm7` (ХММ-регистры);
- формат данных (формат ХММ) размером 128 битов, представляющий собой совокупность из четырех упакованных 32-разрядных чисел с плавающей запятой в коротком формате — SPFP (Single Precision Floating Point);
- набор ХММ-команд;
- 32-разрядный регистр управления/состояния.

Рассмотрим их подробнее.

На рис. 10.15 показаны восемь 128-разрядных ХММ-регистров для хранения данных и один регистр управления/состояния. Каждый из этих регистров доступен соответствующими командами ХММ-расширения.



Рис. 10.15. ХММ-регистры

ХММ-регистры применяются только для хранения данных. Не имеет смысла задействовать их для хранения адресов памяти — для этого достаточно регистров общего назначения. При написании программы программист может смешивать команды обоих типов ММХ-расширения, так как они используют физически раз-

ные наборы регистров. Вследствие этого для работы с ними не нужны команды, подобные EMMS.

Основной тип данных XMM-расширения — число с плавающей запятой в коротком формате. Операндом XMM-команды является XMM-регистр или 128-разрядная ячейка памяти. Операнд содержит четыре числа с плавающей запятой в коротком формате. Поэтому говорят, что числа упакованы, и, соответственно, формат данных называется *упакованным форматом с плавающей запятой* (форматом XMM). В главе 17 учебника рассмотрены различные форматы данных сопроцессора, и указанный формат — один из них. Напомню основные его характеристики:

- длина числа — 32 бита;
- длины полей в формате: мантиссы — 24 бита, порядка — 7 битов, знака — 1 бит;
- диапазон значений:
 - двоичное числа — $2^{-126} \dots 2^{127}$;
 - десятичное числа — $1,18 \times 10^{-38} \dots 1,70 \times 10^{38}$.

Формат XMM-регистров или 128-разрядной ячейки памяти отражает структуру формата XMM. Числа в *упакованном коротком формате с плавающей запятой* в пределах этих объектов нумеруются от 0 до 3. Описание данных формата XMM в программе на языке ассемблера производится по тем же принципам, которыми мы руководствовались при описании данных целочисленного MMX-расширения. В памяти данные в формате XMM располагаются в соответствии с обычным для архитектуры Intel порядком — «младший байт по младшему адресу» (рис. 10.16).



Рис. 10.16. Схема расположения в памяти данных формата XMM

Назначение регистра управления/состояния MXCSR аналогично соответствующим регистрам сопроцессора CWR и SWR (см. главу 17 учебника). Так, большая часть битов этого регистра используется для маскирования/демаскирования исключений, другая часть — для установки способов округления, просмотра флагов состояния (табл. 10.8). Работа с этим регистром осуществляется с помощью команд LDMXCSR и FXRSTOR (загрузка содержимым из памяти) и STMXCSR и FXSAVE (сохранение содержимого в памяти).

Таблица 10.8. Назначение битов регистра mxcsr

Биты	Название	Назначение
0–5	IE, DE, ZE, OE, UE, PE	Фиксация исключений с плавающей запятой (см. главу 17 учебника). Для очистки этих битов предназначена команда LDMXCSR. Возникающие исключения фиксируются для всего упакованного формата, и средств для определения конкретного числа (из четырех), вызвавшего исключение, не существует

продолжение ➤

Таблица 10.8 (продолжение)

Биты	Название	Назначение
7–12	IM, DM, ZM, OM, UM, PM	Маскирование определенных типов исключений с плавающей запятой (см. главу 17 учебника). Начальное состояние этих битов единичное, что означает маскирование всех исключений. Для установки битов используется команда LDMXCSR. Маскирование исключений действует для всего упакованного формата
13–14	RC (Rounding Control)	Задание режима округления. По умолчанию действует режим округления к ближайшему представимому значению числа в коротком формате с плавающей запятой. При желании можно установить следующие режимы округления: 00 — округление к ближайшему; 01 — округление в меньшую сторону; 10 — округление в большую сторону; 11 — округление к нулю с усечением, то есть отбрасыванием дробной части
15	FZ (Flush-to-Zero)	Установка режима «сдвиг к нулю». Используется в ситуации переполнения следующим образом: возвращается нулевое значение вместе с истинным знаком результата; устанавливаются биты UE и PE в регистре MXCSR. По умолчанию бит FZ сброшен в 0. Наличие бита FZ не соответствует стандарту IEEE-754 на вычисления с плавающей запятой, так как этот стандарт требует при возникновении переполнения формировать денормализованный операнд. Данный режим введен из соображений эффективности. Ценой небольшой потери точности можно достигнуть более быстрой работы приложений, для которых ситуация переполнения обычна (так как не требуется вызова обработчика исключения). Демаскирование бита UE имеет приоритет перед режимом FZ

Остальные биты регистра MXCSR (биты 6 и 16–31) зарезервированы (равны 0). При попытке записать ненулевое значение в эти биты будет возбуждено исключение общей защиты.

Система команд

XMM-расширение процессора Pentium III дополнительно вводит в систему команд IA-32 70 новых машинных инструкций (рис. 10.17). Процессор Pentium 4 дополняет систему команд IA-32 еще 144 новыми командами. В основном это команды XMM- и MMX-расширений. Из-за большого их количества в данном разделе ограничимся рассмотрением команд XMM-расширения Pentium III. Этого достаточно для иллюстрации принципов и идей, заложенных в XMM-расширение. Информацию о XMM-командах Pentium 4 можно получить из приложения А учебника.

Команды перемещения данных

Ниже перечислены команды перемещения данных.

- **MOVAPS** приемник, источник — пересылка выравненных 128 битов из источника в приемник. Один из операндов (любой) — XMM-регистр. Другой операнд должен

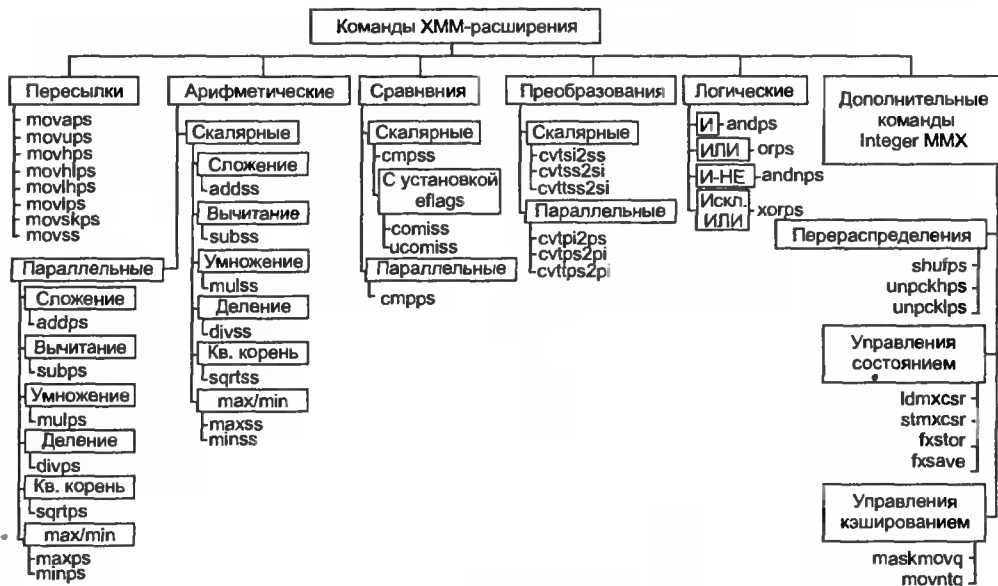


Рис. 10.17. Классификация команд XMM-расширения Pentium III

быть XMM-регистром или 128-разрядной ячейкой памяти. Адрес 128-разрядной ячейки памяти должен быть выровнен по 16-разрядной границе, иначе произойдет исключение общей защиты.

- **MOVUPS** приемник, источник — пересылка не выравненных 128 битов из источника в приемник. Один из операндов (любой) — XMM-регистр. Другой операнд должен быть XMM-регистром или 128-разрядной ячейкой памяти. В отличие от команды **MOVAPS**, данная команда не требует выравнивания по 16-разрядной границе адреса 128-разрядной ячейки памяти.
- **MOVHPS** приемник, источник — пересылка не выравненных 64 битов из источника в приемник. Один из операндов, но не оба одновременно, должен быть XMM-регистром. Другой операнд — 64-разрядная ячейка памяти. Если пересылка производится из памяти, то 64 бита помещаются в старшие 64 бита XMM-регистра, если пересылка производится из регистра, это старшие 64 бита XMM-регистра. Младшие 64 бита XMM-регистра не изменяются. Данная команда не требует выравнивания по 16-разрядной границе адреса 64-разрядной ячейки памяти.
- **MOVNLP** приемник, источник — пересылка 64 битов из источника в приемник. Оба операнда должны быть XMM-регистрами. В результате работы команды изменяется только содержимое младших 64 битов приемника в соответствии со схемой рис. 10.18.
- **MOVLHP** приемник, источник — пересылка 64 битов из источника в приемник. Оба операнда должны быть XMM-регистрами. В результате работы команды изменяются только содержимое старших 64 битов приемника в соответствии со схемой рис. 10.19.

