В. В. Одиноков, В. П. Коцубинский

Программирование на ассемблере

Горячая линия–Телеком

Программирование на ассемблере

«Допущено Сибирским региональным отделением УМО вузов России по образованию в области информационной безопасности для межвузовского использования в качестве учебного пособия по специальностям: 090105 – «Комплексное обеспечение информационной безопасности автоматизированных систем», 090102 – «Компьютерная безопасность», 090106 – «Информационная безопасность телекоммуникационных систем»

Москва
Горячая линия - Телеком
2011

УДК 681.3
ББК 32.973я73
К75

Рецензенты:
доктор техн. наук, профессор *Н. В. Замятин*, канд. техн. наук *Р. В. Мещеряков*

Одиноков В. В., Коцубинский В. П.

К75 Программирование на ассемблере. Учебное пособие для вузов. – М.: Горячая линия – Телеком, 2011. – 280 с.: ил.

ISBN 978-5-9912-0162-9.

Изучение самого «древнего» из языков программирования – *ассемблера* абсолютно необходимо для всех, кто желает в совершенстве овладеть искусством программирования. Почему же ассемблер столь важен? Во-первых, ни один язык программирования кроме него не дает такого ясного представления о строении аппаратуры ЭВМ. Во-вторых, язык ассемблера незаменим для разработки программ для встроенных устройств – микропроцессорных систем и микроконтроллеров, используемых в LCD телевизорах, автомагнитолах, сотовых телефонах, медицинской аппаратуре и других подобных разработках. Эти устройства часто не имеют операционных систем и поэтому управляющие программы должны быть автономными. Материал изложен от написания и отладки простейших программ в среде отладчика *debug* до практической разработки автономных программ, не использующих не только операционную систему, но и *BIOS*. Приведенные сведения иллюстрируются примером достаточно сложной программы – шестнадцатеричного редактора, который позволяет редактировать как содержимое оперативной памяти, так и файлы на диске. Теоретический материал в совокупности с предлагаемыми индивидуальными заданиями оформлен в виде лабораторных работ, что значительно облегчает индивидуальное восприятие предмета читателем и повышает ценность книги при использовании в учебном процессе.

Для студентов технических вузов, будет полезна для широкого круга специалистов, желающих систематизировать свои познания в области программирования и устройства микропроцессорных систем.

ББК 32.973я73

Адрес издательства в Интернет WWW.TECHBOOK.RU

Учебное издание

Одиноков Владимир Викторович, **Коцубинский** Владислав Петрович

ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

Учебное пособие для вузов

Компьютерная верстка Н. В. Дмитриевой
Обложка художника В. Г. Ситникова

Подписано в печать 20.11.10. Формат 70×100/16. Гарнитура Times. Печать офсетная.
Усл. печ. л. 23,35. Тираж 500 экз. Изд. № 10162
ООО «Научно-техническое издательство «Горячая линия–Телеком»

ISBN 978-5-9912-0162-9

В. В. Одиноков, В. П. Коцубинский, 2011
Оформление издательства «Горячая линия–Телеком», 2011

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 8
ЧАСТЬ I. ВЫПОЛНЕНИЕ МАШИННЫХ ПРОГРАММ 10
1. ДВОИЧНЫЕ И ШЕСТНАДЦАТЕРИЧНЫЕ ЧИСЛА 10
1.1. Двоичные числа 10
1.2. Шестнадцатеричные числа 12
1.3. Символьная информация 14
Лабораторная работа 1 14
2. ВЫПОЛНЕНИЕ ПРОГРАММ ПРОЦЕССОРОМ I8086 15
2.1. Структура аппаратных средств 15
2.2. Архитектура процессора i8086 17
2.3. Адресация памяти 19
2.4. Алгоритм работы процессора 21
3. ПРОГРАММИРОВАНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ 23
3.1. Чтение и заполнение регистров 24
3.2. Сложение двух чисел 25
3.3. Вычитание двух чисел 28
3.4. Умножение двух чисел 28
3.5. Деление двух чисел 28
Лабораторная работа 2 29
4. ВЫВОД СИМВОЛОВ НА ЭКРАН 29
4.1. Вывод одного символа 30
4.2. Команда завершения программы 31
4.3. Пересылка данных между регистрами 32
4.4. Вывод на экран строки символов 33
Лабораторная работа 3 35
5. ВЫВОД НА ЭКРАН ДВОИЧНЫХ ЧИСЕЛ 35
5.1. Флаг переноса 35
5.2. Циклический сдвиг 36
5.3. Организация циклов 37
5.4. Отладка программы 38
Лабораторная работа 4 39
6. ВЫВОД НА ЭКРАН ЧИСЕЛ В ШЕСТНАДЦАТЕРИЧНОЙ ФОРМЕ 39
6.1. Флаги состояния 39
6.2. Команды условного перехода 40
6.3. Вывод на экран одной шестнадцатеричной цифры 41
6.4. Вывод старшей цифры двузначного шестнадцатеричного числа 42
6.5. Вывод младшей цифры двузначного шестнадцатеричного числа 43
Лабораторная работа 5 44
7. СПИСКИ И ПРОЦЕДУРЫ 44
7.1. Несвязанные списки 44
7.2. Связанные списки 47
7.3. Программные стеки 48
7.4. Процедуры 49

8. ВВОД С КЛАВИАТУРЫ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ ... 52
8.1. Ввод одной шестнадцатеричной цифры ... 53
8.2. Ввод двузначного шестнадцатеричного числа ... 53
8.3. Более совершенный ввод шестнадцатеричных цифр ... 54
Лабораторная работа 6 ... 56
ЧАСТЬ II. АССЕМБЛЕРНЫЕ ПРОГРАММЫ В СРЕДЕ DOS ... 58
9. СИСТЕМНЫЕ ПРОГРАММЫ ... 58
9.1. Функции системных программ ... 58
9.2. Файлы ... 61
9.3. Язык управления DOS ... 64
9.3.1. Запуск DOS ... 64
9.3.2. Общие сведения о командах DOS ... 65
9.3.3. Некоторые системные команды DOS ... 66
9.3.4. Командные файлы ... 67
9.4. Утилита «DOS Navigator» ... 68
9.4.1. Представление на экране файловой структуры ... 68
9.4.2. Ввод команд DOS ... 70
Лабораторная работа 7 ... 71
10. ПРОСТЫЕ ПРОГРАММЫ НА АССЕМБЛЕРЕ ... 72
10.1. Общая структура простых ассемблерных программ ... 72
10.2. Пример программы на ассемблере ... 73
10.3. Подготовка программы к выполнению ... 73
10.4. Комментарии ... 74
10.5. Метки ... 75
10.6. Еще один пример программы ... 76
10.7. Вывод на экран двузначного шестнадцатеричного числа ... 77
Лабораторная работа 8 ... 79
11. ВЫВОД НА ЭКРАН ДЕСЯТИЧНЫХ И ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ ... 79
11.1. Получение алгоритма ... 79
11.2. Дерево подпрограмм ... 80
11.3. Запись на ассемблере ... 81
11.4. Многофайловая исходная программа ... 82
Лабораторная работа 9 ... 83
12. ДАМПИРОВАНИЕ ПАМЯТИ ... 83
12.1. Дампирование 16 байт ... 83
12.2. Дампирование 256 байт памяти ... 84
12.3. Очистка экрана ... 87
Лабораторная работа 10 ... 89
13. ПЕРЕПИСКА СЕКТОРА ПАМЯТИ ... 89
13.1. Функции переписки сектора ... 90
13.2. Чтение сектора ... 90
13.3. Алгоритмы процедур ... 91
13.4. Макрооператоры ... 92
Лабораторная работа 11 ... 94
14. ДИСПЕТЧЕР КОМАНД ... 95
14.1. Ввод команд ... 95

14.2. Алгоритм диспетчера ... 97
14.3. Выполнение команды ... 98
Лабораторная работа 12 ... 100
15. РЕДАКТИРОВАНИЕ ИНФОРМАЦИИ В ОП ... 100
15.1. Алгоритм редактирования сектора ... 100
15.2. Установка курсора ... 102
15.3. Выполнение команды ... 103
15.4. Запись шестнадцатеричной цифры ... 104
Лабораторная работа 13 ... 105
16. ОПЕРАЦИИ С ФАЙЛАМИ ... 105
16.1. Файловая система FAT ... 105
16.2. Создание и открытие файла ... 109
16.3. Операции чтения и записи ... 111
16.4. Закрытие и уничтожение файла ... 113
16.5. Пример программы ... 113
16.6. Другие операции с файлами ... 117
16.7. Модернизация редактора для работы с файлами ... 117
16.7.1. Алгоритм диспетчера ... 117
16.7.2. Алгоритмы процедур ... 119
Лабораторная работа 14 ... 121
17. РАЗДЕЛЬНАЯ ТРАНСЛЯЦИЯ ПРОГРАММЫ ... 121
17.1. Префикс программного сегмента ... 121
17.2. Получение прикладной программы ... 125
17.3. Программа типа com ... 128
17.4. Программа типа exe ... 132
17.5. Адресация памяти в программе типа exe ... 135
17.6. Применение нескольких объектных модулей ... 139
Лабораторная работа 15 ... 141
18. УПРАВЛЕНИЕ ПАМЯТЬЮ И ЗАПУСК ПРОГРАММ ... 142
18.1. Распределение памяти ... 142
18.2. Запуск прикладных программ ... 145
18.2.1. Описание системного вызова ... 145
18.2.2. Запуск программ из com-программы ... 146
18.2.3. Запуск программ из exe-программы ... 149
18.3. Многопрограммный редактор ... 151
18.3.1. Структура многопрограммного редактора ... 151
18.3.2. Главная программа ... 151
18.3.3. Дочерние программы ... 155
Лабораторная работа 16 ... 157
19. ПРЕРЫВАНИЯ И РЕЗИДЕНТНЫЕ ПРОГРАММЫ ... 157
19.1. Прерывания ... 157
19.1.1. Внешние аппаратные прерывания ... 157
19.1.2. Исключения ... 158
19.1.3. Программные прерывания ... 159
19.1.4. Алгоритм выполнения прерывания ... 160
19.2. Резидентные программы ... 162

19.2.1. Особенности резидентных программ ... 162
19.2.2. Использование свободных прерываний ... 163
19.2.3. Перехват прерываний ... 165
19.3. Резидентный редактор информации ... 168
Лабораторная работа 17 ... 172
ЗАДАНИЯ ДЛЯ ПРОГРАММИРОВАНИЯ ... 172
ЧАСТЬ III. РАЗРАБОТКА АВТОНОМНЫХ ПРОГРАММ ... 176
20. УПРАВЛЕНИЕ УСТРОЙСТВАМИ БЕЗ ПРЕРЫВАНИЙ ... 177
20.1. Управление устройствами на аппаратном уровне ... 177
20.2. Синхронный ввод-вывод ... 179
20.2.1. Логическая схема драйвера ... 179
20.2.2. Программа синхронного драйвера ... 180
20.3. Асинхронный вывод с общей памятью ... 182
20.3.1. Видеоадаптер ... 182
20.3.2. Видеопамять ... 183
20.3.3. Управление курсором ... 185
20.3.4. Логическая схема ... 186
Лабораторная работа 18 ... 188
21. УПРАВЛЕНИЕ УСТРОЙСТВАМИ С ПРЕРЫВАНИЯМИ ... 189
21.1. Контроллер прерываний ... 189
21.2. Алгоритм обработки прерываний ... 191
21.3 Пример драйвера ... 192
21.3.1. Логическая схема ... 192
21.3.2. Программа драйвера ... 194
21.4. Программирование драйвера клавиатуры ... 196
21.4.1. Логическая схема ввода с клавиатуры ... 196
21.4.2. Скан-коды ... 198
21.4.3. Алгоритмы программных модулей ... 198
21.5. Прямой доступ в память ... 200
Лабораторная работа 19 ... 205
22. ПРОСТЕЙШИЙ НАЧАЛЬНЫЙ ЗАГРУЗЧИК ... 206
22.1. Общая схема начальной загрузки ... 206
22.2. Программирование дисковода ... 207
22.3. Укрупненная структура первичного загрузчика ... 211
22.4. Программа начального загрузчика ... 212
22.5. Подготовка начального загрузчика к выполнению ... 216
Лабораторная работа 20 ... 218
23. НАЧАЛЬНАЯ ЗАГРУЗКА С ИСПОЛЬЗОВАНИЕМ ИМЕНИ ФАЙЛА ... 219
23.1. Общая структура информации на диске ... 219
23.2. Структура каталога ... 220
23.3. Структура таблицы FAT ... 221
23.4. Требования к загружаемой программе ... 222
23.5. Алгоритм модифицированного начального загрузчика ... 223
23.6. Алгоритмы процедур ... 226
Лабораторная работа 21 ... 228

24. ВТОРИЧНЫЙ НАЧАЛЬНЫЙ ЗАГРУЗЧИК-РУСИФИКАТОР 229
24.1. Функции вторичного загрузчика 229
24.2. Алгоритм вторичного начального загрузчика 230
24.3. Установка русского шрифта 232
24.4. Близкий косвенный вызов процедуры 236
24.5. Дальний косвенный вызов процедуры 238
24.6. Инициализация векторов прерываний 239
24.7. Выполнение системных вызовов 240
Лабораторная работа 22 241
25. РЕДАКТИРОВАНИЕ ВСЕЙ ОПЕРАТИВНОЙ ПАМЯТИ 242
25.1. Режимы работы процессора 242
25.2. Увеличение разрядности регистров 243
25.3. Аппаратные дескрипторы сегментов 245
25.4. Переход в режим линейной адресации 247
25.5. Совершенствование редактора 251
Лабораторная работа 23 252
ПРИЛОЖЕНИЕ 1. ОСНОВНЫЕ ОПЕРАТОРЫ АССЕМБЛЕРА 253
П1.1. Типы операторов 253
П1.2. Операторы обработки данных 253
П1.2.1. Арифметические операторы 253
П1.2.2. Логические операторы 255
П1.2.3. Операторы передачи данных 256
П1.2.4. Операции над регистром флагов 257
П1.2.5. Операторы сдвига 258
П1.2.6. Цепочечные (строковые) операторы 260
П1.3. Адресация данных 261
П1.4. Определение данных 263
П1.4.1. Метки 263
П1.4.2. Определение байтов 263
П1.4.3. Определение слов 264
П1.4.4. Определение констант 264
П1.4.5. Структуры 265
П1.5. Операторы передачи управления 266
П1.5.1. Операторы условных переходов 266
П1.5.2. Операторы безусловных переходов 267
П1.5.3. Операторы циклов 269
П1.5.4. Операторы процедур 270
П1.5.5. Другие операторы передачи управления 271
П1.6. Псевдооператоры для задания структуры программы 271
П1.7. Вспомогательные псевдооператоры 272
П1.8. Макрооператоры 274
ПРИЛОЖЕНИЕ 2. ВИРТУАЛЬНЫЕ МАШИНЫ «VIRTUAL PC» 276
ЛИТЕРАТУРА 279

ВВЕДЕНИЕ

Данное пособие предназначено для обучения студентов специальностей 090102 – «Компьютерная безопасность», 090105 – «Комплексное обеспечение информационной безопасности автоматизированных систем», 090106 – «Информационная безопасность телекоммуникационных систем», 220201 – «Управление и информатика в технических системах» и 230104 – «Системы автоматизированного проектирования» в рамках учебных курсов «Информатика», «Операционные системы» и «Вычислительные машины, системы и сети», но может быть использовано и при обучении студентов других специальностей, предполагающих углубленную подготовку по программированию.

Как следует из названия пособия, основной целью его изучения является обучение программированию на языке ассемблера. Выбор в качестве объекта изучения этого языка обусловлен следующим. Во-первых, только ассемблер позволяет получить представление об организации и функционировании аппаратуры ЭВМ. Другие языки программирования не предоставляют или почти не предоставляют такой возможности. Во-вторых, программное управление многими аппаратными устройствами предполагает использование ассемблера в качестве языка написания таких программ. И, наконец, в-третьих, программирование на ассемблере позволяет получить наиболее эффективные машинные программы как по времени выполнения, так и по затратам памяти. Данные свойства имеют существенное значение для тех программ, которые выполняются в среде ограниченных аппаратных средств, например в среде встроенных вычислительных систем, установленных непосредственно на управляемых объектах. Поэтому изучение ассемблера было и остается важным средством подготовки специалистов в области вычислительной техники.

Среди многих языков ассемблера в качестве объекта изучения выбран язык NASM для микропроцессоров фирмы INTEL. Подобный выбор обусловлен, во-первых, широким применением указанных микропроцессоров в качестве центральных процессоров для персональных ЭВМ. Во-вторых, микропроцессоры INTEL обладают сравнительно большим числом машинных команд и других аппаратных ресурсов. Знакомство с языком ассемблера для этих процессоров делает сравнительно несложной задачей последующее изучение ассемблера для какого-то другого процессора. В-третьих, распространяемый на бесплатной основе ассемблер NASM обладает очень полезным свойством: результирующая машинная программа ориентирована или на выполнение в среде требуемой операционной системы – DOS, Windows или UNIX, или может выполняться автономно – без использования операционной системы. Следует заметить, что тексты программ на языке NASM очень похожи на тексты программ, записанные на других известных языках ассемблера – TASM и MASM.

Полезно выполнить позиционирование данного пособия среди других книг по обучению ассемблеру. Все эти книги, независимо от их названий, можно разбить на две группы, имеющие условные названия: 1) «Программирование на ассемблере», «Язык ассемблера». Отличительной чертой второй группы книг является их справочный характер. Обилие технической информации делает эти пособия малопригодными для первоначального знакомства с предметом, так как они полностью соответствуют принципу «за деревьями не видно леса». Именно как хранилище технической информации эти книги полезны для опытных программистов на ассемблере. В отличие от них, в кни-

гах первой группы основное внимание уделяется не самому ассемблеру, а технологии его изучения. Именно к этой группе книг относится предлагаемое пособие.

Данное учебно-методическое пособие состоит из введения, трех частей и двух приложений. Каждая из частей пособия состоит из нескольких разделов, большинство из которых заканчивается заданиями на выполнение лабораторных работ. В конце второй части приводится список задач для программирования.

В первой части пособия «Выполнение машинных программ» (гл. 1–8) производится знакомство с языком, занимающим промежуточное положение между машинным языком и ассемблером. Обладая ассемблерной формой записи кодов операций и регистров, данный язык не имеет ассемблерных псевдооператоров и использует численные (не символьные) адреса. Знакомство с данным языком осуществляется с помощью отладчика Debug и предназначено прежде всего для изучения механизма выполнения процессором машинных программ. Кроме того, при изучении первой части пособия создается основа для последующего написания и отладки ассемблерных программ.

Во второй части пособия «Ассемблерные программы в среде DOS» (гл. 9–19) рассматриваются вопросы разработки программ, выполняемых в среде операционной системы DOS. Рассмотрение данной темы осуществляется на примере разработки достаточно большой программы – шестнадцатеричного редактора, предназначенного для редактирования информации в оперативной или во внешней памяти (в файле на магнитном диске). В процессе данной разработки изучаются наиболее важные операторы ассемблера, а также наиболее интересные системные вызовы. Многомодульная структура программы обеспечивается на одном или нескольких уровнях: подпрограммы, исходные файлы, объектные модули, бинарные файлы. Кроме того, разработанная программа может быть резидентной.

В третьей части пособия «Разработка автономных программ» (гл. 20–25) рассматриваются вопросы разработки программ, выполняемых без помощи операционной системы. При этом особое внимание уделяется управлению вводом-выводом на уровне портов, что позволяет создавать автономные программы, не использующие не только операционную систему, но и BIOS. Кроме того, рассматриваются вопросы разработки подпрограмм, выполняющих первоначальную загрузку и обслуживание обрабатывающих программ. Заканчивается третья часть пособия разработкой автономного шестнадцатеричного редактора, позволяющего редактировать любую область оперативной памяти.

В приложении 1 приводятся краткие сведения об основных операторах ассемблера. В приложении 2 содержатся сведения об использовании виртуальной машины «Virtual PC», достаточные для применения этого полезного инструмента при выполнении лабораторных работ.

Часть I. ВЫПОЛНЕНИЕ МАШИННЫХ ПРОГРАММ

1. Двоичные и шестнадцатеричные числа

Прежде чем рассматривать выполнение машинных программ аппаратурой ЭВМ, рассмотрим представление в памяти машины чисел, а также алфавитно-цифровых символов. Это представление потребуется нам в дальнейшем при изучении машинных программ, а также обрабатываемых ими данных.

1.1. Двоичные числа

Чтобы сделать вычислительные системы более надежными и простыми, их аппаратура строится из простейших электронных схем, которые могут находиться только в двух состояниях. Одно из них обозначается **0**, а другое – **1**. Такая схема предназначена для длительного или краткого хранения самой мелкой единицы информации – ***бита*** (от ***BI****nary digi****T*** – двоичная цифра).

Любое число можно представить в виде цепочки битов. Такое представление числа называется ***двоичным числом***. Цепочка из 8 бит называется ***байтом*** (рис. 1.1).

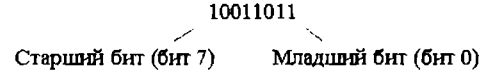

Рис. 1.1. Пример байта

Величина двоичного числа определяется относительной позицией каждого бита и его значением. Позиционный вес младшего бита $2^0 = 1(10)$, где 1(10) – единица в десятичной системе счисления. Следующий бит имеет вес $2^1 = 2(10)$. Вес любой позиции получается удвоением веса предыдущей позиции (рис. 1.2).

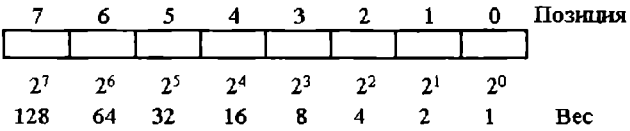

Рис. 1.2. Веса позиций байта

Для преобразования десятичного числа в двоичное число можно использовать один из двух методов – метод деления и метод вычитания. Первый из этих методов широко используется в программах, выполняющих преобразование чисел из одной системы счисления в другую. Этот метод будет рассмотрен нами в других разделах, при описании соответствующих программ. Сейчас мы будем использовать ***метод вычитания***, главное достоинство которого – наглядность. Согласно этому методу для преобразования десятичного числа в двоичное надо сделать ряд вычитаний, каждое из которых даст значение одного бита.

Прежде всего вычтите из десятичного числа наибольший возможный двоичный вес и запишите 1 в эту позицию бита. Затем из результата вычтите новый наибольший возможный двоичный вес и запишите 1 в эту новую позицию бита. Так – до получения нулевого результата. Например, преобразование числа 50 в двоичное число:

```
 50
 32 (бит 5 = 1)
 18
 16 (бит 4 = 1)
  2
  2 (бит 1 = 1)
  0
```

Записывая 0 в остальные позиции битов (биты 0,2,3), получаем окончательный результат: 110010.

Для выполнения обратного преобразования следует сложить десятичные веса тех позиций, в которых стоит 1:

32 (бит 5) + 16 (бит 4) + 2 (бит 1) = 50

Байт может представлять десятичные положительные числа от 0 (00000000) до 255 (11111111). Число 255 может быть получено двумя способами:

1) суммированием весов всех битов байта;

2) по формуле $2^8 - 1$, где 8 – номер первого бита, не вошедшего в состав байта.

Оперативная память ЭВМ состоит из блоков памяти по 1024 байт. Число 1024 есть 2^{10}. Число 1024 имеет стандартное обозначение ***К***. Следовательно, ЭВМ, имеющая 48 К памяти, содержит 48 × 1024 = 49152 байт.