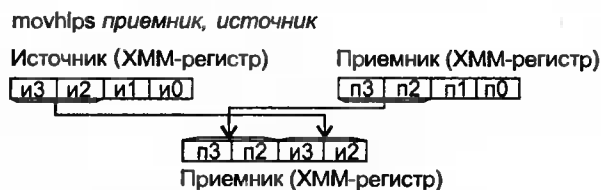


Рис. 10.18. Схема работы команды MOVHLPS

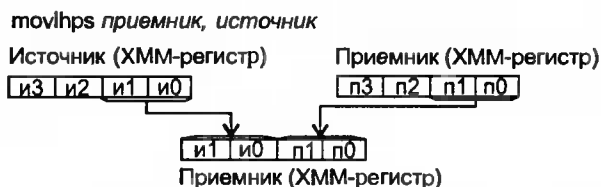


Рис. 10.19. Схема работы команды MOVLPS

- **MOVLPSP** *приемник, источник* — пересылка не выравненных 64 битов из источника в приемник. Один из операндов (любой) — XMM-регистр. Другой операнд должен быть 64-разрядной ячейкой памяти. Если пересылка производится из памяти, то 64 бита помещаются в младшие 64 бита XMM-регистра. Если источник — регистр, то пересылке подлежат младшие 64 бита регистра. Старшие 64 бита XMM-регистра не изменяются. Данная команда не требует выравнивания по 16-разрядной границе адреса 64-разрядной ячейки памяти.
- **MOVMSKPS** *приемник, источник* — пересылка знакового бита каждого из четырех упакованных чисел с плавающей запятой источника в младшие четыре бита приемника (рис. 10.20). Источник должен быть XMM-регистром. Приемник должен быть 32-разрядным регистром общего назначения. В дальнейшем полученную четырехразрядную величину можно использовать для организации условных переходов.

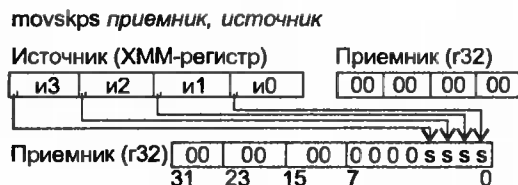


Рис. 10.20. Схема работы команды MOVMSKPS

- **MOVSS** *приемник, источник* — пересылка 32 младших битов из источника в приемник. Один из операндов или все операнды должны быть XMM-регистрами. Если XMM-регистром является только один из операндов, другой операнд должен быть 32-разрядной ячейкой памяти. Если пересылка производится из памяти, то 32 бита помещаются в младшие 32 бита XMM-регистра. Если источник — регистр, то пересылке подлежат младшие 32 бита регистра XMM. Остальные биты XMM-регистра не изменяются.

Арифметические команды

Набор арифметических команд XMM-расширения включает в себя обычные команды для выполнения арифметических операций. Особенность этого набора в том, что он содержит два типа арифметических операций: скалярных и параллельных. Отличить эти команды можно по их мнемонике: скалярные команды имеют суффикс *S*, а параллельные — суффикс *P*. Команды *скалярных* арифметических операций обрабатывают только младшие 32-разрядные двойные слова упакованных операндов, остальные двойные слова в операции не участвуют и не изменяются. Команды *параллельных* арифметических операций обрабатывают одновременно четыре упакованных двойных слова. Заметьте также, что в наборе команд XMM-расширения присутствуют команды для операции деления, которых не было в целочисленном MMX-расширении.

Сложение и вычитание

- **ADDPS (SUBPS)** приемник, источник — параллельное сложение (вычитание) элементов операндов приемник и источник. Операнды должны иметь формат XMM. Результат операции помещается в приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти.
- **ADDSS (SUBSS)** приемник, источник — скалярное сложение (вычитание) операндов приемник и источник. Младшие двойные слова операндов должны быть числами с плавающей запятой в коротком формате. Результат помещается в операнд приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое старших битов операндов не принципиально и не меняется.

Умножение и деление

- **MULPS (DIVPS)** приемник, источник — параллельное умножение (деление) операнда приемник на операнд источник. Операнды должны иметь формат XMM. Результат операции помещается в приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти.
- **MULSS (DIVSS)** приемник, источник — скалярное умножение (деление) операнда приемник на операнд источник. Младшие двойные слова операндов должны быть числами с плавающей запятой в коротком формате. Результат помещается в приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое старших битов операндов не имеет значения и не меняется.

Извлечение квадратного корня

- **SQRTPS** приемник, источник — параллельное извлечение квадратного корня из упакованных чисел с плавающей запятой операнда источник в формате XMM. Результат помещается в операнд приемник — XMM-регистр. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти.
- **SQRTSS** приемник, источник — скалярное извлечение квадратного корня из упакованного числа с плавающей запятой операнда источник в формате XMM. Результат помещается в операнд приемник. Приемник должен быть XMM-регистром.

ром. Источник может быть ХММ-регистром или 32-разрядной ячейкой памяти. Если источник — ХММ-регистр, то в операции извлечения участвует только число с плавающей запятой в его младшем двойном слове. Содержимое старших битов операндов не принимается во внимание и не меняется.

Извлечение максимальных/минимальных значений операндов

- **MAXPS (MINPS)** приемник, источник — параллельное извлечение максимальных (минимальных) значений из каждой пары упакованных чисел с плавающей запятой операндов источник и приемник в формате ХММ. Результат формируется из максимальных (минимальных) значений каждой пары и помещается в приемник — ХММ-регистр. Источник может быть ХММ-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется.
- **MAXSS (MINSS)** приемник, источник — скалярное извлечение максимального (минимального) значения из младшей пары упакованных чисел с плавающей запятой операндов источник и приемник в формате ХММ. Результат формируется из максимального (минимального) значений каждой пары и помещается в приемник — ХММ-регистр. Источник может быть ХММ-регистром или 32-разрядной ячейкой памяти, и его содержимое не меняется. Если источник — ХММ-регистр, то в операции сравнения и извлечения участвует только число с плавающей запятой в его младшем двойном слове.

Команды сравнения

Команды сравнения также делятся на два типа: скалярные и параллельные. Эти команды или их комбинации поддерживают полный набор условий сравнения. Обратите внимание на формат команд — он несколько необычен, так как условие сравнения задается непосредственно в операнде.

- **CMPPS** приемник, источник, условие — параллельное сравнение значений каждой пары упакованных чисел с плавающей запятой операндов источник и приемник в формате ХММ. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется в приемнике — ХММ-регистре. Источник может быть ХММ-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника, которое замещается 32-разрядными значениями 00000000h или 0fffffffh в зависимости от результата (00000000h — если условие не выполняется для данной пары значений упакованных чисел операндов источник и приемник, и 0fffffffh, если условие выполняется).
- **CMPSD** приемник, источник, условие — скалярное сравнение младших пар чисел с плавающей запятой в коротком формате операндов приемник и источник. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется в приемнике — ХММ-регистре. Источник может быть ХММ-регистром или 32-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника, младшее двойное слово которого замещается 32-разрядным значением 00000000h или 0fffffffh в зависимости от результата

(00000000h — если условие не выполняется для младшей пары чисел с плавающей запятой в коротком формате из операндов источник и приемник, и 0fffffffh, если условие выполняется).

Предыдущие две команды сравнивали операнды, изменяя при этом содержимое операнда приемник. Следующие две команды позволяют выполнить операцию сравнения без изменения его содержимого, но с одним ограничением — эти команды являются скалярными. По результатам их работы устанавливаются флаги в регистре EFLAGS.

- **COMISS** приемник, источник; условие — скалярное сравнение младших пар чисел с плавающей запятой в коротком формате операндов приемник и источник. Условие, по которому производится сравнение, определяется третьим операндом, представляющим собой непосредственное значение. Результат сравнения формируется установкой флагов в регистре EFLAGS (устанавливаются флаги ZF, PF, CF и сбрасываются флаги OF, SF, AF). Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое приемника и источника не меняется.
- **UCOMISS** приемник, источник; условие — неупорядоченное скалярное сравнение младших пар чисел с плавающей запятой в коротком формате операндов приемник и источник. Данная команда отличается от команды COMISS тем, что позволяет обнаружить ситуацию, когда младшие 32 бита приемника являются «тихим» или сигнальным не-числом. Результат сравнения формируется установкой флагов в регистре EFLAGS (устанавливаются флаги ZF, PF, CF и очищаются флаги OF, SF, AF). Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 32-разрядной ячейкой памяти. Содержимое приемника и источника не меняется.

Команды преобразования

Эта группа команд обеспечивает взаимное преобразование значений трех форматов: формата XMM, формата MMX и обычного целочисленного двойного формата представления числа в одном из регистров общего назначения. Набор команд также делится на два класса: скалярные и параллельные.

- **CVTPI2PS** приемник, источник — параллельное преобразование двух 32-разрядных целочисленных значений в операнде источник, которым является MMX-регистр целочисленного MMX-расширения или 64-разрядная ячейка памяти. Операнд источник содержит два 32-разрядных целочисленных значения. Результат преобразования представляется в виде двух 32-разрядных чисел с плавающей запятой в коротком формате. Эти значения размещаются в двух младших двойных словах XMM-регистра, представляющего собой операнд приемник. Старшие два двойных слова в приемнике не изменяются. Если не удастся получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра MXCSR.
- **CVTPS2PI** приемник, источник — параллельное преобразование двух 32-разрядных чисел с плавающей запятой в коротком формате, содержащихся в двух младших двойных словах операнда источник, в два 32-разрядных целых значения.

Операнд источник представляет собой ХММ-регистр или 64-разрядную ячейку памяти. Операнд приемник является адресом 64-разрядной ячейки памяти. Данная команда выполняет действие, обратное команде **CVTPI2PS**. Если не удастся получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра **MXCSR**.

В системе команд ХММ-расширения существует другой вариант последней команды — **CVTTPS2PI**, которая перед преобразованием отбрасывает дробную часть операнда источник.

- **CVTSI2SS** приемник, источник — скалярное преобразование одного 32-разрядного целочисленного значения из операнда источник в одно 32-разрядное число с плавающей запятой в коротком формате в операнде приемник. Операнд источник представляет собой один из 32-разрядных регистров общего назначения или 32-разрядную ячейку памяти. Результат преобразования в виде одного 32-разрядного числа с плавающей запятой в коротком формате помещается в младшее двойное слово 128-разрядного операнда приемник, представляющего собой ХММ-регистр. Старшие три двойных слова в операнде приемник не меняются. Если не удастся получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра **MXCSR**.

- **CVTSS2SI** приемник, источник — скалярное преобразование одного 32-разрядного числа с плавающей запятой в коротком формате из операнда источник в одно 32-разрядное целое число в операнде приемник. Операнд источник представляет ХММ-регистр или 32-разрядную ячейку памяти. Результат преобразования в виде одного 32-разрядного целочисленного значения помещается в операнд приемник, который является 32-разрядным регистром общего назначения. Старшие три двойных слова в операнде приемник не меняются. Если не удастся получить точный результат преобразования, то производится его округление в соответствии с режимом, заданным в соответствующих битах регистра **MXCSR**.

В системе команд существует другой вариант последней команды — **CVTTSS2SI**, которая перед преобразованием отбрасывает дробную часть операнда источник.

Логические команды

Все логические команды являются параллельными и реализуют над парами битов в обоих операндах логические операции **И**, **И-НЕ**, **ИЛИ**, **исключающее ИЛИ**.

- **ANDPS** приемник, источник — параллельное выполнение логического умножения (операция логического **И**) над парами битов упакованных чисел с плавающей запятой операндов источник и приемник в формате ХММ. Результат операции логического умножения формируется в операнде приемник — ХММ-регистре. Источник может быть ХММ-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника.
- **ANDNPS** приемник, источник — параллельное выполнение логической операции **И-НЕ** над парами битов упакованных чисел с плавающей запятой операндов источник и приемник в формате ХММ. Результат выполнения логической операции **И-НЕ** формируется в операнде приемник. Приемник должен быть ХММ-

регистром. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое сохраняется. Изменяется только значение приемника.

- **ORPS** приемник, источник — параллельное выполнение логического сложения (операция логического ИЛИ) над парами битов упакованных чисел с плавающей запятой операндов источник и приемник в формате XMM. Результат операции логического сложения формируется в операнде приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое остается прежним. Изменяется только значение приемника.
- **XORPS** приемник, источник — параллельное выполнение логической операции исключающего ИЛИ над парами битов упакованных чисел с плавающей запятой операндов источник и приемник в формате XMM. Результат логической операции исключающего ИЛИ формируется в операнде приемник. Приемник должен быть XMM-регистром. Источник может быть XMM-регистром или 128-разрядной ячейкой памяти, и его содержимое не меняется. Изменяется только значение приемника.

Команды управления состоянием

Ниже перечислены команды управления состоянием.

- **LDMXCSR** источник — команда загрузки регистра MXCSR (регистра состояния и управления XMM-расширения) из памяти. Источник должен быть 32-разрядной ячейкой памяти.
- **STMXCSR** приемник — команда сохранения содержимого регистра MXCSR (регистра состояния и управления XMM-расширения) в памяти. Приемник должен быть 32-разрядной ячейкой памяти.
- **FXRSTOR** источник — команда загрузки состояния сопроцессора (целочисленного MMX-расширения) и XMM-расширения в области памяти размером 512 байтов.
- **FXSAVE** приемник — команда сохранения состояния сопроцессора (целочисленного MMX-расширения) и XMM-расширения в области памяти размером 512 байтов. Подобно команде FSAVE, команда FXSAVE не инициализирует сопроцессор.

Команды перераспределения

Ниже перечислены команды перераспределения.

- **SHUFPS** приемник, источник, маска — команда упакованной перестановки двойных слов из операндов источник и приемник в операнд приемник в соответствии с маской, заданной операндом маска. Источник может быть MMX-регистром или 128-разрядной ячейкой памяти. Результат перераспределения двойных слов обоих операндов помещается в приемник, который является MMX-регистром. Третий операнд является непосредственным операндом. Принцип формирования результата следующий. Младшие два двойных слова результата формируют два из четырех двойных слов приемника. Старшие два двойных слова ре-

зультата формируют два из четырех двойных слов источника. Какие именно двойные слова будут включены в результат в качестве двойных слов, определяют биты маски. Значение первых двух битов маски определяют номер двойного слова источника, которое будет включено в результат в качестве первого двойного слова. Значение следующей пары битов (биты 2 и 3 маски) определяет номер двойного слова источника, которое будет включено в результат в качестве второго двойного слова. Значение следующей пары битов маски (4 и 5) определяет номер двойного слова приемника, которое будет включено в результат в качестве третьего двойного слова. Последняя пара битов маски (6 и 7) определяет номер двойного слова приемника, которое будет включено в результат в качестве четвертого двойного слова.

- **UNPCKHPS** приемник, источник — команда упакованного перемещения с чередованием старших двух двойных слов из операндов источник и приемник в операнд приемник. Источник может быть MMX-регистром или 128-разрядной ячейкой памяти. Результат перемещения двойных слов обоих операндов формируется в приемнике, который является MMX-регистром. Принцип формирования результата иллюстрирует рис. 10.21.

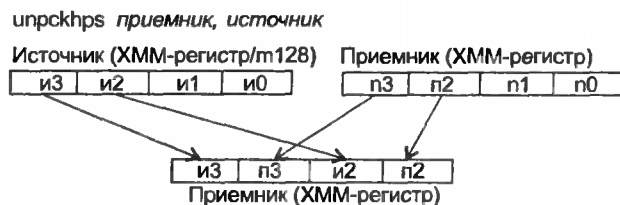


Рис. 10.21. Схема работы команды UNPCKHPS

- **UNPCKLPS** приемник, источник — команда упакованного перемещения с чередованием младших двух двойных слов из операндов источник и приемник в операнд приемник. Источник может быть MMX-регистром или 128-разрядной ячейкой памяти. Результат перемещения двойных слов обоих операндов формируется в операнде приемник, который является MMX-регистром. Выполнение команды иллюстрирует рис. 10.22.

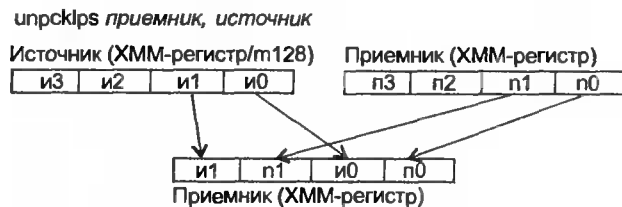


Рис. 10.22. Схема работы команды UNPCKLPS

Команды управления кэшированием

Команды XMM-расширения, рассмотренные в этом разделе, позволяют программисту управлять кэшированием данных. До появления моделей процессора с XMM-

расширением в системе команд IA-32 было несколько команд, которые влияли на работу кэша, но они не отличались гибкостью и с их помощью тяжело было учесть особенности конкретной задачи. Потребность оптимизировать работу кэша возникла из-за большого объема данных, которыми манипулирует большинство мультимедийных приложений. Чтобы предотвратить загрузку в кэш данных, которые впоследствии могут вообще не понадобиться, были введены специальные команды. Более того, с помощью новых команд можно организовать загрузку данных заранее, то есть до того, как они реально будут востребованы. Понятно, что это поднимет скорость работы приложений.

Вначале будут рассмотрены три команды, которые обеспечивают программное управление записью данных в память из регистров целочисленного или потокового MMX-расширений, обеспечивая при этом минимальное «загрязнение» кэш-памяти.

- **MASKMOVQ** источник, маска — команда выборочного сохранения в памяти байтов упакованного целого числа из MMX-регистра. Источник является MMX-регистром целочисленного MMX-расширения. Местоположение приемника в памяти задано неявно содержимым пары DS:DI/EDI. Маска определяет, какие байты будут сохранены в памяти. Это делается следующим образом. Значение для данной команды имеет знаковый разряд каждого упакованного байта маски. Если он равен 1, то значение соответствующего байта из источника (MMX-регистра) копируется в соответствующий байт памяти. Если же он равен 0, то содержимое соответствующего байта из источника в память не переносится. Наглядно этот процесс иллюстрирует рис. 10.23. Особенностью команды **MASKMOVQ** (и, собственно, причиной, по которой она включена в данную группу команд) является то, что при записи информации из регистров в память не производится кэширования этой информации и таким образом отсутствует загрязнение кэш-памяти первого или второго уровня.