Машинным словом будем называть битовую строку длиной 16 бит. Одно слово содержит 2 байта (рис. 1.3). Каждый бит слова имеет свой вес. Просуммировав все веса, найдем максимальное целое число без знака, которое можно записать в одно слово, оно равно $2^{16} - 1 = 65535$.

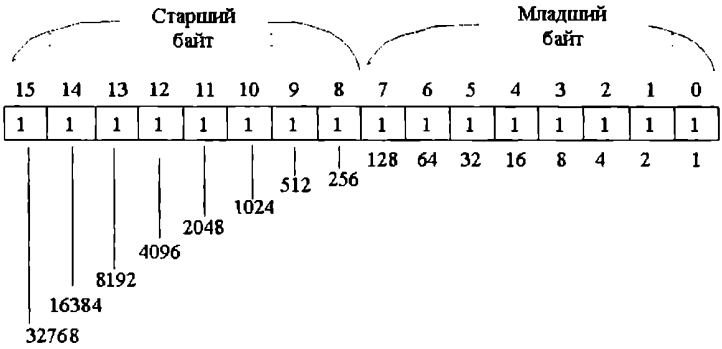

Рис. 1.3. Веса позиций слова

Двоичное содержимое байта или слова может рассматриваться (интерпретироваться) как число без знака и как число со знаком. ***Число без знака*** занимает все 16 бит слова или 8 бит байта. Оно может быть только неотрицательным. Просуммируем два таких числа:

```
  00111100          60 (10)
+ 10110101        + 181 (10)
----------        ----------
  11110001          241 (10)
```

Обратим внимание, что единица, появившаяся в старшем бите результата, свидетельствует лишь о возросшей величине результата, который для беззнаковых чисел может быть только неотрицательным.

Если слово (байт) содержит ***число со знаком***, то в старшем бите содержится знак (0 есть +, 1 есть –), а оставшиеся 15 или 7 бит содержат само число в ***дополнительном коде***. Дополнительный код положительного числа совпадает с записью такого же числа без знака. Для получения дополнительного кода отрицательного числа из его абсолютного значения используется следующее правило:

1) все биты числа (в том числе и знаковый) инвертируются;

2) к полученному числу прибавляется 1.

Например, получим дополнительный код числа –65:

65(10) = 01000001(2).

1) 10111110

2) \+ 1

10111111 = -65(10)

Для получения абсолютного значения отрицательного числа повторяют эти же самые два действия. Например:

–65(10)= 10111111

1) 01000000

2) \+ 1

01000001 = 65(10)

Сумма +65 и –65 должна составить ноль:

01000001 (+65)

\+ 10111111 (-65)

00000000

В данном примере у нас произошли два интересных переноса: 1) в знаковый (7) разряд; 2) за пределы байта. Первая единица переноса обрабатывается как обычно, а вторая теряется. Оба переноса считаются правильными.

Вычитание двух положительных чисел заменяется суммированием первого числа с дополнением второго. Таким образом, как суммирование, так и вычитание выполняет одно и то же аппаратное устройство – сумматор. Кроме этого свойства использование дополнения имеет еще одно достоинство – число ноль имеет единственное представление, т. е. нет двух нулей – положительного и отрицательного. Приведем целые числа в окрестностях 0:

\+ 3 00000011

\+ 2 00000010

\+ 1 00000001

0 00000000

– 1 11111111

– 2 11111110

– 3 11111101

Отсюда видно, что нулевые биты в отрицательном двоичном числе фактически определяют его величину: рассмотрите весовые значения нулевых битов так, как если бы это были единичные биты, сложите эти значения и прибавьте 1.

1.2. Шестнадцатеричные числа

В то время как процессор и другие устройства ЭВМ используют только двоичное представление информации, такое представление очень неудобно для человека, который

анализирует содержимое памяти ЭВМ. Введение шестнадцатеричных чисел значительно облегчает эту задачу. При этом, как будет показано далее, шестнадцатеричное число фактически представляет собой сокращенную форму записи двоичного числа.

Допустим, что мы хотим проанализировать содержимое четырех последовательных байтов (двух слов). Разделим мысленно каждый байт пополам и запишем для каждого полубайта соответствующее десятичное значение:

0101 1001 0011 0101 1011 1001 1100 1110

5 9 3 5 11 9 12 14

Чтобы не использовать для некоторых полубайтов две десятичные цифры, рассмотрим систему счисления: 10 = *A*, 11 = *B*, 12 = *C*, 13 = *D*, 14 = *E*, 15 = *F*. Теперь содержимое тех же самых четырех байтов выглядит более удобно:

59 35 *B9 CE*

Такая система счисления включает «цифры» от 0 до *F*, и так как таких цифр 16, то она называется ***шестнадцатеричной***. В табл. 1.1 приведено соответствие между двоичными, десятичными и шестнадцатеричными числами от 0 до 15(10).

Таблица 1.1. Соответствие между двоичными, десятичными и шестнадцатеричными числами

Двоичное	*Десятичное*	*Шестнад.*	*Двоичное*	*Десятичное*	*Шестнад.*
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	*A*
0011	3	3	1011	11	*B*
0100	4	4	1100	12	*C*
0101	5	5	1101	13	*D*
0110	6	6	1110	14	*E*
0111	7	7	1111	15	*F*

Подобно двоичным и десятичным цифрам каждая шестнадцатеричная цифра имеет вес, кратный основанию счисления. Таким образом, каждая цифра имеет вес в 16 раз больше, чем соседняя справа цифра. Крайняя правая цифра имеет вес $16^0 = 1$, следующая $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, $16^4 = 65536$.

Например, шестнадцатеричное число *3AF* имеет десятичное значение:

$(3 \cdot 16^2) + (A \cdot 16^1) + (F \cdot 16^0) = (3 \cdot 256) + (10 \cdot 16) + (15 \cdot 1) = 943$.

Для обозначения шестнадцатеричного числа часто используют букву ***H*** (или ***h***), например *3AFh*. Над шестнадцатеричными числами можно выполнять арифметические операции подобно тому, как они выполняются над десятичными числами. Но в отличие от десятичных чисел, только две арифметические операции над шестнадцатеричными числами имеют практическое значение: 1) сложение чисел (используется для определения суммарной области памяти, занимаемой программой); 2) вычитание из большего шестнадцатеричного числа меньшего (используется для определения длины области памяти).

Например, найдем сумму и разность чисел *6Ah* и *B5h*:

```
а)  6A        б)  B5
  + B5          - 6A
  ----          ----
    1F            4B
```

Выполнение записанных действий над шестнадцатеричными числами аналогично соответствующим действиям «в столбик» над десятичными числами. Единственное отличие – роль числа 10 при переносе и заеме теперь играет число 16.

1.3. Символьная информация

Для того чтобы хранить в памяти ЭВМ символьную (буквенно-цифровую) информацию, чтобы ее обрабатывать, информацию необходимо преобразовать в последовательность битов. Для такого преобразования используются ***символьные коды***, среди которых наиболее распространен код ASCII (*American Standard Code for Information Interchange* – Американский стандартный код для обмена информацией).

При использовании данного кода каждый символ представляется в виде одного байта. Например, букве *A* соответствует двоичный код 01000001= 41*h*. Таким образом, одна и та же битовая строка обозначает и букву *A*, и число 41*h* = 65(10). Что именно она обозначает, сама битовая строка «не знает», так как никаких пометок в памяти на этот счет нет. Смысл содержимого ячейки памяти определяется тем, как использует (интерпретирует) эту ячейку программа. Более того, разные программы могут интерпретировать одну и ту же битовую строку по-разному. Например, одна программа может выполнить суммирование содержимого двух байтов как беззнаковых чисел, вторая программа выполнит суммирование этих же цепочек битов как чисел со знаками, а третья программа выведет содержимое этих байтов на экран в качестве кодов символов.

Лабораторная работа 1

В отличие от последующих лабораторных работ, данная работа выполняется не с помощью ЭВМ, а вручную на листе бумаги. Задание на выполнение лабораторной работы состоит в том, что требуется без использования калькулятора выполнить следующие действия над двумя 4-значными целыми положительными десятичными числами, выданными преподавателем:

1) записать исходные числа в двоичной системе счисления;
2) найти сумму двоичных чисел из п. 1;
3) найти разность двоичных чисел из п. 1 (от меньшего числа отнять большее);
4) записать исходные числа в шестнадцатеричной системе счисления;
5) найти сумму шестнадцатеричных чисел из п. 4;
6) найти разность шестнадцатеричных чисел из п. 4 (от большего числа отнять меньшее число).

☑ **Примечание**. В качестве проверки рекомендуется найти сумму результатов п. 3 и п. 6, предварительно представив их в одной и той же системе счисления (двоичной или шестнадцатеричной). Указанная сумма должна быть равна 0. В самом деле, если обозначим исходные числа как *A* и *B*, то результат п. 3: $A - B$, а результат п. 6: $B - A$. Найдем искомую сумму: $(A - B) + (B - A) = 0$.

2. ВЫПОЛНЕНИЕ ПРОГРАММ ПРОЦЕССОРОМ i8086

2.1. Структура аппаратных средств

Любая ЭВМ предназначена для выполнения некоторого множества задач по переработке информации. Сущность подобной задачи состоит в том, что имеется некоторая исходная информация, на основе которой требуется получить другую – результирующую информацию. Как следует из предыдущего раздела, любая информация содержится в ЭВМ в виде битовой строки.

Каждая задача, решаемая ЭВМ, имеет алгоритм решения. ***Алгоритм*** – правило, определяющее последовательность действий над исходными данными, приводящую к получению искомых результатов. Форма представления алгоритма решения задачи, ориентированная на машинную реализацию, называется ***машинной программой***. Совокупность аппаратных средств вычислительной системы (*ВС*), предназначенных для выполнения машинных программ, часто называют просто ***аппаратурой***. В процессе любого программирования специалист обязательно имеет в своей голове модель той аппаратуры, которая будет использоваться для выполнения программы. Программирование на ассемблере предъявляет особые требования к данной модели, так как в программе приходится учитывать многие особенности используемой аппаратуры. Перейдем к рассмотрению укрупненной структуры аппаратуры ЭВМ, которая будет достаточна для нас на протяжении двух первых частей пособия.

На рис. 1.4 приведена структура аппаратных средств однопроцессорной ЭВМ с общей шиной. В данной структуре центральным связывающим звеном между основными блоками является ***общая шина*** (***ОШ***). При этом под термином ***шина*** понимается группа параллельных проводов. ОШ в общем случае есть объединение трех шин: 1) управления; 2) адреса; 3) данных. Рассмотрим кратко назначение других аппаратных блоков.

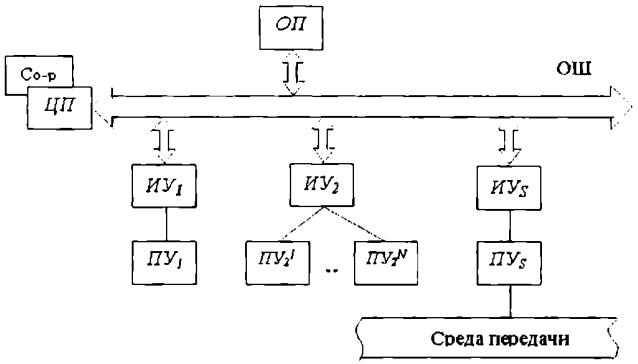

Рис. 1.4. Однопроцессорная ЭВМ с общей шиной

Главным аппаратным блоком любой ЭВМ является ***центральный процессор*** (***ЦП***). Это – «мозг» ЭВМ, который непосредственно выполняет все программы, как системные, так и прикладные. Любая программа представляет собой последовательность машинных команд, каждая из которых требует для своего размещения один, два или большее число байтов. На рис. 1.5 приведена структура наиболее типичной машинной команды. Здесь ***КОП*** – код операции. Это комбинация битов, кодирующая тип операции, которую следует выполнить над операндами (например, суммирование). Операнд 1, операнд 2 – это

или сами данные, над которыми выполняется машинная команда, или адреса в памяти (ОП или регистры), где эти данные находятся.

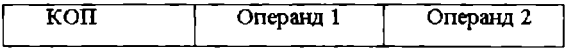

Рис. 1.5. Структура машинной команды

Термин «однопроцессорная ЭВМ» означает, что в рассматриваемой ЭВМ используется единственный центральный процессор. Кроме него практически любая ЭВМ имеет ***специализированные процессоры***. Примером специализированного процессора является ***сопроцессор*** – процессор, расположенный на той же плате, что и ЦП, и имеющий такой же доступ к ОП. В отличие от ЦП сопроцессор предназначен для выполнения не всей прикладной программы, а лишь отдельных ее команд (чаще всего команд обработки данных с плавающей точкой). Каждая такая команда выполняется сопроцессором как отдельная программа, записанная на его специализированном машинном языке.

Оперативная память (**ОП**) предназначена для кратковременного хранения программ и обрабатываемых ими данных. Название обусловлено тем, что операции чтения содержимого ячеек памяти и записи в них нового содержимого производятся достаточно быстро. Иногда используют другое название – ***операционная память***. Это название обусловлено тем, что ЦП может достаточно просто считывать машинные команды из ОП и исполнять их. Логическая структура любой ОП представляет собой линейную последовательность ячеек, которые пронумерованы (рис. 1.6). В зависимости от ЭВМ ячейкой является или байт, или машинное слово. Номер ячейки называется ***физическим*** или ***реальным адресом*** этой ячейки. В простейших ЭВМ поле операнда в машинной команде содержит этот номер.

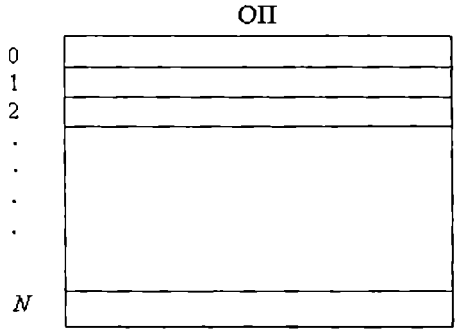

Рис. 1.6. Логическая структура ОП

Периферийные устройства (***ПУ***) – устройства ввода-вывода, устройства внешней памяти, а также устройства сопряжения со средой передачи данных. Посредством ***устройств ввода-вывода*** ЭВМ «разговаривает» с человеком-пользователем. Сюда относятся: клавиатура, экран (дисплей), мышь, принтер и так далее.

Устройство внешней памяти предназначено для работы с ***носителем внешней памяти***. Примером такого устройства является дисковод. Он работает с носителем внешней памяти – магнитным или оптическим диском. ***Внешняя память*** (***ВП***) имеет следующие отличия от ОП:

1) обмен информацией между ЦП и ВП выполняется во много раз медленнее, чем между ЦП и ОП;

2) ЦП не может выполнять команды, записанные в ВП. Для выполнения этих команд их необходимо предварительно переписать в ОП;

3) информация на носителе ВП сохраняется и после выключения питания.

В прошлом емкость носителей ВП многократно превосходила емкость ОП. В настоящее время это выполняется не для всех типов носителей.

Устройство сопряжения со средой передачи данных используется для подсоединения ЭВМ к сети. Например, если в качестве среды передачи данных используется пара проводов, то в качестве устройства сопряжения может быть применен модем.

Интерфейсное устройство (***ИУ***) предназначено для того, чтобы согласовать стандартную для данной ЭВМ структуру ОШ с конкретным типом ПУ, которых существует очень много. Поясним смысл слова: ***интерфейс*** – граница между двумя взаимодействующими системами. В данном случае речь идет о взаимодействии ОШ (с одной стороны) и ПУ (с другой стороны).

В дальнейшем будет сделано более подробное описание перечисленных аппаратных модулей ЭВМ. Естественно, что начать такое описание следует с центрального процессора, в качестве которого на протяжении первых двух частей данного пособия используется микропроцессор ***Intel 8086*** (сокращенно – i8086). Несмотря на то, что этот процессор давно стал достоянием компьютерной истории, его машинный язык аппаратно поддерживается в последующих моделях фирмы INTEL. Собственные машинные языки (и языки-ассемблеры) этих моделей включают язык для i8086 в качестве своего подмножества, отличаясь от него большей сложностью, которая делает их менее пригодными для первоначального знакомства с предметом.

2.2. Архитектура процессора i8086

На рис. 1.7 приведена архитектура (структура) микропроцессора i8086. Рассмотрим назначение его основных модулей.

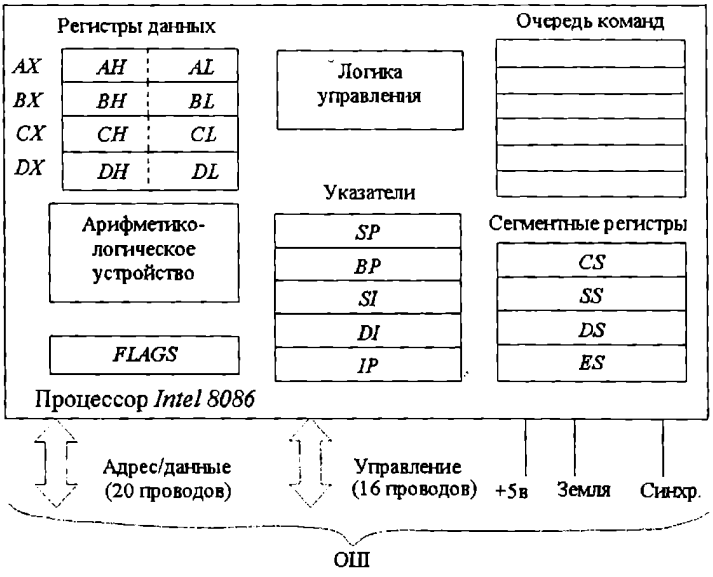

Рис. 1.7. Структура микропроцессора i8086

Блок ***логика управления*** выполняет дешифрирование и исполнение машинных команд. Если очередная команда является арифметической или логической, то она направляется логикой управления для выполнения в ***арифметико-логическое устройство***.

Регистры данных предназначены для хранения операндов и результатов операций. ***Регистр*** – ячейка памяти, скорость обмена с которой намного выше, чем с другими видами памяти: с ОП и тем более с ВП. К регистрам данных относятся регистры ***AX***, ***BX***, ***CX*** и ***DX***. Они допускают адресацию не только целых регистров, но и их младшей и старшей половин. Например, допускается использовать два байта в регистре *AX* вместе, а также указывать отдельные байты – ***AL*** (младший) и ***AH*** (старший). Регистры *BX*, *CX* и *DX*, кроме обычных функций, имеют и специальные назначения.

Очередь команд содержит текущую команду на время ее дешифрации (распознавания) и выполнения. Кроме того, для повышения быстродействия в данную очередь заранее считываются из ОП следующие по порядку в программе команды. Длина очереди команд – 6 байт.

В группу регистров ***указатели*** входят указатель команды *IP* и указатель стека *SP*, а также регистры *BP*, *SI* и *DI*. ***Указатель команды IP*** – очень важный регистр, который содержит относительный адрес (смещение) следующей команды, подлежащей выполнению на ЦП. ***Указатель стека SP*** также очень важен: он содержит относительный адрес вершины стека. Подробнее регистры *IP* и *SP* будут рассмотрены позже.

Остальные регистры-указатели используются для адресации ячеек какого-то массива относительно его начала. При этом ***базовый регистр BP*** часто используется для адресации ячеек программного стека относительно его вершины. А ***индексные регистры SI*** и ***DI*** обычно используются при циклической обработке элементов массива. При этом для того, чтобы обрабатывать одной и той же машинной командой различные ячейки массива, достаточно перед выполнением тела цикла осуществлять изменение содержимого того индексного регистра, который используется для адресации ячеек массива.

Регистр флагов FLAGS отражает текущее состояние ЦП. Структура этого регистра приведена на рис. 1.8.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	*DF*	*IF*	*TF*	*SF*	*ZF*		*AF*		*PF*		*CF*

Рис. 1.8. Регистр флагов микропроцессора i8086

Семь битов в *FLAGS* не используются. Остальные биты (флажки) делятся на условные и управляющие. ***Условные флажки*** отражают результат предыдущей арифметической или логической операции. Это:

1) ***SF – флаг знака***. Равен старшему биту результата. Так как в полнительном коде старший бит отрицательных чисел содержит 1, а у положительных он равен 0, то *SF* показывает знак предыдущего результата.

2) ***ZF – флаг нуля***. Устанавливается в 1 при получении нулевого результата и сбрасывается в 0, если результат не равен 0.

3) ***PF – флаг паритета***. Устанавливается в 1, если младшие 8 бит результата содержат четное число единиц, в противном случае он сбрасывается в 0.

4) ***CF – флаг переноса***. При сложении (вычитании) устанавливается в 1, если возникает перенос (заем) в старший бит (из старшего бита). Обычно данный флаг используется не по прямому назначению, а как признак возврата из подпрограммы: если подпро-

грамма (процедура или обработчик прерываний) завершилась успешно, то она возвращает *CF*=0, а если с ошибкой, то *CF*=1.

5) ***AF – флаг вспомогательного переноса***. Устанавливается в 1, если при сложении (вычитании) возникает перенос (заем) из бита 3. Флаг предназначен только для двоично-десятичной арифметики.

6) ***OF – флаг переполнения***. Устанавливается в 1, если знаковый бит изменился в той ситуации, когда этого не должно было произойти.

Пусть, например, машинная команда *add* (здесь и везде далее используются ассемблерные мнемоники машинных команд) выполнила следующее сложение:

```
  0010 0011 0100 0101
+ 0011 0010 0001 1001
-------------------
  0101 0101 0101 1110 ,
```

тогда после ее выполнения получаются состояния флажков:

SF = 0, *ZF* = 0, *PF* = 0, *CF* = 0, *AF* = 0, *OF* = 0 .

Если *add* выполнила сложение

```
  0101 0100 0011 1001
+ 0100 0101 0110 1010
-------------------
  1001 1001 1010 0011 ,
```

то флажки принимают состояния

SF = 1, *ZF* = 0, *PF* = 1, *CF* = 0, *AF* = 1, *OF* = 1 .

Флажки управления влияют на выполнение специальных функций. Эти флажки устанавливаются лишь несколькими специальными машинными командами. Это флажки:

1) ***DF – флаг направления***. Он используется при выполнении команд, обрабатывающих цепочки – последовательности ячеек памяти. Если флаг сброшен, цепочка обрабатывается с первого элемента, имеющего наименьший адрес. Иначе цепочка обрабатывается от наибольшего адреса к наименьшему.

2) ***IF – флаг разрешения прерываний***. Когда установлен этот флажок, ЦП выполняет маскируемые прерывания. Иначе эти прерывания игнорируются.

3) ***TF – флаг трассировки***. Если этот флажок установлен, то после выполнения каждой машинной команды ЦП генерирует внутреннее аппаратное прерывание (прерывание номер 1).

Рассмотренный регистр *FLAGS* чрезвычайно важен не только для понимания логики работы ЦП, но и всей ЭВМ в целом. Это обусловлено тем, что данный регистр фактически является дескриптором ЦП. ***Дескриптор*** (или ***блок управления***) – структура данных, используемая для управления модулем ЭВМ. При этом каждый сколько-нибудь сложный модуль ВС имеет свой блок управления.

Последний блок ЦП образуют сегментные регистры *CS*, *SS*, *DS* и *ES*. Данные регистры используются для адресации ячеек ОП.

2.3. Адресация памяти

Применение сегментных регистров процессора i8086 вызвано стремлением расширить объем программно адресуемого пространства ОП. Дело в том, что 20-проводная шина адреса (входит в состав ОШ, см. рис. 1.7) позволяет адресовать до 1 млн ячеек (байтов) ОП, так как $2^{20} \approx 1$ млн. Но все регистры в рассматриваемом ЦП 16-битовые. Ни одного 20-битового нет. Максимальный объем памяти, который можно адресовать с по-

мощью 16 бит, составляет 64 К (1 К = 1024). Рассмотрим, как получаются 20-битовые адреса.