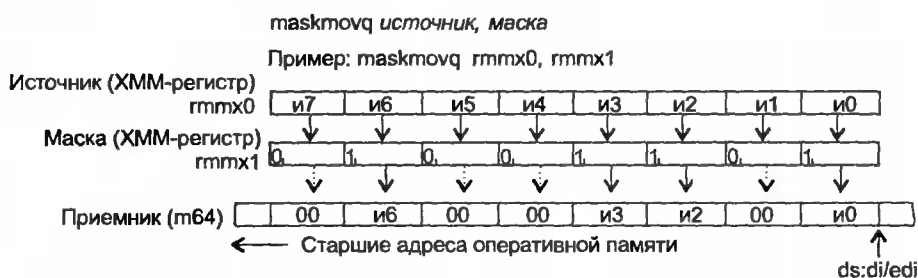


Рис. 10.23. Схема работы команды **MASKMOVQ**

- **MOVNTQ** приемник, источник — команда сохранения в памяти упакованных целых чисел в целочисленном формате MMX. Источник является MMX-регистром целочисленного MMX-расширения. Приемником является 64-разрядная ячейка памяти. Запись в память производится минуя кэш-память.
- **MOVNPS** приемник, источник — команда сохранения в памяти упакованных чисел с плавающей запятой в формате XMM. Источник представляет собой XMM-регистр. Приемником является 128-разрядная ячейка памяти, адрес которой

должен быть выравнен по границе параграфа (по значению, кратному 16). Запись в память производится минуя кэш-память.

Таким образом, приведенные выше три команды отличаются от обычных команд пересылки политикой работы с кэш-памятью.

После общего обзора возможностей ХММ-расширения рассмотрим несколько практических примеров программирования и пути решения специфических проблем, которые при этом могут возникать. Начнем с рассмотрения порядка описания данных, которыми манипулируют ХММ-команды Pentium III(4).

Описание упакованных и скалярных данных

Описание ХММ-данных в приложении обычно производится в одном из двух форматов:

- в массиве структур;
- в структуре, элементами которой являются массивы.

Описание точек изображения в трехмерном пространстве принято задавать в виде четырехмерного вектора (x, y, z, w) . Это связано с тем, что проекционные преобразования, необходимые для показа изображения с различных точек зрения, проще всего описываются матрицами 4×4 . Используя перечисленные выше форматы задания ХММ-данных, совокупность точек в трехмерном пространстве можно описать двумя способами.

- Первый способ — для каждой точки определить свой экземпляр структуры:

```
point_3D      struc
x             dd      0.0
y             dd      0.0
z             dd      0.0
w             dd      0.0
ends
.data
p1            point_3D 4 dup (<>) ; описание пирамиды массивом структур.
                                   ; каждая из которых представляет
                                   ; одну из четырех вершин
...

```

- Второй способ — все точки описать одной структурой, элементами которой являются массивы координат x, y, z, w :

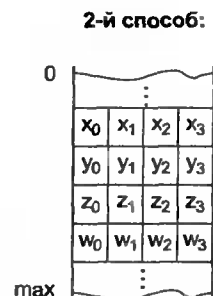
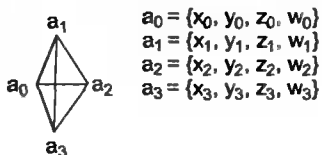


Рис. 10.24. Расположение в памяти описания вершин пирамиды

```

prism_point_3D struc
x          dd      4 dup (0.0)
y          dd      4 dup (0.0)
z          dd      4 dup (0.0)
w          dd      4 dup (0.0)
ends
.data
prism      prism_point_3D <>      ; структура, описывающая
                                   ; треугольную пирамиду (4 вершины)
                                   ; ...

```

Приведенные выше примеры описания пирамиды иллюстрирует рис. 10.24.

В большинстве приложений используется первый способ представления XMM-данных, хотя он и считается менее эффективным. При необходимости можно произвести перевод из одного представления данных в другое. Вариант такого преобразования показан ниже.

```

+-----+
| Программа: prg10_01.asm.                                     |
| Преобразование XMM-данных из одного представления в другое. |
+-----+

;...
prizm      struc
union
struc
xyzw1      dd      0.0      ; описание треугольной пирамиды (1)
xyzw2      dd      0.0
xyzw3      dd      0.0
xyzw4      dd      0.0
ends
struc
; описание треугольной пирамиды (2)
x          dd      4 dup (0.0)
y          dd      4 dup (0.0)
z          dd      4 dup (0.0)
w          dd      4 dup (0.0)
ends
ends
; конец объединения
ends
.data
prizm_1    prizm <>      ; экземпляр объединения
.code
; ...
;----- преобразование представлений вершин пирамиды (на месте)
lea        si, prizm_1
movlps     rxmm0, [si]      ; rxmm0 = ? ? y0 x0
movhps     rxmm0, [si+16]   ; rxmm0 = y1 x1 y0 x0
movlps     rxmm2, [si+32]   ; rxmm2 = ? ? y2 x2
movhps     rxmm2, [si+48]   ; rxmm2 = y3 x3 y2 x2
movaps     rxmm1, rxmm0     ; rxmm1 = y1 x1 y0 x0
shufps     rxmm0, rxmm2, 88h ; rxmm0 = x3 x2 x1 x0
shufps     rxmm1, rxmm2, 0ddh ; rxmm1 = y3 y2 y1 y0
movlps     rxmm2, [si+8]    ; rxmm2 = ? ? w0 z0
movhps     rxmm2, [si+24]   ; rxmm2 = w1 z1 w0 z0
movlps     rxmm4, [si+40]   ; rxmm4 = ? ? w2 z2
movhps     rxmm4, [si+56]   ; rxmm4 = w3 z3 w2 z2
movaps     rxmm3, rxmm2     ; rxmm3 = w1 z1 w0 z0
shufps     rxmm2, rxmm4, 88h ; rxmm2 = z3 z2 z1 z0
shufps     rxmm3, rxmm4, 0ddh ; rxmm3 = w3 w2 w1 w0
;----- на выходе получим следующее состояние XMM-регистров:
; RXMM0 = x3 x2 x1 x0. RXMM1 = y3 y2 y1 y0.
; RXMM2 = z3 z2 z1 z0. RXMM3 = w3 w2 w1 w0.
; теперь их необходимо сохранить в памяти
movups     [si], rxmm0

```



```

:----- сохраняем результат
movss  rez_sum, rxmm0
:....

```

Умножение матрицы на вектор

Умножение матрицы на вектор — наиболее характерная операция для вычислений в области машинной графики. Существуют различные способы формирования трехмерного изображения. В наиболее простом случае изображение на экране задается в виде опорных точек. К примеру, рассмотрим вариант, когда на экране находится трехмерное изображение, состоящее из отрезков прямых. Необходимая для его формирования информация хранится в памяти как список опорных точек — концов отрезков. Если изображение дается в трехмерной проекции, то описание каждой точки удобно задать в виде четырехмерного вектора (x, y, z, w) . Включение в трехмерный вектор (x, y, z) дополнительной координаты w объясняется тем, что проекционные преобразования, необходимые для показа изображения с различных точек зрения, описываются матрицами 4×4 . Поэтому для удобства реализации проекционных преобразований, зачастую сопровождаемых операциями умножения матриц и векторов, трехмерный вектор (x, y, z) представляют в виде четырехмерного вектора (x, y, z, w) , где значение w обычно принимается равным 1. Для выполнения самого преобразования подготовленная заранее опорная матрица умножается на этот вектор, в результате чего получается четырехмерный вектор (x', y', z', w') . Для обратного перехода к требуемому трехмерному вектору необходимо разделить координаты x', y', z' на w' , после чего удалить четвертую координату w' .

Матрица M преобразования и вектор V имеют следующий вид:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

Преобразования координат выполняются по формулам:

$$\begin{aligned}
 x' &= x \times m_{00} + y \times m_{01} + z \times m_{02} + 1 \times m_{03} \\
 y' &= x \times m_{10} + y \times m_{11} + z \times m_{12} + 1 \times m_{13} \\
 z' &= x \times m_{20} + y \times m_{21} + z \times m_{22} + 1 \times m_{23} \\
 w' &= x \times m_{30} + y \times m_{31} + z \times m_{32} + 1 \times m_{33}
 \end{aligned}$$

Для получения преобразованных координат в виде трехмерного вектора (x, y, z) делим x', y', z' на w' :

$$\begin{aligned}
 x &= x' / w' = (x \times m_{00} + y \times m_{01} + z \times m_{02} + 1 \times m_{03}) / (x \times m_{30} + y \times m_{31} + z \times m_{32} + 1 \times m_{33}) \\
 y &= y' / w' = (x \times m_{10} + y \times m_{11} + z \times m_{12} + 1 \times m_{13}) / (x \times m_{30} + y \times m_{31} + z \times m_{32} + 1 \times m_{33}) \\
 z &= z' / w' = (x \times m_{20} + y \times m_{21} + z \times m_{22} + 1 \times m_{23}) / (x \times m_{30} + y \times m_{31} + z \times m_{32} + 1 \times m_{33})
 \end{aligned}$$

Элементы матрицы и векторов представлены числами с плавающей запятой в коротком вещественном формате (4 байта).

Ниже приведены два варианта программы умножения матрицы на вектор — один при помощи стандартного сопроцессора, другой с использованием команд

ХММ-расширения (посредством профилировщика можно сравнить скорость преобразования). Результирующий трехмерный вектор замещает исходный.