Мысленно разобьем ОП на участки по 16 байт, называемые ***параграфами*** (рис. 1.9). Регистр сегмента кода ***CS*** содержит номер параграфа, с которого начинается выделенный нашей программе сегмент. Например, это может быть число *0002h*. Для получения реального адреса начальной ячейки параграфа необходимо его номер (*0002h*) умножить на число 16=10*h*: *0002h* x *10h* = *00020h*.

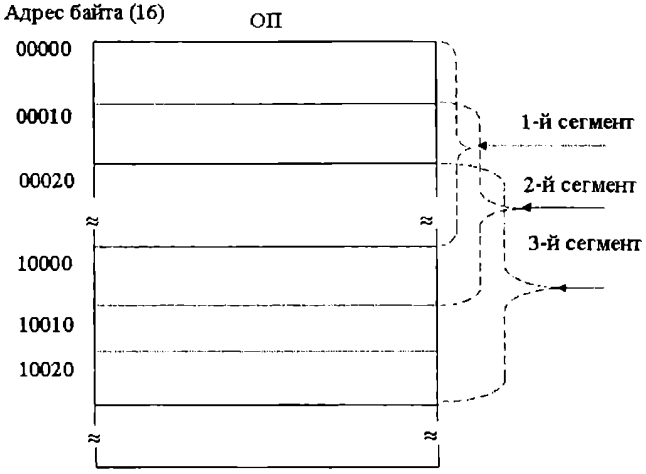

Рис. 1.9. Разбиение ОП на параграфы

Указатель команды ***IP*** содержит относительный адрес (смещение) адресуемой ячейки относительно начала сегмента. Допустим, что это число 100*h*. Физический адрес *R* адресуемой ячейки ЦП получает непосредственно перед обращением к ОП путем суммирования содержимого регистра *CS*, умноженного на 16 (10*h*), с содержимым регистра *IP*:

R = (CS) x 10h + (IP),

где запись регистра в круглых скобках означает содержимое этого регистра. Например, пусть (*CS*) = 0002*h*, а (*IP*) = 0100*h*, тогда *R* = 2*h* x 10*h* + 100*h* = 120*h* (рис. 1.10).

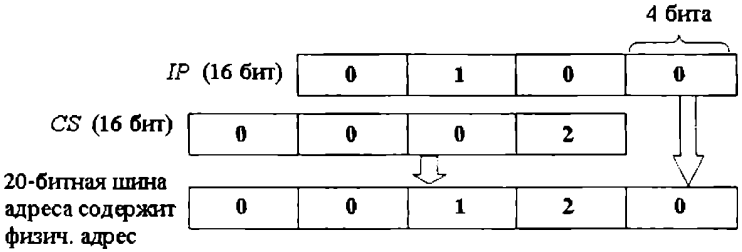

Рис. 1.10. Получение физического адреса

В пределах текущего сегмента *IP* может обращаться к любой ячейке ОП, имеющей смещение относительно начала сегмента 0 ÷2^{16} – 1, т. е. 0 ÷ 65535. Число 2^{16} = 65536 = = 10000*h* = 64 К называется ***длиной сегмента***.

Имея в своем распоряжении четыре сегментных регистра, программа (а точнее, выполняющий ее ЦП) использует их по-разному: *CS* – для адресации машинных команд,

SS – для адресации ячеек стека, *DS* – для основной адресации данных, *ES* – для дополнительной адресации данных. Поэтому в любой момент времени любая машинная программа имеет в своем распоряжении четыре ***логических сегмента*** ОП размером 64 К. Каждый из этих логических сегментов играет строго определенную роль при выполнении команд машинной программы:

Сегмент кода – сегмент ОП, номер начального параграфа которого находится в регистре *CS*. Следующая команда программы выбирается только из сегмента кода.

Сегмент данных – сегмент ОП, номер начального параграфа которого – в регистре *DS*. Команды, у которых ячейки памяти задаются только одним операндом, имеют дело с этим сегментом данных.

Дополнительный сегмент данных – сегмент ОП, номер начального параграфа которого – в регистре *ES*. Команды, у которых ячейки памяти задаются обоими операндами, используют и сегмент данных, и дополнительный сегмент данных.

Сегмент стека – сегмент ОП, номер начального параграфа которого – в регистре *SS*. Команды, выполняющие операции со стеком, имеют дело с тем стеком, на который «указывает» *SS*.

Перечисленные логические сегменты как бы «высвечиваются» ЦП из одномегабайтового адресного пространства ОП. Причем они могут пересекаться друг с другом или вообще совпадать. Выполняя запись в сегментные регистры (например, с помощью команды *mov*) нового содержимого, программа выполняет замену «высвечиваемых» сегментов.

Для того чтобы можно было обращаться к любой ячейке логического сегмента, в распоряжении программы имеется регистр, содержащий смещение искомой ячейки относительно начала сегмента. Это:

1) для сегмента кода – *IP*;
2) для сегмента данных – *BX, SI, DI*;
3) для дополнительного сегмента данных – *BX, DI*;
4) для сегмента стека – *SP*.

Таким образом, логический адрес любой ячейки ОП представляет собой пару:

(S, L) = ((регистр сегмента), (регистр смещения)),

где *S* – начальный адрес сегмента (номер параграфа);

L – смещение ячейки относительно начала сегмента.

Далее будем называть S адресом-сегментом, а L – адресом-смещением.

Преобразование логического адреса в физический адрес происходит при попадании соответствующей машинной команды на ЦП так, как это показано на рис. 1.10. Так как это преобразование происходит во время выполнения программы, то оно называется ***динамическим преобразованием адреса***. (***Статическая*** операция производится до начала выполнения программы.)

2.4. Алгоритм работы процессора

На рис. 1.11 приведен алгоритм выполнения машинных команд в ЦП. Он представляет собой упрощенный вариант работы реального процессора i8086, для которого некоторые этапы выполняются во времени не последовательно, а параллельно.

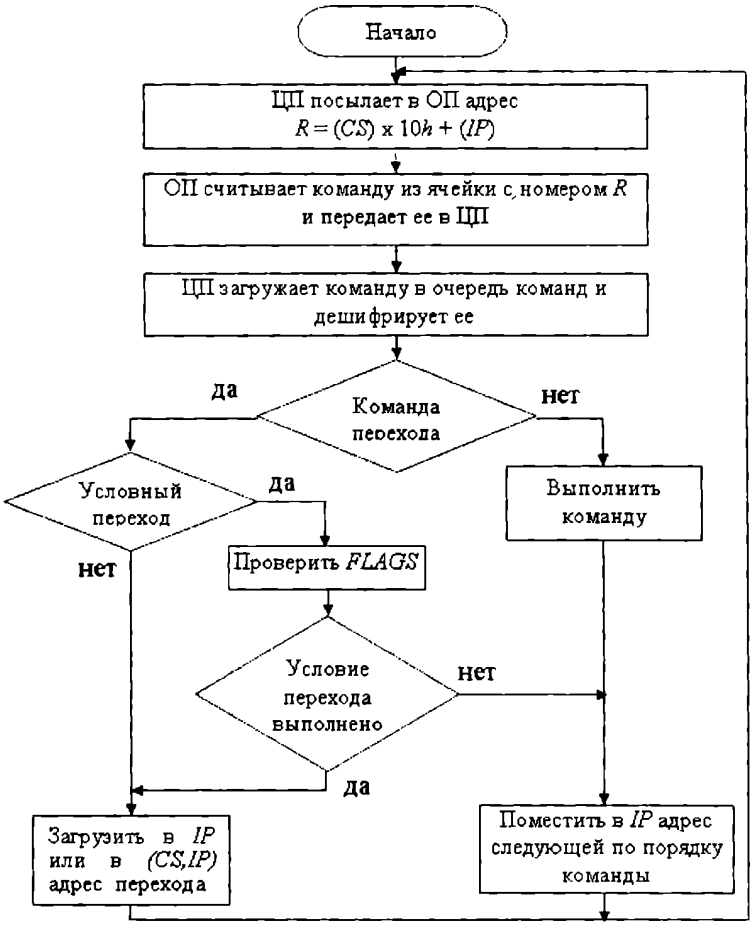

Рис. 1.11. Алгоритм работы ЦП

Данный алгоритм представляет собой бесконечный цикл, который инициируется сразу же после включения питания. На одной итерации алгоритма ЦП выполняет одну машинную команду. Это выполнение начинается с определения реального адреса команды в ОП. Этот адрес выдается из ЦП на шину адреса. Получив его, ОП помещает следующую команду на шину данных, и ЦП вводит команду в очередь команд. Пока дешифрируется эта команда, определяется ее длина в байтах и производится увеличение содержимого *IP* на эту длину, в результате чего *IP* адресует следующую машинную команду. После этого цикл повторяется.

Последовательная выборка команд из памяти и их выполнение продолжаются до тех пор, пока очередной командой, поступившей из ОП на ЦП, не окажется команда перехода. Команды перехода позволяют изменить естественный порядок следования машинных команд. Они делятся на команды безусловного и команды условного перехода.

Команды безусловного перехода обязательно изменяют естественный порядок выполнения команд. Существуют пять машинных команд безусловных переходов. Все они имеют одну и ту же ассемблерную мнемонику ***jmp*** и один операнд. Эти команды можно разбить на две группы: команды близких и команды дальних переходов.

Команда близкого перехода выполняет безусловный переход в пределах текущего сегмента кода. Это делается путем замещения содержимого *IP* (т. е. внутрисегментного адреса следующей по порядку команды) адресом, задаваемым самой командой перехода.

Команда дальнего перехода выполняет безусловный переход на ячейку ОП, расположенную за пределами текущего сегмента кода. Это делается путем замещения содержимого не только *IP*, но и *CS* значениями, содержащимися в самой команде перехода. Следствием этого является замена логического сегмента кода, и поэтому до тех пор, пока на ЦП не поступит следующая команда дальнего перехода, все последующие команды будут выбираться из нового сегмента.

Кроме команд *jmp* безусловный переход выполняют команды *call*, *ret*, *int*, *iret*. Эти команды мы рассмотрим позже, а пока лишь отметим, что мнемоникам *call* и *ret* соответствуют по две машинные команды, одна из которых выполняет близкий, а другая – дальний безусловный переход. Команды *int* и *iret* выполняют только дальние переходы.

Команды условных переходов замещают или не замещают содержимое *IP* в зависимости от результатов предыдущих команд, отраженных в регистре *FLAGS*. Например, если после команды *sub* (вычитание) в программе находится команда *jz* (переход по нулю), то переход осуществляется в том случае, если результат вычитания нулевой, и поэтому *ZF* (флаг нуля в *FLAGS*) установлен. Если же *ZF*=0 (ненулевой результат вычитания) переход не производится. Так как команды условных переходов могут выполнять только очень близкие переходы (не далее 128 байт), то для реализации больших переходов (в том числе и за пределы сегмента кода), каждый такой оператор дополняется оператором безусловного перехода.

3. Программирование арифметических операций

После того как мы рассмотрели логические основы организации аппаратуры ЭВМ, перейдем к построению простейших машинных программ, выполняемых с помощью этой аппаратуры. Выполнение программ на «голой» аппаратуре будет рассматриваться нами в третьей части пособия, а в первых двух частях пособия все наши программы будут выполняться в среде операционной системы DOS. При этом для того, чтобы «не отдаляться» от аппаратуры, при построении своих программ в первой части пособия мы будем использовать единственную системную программу – Debug.

Debug является отладчиком, т. е. программой, предназначенной для оказания помощи программистам в поиске ошибок в программах, исходные тексты которых написаны на ассемблере или каких-то других языках программирования. С помощью этой программы производится анализ и заполнение ячеек регистровой и оперативной памяти, осуществляется пошаговое выполнение программы. Преследуя цель лучше понять механизм выполнения машинных программ, мы не будем пока использовать для записи программ даже язык ассемблера, ограничившись лишь использованием вместо цифровых КОПов машинных команд соответствующих ассемблерных мнемоник.

Рассмотрим происхождение слова Debug (дословно – *насекомые*). В переводе со слэнга программистов означает «ошибки в программе». Используя Debug для пошагового запуска программы и наблюдая, как программа работает на каждом этапе, мы можем найти ошибки и исправить их. Этот процесс называется отладкой (*Debugging*), отсюда и произошло название программы Debug.

3.1. Чтение и заполнение регистров

Прежде чем запустить Debug, необходимо запустить ту операционную систему (ОС), в среде которой будут выполняться и Debug, и получаемые с помощью него наши программы. В качестве такой ОС везде далее используется любая система из семейства DOS. Эти ОС имеют схожие пользовательские и программные интерфейсы.

Примером DOS является свободно распространяемая операционная система FreeDOS. Так как эта ОС является самостоятельной системой, выполняемой на «голой» аппаратуре, то для ее запуска из среды какой-то другой ОС, например из любой Windows или из UNIX, предварительно следует запустить программный имитатор аппаратуры ЭВМ. Примером такого имитатора является виртуальная машина Microsoft «Virtual PC». Пояснения об ее использовании содержатся в приложении 2. Впрочем, при выполнении первых двух частей пособия можно ограничиться запуском из Windows MS-DOS, которая в настоящее время фактически является частью Windows. Запуск производится: <Пуск> → <Программы> → <Стандартные> → <Командная строка>.

В результате запуска DOS на черном экране появится ее приглашение для ввода нами команды, например:

```
C:\>
```

Данное приглашение означает, что в данный момент времени текущим логическим диском является *C:*, а текущим каталогом – корневой каталог (\) на этом логическом диске. Далее на экране находится само приглашение – символ «>».

Запустите Debug, набрав его название после приглашения DOS, например:

```
C:\> DEBUG
```

Debug можно вызвать и с помощью программы DOS Navigator, или с помощью другой аналогичной утилиты, имеющейся на вашей ЭВМ. Для этого надо найти в каталоге файлов файл *Debug.com*, установить на него курсор-маркер и нажать клавишу <Enter>.

Дефис «_», который Вы видите в качестве ответа на вашу команду – это приглашение программы Debug. Это означает, что Debug ждет вашей команды. Чтобы покинуть Debug и вернуться в DOS, напечатайте «***Q***» («*Quit*») около дефиса и нажмите <Enter>.

Попробуйте выйти и затем обратно вернуться в Debug:

```
_Q
C:\ > DEBUG
```

Мы начнем использование Debug с того, что попросим его показать содержимое регистров ЦП (микропроцессора i8086) с помощью команды ***R*** (от *Register*):

```
_R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756  SS=3756  CS=3756  IP=0100 NV UP DI PL NZ NA PO NC
3756:0100  E485    MOV  AL,85
```

Возможно, на своем экране вы увидите другие числа во второй и третьей строках. А сейчас обратим внимание на первые четыре регистра: *AX*, *BX*, *CX* и *DX*, о значениях которых Debug сообщил, что они все равны *0000*. Это регистры общего назначения. Остальные регистры *SP*, *BP*, *SI*, *DI*, *DS*, *ES*, *SS*,*CS*, *IP* являются регистрами специального назначения. Так как каждый из 13 регистров i8086 является словом и имеет длину 16 бит, то его содержимое представлено на экране в виде четырехзначного шестнадцатеричного числа.

Команда Debug *R* не только высвечивает регистры. Если указать в команде имя регистра, то Debug поймет, что мы хотим взглянуть на содержимое именно этого регистра и может быть изменить его. Например, мы можем изменить содержимое *AX*:

```
_R AX
AX=0000
:3A7
```

Теперь можно убедиться в том, что в регистре *AX* содержится *3A7h*:

```
_R
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 E485 MOV AL,85
```

Так и есть. Итак, мы можем помещать шестнадцатеричное число в регистр с помощью команды *R*, указывая имя регистра и вводя его новое значение после двоеточия.

3.2. Сложение двух чисел

Теперь мы перейдем к написанию и выполнению с помощью Debug программы на машинном языке. Простейшая такая программа состоит всего из одной машинной команды. Допустим, что эта команда выполняет сложение двух чисел, предварительно записанных в регистры.

Допустим, что мы хотим сложить числа *3A7h* и *92Ah*. Запишем эти числа соответственно в регистры *AX* и *BX*, воспользовавшись командой *R* Debug. Для суммирования содержимого *AX* и содержимого *BX* будем использовать машинную команду:

```
D801
```

Данная машинная команда имеет длину два байта. Старший байт команды – *D8*, а младший – *01*. Для лучшего восприятия человеком используется ассемблерная (мнемоническая) форма записи этой же самой команды:

```
add  ax, bx
```

Для того чтобы данная машинная команда была исполнена ЦП, необходимо выполнить четыре действия:

1) выбрать место в ОП для размещения машинной команды;
2) выполнить запись команды на выбранное место;
3) сообщить ЦП о том, где расположена наша машинная команда;
4) запустить ЦП для того, чтобы он исполнил команду.

Первое из перечисленных действий заключается в том, что мы должны выбрать из огромного числа ячеек (байтов) ОП всего два соседних байта для размещения нашей двухбайтовой команды. Мы не будем сильно «блуждать» по ОП, а ограничимся той областью памяти, которую нам рекомендует использовать Debug. Данная область (сегмент) имеет длину 65536 байт (64 Кбайт). Напомним, что эта длина обусловлена тем, что 65535 – максимальное число, которое можно записать в 16-битовое слово.

Где расположен в ОП предоставляемый нам сегмент? Для этого достаточно прочитать содержимое регистра *CS*, воспользовавшись командой *R* Debug. Напомним, что регистр *CS* называется регистром сегмента кода и содержит номер параграфа, с которого начинается сегмент. Для того чтобы получить номер (реальный адрес) первой ячейки параграфа, достаточно его номер умножить на 16. На практике такое умножение выпол-

няется очень просто – к номеру параграфа в шестнадцатеричном коде (он и содержится в *CS*) справа приписывается 0.

Из 64 Кбайт сегмента, на который указывает *CS*, нам надо выбрать всего два байта. В принципе можно взять любое смещение, меньшее, чем 65535, относительно начала сегмента. Но чаще всего берут смещение *100h*. Ячейки сегмента с меньшими внутрисегментными адресами резервируют для служебной информации, помещаемой туда DOS.

После того как мы выбрали место для размещения нашей машинной команды в ОП, *запишите* ее туда с помощью команды Debug ***E*** (от Enter), предназначенной для исследования и изменения памяти:

```
_E 100
3756:0100  E4.01
_E 101
3756:0101  85.D8
```

В результате числа *01h* и *D8h* расположены по адресам *3756:0100* и *3756:0101*. Числа *E4h* и *85h* представляют собой старое содержимое указанных ячеек ОП, которое осталось от ранее выполнявшихся программ. Номер начального параграфа сегмента, который вы увидите, возможно, будет другим, но это различие не будет влиять на нашу программу. Обратите внимание, что младший байт команды (*01h*) записан нами в ячейку с меньшим адресом, а старший байт (*D8h*) – с большим адресом. Заметим, что для записи нескольких соседних байтов мы можем использовать всего одну команду *E*, разделяя пробелом уже записанный байт от следующего:

```
_E 100
3756:0100  E4.01  85.D8
```

Прежде, чем идти дальше, *проверьте* результат наших предыдущих действий с помощью команды *R*:

```
_R
AX=03A7 BX=092A CX=0000 DX=0000 SF=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 01D8 ADD AX,BX
```

Теперь вам понятно, что содержит последняя строка, выданная Debug. Она содержит: 1) логический адрес в ОП машинной команды; 2) шестнадцатеричный код этой команды, причем младший байт команды изображен слева, а старший – справа (содержимое регистров показывается наоборот: старший байт слева, а младший справа); 3) ее мнемоническое представление. Почему именно нашу команду показал Debug? Ответ заключается в том, что Debug высвечивает ту команду, младший байт которой имеет адрес: *(CS):(IP)*.

Напомним, что указатель команды *IP* содержит внутрисегментное смещение младшего байта той машинной команды, которая будет исполняться следующей на ЦП. В процессе выполнения машинной программы ее команды сами могут изменять содержимое *IP*. А пока для этой цели мы будем использовать Debug. После своего запуска Debug всегда записывает в *IP 100h*. В процессе выполнения нашей машинной программы это значение будет меняться. Пользуясь командой *R* Debug, мы всегда можем записать в *IP* требуемое нам значение.

Теперь регистры и ОП готовы для исполнения нашей машинной команды. *Попросите* Debug ее выполнить, используя команду ***T*** (от *Trace*), которая выполняет одну команду за шаг, а затем показывает содержимое регистров. После каждого запуска *IP* будет указывать на следующую команду, в нашем случае будет указывать на *102h*. Мы

не помещали никакой команды в *102h*, поэтому в последней строке распечатки мы увидим команду, оставшуюся от предыдущей программы:

```
_T
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0102 NV UP DI PL NZ NA PO NC
3756:0102 41 INC CX
```

Вот и все. Регистр *AX* теперь содержит число *CD1h*, которое является суммой *3A7h* и *92Ah*. А регистр *IP* указывает на адрес *102h*, так что в последней строке распечатки регистров мы видим команду, расположенную в памяти по адресу *102h*, а не по адресу *100h*.

Как отмечалось ранее, указатель команды *IP* вместе с регистром *CS* всегда указывает на следующую машинную команду, которую нужно выполнить процессору. Если мы опять напечатаем «*T*», то выполнится следующая команда. Но не делайте этого сейчас, – ваш процессор может «зависнуть».

Если мы захотим выполнить введенную машинную команду еще раз, т. е. сложить *92Ah* и *CD1h* и сохранить новый ответ в *AX*, то надо объяснить процессору, где найти следующую команду, и чтобы этой следующей командой оказалась та же «*add ax,bx*», расположенная по адресу *100h*. Изменить значение регистра *IP* на *100h* можно, используя команду *R*. После этого:

```
_R
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100 01D8 ADD AX,BX
```

П о п р о б у й т е еще раз ввести команду Debug *T* и убедитесь, что регистр *AX* содержит число *15FBh*.

Следовательно, перед тем как использовать команду *T*, вам необходимо проверить регистр *IP* и соответствующую его значению машинную команду, располагаемую в нижней части распечатки (листинга), выдаваемой командой *R*. В результате вы будете уверены, что процессор выполнит требуемую команду.

До этого арифметические действия совершались над шестнадцатибитными словами. Но рассматриваемый процессор может выполнять действия и над восьмибитными байтами.

Напомним, что каждый регистр общего назначения может быть разделен на два байта – старший байт (первые две шестнадцатеричные цифры) и младший байт (следующие две шестнадцатеричные цифры). Название каждого из полученных регистров складывается из первой буквы названия регистра (от *A* до *D*), стоящей перед *X* в слове, и буквы *H* для старшего байта или буквы *L* для младшего. Например, *DL* и *DH* – регистры длиной в байт, а *DX* – длиной в слово.

Проверим байтовую арифметику на машинной команде *add*. *В в е д и т е* два байта *00h* и *C4h*, начиная с адреса *0100h*. Внизу листинга регистров вы увидите команду «*add ah,al*», которая суммирует два байта регистра *AX* и записывает результат в старший байт *AH*.

Затем *з а г р у з и т е* в *AX* число *0102h*. Таким образом, вы поместите *01h* в регистр *AH* и *02h* в регистр *AL*. Установите регистр *IP* в *100h*, выполните команду *T*, и вы увидите, что регистр *AX* теперь содержит *0302*. Результат сложения *01h* и *02h* будет *03h*, и именно это значение находится в AH.

3.3. Вычитание двух чисел

Мы собираемся написать машинную команду для вычитания *BX* из *AX*, так что после двух вычитаний в регистре *AX* появится результат *3A7h*. Тогда мы вернемся к той точке, с которой начали. *Запишите* с помощью команды *E* команду вычитания в ОП:

```
_E 100
 3756:0100  01.29  D8.D8
```

Листинг регистров (не забывайте установить *IP* в *100h*) должен теперь показать команду «***sub*** *ax,bx*», которая вычитает содержимое регистра *BX* из регистра *AX* и записывает результат в *AX*.

Выполните эту машинную команду с помощью команды Debug *T*. *AX* должен содержать *CD1*. Измените *IP* так, чтобы он указывал на эту машинную команду, и выполните ее опять (не забывайте сначала проверить команды внизу листинга регистров). *AX* теперь должен содержать *03A7h*.

Используйте машинную команду *sub*, чтобы подтвердить свои знания о представлении отрицательных чисел. Вычтем из 0 (в регистре *AX*) единицу (в *BX*). В результате *AX* должен содержать *FFFFh* (-1).

3.4. Умножение двух чисел

Команда умножения называется «***mul***», а машинный код для умножения *AX* на *BX* – *E3F7h*. Так как умножение двух 16-битовых чисел может дать 32-разрядный ответ, то машинная команда *mul* сохраняет результат в двух регистрах – *DX* и *AX*. Старшие 16 бит помещаются в регистре *DX*, а младшие – в *AX*. Эта комбинация регистров записывается как *DX:AX*.

Введите с помощью Debug команду умножения *E3F7h* по адресу *0100h* и установите *AX=7C4Bh* и *BX=100h*. Вы увидите команду в листинге регистров как «*mul bx*», без всяких ссылок на регистр *AX*. При умножении слов процессор i8086 всегда умножает регистр, имя которого вы указываете в машинной команде, на регистр *AX*, и сохраняет ответ в паре регистров *DX:AX*.

Перед запуском команды умножения перемножим *100h* и *7C4Bh* вручную. Три цифры 100 имеют в шестнадцатеричной системе такой же эффект, как и в десятичной. Так что умножение на *100h* просто добавит два нуля справа от шестнадцатеричного числа. Таким образом, *100h * 7C4B = 7C4B00h*. Этот результат слишком длинен для того, чтобы поместиться в одном слове, поэтому мы разбиваем его на два слова *007Ch* и *4B00h*.

Используйте Debug для запуска машинной команды умножения. Вы увидите, что *DX* содержит слово *007Ch*, а *AX* содержит слово *4B00h*.

3.5. Деление двух чисел

При делении сохраняется как результат, так и остаток от деления.

Поместите машинную команду *F3F7h* по адресу *0100h* (и *101h*). Как и команда *mul*, команда деления ***div*** использует пару регистров *DX:AX*, не сообщая об этом, так что все, что мы видим – это «*div bx*». Загрузите в регистры значения: *DX=007Ch* и *AX=4B12h*; регистр *BX* по-прежнему должен содержать *0100h*.

Подсчитаем результат вручную: *7C4B12h/100h=7C4Bh* с остатком *12h*. После выполнения команды деления по адресу *0100h* мы получим для *AX=7C4Bh* результат нашего деления и для *DX=0012h* остаток.

Лабораторная работа 2

В ходе выполнения лабораторной работы требуется выполнить с помощью Debug пошаговое вычисление выражения:

Y=[(X1+X2)X3-X4]/X5 ,

где *X1÷X5* – десятичные целые числа, взятые в соответствии с номером вашего варианта *K* из табл. 1.2.

Таблица 1.2. Исходные числа для лабораторной работы 2

K	*X1*	*X2*	*X3*	*X4*	*X5*	*K*	*X1*	*X2*	*X3*	*X4*	*X5*
1	235	314	11	2320	13	11	417	125	14	3181	11
2	513	248	17	4453	12	12	317	373	13	1920	12
3	197	372	12	3838	17	13	550	241	11	2720	15
4	254	418	14	4118	21	14	632	193	12	3593	21
5	349	517	11	5314	19	15	249	431	17	4111	13
6	267	149	15	2773	14	16	391	463	14	5320	22
7	435	317	13	3815	15	17	561	323	13	6213	23
8	561	273	16	2584	20	18	244	395	15	4713	14
9	321	491	12	4511	13	19	139	456	19	5334	17
10	634	124	19	8416	24	20	286	293	16	4811	21

Результат выполнения работы в виде экранного изображения показывается преподавателю. Одновременно преподавателю показывается на листе бумаги подробный ручной расчет заданного выражения в десятичной системе счисления. При этом итоговое число должно быть записано в виде обыкновенной дроби, например $17\frac{8}{27}$.

4. Вывод символов на экран

В предыдущем разделе мы выполняли с помощью Debug запись в ОП и последующее выполнение всего одной машинной команды, производящей над своими операндами какую-то арифметическую операцию. Теперь пришла пора заняться получением программы, состоящей из нескольких команд. Как и прежде, первая команда программы будет получать управление из Debug. Кроме того, последняя команда нашей программы должна возвращать управление обратно в Debug. В процессе своего выполнения программа будет обращаться за помощью к DOS с целью вывода на экран строки символов. Единственным способом вызова из прикладной программы системных подпрограмм DOS или BIOS является размещение в этой программе машинной команды программного прерывания. Термин «прерывание» означает, что выполнение нашей программы прерывается (приостанавливается) на время, необходимое для выполнения требуемой системной программы. Команда программного прерывания обозначается как ***int*** (от *Interrupt* – прерывание). Команда *int* для функций DOS имеет вид «*int 21h*», в машинном коде *21CDh*.

4.1. Вывод одного символа

Примером функции DOS, выполнение которой мы можем запросить из программы с помощью команды «*int 21h*», является вывод символа на экран. Для того чтобы различать функции DOS, которых много, используется регистр *AH*. При выводе одного символа в него помещается *02h*. В регистр *DL* заносится код ASCII выводимого символа. В табл. 1.3 приведены отображаемые (видимые на экране) коды ASCII.

Таблица 1.3. Шестнадцатеричные коды ASCII-*символов*

Символ	*Код*	*Символ*	*Код*	*Символ*	*Код*	*Символ*	*Код*	*Символ*	*Код*	*Символ*	*Код*
Проб.	20	0	30	@	40	P	50	`	60	p	70
!	21	1	31	A	41	Q	51	a	61	q	71
«	22	2	32	B	42	R	52	b	62	r	72
#	23	3	33	C	43	S	53	c	63	s	73
$	24	4	34	D	44	T	54	d	64	t	74
%	25	5	35	E	45	U	55	e	65	u	75
&	26	6	36	F	46	V	56	f	66	v	76
‘	27	7	37	G	47	W	57	g	67	w	77
(	28	8	38	H	48	X	58	h	68	x	78
)	29	9	39	I	49	Y	59	i	69	y	79
*	2A	:	3A	J	4A	Z	5A	j	6A	z	7A
+	2B	;	3B	K	4B	[	5B	k	6B	{	7B
,	2C	<	3C	L	4C	\	5C	l	6C	|	7C
-	2D	=	3D	M	4D	]	5D	m	6D	}	7D
.	2E	>	3E	N	4E	^	5E	n	6E	~	7E
/	2F	?	3F	O	4F	_	5F	o	6F	DEL	7F

Допустим, что мы хотим вывести символ *A*, тогда в регистр *DL* мы должны поместить число *41h*. *Подготовьте* регистры и память для последующего выполнения команды *«int 21h»*. Для этого в регистры *AX* и *DX* запишем с помощью Debug числа *0200h* и *0041h*, а по адресу *0100h* в ОП запишем *21CDh*. После этого можно перейти к выполнению машинной команды программного прерывания. Для этого не рекомендуем использовать команду *T* Debug. Дело в том, что в результате выполнения *«int 21»* начинает выполняться системная подпрограмма вывода символа, состоящая из многих машинных команд. Пошаговое выполнение этой подпрограммы вам скоро наскучит. Но если вы не доведете его до конца, то ваш компьютер «зависнет».

Но если вы все-таки протрассируете несколько шагов, можно выйти из Debug с помощью команды *Q*, которая ликвидирует беспорядок. (При выполнении трассировки обратите внимание на то, что изменилось первое число, являющееся составляющей адреса. Это обусловлено тем, что единственная машинная команда нашей программы и подпрограмма DOS находятся в разных сегментах ОП.)

В этом случае намного удобнее использовать команду Debug ***G*** (от *GO*), после которой пишется адрес-смещение, на котором мы хотим остановиться:

```
_G 102
A
AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0102 NV UP DI PL NZ NA PO NC
```

```
3970:0102 BBE5   MOV SP,BP
```

DOS вывел на экран букву *A* и возвратил затем управление в нашу программу. (Машинная команда, размещенная по адресу *102h*, осталась от другой программы, поэтому последняя строка вашего листинга может выглядеть по-другому.)

4.2. Команда завершения программы

Машинная команда «*int 20h*» сообщает DOS о том, что мы хотим выйти из нашей программы, и чтобы управление опять вернулось в DOS. В нашем случае эта команда вернет управление в Debug, так как мы запускаем нашу программу не непосредственно из DOS, а из Debug.

Введите команду *20CDh*, начиная с адреса *100h*, а затем проделайте следующее (не забудьте проверить команду «*int 20h*» с помощью команды *R* Debug):

```
_G 102
Program terminated normally
_R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970  ES=3970  SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100 CD20   INT 20
_G
Program terminated normally
_R
```

Результат команды *G* аналогичен результату команды «*G 102*». Любая из этих команд Debug выполняет всю программу (сейчас она состоит всего из одной команды – «*int 20h*») и затем возвращается к началу. Когда мы начали выполнение, *IP* был установлен в *100h*, так как мы заново запустили Debug. После выполнения *G IP* опять содержит *100h*.

Мы можем поместить машинную команду «*int 20h*» в конец любой программы для того, чтобы красиво передать управление DOS (или Debug). Для начала поместим ее после команды «*int 21h*» и получим программу из двух команд, выполняющую вывод символа на экран. Для этого начиная с адреса *100h введите* одну за другой две машинные команды – *21CDh* и *20CDh*.

Когда у нас была только одна машинная команда, то мы могли «пролистать» ее командой *R* Debug, но теперь у нас две команды. Чтобы увидеть их, воспользуемся командой Debug ***U*** (от *Unassemble* – разассемблирование):

```
_U  100
3970:0100  CD21   INT 21
3970:0102  CD20   INT 20
```

Далее идут еще 12 строк листинга, содержащие команды, оставшиеся в памяти от предыдущих программ.

Поместите в регистр *AH* значение *02h*, а в регистр *DL* код любого символа, например код символа *F* – *46h*. Затем введите команду *G*, чтобы увидеть символ на экране:

```
_G
F
Program terminated normally
```

До этого мы вводили команды программы в виде чисел, например *21CDh*. Но это слишком тяжелая работа и избавиться от нее помогает команда Debug ***A*** (от *Assemble* –

ассемблирование). Эта команда помогает вводить мнемонические (человекочитаемые) машинные команды. Применим команду *A* для ввода нашей программы:

```
_A '100
3970:0100    INT 21
3970:0102    INT 20
3970:0104
```

Команда *A* сообщает Debug о том, что мы хотим ввести машинные команды в мнемонической форме, а число 100 в команде означает, что ввод машинных команд начинается с ячейки *100h*.

4.3. Пересылка данных между регистрами

До сих пор мы записывали требуемые числа в регистры с помощью команды *R* Debug. Но обычно это делает команда самой программы – ***mov***. Эта же команда выполняет пересылку чисел между регистрами.

П о м е с т и т е 1234h в *AX* (*12h* в регистр *AH* и *34h* в *AL*) и *ABCDh* в *DX* (*ABh* в *DH* и *CDh* в *DL*). С помощью команды *A* введите машинную команду «*mov ah,dl*». Эта команда пересылает (копирует) число из *DL* в *AH*. *AL* при этом не используется. Если вы протрассируете эту строку, то увидите, что *AX=CD34h* и *DX=ABCDh*. Изменился только *AH*. Теперь он содержит копию числа из *DL*.

Команда *mov* пересылает число из второго регистра в первый, и по этой причине мы пишем *AH* перед *DL*. Машинный код данной команды *D488h*. Существуют другие формы этой же команды *mov*. Они имеют другие машинные коды и выполняют другие операции пересылки. Например, следующая команда (*C389h*) выполняет пересылку не байтов, а слов между двумя регистрами *AX* и *BX*:

```
3970:0100  89C3   MOV BX, AX
```

Следующая форма команды *mov* записывает значение числа в регистр, не используя другой регистр-источник:

```
3970:0100  B402   MOV AH, 02
```

Эта команда загружает число *02h* в регистр *AH*. Старший байт команды, *02h* является числом, которое мы хотим загрузить. *З а п и ш и т е* эту команду в ОП и выполните ее. Затем загрузите в *AH* другое число: с помощью команды «*E 101*» измените старший байт, чтобы он был равен, например, *C1h*.

Сложим все части вместе и построим длинную программу. Она будет печатать звездочку «*», выполняя все операции сама, не требуя от нас установки регистров (*AH* и *DL*). Программа использует команды *mov* для того, чтобы установить регистры *AH* и *DL* перед выполнением команды «*int 21h*», выполняющей вызов функции DOS:

```
15AC:0100  B402    mov   ah, 02
15AC:0102  B22A    mov   dl, 2a
15AC:0104  CD21    int   21
15AC:0106  CD20    int   20
```

В в е д и т е программу и проверьте ее командой *U 100*. Убедитесь, что *IP* указывает на ячейку *100h*. Запустите программу командой *G*. В итоге на экране должен появиться символ «*».

Теперь у нас есть законченная программа. Запишем ее на диск в виде *com*-файла для того, чтобы мы могли запускать ее прямо из DOS, просто набрав ее имя. Так как у программы пока нет имени, то мы должны его присвоить.

Команда Debug *N* (от *Name*) присваивает файлу имя перед записью на диск. *Наберите*:

```
_N  Writestr.com
```

Эта команда не запишет файл на диск – она только назовет его *Writestr.com* .

Далее мы должны сообщить Debug о том, сколько байт занимает программа, для того, чтобы он знал размер файла. Если вы посмотрите на разассемблированный листинг программы, то увидите, что каждая машинная команда в нем занимает 2 байта (в общем случае это не выполняется). У нас четыре команды, следовательно, программа имеет длину восемь байт.

Полученное число байт надо куда-то записать. Для этого Debug использует пару регистров *BX:CX*, и поэтому, поместив *8h* в *CX*, мы сообщим Debug о том, что программа имеет длину в восемь байт. *BX* должен быть предварительно установлен в ноль.

После того как мы установили имя и длину программы, мы можем записать ее на диск с помощью команды Debug *W* (от *Write*):

```
_W
Writing 0008 bytes
```

Теперь на диске есть программа *Writestr.com*, а мы с помощью *Q* покинем Debug и посмотрим на нее. *Используйте* команду DOS *dir* , чтобы увидеть справочную информацию о файле:

```
C:\ >dir  Writestr.com
Volume in drive C has no label
Directory  of  C:\
WRITESTR.COM  8  6-30-93  10:05a
1 File (S)    18432  bytes  free
```

Листинг директории сообщает, что файл *Writestr.com* находится в корневом каталоге ( ) на диске «*C:*», и его длина составляет 8 байт. Чтобы загрузить и выполнить программу, наберите в ответ на приглашение DOS *writestr.com* и нажмите <Enter>. Вы увидите символ «*».

Если мы хотим запустить свою *com*-программу не из DOS, а из Debug, то запуск Debug следует выполнить вместе с требуемым загрузочным модулем. Пример такого запуска: Debug *Writestr.com*. После этого с данной программой можно работать так, как будто мы создали ее только что с помощью Debug, а не считали с диска. Для сохранения скорректированной программы на диске следует выполнить те же операции, что и для нового файла.

4.4. Вывод на экран строки символов

Функция номер *02h* для прерывания «*int 21h*» печатает один символ на экране. Другая функция, номер *09h*, выводит на экран целую строку и прекращает вывод, когда находит в строке символ «*$*».

Поместим строку в память, начиная с ячейки *200h*, чтобы эта строка не перепуталась с кодом самой программы. *Введите* следующие числа, используя команду *E 200*:

```
48 65 6C 6C 6F 2C 20 44
4F 53 20 68 65 72 65 2E
24
```

Последнее число *24h* является ASCII–кодом для символа *$*, и оно сообщает DOS, что это конец строки символов. Теперь посмотрим, что содержит эта строка, запустив следующую программу:

```
15AC:0100    B409      mov    ah, 09
15AC:0102    BA0002    mov    dx, 0200
15AC:0105    CD21      int    21
15AC:0107    CD20      int    20
```

200h – адрес строки, которую мы ввели, а загрузка *200h* в регистр *DX* сообщает DOS о том, где ее искать. *Проверьте* программу командой *U* и затем запустите ее командой *G*:

```
_G
Hello, DOS here.
Program terminated normally
```

Команда Debug ***D*** (от *Dump*) дампирует (выводит содержимое) памяти на экран. Это похоже на действия, совершаемые командой *U* при распечатке машинных команд. Подобно *U* поместите после *D* адрес, чтобы сообщить Debug, откуда начинать дамп.

Наберите команду *D 200*. Она выведет содержимое участка памяти, в котором хранится только что введенная строка:

```
_D 200
15AC:0200  48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E  Hello, DOS here.
15AC:0210  24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8 $J.U..0...
```

После каждого числа, обозначающего адрес (как *15AC:0200* в примере), мы видим 16 пар шестнадцатеричных чисел, вслед за которыми записаны 16 ASCII–символов для этих пар (байтов). Например, в первой строке записаны символы, которые вы ввели. Символ *$* является первым символом в следующей строке, остальная часть строки представляет собой беспорядочный набор символов.

Точка «.» в окне *ASCII* означает, что это может быть как точка, так и специальный символ, например греческая буква «*pi*». Команда Debug *D* выдает только 96 из 256 символов символьного набора IBM PC, поэтому точка используется для обозначения остальных 160 символов. Часть специальных символов представляет собой прописные и строчные буквы русского алфавита. Соответствующие шестнадцатеричные коды приведены в табл. 1.4.

Таблица 1.4. Шестнадцатеричные коды букв русского алфавита (кириллицы)

Символ	*Код*	*Символ*	*Код*	*Символ*	*Код*	*Символ*	*Код*
А	80	Р	90	а	A0	р	E0
Б	81	С	91	б	A1	с	E1
В	82	Т	92	в	A2	т	E2
Г	83	У	93	г	A3	у	E3
Д	84	Ф	94	д	A4	ф	E4
Е	85	Х	95	е	A5	х	E5
Ж	86	Ц	96	ж	A6	ц	E6
З	87	Ч	97	з	A7	ч	E7
И	88	Ш	98	и	A8	ш	E8
Й	89	Щ	99	й	A9	щ	E9
К	8A	Ъ	9A	к	AA	ъ	EA

Символ	Код	Символ	Код	Символ	Код	Символ	Код
Л	8B	Ы	9B	л	AB	ы	EB
М	8C	Ь	9C	м	AC	ь	EC
Н	8D	Э	9D	н	AD	э	ED
О	8E	Ю	9E	о	AE	ю	EE
П	8F	Я	9F	п	AF	я	EF

Теперь запишем программу, выводящую строку на экран, на диск. Программа начинается со строки *100h*, и из выполненного дампа памяти можно видеть, что символ, следующий за знаком *$*, заканчивающим нашу строку, расположен по адресу *211h*. *Сохраните* разность *211h–100h* в регистре *CX*, опять установив *BX* в ноль. Используйте команду *N*, чтобы дать имя программе (добавьте расширение *com*, чтобы запускать программу прямо из DOS), и затем командой *W* запишите программу и данные в дисковый файл.

Лабораторная работа 3

Требуется разработать программу, которая выводит на экран ваши имя и фамилию, записанные сначала русскими, а затем английскими буквами. Программа должна вызываться непосредственно из DOS.

5. Вывод на экран двоичных чисел

Приступим к решению задачи отображения на экран двоичных чисел, содержащихся в ячейках памяти, называемых регистрами. Напомним, что в ячейке любой памяти (регистры, ОП, ВП) вся информация содержится в виде последовательности битов. Поэтому отображение этой информации на экране в виде двоичного числа имеет практический смысл.

В ходе решения указанной задачи нам потребуется рассмотреть новые машинные команды, а также произвести знакомство с новыми понятиями. При этом мы начнем с более близкого знакомства с одним из битов рассмотренного ранее регистра *FLAGS* – флага переноса.

5.1. Флаг переноса

Если выполнить сложение чисел 1 и *FFFFh*, то получим *10000h*. Это число не может быть записано в шестнадцатибитовое слово, так как в нем помещаются только четыре шестнадцатеричные цифры. Единица в результате называется ***переполнением***. Она записывается в специальную ячейку, называемую флагом переноса *CF* (от *Carry Flag*). Флаг содержит число из одного бита, т. е. содержит или единицу, или ноль. Если флаг содержит единицу, то говорят, что он «установлен», а если ноль – «сброшен». Напомним, что флаг *CF* является одним из шестнадцати битов в регистре флагов *FLAGS*.

Выполните загрузку чисел 1 и *FFFFh* в регистры *BX* и *AX* и запишите в память команду «*add ax,bx*». После этого протрассируйте эту команду. В конце второй строки распечатки, полученной с помощью команды *R* Debug, вы увидите восемь пар букв. Последняя пара выглядит как *CY* (от *Carry Ye* – перенос есть), т. е. флаг переноса установлен.

Установите IP в *100h* и прибавьте единицу к нулю в *AX*, повторив трассировку команды сложения. Флаг переноса переустанавливается в каждой операции сложения, и так как на этот раз переполнения не будет, то флаг будет сброшен. С помощью команды *R* проверьте, что в качестве состояния флага *CF* листинг содержит *NC* (от *No Carry* – нет переноса).

5.2. Циклический сдвиг

Допустим, что нам надо выполнить вывод на экран двоичного числа. За шаг мы выводим только один символ, и нам надо произвести выборку всех битов двоичного числа, одного за другим, слева направо. Например, пусть требуемое число есть *10000000b*. Если мы сдвинем весь этот байт влево на одну позицию, помещая единицу во флаг переноса и добавляя ноль справа, а затем повторим этот процесс для каждой последующей цифры, то во флаге переноса будут по очереди содержаться все цифры нашего двоичного числа.

Команда ***rcl*** (от *Rotate Carry Left* – циклический сдвиг влево с переносом) сдвигает крайний левый бит во флаг переноса (в примере это 1), в то время как бит, находившийся до этого во флаге переноса, сдвигается в крайне правую позицию (т. е. в нулевой бит). В процессе сдвига все остальные биты сдвигаются влево. После определенного количества циклических сдвигов (17 для слова, 9 для байта) биты возвращаются на их начальные позиции, и вы получаете исходное число. На рис. 1.12 показано наглядное представление работы команды *rcl*.

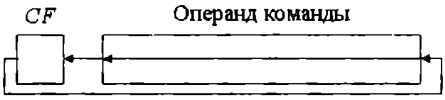

Рис. 1.12. Команда циклического сдвига влево через перенос *rcl*

Выполните с помощью Debug размещение по адресу *100h* команды «*rcl bl,1*», которая циклически сдвигает байт в *BL* влево на один бит, используя флаг переноса. Поместите в регистр *BX* число *B7h* и протрассируйте эту команду несколько раз. Убедитесь, что после 9 циклов регистр *BX* содержит опять *B7h*.

Как вывести на экран двоичное значение флага переноса? Из таблицы кодов ASCII видно, что символ «0» есть *30h*, а символ «1» есть *31h*. Таким образом, сложение флага переноса и *30h* дает символ «0», когда флаг сброшен, и символ «1», когда он установлен. Для выполнения такого сложения удобно использовать команду ***adc*** (от *Add with Carry* – сложение с переносом). Эта команда складывает три числа: два числа, как и команда *add*, а также один бит из флага переноса.