Подпрограмма умножения матрицы 4×4 на четырехмерный вектор с использованием стандартного сопроцессора состоит из двух частей. Сначала вычисляется собственно произведение четырехмерной матрицы на четырехмерный вектор. Затем полученный четырехмерный вектор преобразуется в трехмерный посредством деления его компонентов на его четвертую координату.

```
: prg10_03.asm
.data
xyzw      dd      3 dup (0.0), 1.0  ; исходные координаты точки
                                         ; (необходимо инициализировать)
                                         ; и w=1.0
M          dd      16 dup (0.0)      ; матрица преобразования расположена
                                         ; по строкам (нужно инициализировать)

.code
;...
lea       di, xyzw                    ; адрес вектора
lea       bx, M                      ; адрес матрицы M
finit                                           ; инициализировать сопроцессор
fld       dword ptr [di+12]           ; включить в стек w
fld       dword ptr [di+8]            ; включить в стек z
fld       dword ptr [di+4]            ; включить в стек y
fld       dword ptr [di]              ; включить в стек x
xor       ecx, ecx
mov       cx, 4                      ; счетчик строк матрицы M

row:      ;----- умножаем очередную строку M на (x, y, z, w)
fld       dword ptr [bx]              ; первый элемент очередной строки
fmul      st, st(1)                   ; умножить на x
fld       dword ptr [bx+4]            ; второй элемент строки M
fmul      st, st(3)                   ; умножить на y
fadd                                             ; накопление суммы
fld       dword ptr [bx+8]            ; третий элемент строки M
fmul      st, st(4)                   ; умножить на z
fadd                                             ; накопление суммы
fld       dword ptr [bx+12]           ; четвертый элемент строки M
fmul      st, st(5)                   ; умножить на w
fadd                                             ; накопление суммы
fstp      dword ptr [di]              ; запомнить элемент вектора
add       di, 4                      ; на следующую координату в xyzw
add       bx, 16                     ; к следующей строке M
dec       cx
jnz       row                        ; повторить (всего 3 раза)

;----- теперь необходимо разделить полученные x', y', z', w' на w'
lea       di, xyzw                    ; адрес вектора
fld       dword ptr [di+12]           ; w' в стек
mov       cx, 4
cyc1:     fld       dword ptr [di]      ; очередную координату в стек
fddiv     st(0)/st(1)->st(0)           ; делим на w'
fstp      dword ptr [di]              ; запоминаем очередную координату
add       di, 4                      ; к следующей координате
dec       cx
jnz       cyc1

;----- получили (x, y, z, 1) => убираем w=1
; и получаем результат преобразования (x, y, z)
;...

```

Подпрограмму умножения матрицы 4×4 на четырехмерный вектор с использованием ХММ-расширения иллюстрирует следующий листинг.

```
: prg10_04.asm
.data
;ALIGN 16

```

```

xyzw      dd 3 dup (2.0), 1.0      ; исходные координаты точки
                                                ; (необходимо инициализировать)
                                                ; и w=1.0
M          dd 16 dup (0.0)          ; матрица преобразования расположена
                                                ; по строкам (нужно инициализировать)

.code
;...
lea     edi, xyzw      ; адрес вектора
lea     ebx, M          ; адрес матрицы M
mov     ecx, 4          ; повторений цикла
row:    movups rxmm0, xyzw ; загружаем вектор
        movups rxmm1, rxmm0 ; копируем вектор
        movups rxmm2, [ebx] ; загружаем очередную строку M
        mulps  rxmm2, rxmm1 ; умножаем вектор на строку M
;----- складываем частичные произведения
        shufps rxmm1, rxmm2, 0ah
        addps  rxmm2, rxmm1
        shufps rxmm3, rxmm2, 1
        addss  rxmm3, rxmm2 ; скалярное сложение
        movss  [edi], rxmm3 ; скалярная пересылка
;----- готовимся к вычислению следующей координаты
        add     edi, 4
        loop   row
;----- теперь необходимо разделить полученные x', y', z', w' на w'
lea     edi, xyzw      ; адрес вектора
movups  rxmm0, xyzw    ; загружаем вектор (x', y', z', w')
movss   rxmm1, rxmm0   ; пересылка w'
shufps  rxmm1, rxmm1, 0 ; размножаем w'
divps   rxmm0, rxmm1
movups  xyzw, rxmm0    ; сохраняем новое значение вектора
;----- получили (x, y, z, 1) => убираем w=1
;
; получаем результат преобразования (x, y, z)
;...

```

Поворот изображения

Следующая проблема, постоянно решаемая в задачах двумерной графики, — поворот изображения на определенный угол. Преобразование координат (x_i, y_i) опорных точек при повороте на угол α может выполняться по следующим формулам:

$$\begin{aligned}
 x'_i &= x_i \times \cos \alpha + y_i \times \sin \alpha, \\
 y'_i &= -x_i \times \sin \alpha + y_i \times \cos \alpha.
 \end{aligned}$$

Здесь x_i и y_i — координаты опорных точек, $i = 0..n-1$, n — число опорных точек изображения.

Аналогично программе умножения матрицы на вектор приведем два варианта программы поворота двумерного изображения (пусть это будет прямоугольник) — с использованием сопроцессора и при помощи XMM-расширения.

В представленной ниже программе поворота изображения с использованием стандартного сопроцессора предполагается, что координаты прямоугольника хранятся в виде массива вещественных чисел короткого формата. Элементы массива логически представляют собой последовательность — $x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3$. Угол поворота в радианах находится в переменной a .

```

; prgl0_05.asm
.data
;----- координаты квадрата (необходимо инициализировать)
;
; x0, y0, x1, y1, x2, y2, x3, y3
mas_xy   dd 8 dup (0.0)
a        dd 0.0          ; угол (необходимо инициализировать)

```

```

.code
;....
lea     si, mas_xy
mov     ecx, 4           ; цикл 4 раза - по количеству вершин
finit

:----- вычисляем sin a и cos a
fld     a                ; включаем в стек угол
fsin    ; вычисляем sin a
fxch    ; меняем st(0)<->st(1)
fcos    ; вычисляем cos a
fxch    ; меняем st(0)<->st(1)
fstp    a                ; выталкиваем a

:----- поворот изображения
cyl1:   fld     word ptr [si] ; включить в стек координату x элемента
        fld     word ptr [si+2] ; включить в стек координату y элемента
        fld     st(1)          ; дублируем их
        fld     st(1)
        fmul    st, st(5)      ; вычислить y * sin
        fxch    ; меняем st(0)<->st(1)
        fmul    st, st(4)      ; вычислить x * cos
        fadd    ; новая координата x
        fistp   word ptr[si]   ; передать новое значение x в память
        fmul    st, st(2)      ; вычислить y * cos
        fxch    ; меняем st(0)<->st(1)
        fmul    st, st(3)      ; вычислить x * sin
        fsubr   ; новая координата y
        fistp   word ptr[si+4] ; передать ее в память
        add     si, 8          ; продвинуть указатель массива mas_xy
        dec     ecx
        jnz     cyl1          ; повторить еще 3 раза
:----- в mas_xy преобразованные для поворота координаты квадрата
;....

```

Программа поворота изображения с использованием XMM-расширения представлена ниже.

```

; prg10_06.asm
.data
;ALIGN 16
:----- координаты квадрата (необходимо инициализировать)
:      x0, y0, x1, y1, x2, y2, x3, y3
mas_xy dd      8 dup (0.0)
a      dd      0.0           ; угол (необходимо инициализировать)
sin_a  dd      0.0
cos_a  dd      0.0
null   dd      0.0
.code
;....
lea     esi, mas_xy
mov     ecx, 4           ; цикл 4 раза - по количеству вершин
finit

:----- вычисляем sin a и cos a:
fld     a                ; включаем в стек угол
fsin    ; вычисляем sin a
fxch    ; меняем st(0)<->st(1)
fcos    ; вычисляем cos a
fxch    ; меняем st(0)<->st(1)
fstp    a                ; выталкиваем a
fstp    cos_a            ; выталкиваем cos_a
fstp    sin_a            ; выталкиваем sin_a

:----- поворот изображения
:----- готовим xmm-регистр RMM2 со значениями углов
movlps  xmm2, sin_a
movhps  xmm2, sin_a      ; RMM2 = cos_a sin_a cos_a sin_a

```

```

movss    rxmm2, null
subss    rxmm2, sin_a    ; RMM2 = cos_a sin_a cos_a -sin_a
cyc1:    movlps  rxmm0, [esi] ; RMM0 = ? ? y1 x1
        movhps  rxmm0, [esi] ; RMM0 = y1 x1 y1 x1
        shufps  rxmm0, rxmm0, 0b0h ; RMM0 = x1 y1 y1 x1
        mulps    rxmm0, rxmm2    ; RMM0 = RMM0*RMM2 =
        ; x1*cos_a y1*sin_a y1*cos_a x1*(-sin_a)
        shufps  rxmm1, rxmm0, 31h ; RMM1 =? x1*cos_a ? y1*cos_a
        addps    rxmm0, rxmm1    ; RMM0 = ? (x1*cos_a+y1*sin_a)?
        ; (y1*cos_a+x1*(-sin_a))
        shufps  rxmm0, rxmm0, 2  ; RMM0 = ? ? (y1*cos_a+x1*(-sin_a))
        ; (x1*cos_a+y1*sin_a)

;----- сохраняем результат:
movlps    [esi], rxmm0
;----- готовимся к вычислению нового положения следующей координаты
add        esi, 8
dec        cx
jnz        cyc1
;...

```

На этом мы закончим рассмотрение примеров программирования XMM-расширения. При разработке приведенных выше программ мы считали, что наш транслятор ассемблера поддерживает любые команды процессора Intel, в том числе и XMM-команды. Реально ситуация далека от этой идиллии. Мы уже упоминали, что трансляторы ассемблера не всегда успевают за процессом развития системы команд. Особенно это касается транслятора TASM, который «брошен на произвол судьбы» фирмой Borland (Inprize), и на сегодняшний день его судьба выглядит достаточно неопределенной. Другие фирмы-разработчики трансляторов ассемблера мы не рассматриваем (не потому, что они хуже — просто обсуждение достоинств и недостатков трансляторов ассемблера не является предметом данной книги). Поэтому на практике всегда можно столкнуться с ситуацией, когда любимый и хорошо знакомый транслятор не знает системы команд новейшего процессора. Как быть в этой ситуации? Как, не меняя транслятора, решить задачу понимания неизвестных команд процессора?

Моделирование команд XMM-расширения

Задачу адаптации компилятора (TASM) к новым командам процессора и, в частности, к MMX-командам мы уже решали. Приступая к практическому использованию XMM-команд, вы снова столкнетесь с этой проблемой. Задачу адаптации TASM (или другого компилятора ассемблера) к командам XMM-расширения можно решить двумя способами.

- Во-первых, разработать включаемый файл, в котором для каждой XMM-команды реализовать макрокоманду, моделирующую на базе существующих команд нужную XMM-команду. Традиционно, фирма Intel, зная об инерционности процесса разработки новых версий трансляторов ассемблера, вместе с подмножеством новых команд разрабатывает соответствующий включаемый файл для их поддержки в ассемблерных программах. Для подмножества XMM-команд такой файл называется `iaxmm.inc`. Он ориентирован на транслятор MASM (фирмы Microsoft) и не пригоден (требует доработки) для TASM. Однако при доработке TASM необходимо иметь в виду вопрос об авторских правах. Некоторые проблемы использования файла `iaxmm.inc` совместно с TASM обсуждены ниже.

- Во-вторых, можно разработать программу-препроцессор, на вход которой подавать исходный файл с программой на ассемблере, содержащей новые команды процессора, а на выходе получать текст, адаптированный для компиляции старым транслятором ассемблера. Этот путь имеет то преимущество, что теперь при появлении новых команд можно, не внося больших корректив в технологию разработки программ, всего лишь определенным образом модифицировать файл-препроцессор, дополнив его возможностями по обработке новых команд процессора. Более того, дополнив препроцессор средствами распознавания процессора (Intel или AMD), можно разрабатывать программы с использованием расширения 3DNow!. Рабочий материал для реализации этого способа вы можете найти среди файлов к книге в каталоге этой главы.

Поддержка ХММ-команд в файле `iaxmm.inc`

Текст файла `iaxmm.inc` доступен для загрузки с официального сайта компании Intel (<http://www.intel.com>). С точки зрения структуры включаемый файл `iaxmm.inc` представляет собой набор макрокоманд двух типов — основных и вспомогательных. Названия основных макрокоманд полностью совпадают с названиями ХММ-команд, моделирование которых они обеспечивают. Вспомогательные макрокоманды расположены в начале файла и служат для обеспечения работы основных макрокоманд. В частности, эти макрокоманды устанавливают тип операндов, указанных при обращении к основной макрокоманде, причем делают это, исходя из режима функционирования транслятора — 16- или 32-разрядного. Другое важное действие — установление соответствия между названиями ХММ-регистров и регистров общего назначения. Дело в том, что для моделирования ХММ-команд в 16- или 32-разрядном режиме работы ассемблера используются разные регистры общего назначения — 16-разрядные регистры в 16-разрядном режиме и 32-разрядные в 32-разрядном режиме.

Рассмотрим процесс моделирования ХММ-команд. В качестве основы для моделирования выступает одна из команд основного процессора. Эта команда должна удовлетворять определенным требованиям. Каковы они? В поисках ответа посмотрим на машинные коды ХММ-команд в приложении А учебника. Видно, что общими у них являются два момента:

- поле кода операции ХММ-команд состоит из двух или трех байтов, один из которых равен 0fh;
- большинство ХММ-команд использует форматы адресации с байтами `modR/M` и `sib` и соответственно допускает сочетание операндов как обычных двухоперандных команд целочисленного устройства — регистр-регистр или память-регистр.

Для моделирования ХММ-команд нужно подобрать такую команду основного процессора, которая удовлетворяет этим двум условиям. Во включаемом файле `iaxmm.inc` в качестве таких команд присутствуют две — `CMPSXCHG` и `ADD`. В процессе моделирования на место нужного байта кода операции этих команд помещаются байты со значениями кода операции соответствующей ХММ-команды. Когда процессор «видит», что очередная команда является ХММ-командой, то он начинает трактовать коды регистров в машинной команде как коды ХММ-регистров и ссылки на память размерностью, соответствующей данной команде. В машинном фор-

мате команды нет символических названий регистров, которыми мы пользуемся при написании исходного текста программы, например AX или BX. В этом формате они определенным образом кодируются. Например, регистр AX кодируется в поле **reg** машинной команды как 000 (двоичное). Если заменить код операции команды, в которой одним из операндов является регистр AX, кодом операции некоторой XMM-команды, то это же значение в поле **reg** процессор будет трактовать как регистр **xmm0**. Таким образом, в XMM-командах коды регистров воспринимаются соответственно коду операции. В табл. 10.9 приведены коды регистров общего назначения и соответствующих им XMM-регистров. В правом столбце этой таблицы содержится условное обозначение XMM-регистров, принятое в файле `iaxmm.inc`. Это же соответствие закреплено рядом определений в этом файле, которые иллюстрирует следующая программа.

```
DefineXMMxRegs Macro
IFDEF APP_16BIT
    xmm0 TEXT EQU <AX>
    xmm1 TEXT EQU <CX>
    xmm2 TEXT EQU <DX>
    xmm3 TEXT EQU <BX>
    xmm4 TEXT EQU <SP>
    xmm5 TEXT EQU <BP>
    xmm6 TEXT EQU <SI>
    xmm7 TEXT EQU <DI>
    RXMM0 TEXT EQU <AX>
    RXMM1 TEXT EQU <CX>
    RXMM2 TEXT EQU <DX>
    RXMM3 TEXT EQU <BX>
    RXMM4 TEXT EQU <SP>
    RXMM5 TEXT EQU <BP>
    RXMM6 TEXT EQU <SI>
    RXMM7 TEXT EQU <DI>
ELSE
    xmm0 TEXT EQU <EAX>
    xmm1 TEXT EQU <ECX>
    xmm2 TEXT EQU <EDX>
    xmm3 TEXT EQU <EBX>
    xmm4 TEXT EQU <ESP>
    xmm5 TEXT EQU <EBP>
    xmm6 TEXT EQU <ESI>
    xmm7 TEXT EQU <EDI>
    RXMM0 TEXT EQU <EAX>
    RXMM1 TEXT EQU <ECX>
    RXMM2 TEXT EQU <EDX>
    RXMM3 TEXT EQU <EBX>
    RXMM4 TEXT EQU <ESP>
    RXMM5 TEXT EQU <EBP>
    RXMM6 TEXT EQU <ESI>
    RXMM7 TEXT EQU <EDI>
ENDIF
endm
```

Таблица 10.9. Кодировка регистров в машинном коде команды

Код в поле reg	Регистр целочисленного устройства	XMM-регистр
000	AX/EAX	RXMM0
001	CX/ECX	RXMM1
010	DX/EDX	RXMM2

продолжение ➤

Таблица 10.9 (продолжение)

Код в поле <i>reg</i>	Регистр целочисленного устройства	XMM-регистр
011	BX/EBX	RXMM3
100	SP/ESP	RXMM4
101	BP/EBP	RXMM5
110	SI/ESI	RXMM6
111	DI/EDI	RXMM7

Теперь в исходном тексте программы можно использовать символические имена XMM-регистров в качестве аргументов макрокоманд, моделирующих XMM-команды.

Рассмотрим, как в файле `iamtx.inc` описано макроопределение для моделирования XMM-команды скалярной пересылки `MOVSS`.

```
; F3 0F 10 /r movss xmm1, xmm2/m32
; F3 0F 11 /r movss xmm2/m32, xmm1
movss macro dst:req, src:req
    XMMld_st_f3 opc_movss, dst, src
endm
```

Понимание структуры приведенного макроопределения не должно вызвать у вас трудностей. Начать следует с того, что данная команда содержит вложенный вызов макрокоманды `XMMld_st_f3`, у которой две задачи — определить вариант сочетания операндов, после чего сформировать правильный код операции и подставить его на место соответствующих байтов в команде `CMPSCHG`. В результате этих действий команда `CMPSCHG` «превращается» в XMM-команду `MOVSS`.

```
<1> XMMld_st_f3 macro op:req, dst:req, src:req
<2> local x, y
<3> DefineXMMxRegs
<4> IF (DPATTR(dst)) AND 00010000y : register
<5> x: lock cmpxchg src, dst
<6> y: org x
<7> byte 0F3H, 0Fh, op_ld
<8> org y
<9> ELSE
<10> x: lock cmpxchg dst, src
<11> y: org x
<12> byte 0F3H, 0Fh, op_st
<13> org y
<14> ENDIF
<15> UnDefineXMMxRegs
<16> endm
```

Центральное место в макроопределении `XMMld_st_f3` занимают команда целочисленного устройства (в данном случае — `CMPSCHG`) и директива `ORG`. Первое действие данной макрокоманды — выяснить тип операнда приемника (`dst`) в макрокоманде `MOVSS`, так как он может быть и регистром, и ячейкой памяти. Это необходимо для правильного определения кода операции, которая будет управлять направлением потока данных. После того как определен приемник данных, с помощью условного перехода осуществляется переход на ветвь программы, где будет выполняться собственно формирование соответствующего XMM-команде `MOVSS` кода операции.