Поместите в память после команды «*rcl bl,1*» команду «*adc dl,30*», которая выполнит сложение содержимого *DL* (0), *30h* и флага переноса, поместив результат в *DL*. Записав далее команды, обеспечивающие вывод символа на экран и завершение программы, получим программу, выполняющую вывод на экран старшего бита регистра *BL*:

```
mov   dl, 00
rcl   bl, 1
adc   dl, 30
mov   ah, 02
int   21
int   20
```

Выполните эту программу для обоих значений старшего бита *BL*. Для записи в *BX* используйте команду *R* Debug.

5.3. Организация циклов

Если мы хотим вывести на экран все биты *BL*, то мы должны повторить операции циклического сдвига и распечатки флага переноса *CF* восемь раз (число битов в *BL*). Неоднократное повторение одних и тех же операций называется ***циклом***.

Соответствующий алгоритм приведен на рис. 1.13. На первом этапе переменной *CX* присваивается значение 8. Именно столько раз мы хотим повторить выполнение этапов, образующих «тело цикла». Переменная *CX* называется ***счетчиком повторений***. Тело цикла образуют этапы «Получение кода ASCII для бита» и «Вывод символа». После того как тело цикла выполнено, значение *CX* уменьшается на 1. На следующем этапе алгоритма значение *CX* сравнивается с 0. Если это значение ненулевое, то делается возврат для повторения тела цикла. Иначе – выполняется этап алгоритма, расположенный после цикла. На рис. 1.13 это этап «Завершение алгоритма».

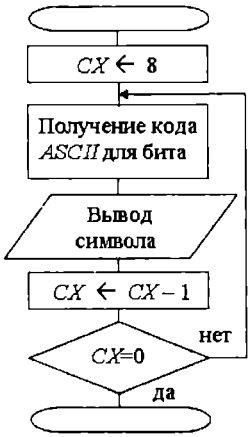

Рис. 1.13. Алгоритм вывода на экран содержимого байта: CX – счетчик повторений

Для организации циклов используются специальные машинные команды. Одной из них является команда ***loop***. Она записывается в конце цикла, т. е. после тех команд, выполнение которых следует повторить. Данная команда имеет один операнд – адрес первой из повторяемых машинных команд. Счетчик повторений цикла содержится в регистре *CX*. Данный регистр используется потому, что буква *C* в названии регистра *CX* означает «счетчик» (от *Count*). *CX* может использоваться так же как регистр общего назначения.

Выполнение команды *loop* сводится к следующему. Во-первых, она вычитает из содержимого регистра *CX* единицу. Во-вторых, она сравнивает полученное содержимое регистра *CX* с нулем, и если оно не равно 0, то делает переход по адресу, заданному в качестве операнда команды *loop*. В-третьих, если переход не делается, то на ЦП начинает выполняться машинная команда, расположенная в программе сразу же за командой *loop*. Таким образом, наличие данной команды обеспечивает реализацию двух этапов алгоритма, приведенного на рис. 1.13.

В качестве примера применения команды *loop* приведем программу, выполняющую вывод на экран четырех звездочек:

```
100  mov  ah,02
102  mov  dl,2a
104  mov  cx,4
107  int  21
109  loop 107
10B  int  20
```

Выполните трассировку данной программы, наблюдая за содержимым регистров *IP* и *CX*. При этом следует вспомнить, что не следует использовать команду *T* Debug для команды *int*. При достижении этой команды следует набрать команду Debug «*G d*», где *d* – адрес в памяти команды, следующей за *int*. При достижении команды «*int 20*» вводится команда *G* Debug.

При применении команды «*G d*» необходимо знать адрес останова *d*. Исключить трассировку при достижении машинной команды *int* проще, если использовать команду Debug ***P*** (от *Proceed* – переходить, продолжать). Эта команда является удобным средством обхода команд *int*, вызывающих подпрограммы DOS.

Выполните написание и ввод в память программы вывода двоичного содержимого байта (содержится в регистре *BL*) на экран (алгоритм на рис. 1.13).

5.4. Отладка программы

Отладка программы включает поиск ошибок (тестирование программы) и их исправление. Пока наши программы достаточно просты, и каждая из них включает всего одну подпрограмму. Что касается отладки программ, состоящих из нескольких подпрограмм, то она будет рассматриваться позднее. Пока лишь заметим, что такая отладка может рассматриваться как последовательность отладок подпрограмм.

Тестирование программы выполняется при различных значениях ее входных данных. Если очередной прогон программы показал наличие в ней ошибки, то производится ее поиск. Как раз для такого поиска и предназначена трассировка программы.

Если программа длинная, то ее пошаговая трассировка не очень удобна. В этом случае сначала желательно локализовать ошибку, определив содержащий ее фрагмент программы. Далее этот фрагмент исследуется более подробно. Для локализации ошибки мы выбираем несколько точек останова. Для выбора этих адресов, разбивающих программу на фрагменты, используется листинг программы, а также ее блок-схема. Далее, с помощью команды «*G d*» производится анализ работы программы в каждой из точек останова. Если в очередной точке останова результаты работы программы неверны, то данная точка завершает искомый фрагмент.

Выполните отладку введенной ранее в память программы вывода двоичного содержимого байта. Тестирование программы проведите с помощью команды *G*, предварительно загружая с помощью команды *R* в регистр *BX* различные пары шестнадцатеричных цифр. При этом заметим, что Debug не имеет команд для работы с однобайтовыми регистрами, и поэтому *BL* заполняется как часть *BX*. В случае обнаружения ошибки выполните пошаговую трассировку или используйте точки останова.

Лабораторная работа 4

Требуется разработать программу, вызываемую из DOS, которая за одно свое выполнение выводит на экран двоичное содержимое двух заданных регистров. Для выбора регистров используйте табл. 1.5.

Таблица 1.5. Задание регистров

K	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Рег.1	AX	BX	CX	DX	BP	SI	DI	BX	CX	DX
Рег.2	BX	CX	AX	BX	AX	BX	CX	DX	BP	BP
K	11	12	13	14	15	16	17	18	19	20
Рег.1	BP	SI	DI	AX	BX	CX	DX	BP	SI	DI
Рег.2	DI	AX	DX	DX	DI	DX	SI	BX	CX	BP

Пример сообщения, выводимого программой на экран:
(AX) = 1010101011111111 (BX) = 0111100100111001

☑ **Примечания:**

1. Так как программа вызывается не из Debug, а из DOS, то она сама должна задавать первоначальное содержимое регистров, выводимых на экран. Для этого рекомендуется использовать команду *mov*.

2. Если регистр, содержимое которого выводится на экран, используется в вашей программе и для других целей (например, *CX* – счетчик цикла), то первоначальное содержимое этого регистра необходимо переписать с помощью команды *mov* в какой-то другой, свободный регистр данных и выполнять вывод на экран содержимого этого регистра.

6. Вывод на экран чисел в шестнадцатеричной форме

В ходе рассмотрения предыдущего раздела мы научились выводить на экран двоичное содержимое регистров. Аналогично делается вывод на экран и двоичного содержимого ячеек ОП. Но гораздо удобнее для человека увидеть содержимое этих же ячеек не в двоичной, а в шестнадцатеричной системе. Напомним, что шестнадцатеричное число фактически есть сокращенная форма записи двоичного числа. При этом каждая шестнадцатеричная цифра соответствует четырем двоичным цифрам, т. е. описывает содержимое четырех битов памяти.

В ходе рассмотрения данного вопроса производится практическое знакомство с тремя флагами состояния, командами для работы с этими флагами, а также с некоторыми другими важными машинными командами.

6.1. Флаги состояния

Кроме флага переноса *CF* существуют другие флаги состояния. Рассмотрим три из них, которые описывают результат последней арифметической операции.

Допустим, что мы выполнили машинную команду вычитания *sub*. Одним из флагов состояния, устанавливаемых в зависимости от результата этой команды, является ***флаг нуля ZF*** («*Zero Flag*»). Если результат команды *sub* есть 0, то флаг нуля будет установ-

лен в 1. На распечатке регистров это значение обозначается как *ZR* (от *Zero* – нуль). Если результат арифметической операции не равен нулю, то флаг нуля сбрасывается в 0 – *NZ* (от *Not Zero* – не нуль).

Введите в память команду «*sub ax,bx*». Протрассируйте ее с одинаковыми и с разными числами в регистрах *AX* и *BX*, наблюдая за состоянием флага нуля (*ZR* или *NZ*).

Флаг знака SF («*Sign Flag*») принимает значение 1 (на распечатке регистров *NG* – от *Negative*), если результат предыдущей арифметической операции отрицательный. Если результат неотрицательный (ноль или положительный), то флаг знака принимает значение 0 (на распечатке *PL* – «*Plus*»). *Выполните* трассировку команды «*sub ax,bx*» при разных содержимых *AX* и *BX*, и наблюдая за флагом знака.

Флаг переполнения OF устанавливается в 1 в том случае, если знаковый бит изменился в той ситуации, когда этого не должно было произойти. Например, если мы сложим два положительных числа *7000h* и *6000h*, то получим отрицательное число *D000h* (представление в дополнительном коде числа -2288). Это ошибка, так как результат переполняет слово. На листинге регистров переполнение обозначается как *OV* (*Overflow* – переполнение). Если предыдущая арифметическая команда не дала переполнения, то флаг сбрасывается в 0. На распечатке регистров это значение обозначается как *NV* (*No Overflow*). *Проверьте* установку флага переполнения, протрассировав команду *sub* или *add*.

Использование команды *sub* для сравнения двух чисел неудобно, так как эта команда производит изменение первого из чисел. Другая команда, ***cmp*** (от *Compare* – сравнение), производит сравнение двух чисел без их изменения. Результат сравнения используется только для установки флагов.

Загрузите в регистры *AX* и *BX* одинаковые числа, например F5*h*, и протрассируйте команду «*cmp ax,bx*». При этом убедитесь, что установлен флаг нуля (*ZR*), но оба регистра сохранили свое значение – F5*h*.

6.2. Команды условного перехода

Напомним, что флаги состояния устанавливаются для того, чтобы можно было менять ход выполнения программы в зависимости от текущей ситуации. Анализ флагов состояния и соответствующие переходы в программе выполняют команды условного перехода.

Команда ***jz*** (от *Jump if Zero* – перейти, если ноль) проверяет флаг нуля, и если он установлен (*ZR*), то выполняется переход на новый адрес. Таким образом, если мы вслед за командой *sub* напишем, например, «*jz 15a*», то нулевой результат вычитания означает, что ЦП начнет выполнять не следующую по порядку команду, а команду, находящуюся по адресу *15A*.

Противоположной по отношению к *jz* является команда *jnz* («*Jump if Not Zero*» – перейти, если не ноль). В следующей простой программе из числа вычитается единица до тех пор, пока в результате не получится ноль:

```
100  sub  al,01
102  jnz  100
104  int  20
```

Поместите небольшое число в *AL* и протрассируйте программу, чтобы увидеть, как работает условное ветвление. При достижении последней команды введите команду *G* Debug.

Команда условного перехода ***ja*** (от *Jump if Above* – перейти, если больше) осуществляет переход на указанный в команде адрес, если по результатам предыдущей команды флаг переноса *CF* сброшен (на листинге *CF = NC*). Флаг нуля *ZF* также должен быть сброшен. Данная команда обычно записывается сразу за командой сравнения (*cmp*) двух беззнаковых чисел. Если первое сравниваемое число больше второго, то команда *ja* осуществляет переход.

6.3. Вывод на экран одной шестнадцатеричной цифры

Любое число между *0* и *Fh* соответствует одной шестнадцатеричной цифре. Переведя выбранное число в ASCII-символ, его можно вывести на экран. ASCII-символы от 0 до 9 имеют значения от *30h* до *39h*; символы от *A* до *F*, однако, имеют значения от *41h* до *46h*. В результате переход в ASCII будет затруднен из-за наличия двух групп чисел (от 0 до 9 и от *Ah* до *Fh*), так что мы должны обрабатывать отдельно каждую группу. На рис. 1.14 приведена блок-схема программы, выполняющей вывод на экран одной шестнадцатеричной цифры.

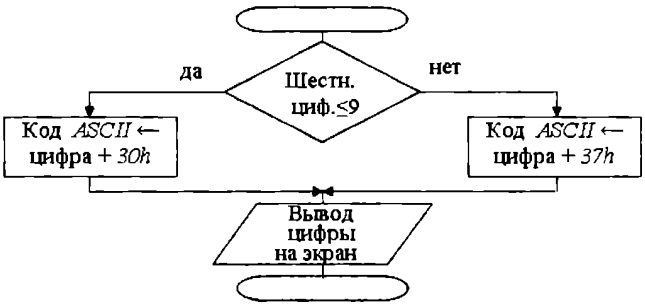

Рис. 1.14. Алгоритм программы вывода одной шестнадцатеричной цифры

Текст программы вывода шестнадцатеричной цифры:

```
100  mov  dl,bl
102  cmp  dl,09
105  ja   10c
107  add  dl,30
10A  jmp  10f
10C  add  dl,37
10F  mov  ah,02
111  int  21
113  int  20
```

Для передачи значения шестнадцатеричной цифры на вход программы используется регистр *BL*. Команда *cmp* вычитает два числа ((*DL*)–*9h*), чтобы установить флаги, но она не изменяет регистр *DL*. Поэтому, если содержимое *DL* больше, чем 9, команда «*ja 10c*» осуществляет переход к команде по адресу *10C*.

Запишите приведенную выше программу в ОП и протрассируйте ее, предварительно записав в *BL* шестнадцатеричное число, состоящее из одной цифры. Не забывайте использовать или команду *G* Debug с указанием точки останова, или команду *P*, когда запускаете машинную команду *int*. Затем проверьте правильность работы программы, используя команду *G*, предварительно загружая в *BX* граничные данные: 0; 9; *Ah* и *Fh* .

Общее правило: при тестировании любой программы рекомендуется проверить граничные данные.

6.4. Вывод старшей цифры двузначного шестнадцатеричного числа

Одна шестнадцатеричная цифра занимает четыре бита (четыре бита часто называют полубайтом или тетрадой). Двузначное шестнадцатеричное число занимает 8 бит (1 байт). Это может быть, например, регистр *BL*. При выводе числа на экран сначала выводится старшая цифра, а затем – младшая. Соответствующий укрупненный алгоритм приведен на рис. 1.15. При этом детальный алгоритм каждого из двух этапов «Вывод цифры на экран» приведен ранее на рис. 1.14. Рассмотрим этап «Выделение старшей цифры». В результате него содержимое старшего полубайта должно быть переписано (сдвинуто) в младший полубайт.

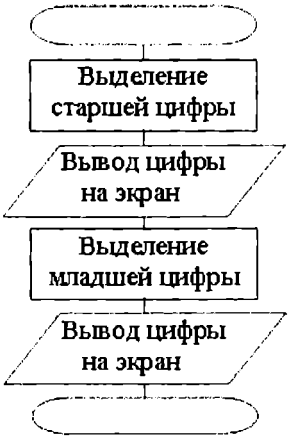

Рис. 1.15. Алгоритм вывода двузначного шестнадцатеричного числа

Несмотря на то, что нам надо выполнить сдвиг вправо, вспомним команду *rcl*, которая циклически сдвигает байт или слово влево через флаг переноса. Ранее мы использовали команду «*rcl bl,1*», в которой единица есть сообщение для ЦП о том, что надо сдвинуть содержимое *BL* на 1 бит. Мы можем осуществить циклический сдвиг более чем на один бит, но для процессора i8086 мы не можем написать команду «*rcl bl,2*». Для циклического сдвига необходимо поместить счетчик сдвигов в регистр *CL*, который используется здесь так же, как регистр *CX* применялся командой *loop* при определении числа повторений цикла. Так как не имеет смысла осуществлять циклический сдвиг более чем 16 раз, то для записи числа сдвигов вполне подойдет восьмибитовый регистр *CL*.

Для сдвига старшего полубайта вправо на четыре бита будем использовать команду сдвига ***shr*** («*Shift Right*» – логический сдвиг вправо). Данная команда не только выполняет сдвиг вправо, но и записывает в освобождающиеся старшие биты нули. В этом проявляется разница между терминами «логический» и «циклический», так как команда циклического сдвига записывает в освобождающиеся биты содержимое флага переноса. Что касается выталкиваемых младших битов байта (или слова), то они по очереди записываются во флаг переноса аналогично циклическому сдвигу.

З а г р у з и т е числа 4 в *CL* и *5Dh* в *DL*, а затем введите и протрассируйте следующую команду сдвига:

```
100   shr  dl, cl
```

DL должен теперь содержать число *05h*, т. е. этот регистр содержит в своем младшем полубайте старшую цифру числа *5Dh*.

Если мы используем процессор, более современный, чем i8086, то величина сдвига может быть задана непосредственно в виде числа. Например, следующие команды осуществляют реализацию этапа «Выделение старшей цифры»:

```
mov  dl,bl
shr  dl,04
```

П о м е с т и т е эти команды в ОП, дополнив их командами этапа «Вывод цифры на экран». При этом не забудьте скорректировать адреса переходов. (Можно записать программу со старыми адресами, а затем скорректировать команды переходов.) Выполните программу, предварительно загрузив в регистр *BL* любую пару шестнадцатеричных цифр.

6.5. Вывод младшей цифры двузначного шестнадцатеричного числа

Для вывода младшей цифры достаточно обнулить старший полубайт в восьмибитовом регистре, содержащем пару шестнадцатеричных цифр.

Обнуление любых битов в байте или в слове удобно выполнить, используя команду ***and*** (логическое «*И*»). Данная команда побитно сравнивает два заданных в ней байта (слова). Если в обоих байтах соответствующий бит имеет значение «1», то и в результирующий байт на место соответствующего бита записывается 1. Если хотя бы один из сравниваемых битов имеет значение «0», то и результирующий бит принимает нулевое значение. Результирующий байт записывается на место первого из сравниваемых байтов. Например, команда «*and bl,cl*» последовательно выполняет операцию *and* сначала над битами 0 регистров *BL* и *CL*, затем над битами 1, битами 2 и так далее, и помещает результат в *BL*.

Выполняя операцию *and* над *0Fh* и каким-либо байтом, мы можем обнулить старший полубайт этого байта:

```
        1011 0101
AND     0000 1111
        0000 0101
```

Следующие две команды реализуют этап «Выделение младшей цифры»:

```
mov  dl,bl
and  dl,0f
```

З а п и ш и т е в память программу для вывода младшей цифры. Протестируйте эту программу, загружая в *BL* различные пары шестнадцатеричных цифр. Далее запишите в память всю программу вывода на экран двузначного шестнадцатеричного числа и протестируйте ее. (Не забудьте при этом скорректировать адреса переходов во второй части программы, а также исключить первую команду «*int 20*».)

Лабораторная работа 5

Требуется разработать программу вывода на экран двух четырехзначных шестнадцатеричных чисел, содержащихся в заданных регистрах (те же регистры, что и в лабораторной работе 4).

Пример сообщения, выводимого программой на экран:

(AX) = 67FE (BX) = BA59

При написании программы учтите примечания к лабораторной работе 4, а также следующие:

☑ **Примечания:**

1. Некоторые 8-битовые регистры следует заменить 16-битовыми.

2. При реализации вывода второй и третьей цифр числа сдвигу числа вправо должен предшествовать его сдвиг влево. Для выполнения сдвига влево используйте команду ***shl*** (*«Shift Left»* – логический сдвиг влево). Использование этой команды аналогично *shr*. Выполнение *shl* имеет такой же эффект, как и умножение на два, четыре, восемь и так далее, в зависимости от числа (соответственно единицы, двойки или тройки), хранящегося в *CL* или записанного в качестве второго операнда.

7. СПИСКИ И ПРОЦЕДУРЫ

7.1. Несвязанные списки

Важное место при разработке программ занимают операции с информационными структурами. К таким структурам относятся ***списки***. Они очень широко используются для работы с информацией, находящейся как в ОП, так и в ВП.

Каждый список состоит в общем случае из множества более мелких информационных структур, называемых ***записями***. Записи, входящие в список, содержат «родственную» информацию. Примером списка является список, содержащий сведения о служащих организации. Одна запись такого списка содержит сведения об одном сотруднике организации.

В свою очередь, запись состоит из одного или нескольких ***полей*** (рис. 1.16). Содержанием поля может быть, например, число или набор символов. Для приведенного выше примера поля записи могут содержать фамилию, имя, отчество, год рождения и так далее. Будем считать, что все записи, входящие в один и тот же список, имеют одинаковую (фиксированную) длину. Кроме того, будем считать для удобства, что все записи списка имеют одинаковую ***структуру***, т. е. одинаковый состав и порядок полей.

Поле 1	Поле 2	.....	Поле *M*

Рис. 1.16. Структура записи

В списке записи расположены, не как попало, а в определенном ***логическом порядке***. В зависимости от этого порядка все списки делятся на линейные и нелинейные. ***Линейным списком*** называется список, логическое расположение записей в котором наглядно описывается прямой линией (рис. 1.17).

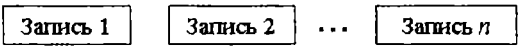

Рис. 1.17. Линейный список

Пример списка, который не является линейным, приведен на рис. 1.18.

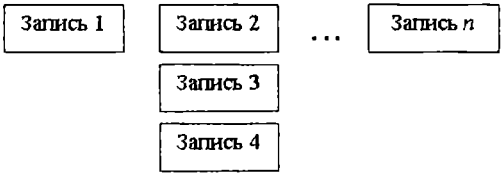

Рис. 1.18. Пример нелинейного списка

Логическое расположение записей в списке в общем случае отличается от их ***физического размещения*** в памяти (оперативной или внешней). Это значит, что две записи, являющиеся в списке соседними, могут занимать участки памяти, расположенные далеко друг от друга.

Логический и физический порядки размещения записей совпадают для линейных списков, называемых ***несвязанными***. Для несвязанного списка характерно то, что, зная адрес памяти, которую занимает данная запись, мы без труда можем найти в памяти логически соседние к ней записи. Допустим пока, что мы имеем дело с несвязанными линейными списками. В зависимости от состава допустимых операций над списком будем различать следующие типы линейных списков: 1) линейный список общего вида; 2) стек; 3) очередь. ***Линейный список общего вида*** допускает наибольшее число операций:

1) получение доступа к k-й записи списка, чтобы проанализировать или изменить содержимое ее полей;

2) включение новой записи непосредственно перед записью k;

3) исключение записи k из списка;

4) объединение двух или более линейных списков в один;

5) определение числа записей в списке;

6) поиск записи в списке с заданным значением некоторого поля записи;

7) сортировка записей списка в порядке возрастания или убывания значения некоторого поля записи.

Для удобства выполнения перечисленных операций над линейным списком общего вида вводятся две вспомогательные переменные (***переменная*** – небольшая область в ОП или в ВП, или это – регистр):

S – содержит адрес в памяти, где расположена первая запись списка;

F – содержит адрес в памяти последней записи списка.

Переменные S и F часто называют указателями соответственно на начало и конец списка. Пользуясь этими переменными, нетрудно входить в список как с начала, так и с конца (рис. 1.19).

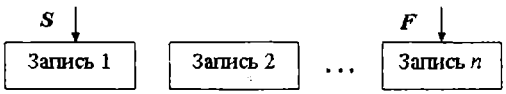

Рис. 1.19. S и F – указатели на начало и конец линейного списка

Стек – линейный список, над которым допустимы только 2 операции:

1. Включение новой записи в начало списка.

2. Исключение записи, стоящей первой от начала списка. При работе со стеком выполняется правило «последним пришел, первым ушел». Это аналогично стопке подносов в столовой: поднос, который был положен на стопку последним, первым будет с нее снят. Так как включения и исключения из стека производятся только в его начале, то нужна только одна переменная *S*. На рис. 1.20 и рис. 1.21 приведены примеры включения и исключения записи.

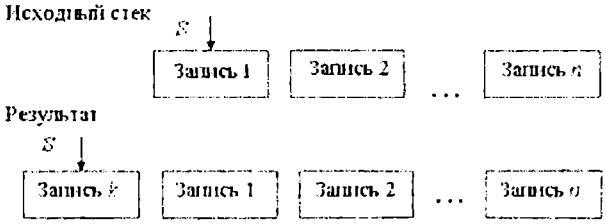

Рис. 1.20. Включение записи *k* в стек

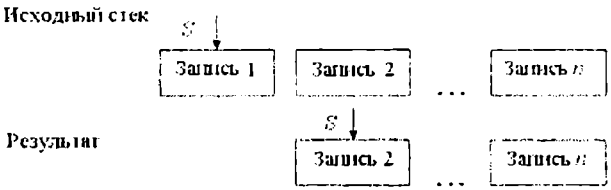

Рис. 1.21. Исключение записи из стека

Очередь – линейный список, над которым допустимы только две операции:

1) включение новой записи в конец очереди;

2) исключение записи, стоящей в начале очереди.

На рис. 1.22 и рис. 1.23 приведены примеры включения и исключения записи.

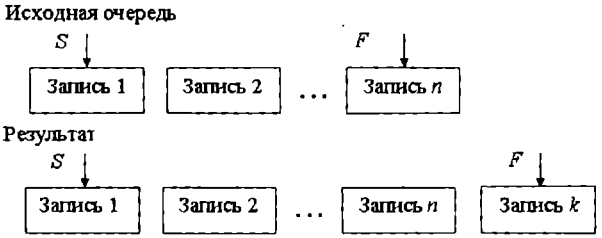

Рис. 1.22. Включение записи *k* в несвязанную очередь

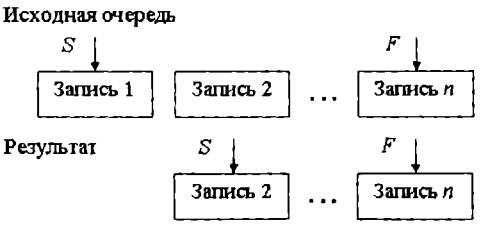

Рис. 1.23. Исключение записи из несвязанной очереди

7.2. Связанные списки

Снимем теперь допущение, что записи, расположенные по соседству в линейном списке, занимают и соседние места в памяти. Подобное несоответствие между логическим и физическим расположением записей допускается в ***связанных списках***. Связанные линейные списки бывают одно- и двухсвязанные. В ***односвязанном списке*** каждая запись имеет одно специальное поле, содержащее указатель на соседнюю запись в списке (рис. 1.24).

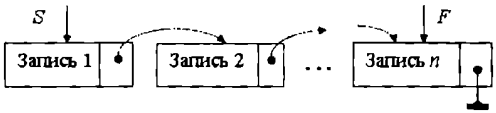

Рис. 1.24. Пример линейного односвязанного списка

Указатель – это адрес соседней записи, пользуясь которым можно найти эту соседнюю запись в памяти. Переменной F может и не быть, так как двигаться справа налево мы все равно не сможем. Единственное применение данной переменной – добавление новой записи в конец списка. Указатель ⊥ – пустой указатель. Он часто кодируется помещением в поле указателя нуля.

В ***двухсвязанном списке*** каждая запись имеет два поля указателей на обе соседние записи (рис. 1.25).

Для связанного линейного списка общего вида определены те же операции, что и для несвязанного. Часто применяются связанные стеки и очереди. Исходя из определений этих списков, связанный стек может быть только одно-, а очередь как одно- так и двухсвязанной. Операции над этими списками аналогичны операциям над соответствующими несвязанными списками.

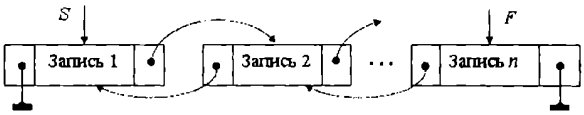

Рис. 1.25. Двухсвязанный линейный список

В отличие от несвязанных списков, связанные списки могут быть нелинейными. Важнейшим видом нелинейных списков является ***дерево*** (рис. 1.26). Запись 1 называется ***корнем дерева***. Записи, у которых нет ***сыновей***, называются ***листьями*** – 1.1, 1.2.1, 1.2.2.1, 1.2.2.2, 1.2.3. Совокупность записей, начиная от корня и кончая каким-то листом, называется ***путем***. Пример пути: «1, 1.2, 1.2.1».

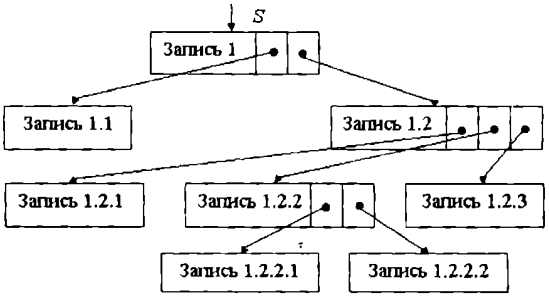

Рис. 1.26. Пример дерева

7.3. Программные стеки

В принципе программа может обрабатывать любые списки, но несвязанный стек – единственный вид списков, для работы с которым ЦП имеет специальные регистры и машинные команды.

Допустим, что для нашей программы операционная система (ОС) выделила область (сегмент) ОП, начиная с параграфа 20*h* (рис. 1.27). Регистр *CS* содержит число 20*h* и представляет собой указатель на начало этой области. Одновременно с назначением области ОП для размещения программы ОС назначает сегмент для размещения стека, который будет обслуживать нашу программу и который поэтому называется ***программным стеком***.

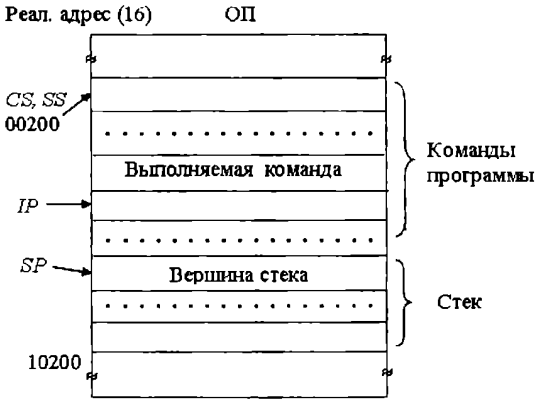

Рис. 1.27. Пример размещения программы в ОП

Где находится стек в ОП? Для того чтобы выполнять операции со стеком, достаточно знать адрес в памяти вершины стека. Этот адрес всегда содержится в паре регистров: 1) регистр сегмента стека – ***SS***; 2) указатель стека – ***SP*** (от *Stack Pointer* – указатель стека). Реальный адрес ячейки, являющейся вершиной стека, получается аппаратно путем суммирования содержимого *SS*, умноженного на 16, с содержимым *SP*.

Содержимое регистров *SS* и *SP* вы многократно наблюдали ранее, получая листинг регистров по команде *R* или какой-то другой команде Debug. Первоначальную запись в данные регистры выполняет DOS, руководствуясь следующим. Во-первых, в *SS* записывается тот же номер параграфа, что и в регистр сегмента кодов *CS*. Это означает, что для стека отводится тот же сегмент ОП, что и для программы (см. рис. 1.27).

Первоначально в указатель стека DOS записывает максимально возможное число без знака (*FFFF*) или чуть меньшее число, в результате чего вершиной стека является последняя ячейка сегмента стека (и сегмента программы), называемая «дном» стека. «Растет» стек в отличие от программы в сторону не больших, а меньших адресов. При добавлении слова данных в стек содержимое *SP* уменьшается на 2, а при исключении слова из стека – увеличивается на 2.

Программа может выполнить включение слова данных в стек, используя машинную команду «***push*** *b*», где *b* – адрес ячейки памяти (ОП или регистровой памяти), содержимое которой следует включить в стек. Исключение слова данных из стека выполняет команда «***pop*** *b*», где *b* – адрес ячейки памяти, в которую следует считать слово данных из стека.

Проверьте с помощью команды *R* Debug содержимое регистров *SS* и *SP*. Выполните команды «*push ax*» и «*pop ax*», наблюдая за содержимым регистра *SP*.

В заключение заметим, что многоцелевое использование одного и того же стека делает необходимым повышенное внимание при работе с ним: каждое записанное в стек слово должно быть вовремя извлечено оттуда.

7.4. Процедуры

За исключением очень небольших программ, разработка программы предполагает ее представление в виде совокупности относительно независимых частей (модулей), называемых ***подпрограммами***. Применение подпрограмм предоставляет следующие выгоды:

1) существенное уменьшение трудоемкости программирования за счет возможности выполнять разработку программы не целиком, а по частям;

2) существенное сокращение памяти для размещения программы, так как, выделив в подпрограмму многократно повторяющийся участок кода программы, достаточно выделить память лишь для одной подпрограммы, инициируя ее из различных мест программы;

3) уменьшение трудоемкости программирования за счет того, что одну и ту же подпрограмму можно использовать не в одной, а в нескольких программах.

Машинные подпрограммы бывают двух типов: процедуры и обработчики прерываний. Любой обработчик прерываний можно инициировать (запустить) из своей программы, поместив в нее машинную команду «*int n*», где *n* – номер прерывания. Более подробно прерывания и их обработчики будут рассмотрены во второй части пособия, а сейчас остановимся на процедурах.

Процедура – список машинных команд, который можно вызывать из различных мест программы. Переход к процедуре называется ***вызовом процедуры***, а соответствующий переход назад называется ***возвратом из процедуры***. Вызов процедуры выполняет машинная команда «***call*** *b*», где *b* – адрес, по которому находится первая команда процедуры. Возврат осуществляет команда ***ret*** (рис. 1.28). Возврат после каждого вызова осуществляется к команде, которая находится в памяти сразу за командой *call*.

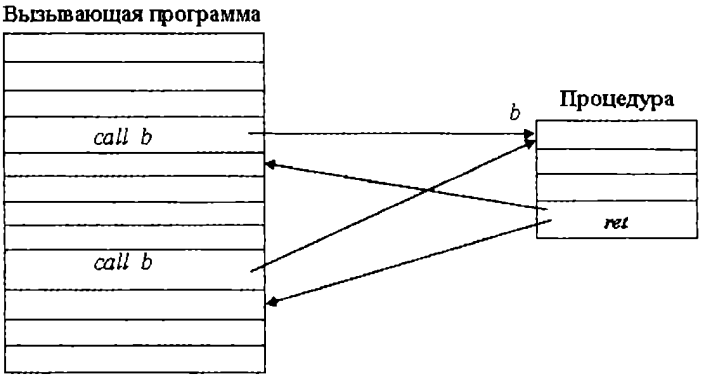

Рис. 1.28. Вызов процедуры и возврат из нее

Подобно команде *jmp*, команды *call* и *ret* выполняют или близкие переходы, когда вызывающая программа и процедура находятся в одном сегменте кода, или дальние переходы – программа и процедура находятся в разных сегментах. В первом случае адрес перехода *b* представляет собой число – смещение первой команды процедуры относи-

тельно начала сегмента (новое значение регистра *IP*). Во втором случае это пара чисел: (*CS, IP*). Так как пока наши программы небольшие, то далее речь будет идти только о близком вызове процедур.

Допустим, что для вызова процедуры мы будем использовать команду «*call 200h*», где *200h* – адрес, по которому находится первая команда процедуры. Вспомним, что первая команда нашей программы находится по адресу *100h*. Располагая процедуру по адресу *200h*, мы стремимся убрать ее подальше от основной программы. Если мы запишем список команд процедуры (тело процедуры) по другому адресу, то этот адрес следует записать вместо адреса *200h*.

Следующая программа вызывает десять раз процедуру, которая каждый раз печатает один символ, начиная от *A* и заканчивая *J*:

```
100  mov  dl,41
102  mov  cx,000a
105  call 0200
108  loop 0105
10A  int  20
```

Текст процедуры:

```
200  mov  ah,02
202  int  21
204  inc  dl
206  ret
```

В тексте процедуры содержится новая команда ***inc***, которая увеличивает содержимое своего операнда (в примере – регистр *DL*) на единицу. Команда *ret* возвращает управление в то место основной программы, откуда процедура была вызвана. Следующей командой, выполняемой после команды *ret*, будет та команда основной программы, которая следует за командой *call*. В примере это команда *loop*.

В в е д и т е основную программу и процедуру в память. С помощью команды *G* выполните программу, а затем протрассируйте ее с целью детального изучения работы. При этом особое внимание уделите изменению регистра *IP*.

При вызове процедуры и возврате из нее необходимо выполнить следующие три требования:

1) при вызове процедуры необходимо запомнить адрес следующей команды, чтобы можно было впоследствии осуществить возврат в нужное место вызывающей программы;

2) используемые процедурой регистры необходимо запомнить до изменения их содержимого, а перед самым выходом из процедуры восстановить;

3) процедура должна иметь возможность выполнять обмен данными с вызывающей ее программой.

Первое требование реализуют команды *call* и *ret*. При выполнении команды вызова процедуры *call* программа «перескакивает» в другую область ОП (например, по адресу *200h*), где находится первая команда тела вызываемой процедуры. Так как при выполнении команды процедуры *ret* мы возвращаемся в основную программу, то необходимо где-то хранить адрес возврата (адрес команды, следующей за командой *call*). В качестве места хранения адреса возврата используется программный стек. При близком вызове команда *call* помещает адрес следующей за ней команды (находится в *IP*) в программный стек, а *ret* извлекает этот адрес из стека и помещает его в указатель команды *IP*. При

дальнем вызове *call* помещает в стек, а *ret* извлекает оттуда не одно, а два слова – содержимое регистров *CS* и *IP*.

Заметим, что вызов процедуры не требует от нас использования команд *push* и *pop*, так как работа со стеком в данном случае производится «автоматически» – команда *call* помещает адрес возврата в стек, а команда *ret* извлекает его оттуда.

Введите в память (если они там не сохранились) предыдущие программу и процедуру. Протестировав программу, наблюдайте за содержимым указателя стека *SP* до и после исполнения команд *call* и *ret*. Сразу же после любого выполнения команды *call* найдите в стеке адрес возврата (108), используя команду *D* Debug.

Свойство стека «пришел первым, ушел последним» идеально подходит для реализации вложенных вызовов процедур (рис. 1.29), когда одна вызываемая процедура вызывает другую процедуру, которая, в свою очередь, вызывает третью и т. д. При этом размещение в программном стеке адреса возврата вызывающей процедуры производится раньше, а его выборка позже, чем соответствующие операции с адресом возврата вызываемой процедуры, что соответствует правилу «последним пришел, первым ушел».

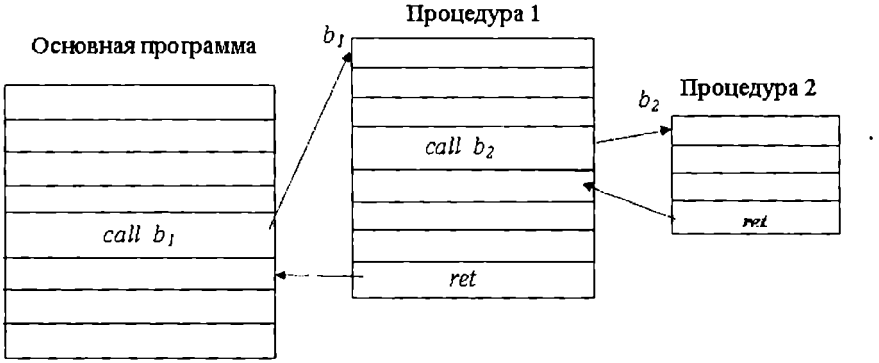

Рис. 1.29. Вложенный вызов процедур

Допустим, что программа состоит из главной программы и трех процедур. Главная программа имеет вид

```
100  call  200
103  int   20
```

По адресу 200 находится процедура, которая печатает букву *A* и вызывает вторую процедуру, записанную по адресу 300, которая выводит *B*. Вторая процедура вызывает третью процедуру, первая команда которой находится по адресу 400. Эта третья процедура выполняет только вывод на экран буквы *C*.

Введите данную программу в память и протрассируйте ее, наблюдая за изменением регистра *SP* и содержанием стека. Обязательно найдите в стеке адреса возврата, когда вызваны все три процедуры.

Требование восстановления содержимого регистров, используемых в вызывающей программе, также выполняется с помощью стека. В начале процедуры ее команды *push* помещают в стек содержимое запоминаемых регистров, а в конце ее команды *pop* извлекают из стека запомненные слова и помещают их в регистры. Порядок регистров при восстановлении обратный их порядку при запоминании. Пример процедуры, в которой производится такое восстановление содержимого регистров:

```
200  push  cx
201  push  dx
202  mov   dl,0a
205  call  300
208  inc   dl
20C  pop   dx
20D  pop   cx
20E  ret
```

Обратите внимание, что команды *pop* расположены по отношению к командам *push* в обратном порядке, так как *pop* удаляет слово, помещенное в стек последним, а старое значение *CX* находится в стеке «глубже» старого значения *DX*.

Сохранение и восстановление регистров *CX* и *DX* позволяет произвольно изменять их значения внутри процедуры (которая начинается с адреса *200h*), в то же время значения регистров, используемых процедурами, вызывающими эту, сохранены. Следовательно, мы можем использовать эти регистры для хранения ***локальных переменных*** – переменных, которые применяются внутри процедуры, не влияя на значения переменных, используемых вызывающей программой.

В дальнейшем обязательно выполняйте правило: все регистры, содержимое которых меняется внутри процедуры, за исключением регистров, используемых для передачи выходных данных процедуры в вызвавшую ее программу, в конце процедуры должны восстанавливать свои прежние значения. При несоблюдении этого правила отладка ваших программ затянется на долгие недели, а возможно, и месяцы.

Рассмотрим теперь обмен данными между процедурой и вызывающей ее программой. Данные, передаваемые процедуре при ее вызове, называются ***входными параметрами процедуры***. А данные, которые передаются процедурой в вызывающую ее программу, называются ***выходными параметрами***. Информационный обмен между программой и процедурой обеспечивается путем использования областей памяти одного или нескольких следующих видов: 1) регистры данных; 2) программный стек; 3) другие области ОП. Если передаваемых данных немного, то достаточно регистров данных, большее количество данных может быть передано с помощью стека. Если же этих данных слишком много, то они помещаются в область ОП, а начальный адрес этой области записывается или в регистр данных, или в стек.

В завершение заметим, что любой язык программирования позволяет записывать ***виртуальные программные процедуры***. В зависимости от языка программирования они называются процедурами, функциями или подпрограммами. В любом случае текст любого из перечисленных модулей преобразуется соответствующим транслятором в код машинной программной процедуры.

8. Ввод с клавиатуры шестнадцатеричных чисел

В предыдущем разделе были рассмотрены теоретические вопросы, связанные с использованием в программе стека и процедур. Теперь перейдем к практическому рассмотрению этих вопросов в процессе решения важной задачи ввода с клавиатуры шестнадцатеричных чисел. Попутно рассмотрим некоторые новые машинные команды.

8.1. Ввод одной шестнадцатеричной цифры

Для того чтобы ввести с клавиатуры в программу ASCII-символ, можно воспользоваться командой программного прерывания «*int 21h*» с функцией номер 1. Вызванная в результате данной команды подпрограмма DOS помещает ASCII-символ, соответствующий нажатой клавише, в регистр *AL*.

Введите команду «*int 21h*» по адресу *100h*, поместите номер функции 1 в регистр *AH*, а затем запустите команду с помощью «*G 102*» либо *P*. В результате DOS переходит в состояние ожидания нажатия вами клавиши (на экране вы видите мерцающий курсор). Нажмите любую клавишу, соответствующую шестнадцатеричной цифре (*0÷F*). Убедитесь, что в результате регистр *AL* содержит соответствующий код ASCII.

При преобразовании ASCII-символа, который содержится в регистре *AL*, в шестнадцатеричную цифру, решается задача, обратная той, которую мы решали при выводе цифры на экран. На рис. 1.30 приведена блок-схема программы, которая выполняет ввод цифры с клавиатуры в регистр *AL*.

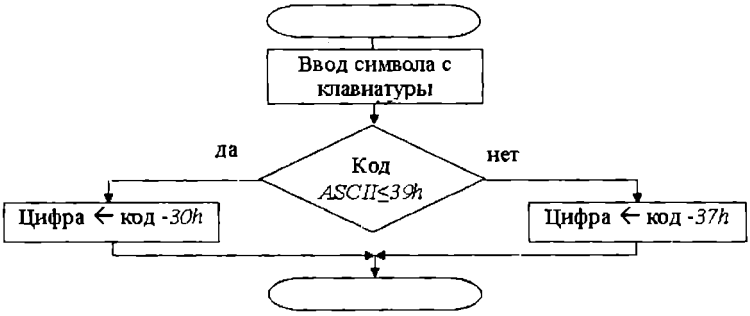

Рис. 1.30. Алгоритм ввода шестнадцатеричной цифры

Запишите текст программы ввода шестнадцатеричной цифры и поместите его в память. Для программирования условия можно использовать не только уже знакомую нам команду условного перехода *ja* (перейти, если больше), но и обратную ей команду ***jbe*** (перейти, если меньше или равно). Обе команды используются после сравнения беззнаковых величин, каковыми коды ASCII и являются.

Так как результат данной программы содержится в регистре *AL*, то этот регистр необходимо проанализировать прежде, чем исполнится команда «*int 20*» (Debug восстанавливает регистры после этой команды). Поэтому для запуска программы используйте команду Debug «*G d*», где *d* – смещение команды «*int 20*».

Выполните программу несколько раз, нажимая не только клавиши, соответствующие шестнадцатеричным цифрам, но и другие клавиши. При этом убедитесь, что программа реагирует на неправильное нажатие клавиши точно так же, как и на правильное. В дальнейшем этот недостаток будет устранен.

8.2. Ввод двузначного шестнадцатеричного числа

Такой ввод можно осуществить следующим образом. Введем старшую цифру числа, записав ее в четыре младших бита регистра *DL*. Далее умножим регистр *DL* на 16, в результате чего цифра переместится в старшие четыре бита *DL*. После этого введем с кла-

виатуры младшую цифру числа и, просуммировав *AL* с *DL*, получим в *DL* все двузначное шестнадцатеричное число. Соответствующий алгоритм приведен на рис. 1.31.

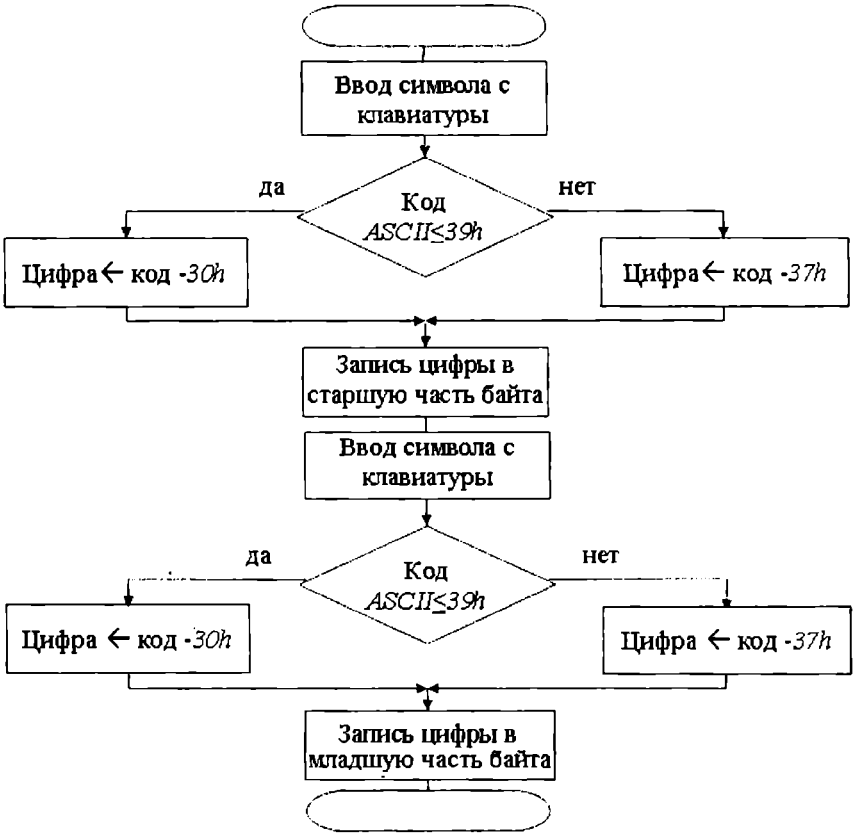

Рис. 1.31. Алгоритм ввода двузначного шестнадцатеричного числа

З а п и ш и т е программу ввода с клавиатуры в регистр *DL* двузначного шестнадцатеричного числа. (Для сдвига регистра *DL* влево используйте рассмотренную ранее команду *shl*.) Введите данную программу в память и протрассируйте ее при различных парах чисел, вводимых с клавиатуры.