Формирование кода операции XMM-команды MOVSS производится с помощью директивы ORG, которая предназначена для изменения значения счетчика адреса. В строках 6 или 11 директива ORG устанавливает значение счетчика адреса равным адресу метки *x*. Адрес метки *x* является адресом первого байта машинного кода команды CMPXCHG. Директива DB в следующих строках размещает по этому адресу байтовые значения 0F3H, 0Fh, *op&_ld* или 0F3H, 0Fh, *op&_st*, в зависимости от того, какое действие производится — загрузка (*_ld*) или сохранение (*_st*). Значения *opc_Movss*, с помощью которого формируются *op&_st* и *op&_ld*, определены в начале файла *iaxmm.inc*:

```
: ...
opc_Movss_ld = 010H
opc_Movss_st = 011H
: ...
```

Интерес представляет еще один характерный момент. Он касается порядка следования операндов в команде и роли бита *d* второго байта кода операции машинной команды. Мы обсуждали этот момент для моделирования команд MMX-расширения. Для XMM-команд рассуждения аналогичны, за исключением команды, на базе которой проводится моделирование — CMPXCHG. Например, значение второго байта кода операции *opc_Movss_st* = 011h (0001 0001b). Его бит *d* = 1, то есть данные передаются из регистра в память (процессор-память). Это нам и позволило в команде CMPXCHG изменить порядок следования операндов, в противном случае транслятор ассемблера команду не пропустит. Но команда MOVSS позволяет также производить передачу и в обратном направлении — память-процессор. Для того чтобы правильно смоделировать машинное представление XMM-команды, необходимо определить, каким объектом является операнд *dest* (приемник) — ячейкой памяти или регистром. После этого будет ясен тип второго операнда, так как реализованы могут быть лишь две схемы расположения операндов: регистр-регистр и память-регистр. Для выяснения типа операнда ассемблер MASM (не TASM) предоставляет оператор OPATTR, который имеет следующий синтаксис:

OPATTR (выражение)

Его эквивалент — другой оператор, работающий и в TASM:

.type выражение

Эти операторы в зависимости от типа операнда, указанного в качестве выражения, возвращают байтовые значения, которые приведены в табл. 10.3. Анализ возвращаемых значений производится строкой:

```
IF (OPATTR(dst)) AND 00010000b : register
```

Если операнд *dest* — регистр, то вырабатывается один код операции, если нет, то — другой.

Адреса операндов в памяти формируются таким же образом, как и для целочисленных команд, а их трактовка зависит от конкретной XMM-команды.

Модельно-зависимые регистры

Физически модельно-зависимые средства процессора представляют собой набор модельно-зависимых регистров (Model Specific Registers, MSR). Наличие и назначение этих регистров по определению привязано к конкретной модели процессо-

ра, что, в свою очередь, не гарантирует их поддержку будущими процессорами архитектуры Intel. Некоторые наиболее удачные из этих средств становятся стандартными для будущих моделей, а некоторые из них могут и не прижиться либо включаться лишь в архитектуру отдельных версий процессоров для поддержки специфических условий их эксплуатации. Один из модельно-зависимых регистров, ставший, если можно так выразиться, стандартом де-факто, будет рассмотрен в практической части данного раздела.

MSR-регистры обеспечивают управление различными аппаратно- и программно-зависимыми средствами, включая:

- счетчики мониторинга производительности;
- расширения отладки;
- средства поддержки исключения машинной ошибки и архитектуры машинного контроля (MCA);
- регистры MTRR для поддержки расширенных средств кэширования.

Чтение и запись в MSR-регистры осуществляются командами RDMSR и WRMSR. Большинство MSR-регистров инициализируются при программной инициализации процессора, многие из них можно впоследствии установить в соответствии с конкретными потребностями программы. Приведем краткое описание команд RDMSR и WRMSR.

Команды RDMSR и WRMSR

Команда RDMSR (ReaD from Model Specific Register) выполняет чтение из MSR-регистра. Действие команды заключается в контроле двух условий: во-первых, проверяется наличие нулевого уровня привилегированности кода, во-вторых, проверяется наличие в регистре ECX значения, адресующего один из MSR-регистров. Если хотя бы одно из этих условий не удовлетворяется, то выполнение команды RDMSR заканчивается. Если удовлетворяются оба условия, то значение MSR-регистра, адресуемого содержимым регистра ECX, помещается в пару 32-разрядных регистров EDX:EAX.

Команда WRMSR (WRIte to Model Specific Register) производит запись значения в один из 64-разрядных MSR-регистров. Действие команды заключается в контроле тех же двух условий: во-первых, проверяется наличие нулевого уровня привилегированности кода, во-вторых, команда убеждается в наличии в регистре ECX значения, адресующего один из MSR-регистров. Если хотя бы одно из этих условий не удовлетворяется, то работа команды завершается. Если соблюдены оба условия, то значение пары 32-разрядных регистров EDX:EAX пересылается в 64-разрядный MSR-регистр, номер которого задан в регистре ECX.

Заключение

Мы учимся писать, читая. По-видимому, причина того, что мы таким же образом не учимся программировать (то есть, читая чужие программы), — нечитабельность большинства программ.

Динар Нурмухамедович Бибишев

Рано или поздно все заканчивается. Подошло к концу и изложение материала данной книги. Наверняка найдется читатель, который скажет, что такой-то алгоритм более оптимально можно было реализовать другим способом, а такой-то материал изложить как-то иначе. Заранее принимая во внимание все возможные точки зрения, хотелось бы, тем не менее, заметить, что, на взгляд автора, материал в книге должен быть несколько избыточным, если хотите, «рыхлым». Это нужно для того, чтобы с ним могли разобраться читатели с различным уровнем подготовки. «Заоптимизировать» можно все, что угодно, только кому под силу будет со всем этим разбираться? Перефразируя известную поговорку, можно сказать: «чужой код — потемки».

Не все вопросы, востребованные на практике, получили свое отражение в книге. Признаюсь, первоначальный план книги был раза в два больше по объему, чем тот, что был реализован в конечном итоге. За кадром остались в основном проблемы системного программирования. Это сделано намеренно. Задачи системного программирования не менее важны, чем прикладного, но они тоже должны быть реализованы на базе обычных прикладных программ, способных достаточно профессионально взаимодействовать с пользователем и средой, в которой они функционируют. Поэтому автор посчитал нужным уделить основное внимание базовым вопросам прикладного программирования, а тонкости системных вопросов оставить «на потом». Насколько это обоснованно и удачно получилось — судить вам, уважаемый читатель...

Список литературы

Приводимый ниже список литературы содержит ссылки на книги двух типов:

- источники, материал которых был использован в процессе подготовки данной книги;
- источники, которые рекомендуются читателю для дальнейшего, более глубокого изучения как самого языка ассемблера, так и тех проблем, которые были затронуты в книге.

1. *Лебедев В. Н.* Введение в системы программирования. — М.: Статистика, 1975.
2. *Костин А. Е., Шаньгин В. Ф.* Организация и обработка данных в вычислительных системах: Учеб. пособ. для вузов. — М.: Высш. шк., 1987.
3. *Буза М. К.* и др. Лабораторный практикум по математическому обеспечению ЭВМ. Ч. 2. Структуры данных и методы трансляции: Учеб. пособ. для вузов. — Минск: Высш. шк., 1983.
4. Книга в двух изданиях, достаточно сильно отличающихся друг от друга:
 - *Вирт Н.* Алгоритмы + структуры данных = программы/Пер. с англ. — М.: Мир, 1985;
 - *Вирт Н.* Алгоритмы и структуры данных/Пер. с англ. — М.: Мир, 1989.
5. *Кнут Д.* Искусство программирования, том 2. Получисленные алгоритмы: Учеб. пособ. 3-е изд./Пер. с англ. — М.: Издательский дом «Вильямс», 2000.
6. *Советов Б. Я., Яковлев С. А.* Моделирование систем: Учебник для вузов по спец. «Автоматизированные системы управления». — М.: Высш. шк., 1985.
7. *Сван Т.* Освоение Turbo Assembler. — Киев: Диалектика, 1996.
8. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов/Пер. с англ. — М.: Мир, 1979.
9. *Гук М.* Аппаратные средства локальных сетей. Энциклопедия. — СПб.: Питер, 2000.
10. *Гудман С., Хидетниими С.* Введение в разработку и анализ алгоритмов. — М.: Мир, 1981.
11. Книга в двух изданиях, достаточно сильно отличающихся друг от друга:
 - *Рихтер Д.* Windows для профессионалов (программирование в Win32 API для Windows NT 3.5 и Windows 95)/Пер. с англ. — М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd», 1995;