Необходимо убедиться, что программа правильно работает при граничных условиях. Это пары чисел: *00; 09; 0A; 0F; 90; A0; F0*. Используйте точку останова для запуска программы без выполнения команды «*int 20h*». (Для ввода шестнадцатеричных чисел используйте только заглавные буквы.)

8.3. Более совершенный ввод шестнадцатеричных цифр

Записанные ранее программы ввода одно- и двузначного шестнадцатеричных чисел реагировали на нажатие «нецифровых» клавиш точно так же, как и «цифровых». Теперь мы получим программу, которая не будет реагировать на нажатие «нецифровых» клавиш.

Для этого в состав программы введем процедуру ввода шестнадцатеричной цифры. Эта процедура возвратит управление в главную программу только тогда, когда она по-

лучит с клавиатуры правильную шестнадцатеричную цифру. При нажатии «нецифровой» клавиши ее код на экран не выводится, и управление в программу не возвращается.

Ранее для ввода символа с клавиатуры использовалась функция 1 программного прерывания *21h*. Эта функция не только вводит символ с клавиатуры, но и выводит его на экран. Подобный вывод символа во время его ввода называется «эхом» символа. Теперь мы будем использовать функцию 8 прерывания 21, которая не выводит «эхо» символа.

На рис. 1.32 приведена блок-схема процедуры ввода шестнадцатеричной цифры.

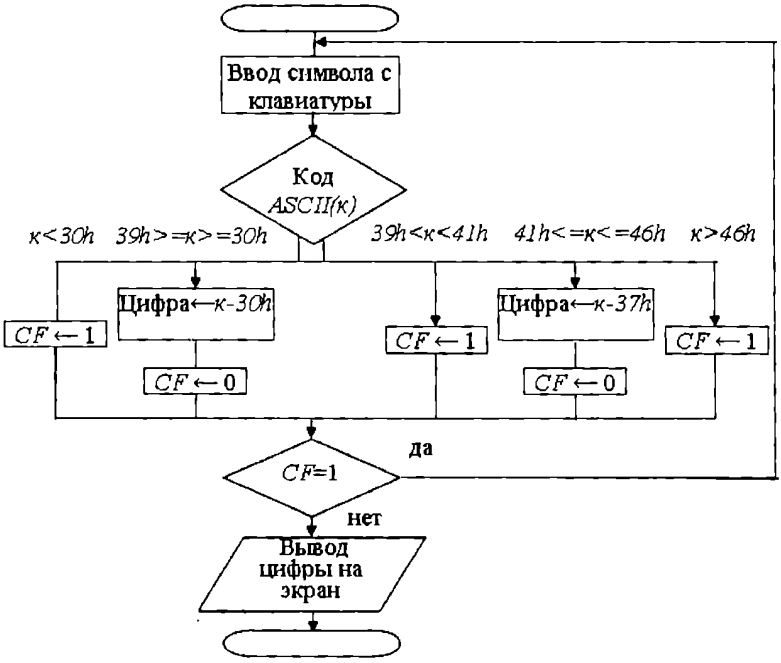

Рис. 1.32. Алгоритм ввода одной шестнадцатеричной цифры:
CF – флаг ошибки (0 – ошибки нет, 1 – ошибка есть)

Текст этой процедуры (возвращает цифру в регистре *BL*):

```
200  push  ax
201  push  dx
202  mov   ah,8
204  int   21
206  mov   dl,al
208  cmp   al,30
20A  jb    222
20C  cmp   al,39
20E  ja    215
210  sub   al,30
212  clc
213  jmp   223
215  cmp   al,41
217  jb    222
219  cmp   al,46
```

```
21B  ja    222
21D  sub   al,37
21F  clc
220  jmp   223
222  stc
223  jc    202
225  mov   bl,al
227  mov   ah,2
229  int   21
22B  pop   dx
22C  pop   ax
22D  ret
```

Изучите внимательно алгоритм и найдите реализацию его этапов в тексте процедуры. Во-первых, обратите внимание, что управляющей структурой верхнего уровня является цепочка из цикла ПОКА и этапа «Вывод цифры на экран». В качестве условия повторения цикла выступает равенство единице флага ошибки *CF*.

Имя *CF* флага ошибки обусловлено тем, что он реализуется в программе с помощью одноименного флага переноса. При этом флаг переноса используется совсем не для того, для чего он был создан. Данный прием широко распространен, и мы также будем его использовать.

Существуют специальные команды для работы с флагом *CF*. Команда ***stc*** устанавливает флаг (*CF=1*), а ***clc*** сбрасывает его (*CF=0*). Команда условного перехода ***jc*** выполняет переход при *CF=1*, а команда ***jnc*** – при *CF=0*.

Введите данную процедуру в память. Кроме того, запишите в память следующую программу для проверки процедуры:

```
100  call  200
103  int   20
```

Протрассируйте программу, используя команду *P* Debug для перехода через команды *int*. Курсор появится в левой части экрана и будет ждать ввода символа. Напечатайте символ «К», который не является правильным символом. Ничего не произойдет. Теперь напечатайте один из заглавных шестнадцатеричных символов. Вы должны увидеть шестнадцатеричную цифру в регистре *BL*, а также на экране. Испытайте эту процедуру на граничных условиях: «\» (символ, стоящий перед нулем), «0», «9», «:» (символ, стоящий после 9) и так далее.

Теперь, когда у нас есть процедура ввода одной шестнадцатеричной цифры, программа, считывающая двузначное шестнадцатеричное число в регистр *DL* и обрабатывающая ошибки, стала достаточно простой. Ее алгоритм приведен на рис. 1.33.

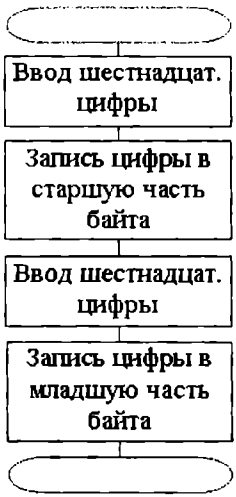

Рис. 1.33. Алгоритм ввода двузначного шестнадцатеричного числа

Лабораторная работа 6

Требуется дополнить разработанную в лабораторной работе 4 программу вывода на экран двоичного содержимого двух регистров так, чтобы первоначальное содержимое этих регистров вводилось с клавиатуры в виде шестнадцатеричных чисел.

Пример информации на экране:

```
ВВЕДИТЕ СОДЕРЖИМОЕ РЕГИСТРА AX    F46B
ВВЕДИТЕ СОДЕРЖИМОЕ РЕГИСТРА BX    5A0C
(AX) = 1111010001101011    (BX) = 0101101000001100
```

☑ **Примечания:**

1. Рекомендуется дополнительно разработать процедуры, одна из которых выполняет ввод шестнадцатеричного числа в 16-битовый регистр, а другая – вывод содержимого регистра в двоичном виде.

2. Следует обратить особое внимание на недопустимость использования одного и того же регистра одновременно для нескольких целей. При этом для временного хранения содержимого регистра удобно использовать стек. Например, с помощью команды «*push ax*» можно записать прежнее содержимое *AX* в стек, затем использовать регистр *AX* для других целей, а затем вернуться к прежнему содержимому этого регистра с помощью команды «*pop ax*». (При этом если удобно, слово из стека можно «вытолкнуть» не в *AX*, а в любой другой регистр.)

3. Для того чтобы выполнить перевод экранной строки, следует последовательно вывести на экран два символа (в любом порядке): *0ah* (перевод строки) и *0dh* (возврат каретки).

Часть II. АССЕМБЛЕРНЫЕ ПРОГРАММЫ В СРЕДЕ DOS

9. Системные программы

До сих пор мы использовали при получении своих программ единственного помощника – программу Debug. Такое использование не принесло нам особого удовлетворения из-за чрезвычайно большой трудоемкости процесса программирования. Причиной этого является то, что Debug предназначен не для получения текстов программ, а для отладки программ. Теперь мы перейдем от процесса написания программ на машинном языке, которым мы фактически занимались, к написанию программ на языке ассемблера. Для этого нам потребуется помощь некоторых новых системных программ. Прежде чем рассматривать эти программы, полезно рассмотреть общие свойства и различия системных программ.

9.1. Функции системных программ

Несмотря на то, что наличие аппаратуры и прикладных программ является минимально достаточным условием для решения задач по переработке информации, ЭВМ, состоящие только из этих двух подсистем, на практике не используются. Обязательной подсистемой любой ЭВМ являются также системные программы. Благодаря им и пользователи ЭВМ, и их прикладные программы имеют дело не с реальной («голой») аппаратурой, а взаимодействуют с ***виртуальной*** (кажущейся) ***ЭВМ***.

На рис. 2.1 показано наиболее укрупненное представление ЭВМ, в том числе наиболее важные системные интерфейсы. При этом интерфейс пользователя ЭВМ с аппаратурой представляет собой клавиши мыши и клавиатуры, поверхность экрана, а также различные кнопки на системном блоке и на периферийных устройствах. Интерфейс между пользователем и системными программами представляет собой ***язык управления операционной системой***, называемый также ***командным языком*** или ***языком директив***. Одна команда этого языка задаёт выполнение одной обрабатывающей программы (прикладной программы, утилиты или лингвистического процессора). Интерфейс пользователя и прикладной программы представляет собой язык диалога этой программы.

Интерфейс между аппаратурой и программами представляет собой язык машинных команд. ***Системный программный интерфейс*** – множество машинных команд, называемых ***системными вызовами***, посредством которых прикладные программы обращаются за помощью к системным программам. При выполнении прикладной программы в среде DOS в качестве системных вызовов используются исключительно команды *int*.

Совокупность интерфейсов, используемых пользователем или прикладной программой, и представляет собой соответствующую виртуальную машину. Например, прикладная программа «выполняется» на виртуальной ЭВМ, предоставляемой языком машинных команд, а также совокупностью системных вызовов. При этом каждый системный вызов может рассматриваться как команда виртуальной ЭВМ, которой в действительности соответствуют многие десятки или сотни машинных команд, образующие вызываемую системную подпрограмму.

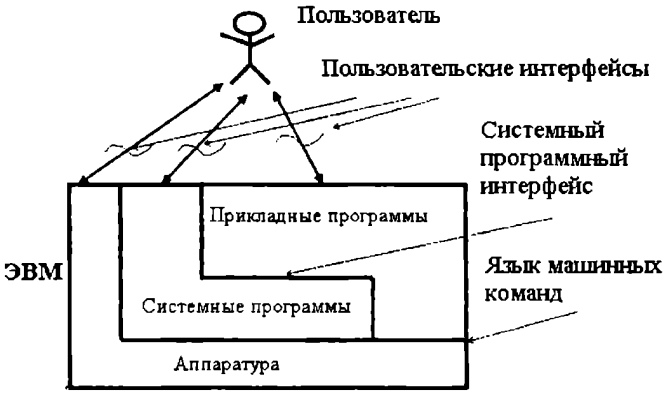

Рис. 2.1. Укрупненное представление ЭВМ

На рис. 2.2 приведена классификация системных программ. ***Обрабатывающие системные программы*** отличаются от управляющих системных программ как по своим функциям, так и по способу их инициирования (запуска). Основные функции обрабатывающих программ:

- Перенос информации. Перенос может выполняться между различными устройствами или в пределах одного устройства. При этом под устройствами понимаются: ОП, устройства ВП, устройства ввода-вывода.
- Преобразование информации, т. е. после считывания информации с устройства обрабатывающая программа преобразует эту информацию, а только затем записывает ее на это же или на другое устройство.

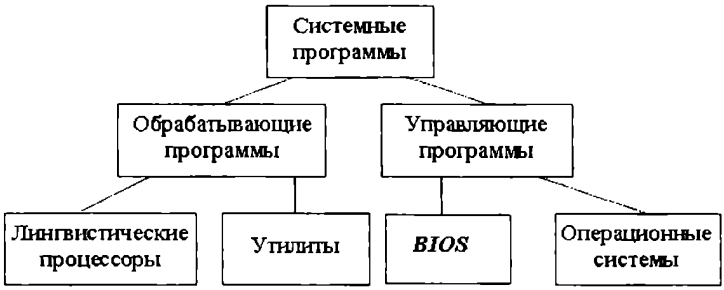

Рис. 2.2. Классификация системных программ

В зависимости от того, какая из этих двух функций является основной, обрабатывающие системные программы делятся на утилиты и лингвистические процессоры. Основной функцией ***утилиты*** является перенос информации, а основная функция ***лингвистического процессора*** – перевод описания алгоритма с одного языка на другой. Сущность алгоритма при этом сохраняется, но форма его представления, ориентированная на программиста, преобразуется в форму, ориентированную на ЦП. Лингвистические процессоры делятся на трансляторы и интерпретаторы.

В результате работы ***транслятора*** алгоритм, записанный на языке программирования (***исходная виртуальная программа***), преобразуется в алгоритм, записанный на машинном языке. (На самом деле, как будет показано позже, машинная программа является результатом совместной работы нескольких лингвистических процессоров.)

Трансляторы делятся на компиляторы и ассемблеры. Исходная программа для ***транслятора-ассемблера*** записывается на ***языке-ассемблере***. Один оператор данного языка транслируется в одну машинную команду. Исходная программа для ***компилятора*** записывается на ***языке программирования высокого уровня***. Каждый оператор такого языка транслируется в несколько машинных команд. Примерами языков высокого уровня являются Паскаль и Си.

Интерпретатор в отличие от транслятора не выдаёт машинную программу целиком. Выполнив перевод очередного оператора исходной программы в соответствующую совокупность машинных команд, интерпретатор обеспечивает их выполнение. Затем преобразуется тот исходный оператор, который должен выполняться следующим по логике алгоритма, и так далее.

Примером интерпретатора является ***интерпретатор командного языка (ИК)***. Другие названия этого модуля: ***командный процессор***, ***командная оболочка***. ИК представляет собой лишь «видимую», сравнительно небольшую часть операционной системы. На самом деле, чтобы интерпретировать, т. е. выполнить (преобразовав в совокупность машинных команд), очередную команду пользователя, ИК инициирует многие другие модули ОС. Одни из них ищут текст требуемой прикладной программы на диске, другие выделяют необходимые аппаратные ресурсы, третьи загружают программу в ОП и инициируют ее. Попутно заметим, что, так как одна команда пользователя задаёт выполнение целой программы (прикладной, утилиты или системы программирования), то язык управления ОС может рассматриваться как язык программирования очень высокого уровня.

Утилиты – обрабатывающие системные программы, которые выполняют:

- Перенос данных с одного периферийного устройства на другое или перенос данных в пределах одного устройства. Примеры: программа копирования данных на магнитном диске; программа ввода данных с клавиатуры терминала на диск; программа распечатки информации на диске.
- Программы изменения расположения данных. Это различные программы сортировки.
- Программы для изменения представления данных. Сюда относятся редакторы, которые осуществляют редактирование виртуальных программ и других текстов, а также программы перекодировки данных для согласования программ, использующих разные кодировки, или для обеспечения секретности.

Примером сложной утилиты является DOS Navigator. Эта утилита переносит с диска на экран информацию, содержащуюся в любом каталоге любого логического диска. По запросу пользователя она выводит на экран файловую структуру любого логического диска. Кроме того, она предоставляет пользователю удобный язык управления операционной системой DOS за счет того, что она переносит имя исполняемого файла программы из позиции экрана, отмеченной пользователем с помощью курсора-маркера, в то место памяти, откуда это имя может взять интерпретатор команд ОС. ***Курсор-маркер*** – светящийся прямоугольник, генерируемый не аппаратурой экрана (как настоящий курсор), а получаемый программно. Ниже мы еще вернемся к рассмотрению DOS Navigator.

В отличие от лингвистических процессоров, утилиты используются не только программистами, но и ***пользователями-непрограммистами***. Эта наиболее многочисленная категория пользователей ЭВМ работает на виртуальных машинах, предоставляемых готовыми прикладными программами, а также утилитами. При этом заметим, что запуск любой обрабатывающей системной программы (утилиты или лингвистического процес-

сора) аналогичен запуску прикладной программы. Более того, с точки зрения других системных программ, а также аппаратуры, системные обрабатывающие программы ничем не отличаются от обычных прикладных программ.

Основные функции ***управляющих программ***:

- Оказание помощи прикладным и системным обрабатывающим программам в использовании ими ресурсов ЭВМ. При этом различают информационные, программные и аппаратные ресурсы. Данная функция реализуется во всех системах.
- Обеспечение ***однопользовательской мультипрограммности*** – одновременное выполнение нескольких прикладных и (или) системных обрабатывающих программ в интересах одного пользователя. Эта функция реализуется лишь в мультипрограммных системах. В однопользовательских однопрограммных системах эта функция отсутствует.
- Обеспечение ***многопользовательской мультипрограммности*** – одновременное выполнение нескольких обрабатывающих программ (прикладных и системных) в интересах нескольких пользователей. Данная функция реализуется лишь в многопользовательских системах.

Управляющие системные программы делятся на две группы: программы BIOS и программы операционной системы. BIOS – базовая система ввода-вывода. Сюда относятся системные программы, находящиеся в ПЗУ (постоянное запоминающее устройство). Эти программы выполняют многие функции обмена с периферийными устройствами, участвуя, таким образом, в выполнении первой из перечисленных выше функций управляющих программ.

Операционная система (ОС) – множество управляющих программ, предназначенных для выполнения всех трех перечисленных выше функций. Примером однопользовательской однопрограммной ОС является любая DOS. Примерами однопользовательских мультипрограммных ОС являются различные Windows. Операционная система UNIX является примером многопользовательской системы.

При рассмотрении любого командного языка, а этим мы займемся чуть позже, невозможно обойтись без понятия файла. В отличие от многих других объектов, управляемых ОС, файлы «видимы» для пользователя и используются им при формировании своих команд для ОС.

9.2. Файлы

Вся информация, обрабатываемая ВС, содержится в ней на устройствах: ОП, ВП, устройства ввода-вывода. При этом на любом из устройств информация хранится в виде длинной битовой строки, т. е. в виде последовательности нулей и единиц. Длина одной такой битовой строки может составлять многие сотни Гбит ($1Г = (1024)^3$). Так как работать с такой длинной строкой чрезвычайно неудобно, то она разделяется на поименованные части разной длины, называемые файлами. Более точное определение: ***файл*** – часть пространства носителя ВП (разрывная или непрерывная), которой присвоено имя, уникальное для данной ЭВМ.

Информация на носителе делится на части-файлы по смысловому принципу. Например, один файл может содержать текст исходной программы, второй – ее объектный модуль, а третий – загрузочный модуль. Кроме того, в большинстве современных ОС каждое устройство ввода-вывода также считается файлом. Что касается ОП, то информация в ней делится не на файлы, а на сегменты.

Применительно к файлу существуют три пользовательских имени. Наиболее короткое из них – ***простое имя файла***. Это имя дает файлу пользователь при его создании. Ограничения на выбор простого имени файла определяются типом используемой ОС. В достаточно старых операционных системах используется короткое простое имя файла. Например, в DOS (за исключением новых версий) длина простого имени не может превышать 12 символов по схеме «8.3». При этом до восьми символов имеет ***собственно имя файла***, до трех символов – ***расширение имени файла***, один символ – разделительная точка. Применение расширений позволяет давать «родственным» файлам «родственные» имена, что весьма удобно для пользователя. Обычно расширение указывает на тип файла и учитывается многими системными программами. Например, если текст ассемблерной программы записывается текстовым редактором в файл *Abc.asm*, то объектный модуль помещается транслятором в файл *Abc.obj*, а загрузочный модуль записывается редактором связей в файл *Abc.com* или *Abc.exe*. В новых версиях DOS и в различных Windows длина простого имени файла может достигать 255 символов.

Что касается символов, которые можно использовать для записи имен файлов, то к их составу старые версии DOS также предъявляют гораздо более жесткие требования. В соответствии с этими требованиями в состав символов могут входить строчные и прописные латинские буквы, цифры, а также некоторые служебные символы: «-»; «_»; «$»; «#»; «&»; «@»; «!» ; «%»; «~»; «^»; «(»; «) »; «{»; «}». Придерживаясь далее этих требований, мы обеспечим выполнимость своих программ в среде любой DOS. Попутно заметим, что если в состав имени входят строчные (малые) буквы, то DOS (как и Windows) всегда воспринимает их как прописные (большие).

Файл является достаточно крупной единицей информации. Для удобства работы с ним битовая строка, образующая файл, делится на части, называемые ***записями файла***. При этом файл может рассматриваться как несвязанный линейный список, расположенный на конкретном носителе информации. Заметим, что разные программы могут использовать разное разбиение одного и того же файла на записи.

В реальной ВС одновременно существуют сотни или даже тысячи файлов. Для того чтобы ориентироваться в этом «море», и ОС, и ее пользователь прибегают к помощи ***файловой структуры системы***. На верхнем уровне этой структуры находятся логические диски. ***Логический диск*** – магнитный или оптический диск целиком или его часть, имеющий имя, уникальное для данной ЭВМ. Имя логического диска – буква с двоеточием, например *a:* или *c:*. Каждый логический диск имеет отдельную ***файловую структуру логического диска*** в виде дерева, пример которой приведен на рис. 2.3.

Для построения древовидной структуры применяются специальные файлы, называемые каталогами (в Windows каталоги называются ***папками***). ***Каталог*** – служебный файл, содержащий сведения о других файлах (в том числе, возможно, и о других каталогах). Файл-каталог состоит из 32-байтовых записей, каждая из которых содержит информацию об одном файле. Поля записи содержат сведения о файле: имя, размер, начальный адрес, атрибуты, дату и время последней модификации. При этом поле начального адреса в такой записи играет роль указателя на соответствующий файл (каталог). Две записи каталога всегда используются особо. В одной из них содержится адрес размещения на диске самого каталога, а во второй – адрес «родительского» каталога.

Так как в каталоге могут быть «зарегистрированы» не только файлы данных, но и каталоги, то появляется возможность связать все файлы системы в единую ***иерархическую (древовидную) структуру***. При этом корнем дерева является ***корневой каталог***. На следующем уровне дерева находятся те файлы и каталоги, сведения о которых содержатся в

корневом каталоге. Аналогично, каталоги первого уровня дерева «порождают» файлы и каталоги второго уровня и так далее. Корневой каталог расположен в фиксированной области диска и состоит из 32-байтовых записей, структура которых аналогична структуре записей обычного каталога. Корневой каталог имеет имя «\» (обратный слеш). Простое имя любого другого каталога отличается от имени обычного файла данных тем, что оно не может иметь расширения имени.