- *Рихтер Д.* Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. 4-е изд./Пер. с англ. — СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.
- 12. *Дмитриева М. В., Кубенской А. А.* Турбо Паскаль и Турбо Си: Построение и обработка структур данных: Учеб. пособ. — СПб.: Издательство С.-Петербургского университета, 1996.
- 13. *Кнут Д.* Искусство программирования. Учеб. пособ. Том 1. Основные алгоритмы. 3-е изд./Пер. с англ. — М.: Издательский дом «Вильямс», 2000.
- 14. *Финогенов К. Г.* Самоучитель по системным функциям MS-DOS. — М.: МП «Малип», 1993.
- 15. *Пильщиков В. Н.* Сборник упражнений по языку Паскаль: Учеб. пособ. для вузов. — М.: Наука, 1989.
- 16. *Гилл А.* Введение в теорию конечных автоматов. — М.: Наука, 1966.
- 17. *Хантер Р.* Проектирование и конструирование компиляторов/Пер. с англ. — М.: Финансы и статистика, 1984.
- 18. Microsoft Corporation. Руководство программиста по Microsoft Windows 95/Пер. с англ. — М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd», 1997.
- 19. *Бек Л.* Введение в системное программирование: Пер. с англ. — М.: Мир, 1988.
- 20. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин/Пер. с англ. — М.: Мир, 1975.
- 21. Язык Си для профессионалов. — М.: И.В.К.-Софт, 1991.
- 22. *Гук М.* Аппаратные средства IBM PC. Энциклопедия. — СПб.: Питер, 1999.
- 23. Использование Turbo Assembler при разработке программ. — Киев: Диалектика, 1994.
- 24. *Григорьев В. Л.* Микропроцессор i486. Архитектура и программирование (в 4 книгах). — М.: Гранал, 1993.
- 25. *Кулаков В.* Программирование на аппаратном уровне. Специальный справочник. — СПб.: Питер, 2001.
- 26. *Ахо А. В., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. Учеб. пособ./Пер. с англ. — М.: Издательский дом «Вильямс», 2000.
- 27. *Кнут Д.* Искусство программирования, том 3. Сортировка и поиск. 3-е изд. Учеб. пособ./Пер. с англ. — М.: Издательский дом «Вильямс», 2000.
- 28. *Бентли Д.* Жемчужины творчества программиста/Пер. с англ. — М.: Радио и связь, 1990.
- 29. *Пустоваров В. И.* Ассемблер: программирование и анализ корректности машинных программ. — Киев: Издательская группа ВНУ, 2000.
- 30. *Касперски К.* Техника и философия хакерских атак. — М.: Солон-Р, 1999.
- 31. *Аммерал Л.* Интерактивная трехмерная графика/Пер. с англ. — М.: Сол Систем, 1992.

32. Кохонен Т. Ассоциативные запоминающие устройства/Пер. с англ. — М.: Мир, 1982.
33. Компаниец Р. И., Маньков Е. В., Филатов Н. Е. Системное программирование. Основы построения трансляторов: Учеб. пособ. для высших и средних учебных заведений. — СПб.: Корона принт, 2000.
34. Рудаков П. И., Финогенов К. Г. Программируем на языке ассемблера IBM PC: в 4 частях. — М.: Энтроп, 1995.
35. Зубков С. В. Ассемблер для DOS, Windows и UNIX. — М.: ДМК Пресс, 2000.
36. Брой М. Информатика: В 4 ч./Пер. с нем. — М.: Диалог-МИФИ, 1996.
37. Шилдт Г. Программирование на C и C++ для Windows 95. — Киев: Торгово-издательское бюро BVH, 1996.
38. <http://doc.lipetsk.ru/fravia/crcutut1.htm>, <http://silkerok.chat.ru/rus/doc/crc.htm>.
39. Юров В. Assembler. — СПб.: Питер, 2003.
40. Юров В. Assembler: Специальный справочник. — СПб.: Питер, 2000.
41. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. — СПб.: Питер, 2000.
42. Григорьев В. Л. Архитектура и программирование арифметического сопроцессора. — М.: Энергоатомиздат, 1991.
43. Assembly-Language Developer System, Version 6.1, for MS-DOS and Windows Operation System Microsoft Corporation.
44. Хаммел Р. Л. Последовательная передача данных: Руководство для программиста/Пер. с англ. — М.: Мир, 1996.
45. Культин Н. Б. C/C++ в задачах и примерах. — СПб.: BVH-Петербург, 2001.
46. Брукс Ф. Мифический человек-месяц, или Как создаются программные системы/Пер. с англ. — СПб.: Символ-Плюс, 1999.
47. Ларин Л. К. Математическое обеспечение ЭВМ. — Киев: КВИРТУ ПВО, 1985.
48. Бибишев Д. Н. Технология разработки и эксплуатации программного обеспечения. Методы и средства проектирования, реализации и сопровождения программных комплексов: Курс лекций. — Киев: КВИРТУ ПВО, 1990.

Юров Виктор Иванович

Assembler. Практикум

2-е издание

Главный редактор	<i>Е. Строганова</i>
Заведующий редакцией	<i>А. Кривцов</i>
Руководитель проекта	<i>Ю. Суркис</i>
Литературный редактор	<i>Е. Васильев</i>
Художник	<i>Н. Биржаков</i>
Корректоры	<i>С. Беляева, И. Смирнова</i>
Верстка	<i>Л. Харитонов</i>

Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 21.07.05. Формат 70×100/16. Усл. п. л. 32,25.

Доп. тираж 4000 экз. Заказ № 2228.

ООО «Питер Принт», 194044, Санкт-Петербург, пр. Б. Сампсониевский, д. 29а.

Налоговая льгота — общероссийский классификатор продукции

ОК 005-93, том 2; 953005 — литература учебная.

Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Федерального агентства по печати и массовым коммуникациям.

197110, Санкт-Петербург, Чкаловский пр., 15.

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Павелецкая», 1-й Кожовнический переулок, д.10; тел./факс (095) 234-38-15, 255-70-67, 255-70-68; e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а;
тел./факс (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Воронеж ул. 25 января, д. 4; тел./факс (0732) 39-43-62, 39-61-70;
e-mail: pitervm@comch.ru

Екатеринбург ул. 8 Марта, д. 2676, офис 203, 205; тел./факс (343) 225-39-94, 225-40-20;
e-mail: piter-ural@isnet.ru

Нижний Новгород ул. Совхозная, д. 13; тел. (8312) 41-27-31;
e-mail: piter@infonet.nnov.ru

Новосибирск ул. Немировича-Данченко, д. 104, офис 502;
тел./факс (383) 354-13-09, 211-27-18; e-mail: piter-sib@nisp.ru

Ростов-на-Дону ул. Ульяновская, д. 26; тел. (863) 269-91-22, 269-91-30;
e-mail: jupiter@rost.ru

Самара ул. Аминева, д.17; тел. (846) 994-22-62, 994-69-53; e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10-11; тел./факс (10-38-057) 712-27-05,
751-10-02, (0572) 58-41-45; e-mail: piter@kharkov.piter.com

Киев пр. Московский, 6, кор. 1, оф.33; тел./факс (10-38-044) 490-35-68, 490-35-69;
e-mail: office@piter-press.kiev.ua

БЕЛАРУСЬ

Минск ул. Бобруйская, д. 21, офис 3; тел./факс (10-375-17) 226-19-53;
e-mail: office@minsk.piter.com



Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73.**

E-mail: grigorjan@piter.com



Издательский дом «Питер» приглашает к сотрудничеству авторов.

Обращайтесь по телефонам: **Санкт-Петербург — (812) 103-73-72,**

Москва — (095) 974-34-50.



Заказ книг для вузов и библиотек: **(812) 703-73-73.**

Специальное предложение — e-mail: kozin@piter.com

В. И. Юров

ASSEMBLER

ПРАКТИКУМ

2-е издание

Преподавателям и студентам вузов, школьникам, специалистам, а также всем, кто интересуется практическими аспектами низкоуровневого программирования.

Книга является пособием для практического низкоуровневого программирования. С ее помощью вы научитесь создавать сложные, многофункциональные программы на языке ассемблера. Характер подобранного материала — сугубо прикладной. Практические советы дополняются примерами изящного решения популярных задач программирования. Полные тексты изучаемых программ можно найти на сайте издательства. Впрочем, и сама книга пестрит ассемблерными вставками, содержащими особо значимые и интересные места исходного кода. Практикум «Assembler» дополняет одноименный учебник того же автора, выпущенный издательством «Питер». Книга рассчитана прежде всего на студентов и специалистов, применяющих ассемблер для решения задач прикладного и системного программирования.

ISBN 5-94723-671-0



9 785947 236712

ПИТЕР®
WWW.PITER.COM

Темы, рассмотренные в книге:

- сложные структуры данных;
- рекурсивные процедуры;
- алгоритмы преобразования чисел;
- работа с файлами;
- эффективность исходных кодов;
- использование команд MMX и XMM.

**Не забудьте
купить учебник!**



Посетите наш web-магазин: <http://www.piter.com>