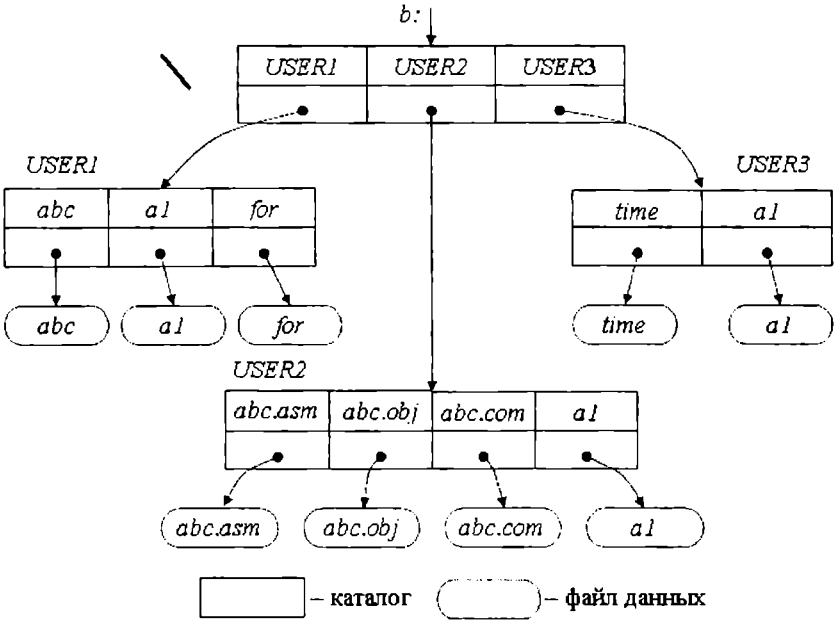

Рис. 2.3. Пример файловой структуры логического диска

Большим достоинством любой древовидной файловой структуры является то, что она позволяет пользователю не заботиться об уникальности простых имен файлов. Это объясняется тем, что ОС работает не с этими пользовательскими именами файлов, а с путями. Единственное ограничение: все файлы, связанные с одним каталогом, должны иметь разные простые имена. ***Имя-путь файла***, называемое также ***абсолютным именем файла***, представляет собой последовательность всех имен, начиная с корневого каталога и кончая простым именем файла. При этом имя каждого промежуточного каталога в имени-пути завершается символом «\». Например, на рис. 2.3 три файла имеют одинаковое простое имя *a1*, но абсолютные имена у них разные: *\user1\a1*, *\user2\a1*, *\user3\a1*.

Так как ЭВМ имеет в общем случае несколько логических дисков, то имя каждого такого диска может рассматриваться в качестве переменной *S* (см. п. 7.2), позволяющей ОС однозначно выбрать среди нескольких древовидных файловых структур требуемую. Один из логических дисков ОС считает ***текущим логическим диском***. Смена текущего диска выполняется командой пользователя для ОС (в DOS для этого достаточно набрать имя требуемого логического диска). Если пользователь хочет задать имя файла, расположенного на логическом диске, отличном от текущего, то он добавляет имя логического диска к имени-пути файла. Пример: *b:\USER2\abc.asm*.

Следует отметить, что ОС «помнит» не только текущий логический диск, но и ***текущий каталог*** на этом диске. Поэтому если искомый файл «зарегистрирован» в текущем каталоге, то его можно задать для ОС не с помощью имени-пути, а используя его простое имя. ОС сама получит имя-путь файла, соединив имя-путь каталога с простым именем файла. С помощью команды ОС пользователь может сменить текущий каталог.

Если адресуемый файл является «потомком» текущего каталога, то в качестве имени этого файла можно использовать «смещение» относительно текущего каталога. Такое пользовательское имя файла называется ***относительным именем***. Например, если текущим каталогом является \, то имя-путь файла *\user1\a1* может быть заменено чуть более коротким именем *user1\a1*. Обратите внимание на отсутствие в начале этого имени символа «\». Его наличие всегда говорит о том, что записано полное имя-путь.

Принято использовать для обозначения текущего каталога символ «.», а для обозначения родительского каталога (по отношению к текущему каталогу) – «..». Например, если текущим каталогом является *\user1*, то для задания файла *\user3\time* пользователь может использовать относительное имя *..\user3\time* . Так как это относительное имя оказалось даже длиннее имени-пути файла на целых два символа, то никакого практического смысла в его использовании нет. Подобный смысл появляется лишь при использовании многоуровневых файловых структур.

9.3. Язык управления DOS

DOS – дисковая операционная система. Этим названием обозначается не одна, а целая группа ОС, имеющих схожие пользовательские и программные интерфейсы. Примерами являются MS-DOS – DOS фирмы Microsoft, а также FreeDOS – DOS, распространяемая свободно. Любая DOS – простая однопользовательская, однопрограммная операционная система. К ее достоинствам относятся весьма малый объем занимаемой памяти, а также большое число разработанных для нее прикладных программ. Кроме того, DOS очень полезна при изучении программирования на ассемблере, так как почти не препятствует любому взаимодействию программ с аппаратурой системы.

9.3.1. Запуск DOS

Запуск операционной системы DOS зависит от ее типа. При этом MS-DOS версии 7.0 не является самостоятельной ОС. Поэтому ее запуск и последующее выполнение производится в среде Windows. Способы запуска имитатора MS-DOS из Windows 2000 или XP:

1) <Пуск> → <Программы> → <Стандартные> → <Командная строка>

2) <Пуск> → <Выполнение> → «*cmd*»;

3) запустить DOS Navigator, FAR или другую подобную утилиту, а затем воспользоваться командной строкой MS-DOS внизу экрана.

В отличие от MS-DOS, FreeDOS является самостоятельной ОС, выполняемой на «голой» аппаратуре. Поэтому для ее запуска требуется или отсутствие на ЭВМ другой ОС, или, что более удобно, – предварительный запуск из имеющейся на ЭВМ операционной системы (Windows или UNIX) программного эмулятора аппаратуры. Для запуска FreeDOS требуется, чтобы на логическом системном диске находились файлы, образующие эту ОС:

- *Kernel.sys* – содержит ядро операционной системы;
- *Command.com* – интерпретатор команд DOS.

Эти два файла нельзя корректировать никогда.

Следующие два файла, относящиеся к FreeDOS, можно корректировать:

- *Fdconfig.sys* – перечень сведений об устройствах, имеющихся в ЭВМ, и информация о том, как с этими устройствами работать;
- *Autoexec.bat* – перечень команд ОС, которые она должна выполнить сразу же после своей загрузки в ОП.

Если запуск FreeDOS производится на «голой» аппаратуре, то этот запуск производится автоматически после включения питания. Если запуск этой (или другой) ОС производится при наличии уже работающей ОС, например Windows или UNIX, то предварительно необходимо запустить программный имитатор аппаратуры. В качестве такого имитатора можно использовать бесплатную программу *Virtual PC* фирмы Microsoft, запускаемую из Windows.

9.3.2. Общие сведения о командах DOS

После запуска DOS любым из перечисленных выше способов на экране появляется ее приглашение, например следующего типа:

```
C:\SIMP>
```

Это приглашение означает, что в текущий момент времени DOS ожидает от вас команды и что она находится в точке файловой структуры – *C:\SIMP*, где *SIMP* – текущий каталог на логическом диске *C:* .

Строка экрана, в которой появилось приглашение, называется ***командной строкой***. В этой строке вы набираете команду:

```
< имя программы >  [< параметры >],
```

где часть команды в квадратных скобках не обязательна.

Имя программы – имя файла, содержащего загрузочный модуль программы (расширение имени файла – *com* или *exe*), или это имя командного файла (расширение – *bat*). Расширение имени файла в команде может отсутствовать.

Набрав команду, вы нажимаете клавишу <Enter>. В результате команда поступает в интерпретатор команд DOS, который поступает следующим образом. Во-первых, он определяет имя программы, которая должна быть выполнена – это последовательность символов, заканчивающаяся первым пробелом. Во-вторых, ИК ищет адрес требуемой программы – определяет, на каком логическом диске находится загрузочный модуль программы, а также ее адрес на физическом диске. С учетом того, что в файловой структуре системы могут находиться несколько файлов с требуемым именем, необходимо четко представлять алгоритм поиска требуемого файла, по которому работает интерпретатор команд DOS:

- Просматривает свои внутренние таблицы. Если при этом имя файла найдено, то запускается соответствующая программа. Иначе – переход к шагу 2.
- Просматривает все файлы текущего каталога.
- Ищет файл во всех каталогах, указанных в команде *PATH* файла *Autoexec.bat*. Например, команда «*path c:\len; c:\auto*» сообщает о том, что программные файлы следует искать в каталогах *LEN* и *AUTO* логического диска *C:* .

Если в своей команде вы не задали расширение имени файла, то в любом из трех перечисленных шагов приоритет поиска следующий:

1) ищется файл с расширением *com*;

2) с расширением *exe*;

3) с расширением *bat*.

Если ни один из трех перечисленных шагов поиска не увенчался успехом, на экран выводится: *Bad command or file name* (имя команды или файла указано неверно) и вновь выдается приглашение DOS.

9.3.3. Некоторые системные команды DOS

Естественно, что перечислить все команды DOS невозможно, так как имя любой программы может рассматриваться как команда. Речь может идти только о перечислении команд, требующих выполнение системных программ – утилит, лингвистических процессоров и драйверов. Соответствующая системная программа может находиться внутри DOS или существовать в виде отдельного *com-* или *exe*-файла. Команды, соответствующие первому типу программ, называются внутренними, а второму – внешними. Вот некоторые из системных команд.

1. **Задание текущего логического диска** (внутренняя команда). Для этого в ответ на приглашение DOS достаточно набрать требуемое имя логического диска, например:

C: или D:

2. **Задание текущего каталога** (внутренняя команда):

CD <имя каталога>

Например, в результате выполнения команды «*CD \SIMP\SET*» текущим каталогом станет дочерний каталог каталога *SIMP – SET*. Текущий логический диск при этом не меняется.

3. **Вывод на экран содержимого каталога** (внутренняя команда):

DIR [имя лог. диска, или имя каталога, или имя файла][/**p**][/**w**] ,

где квадратными скобками выделены необязательные параметры.

Если параметры-имена отсутствуют, то на экран выводится содержимое текущего каталога:

DIR

Если задано имя логического диска, то выводится содержимое корневого каталога на этом диске. Например, следующая команда выводит на экран содержимое корневого каталога на логическом диске *A:*

DIR A:

Если задано имя каталога, то на экран выводится его содержимое. Например, следующая команда выводит на экран содержимое каталога *SIMP*, являющегося дочерним каталогом по отношению к текущему каталогу:

DIR SIMP

Если задано имя файла, то на экран выводятся сведения только об этом файле. При задании имени файла разрешается вместо любой последовательности символов в имени (в том числе и вместо расширения имени) задать символ «*». В этом случае на экран будут выведены сведения обо всех файлах, имеющих в своих именах последовательности символов, заданные в команде. Например, следующая команда выводит сведения обо всех файлах текущего каталога, имеющих расширение *exe*:

DIR *.exe

Если информация в каталоге слишком велика, чтобы уместиться на одном экране, то используют параметр «*/p*» . В этом случае заполнение экрана приводит к приостановке вывода до нажатия вами любой клавиши.

Параметр «*/w*» используется для сжатия выводимой на экран информации за счет того, что для каждого файла выводится лишь имя, а атрибуты (размер, дата и время создания) опускаются. Допускается одновременное применение и параметра «*/p*», и параметра «*/w*».

4. **Создание каталога**:

MD <имя каталога>

5. **Уничтожение каталога**:

RD < имя каталога >

6. **Копирование файла** (внутренняя команда):

COPY <имя файла1> <имя каталога или имя файла2>

Данная команда или создает копию файла с именем «имя файла 1» в заданном каталоге, или создает копию файла в прежнем каталоге, но с новым именем файла «имя файла 2». Например, команда

```
COPY abc.exe c:\simp
```

копирует файл *abc.exe*, расположенный в текущем каталоге текущего логического диска, в файл с таким же простым именем, но расположенный в каталоге *simp* на логическом диске *c:* .

В результате выполнения команды

```
COPY abc.exe 123.exe
```

текущий каталог содержит 2 файла с одинаковым содержанием, но с разными именами.

7. **Удаление файла** (внутренняя команда):

DEL <имя файла> ,

где <имя файла> – имя удаляемого файла.

8. **Переименование файла** (внутренняя команда):

REN <имя файла 1> <имя файла 2> ,

где <имя файла 1> – старое имя файла;
<имя файла 2> – новое имя файла.

9. **Вывод содержимого текстового файла на экран:**

TYPE <имя файла> ,

где <имя файла> – имя текстового файла в коде ASCII.

10. **Создание или корректировка текстового файла:**

EDIT <имя файла>

9.3.4. Командные файлы

Командный файл – файл, содержащий несколько команд DOS. Он создается пользователем как обычный текстовый файл, имеющий любое собственное имя, но обязательно расширение имени – *bat*. При этом каждая команда набирается на отдельной строке экрана. Один командный файл играет особую роль – *Autoexec.bat*. Он запускается автоматически (без нашего участия) DOS в самом начале ее работы. Обычно среди прочих команд этот файл содержит:

1) команду запуска подпрограммы *PATH*;
2) команды запуска драйверов клавиатуры и экрана;
3) команду запуска утилиты DOS Navigator, FAR или другой подобной утилиты.

Пример простого командного файла, выполняющего переход в заданный каталог *\katalog*, вывод текстового файла *1.txt*, содержащегося в этом каталоге, а затем уничтожение этого файла:

```
cd \katalog
type 1.txt
del 1.txt
```

Если командный файл, содержащий перечисленные команды, имеет имя *2.bat*, то для запуска этого файла на выполнение достаточно использовать команду DOS:

```
c:\> 2.bat
```

9.4. Утилита «DOS Navigator»

9.4.1. Представление на экране файловой структуры

Утилита DOS Navigator предназначена для того, чтобы предоставить пользователю DOS удобный интерфейс для общения с этой системой. Это обеспечивается, во-первых, наглядным выводом на экран информации о файловой структуре системы. Во-вторых, DOS Navigator существенно упрощает для пользователя ввод команд DOS.

Для того чтобы запустить DOS Navigator, достаточно набрать команду DOS – *dn*. Часто эту команду включают в файл *Autoexec.bat*, и поэтому она выполняется автоматически. В любом случае в верхней части экрана появляются две серых панели, каждая из которых содержит перечень файлов, расположенный в одном из каталогов файловой структуры системы. При этом в заголовке каждой панели указаны имя логического диска и имя каталога. Ниже панелей располагается командная строка DOS с обычным ее приглашением и мерцающим курсором. В последней строке экрана находится список клавиш <F1> ÷ <F10> с кратким обозначением их функций.

Каталоги, отображаемые на левой и правой панелях, могут совпадать или не совпадать, но в любом случае одна из панелей, называемая ***активной панелью***, отображает текущий каталог. Заголовок активной панели выделяется зеленым цветом. Кроме того, одно из имен файлов на активной панели выделено курсором-маркером. Смена активной панели производится нажатием клавиши <Tab>.

Перемещение курсора-маркера внутри активной панели производится с помощью клавиш управления курсором – ↓, ↑, ←, →. Нажатие клавиши <End> приводит к установке курсора-маркера на последнюю, а <Home> – на первую строку панели. Щелчком левой клавиши мыши можно установить курсор-маркер в любую позицию не только активной, но и соседней панели (происходит смена активной панели).

В общем случае панель содержит строки трех типов:

1) строку «..», обозначающую выход в «родительский каталог» данного каталога;

2) строки с именами подкаталогов данного каталога (выделяются белым цветом);

3) строки с именами файлов данного каталога. При этом цвет строки зависит от типа файла: серый для обычных текстовых файлов; светло-зеленый для исполняемых файлов (расширение имени файла – *com*, *exe* или *bat*); ярко-зеленый для архивных файлов; голубой для скрытых файлов; розовый для текстовых файлов редактора WORD.

В последней строке панели, как правило, записано имя выделенного файла, его длина, дата и время создания или последней модификации. Еще ниже выводится общее чис-

ло свободной памяти (в байтах) на логическом диске, к которому относится рассматриваемый на панели каталог.

Наличие у DOS Navigator своих команд для работы с каталогами и файлами существенно упрощают работу пользователя по управлению ОС. Рассмотрим основные из этих команд.

1. **Смена логического диска, соответствующего панели**, производится одновременным нажатием двух клавиш:

а) для левой панели – <Alt>+<F1>;

б) для правой панели – <Alt>+<F2>.

На экране появится диалоговое окно-меню из имен логических дисков. После этого следует установить курсор-маркер на требуемое имя и нажать клавишу <Enter>.

2. **Переход на подкаталог текущего каталога** (смена текущего каталога + вывод на экран его содержимого) производится установкой на него курсора-маркера и нажатием <Enter>. Для возврата в родительский каталог требуется установить курсор-маркер на «..» и нажать <Enter>. Перемещаясь подобным образом вверх-вниз по файловой структуре логического диска, можно сделать текущим любой каталог на нем.

3. **Создание каталога**. Для этого достаточно установить в заголовке активной панели «родительский» каталог по отношению к вновь создаваемому каталогу, а затем нажать <F7>. На экране появится диалоговое окно с приглашением набрать имя нового каталога. В ответ следует набрать имя каталога прописными или строчными буквами и нажать <Enter>. В результате на активной панели появится имя нового каталога.

4. **Копирование файла**. Для этого требуется установить в заголовке одной из панелей «родительский» каталог по отношению к копируемому файлу, а в заголовке другой панели – «родительский» каталог по отношению к файлу-копии. Далее следует установить курсор-маркер на копируемый файл и нажать клавишу <F5>. На экране появится диалоговое окно с сообщением о готовности выполнить копирование. В ответ достаточно нажать клавишу <Enter>. (Для отмены этой или другой команды следует нажать <Esc>.) В результате на второй панели появится имя скопированного файла.

Для того чтобы создать копию файла в том же каталоге, что и исходный файл, необходимо обеспечить, чтобы старый и новый файлы имели разные имена. Для этого следует в диалоговом окне, появившемся после нажатия <F5>, набрать имя файла-копии, а уж затем нажать <Enter>.

Копирование каталога выполняется аналогично копированию обычного файла данных. При этом копируется все поддерево файловой структуры, начинающееся с данного каталога.

Копирование нескольких «дочерних» файлов (каталогов) текущего каталога можно ускорить, используя ***выделение группы файлов***. Для выделения файла необходимо установить на его имя курсор-маркер и нажать клавишу <Ins>. В результате имя файла высвечивается желтым цветом, и оно включается в группу. (Для исключения файла из группы необходимо повторить эти же два действия – установку курсора-маркера и нажатие <Ins>.) После того, как требуемая группа файлов выделена (в нижней строке панели информация об общем числе выделенных файлов и их общем объеме), ее можно скопировать последовательным нажатием <F5> и <Enter>.

5. **Перенос файла или каталога**. Для этого требуется установить курсор-маркер на имя переносимого файла (каталога) и нажать <F6>. На экране появится диалоговое окно с просьбой подтвердить намерение перенести файл. В ответ достаточно нажать <Enter> (для отмены – <Esc>).

6. **Уничтожение файла или каталога**. Для этого требуется установить курсор-маркер на имя уничтожаемого файла (каталога) и нажать <F8>. На экране появится диалоговое окно с просьбой подтвердить намерение удалить файл. В ответ достаточно нажать <Enter> (для отмены – <Esc>).

Выделив группу файлов (аналогично выделению при копировании), ее можно уничтожить нажатием <F8> и последующими нажатиями <Enter> в ответ на вопросы DOS Navigator.

7. **Переименование файла**. Для этого следует установить курсор-маркер на требуемый файл и нажать <F6>. В появившемся диалоговом окне следует набрать новое имя файла, а затем нажать <Enter>.

8. **Просмотр содержимого текстового файла** осуществляется установкой курсора-маркера на требуемый файл и последующим нажатием <F3>. Если во время просмотра нажать <F8>, то произойдет смена используемой кодировки русских букв с кодировки, предназначенной для DOS на кодировку, предназначенную для Windows (или наоборот).

Если во время просмотра вы нажмете <F4>, то файл будет просматриваться в шестнадцатеричном режиме. Такой режим особенно полезен для просмотра не текстовых, а двоичных файлов.

9. **Создание текстового файла**. Для создания нового файла необходимо одновременно нажать клавиши <Shift>+<F4>. В появившемся на экране окне необходимо набрать имя создаваемого файла и нажать <Enter>. В ответ на последующий вопрос также нажимается эта клавиша. Далее выполняется ввод с клавиатуры содержимого файла, по завершении которого следует нажать клавишу <F2>.

10. **Редактирование текстового файла**. Для работы с ранее созданным текстовым файлом необходимо установить курсор-маркер в открытом каталоге на нужный файл и нажать <F4>. Для того чтобы сохранить этот файл, скопировав его из ОП на диск, необходимо нажать <F2>.

Во время редактирования файла можно выделить любой его фрагмент, одновременно нажав <Ctrl>+<←,→,↓,↑>. Выделенный фрагмент можно скопировать в буфер редактора, нажав <Ctrl>+<C>. Далее скопированный фрагмент можно вставить в любое место данного, или другого файла, на которое указывает курсор. Для этого достаточно нажать <Ctrl>+<V>. Для уничтожения выделенного фрагмента требуется нажать <Ctrl>+<D>.

11. **Прекращение работы** DOS Navigator происходит в результате нажатия нескольких клавиш на клавиатуре и нарисованных на экране:
<F10> → [File] → [Exit] → [Yes]

Заметим, что в результате нажатия <F10> на экран выводится главное меню DOS Navigator, через которое можно осуществить доступ к различным функциям этой программы.

9.4.2. Ввод команд DOS

В отличие от рассмотренных выше операций с файлами, для выполнения которых DOS Navigator предоставляет пользователю свои команды, формат остальных команд DOS остается без изменений. При этом помощь DOS Navigator заключается в ускорении набора этих команд.

1. Крайний случай – пользователь набирает команду в командной строке DOS полностью вручную, не пользуясь помощью DOS Navigator. Такой метод используется для ввода коротких или редко используемых команд.

2. Второй метод – для заполнения командной строки используются имена файлов, высвечиваемые на панелях. Для того чтобы скопировать имя файла с панели в командную строку, достаточно установить курсор-маркер на требуемое имя файла и одновременно нажать две клавиши – <Ctrl>+<Enter>. При этом введенное имя файла можно редактировать точно так же, как и остальную часть командной строки.

3. Третий метод позволяет обойтись вообще без записи в командную строку. Установив курсор-маркер на имени исполняемого файла (расширение имени – *com*, *exe* или *bat*), следует нажать <Enter>. Данный метод обычно применяется тогда, когда команда не имеет параметров.

4. Четвертый метод заключается в том, что команда DOS переписывается в командную строку из протокола команд. ***Протокол команд*** – список последних команд, сохраненный в DOS Navigator.

Поиск нужной команды в протоколе может быть выполнен двумя способами. В первом способе на экран выводится весь список (меню) команд. Для этого достаточно нажать комбинацию клавиш – <Alt>+<F8>. Установив курсор-маркер на требуемую команду в меню, следует нажать <Enter>.

Во втором способе список команд в протоколе просматривается последовательно, команда за командой. Для перехода к первой из них (самой новой) требуется нажать комбинацию <Ctrl>+<E>. При этом данная команда будет скопирована в командную строку, где она может быть скорректирована до нажатия <Enter>. Если данная команда не устраивает, то вместо нажатия <Enter> следует опять нажать <Ctrl>+<E>. В результате в командной строке окажется следующая команда из протокола. Для передвижения по протоколу на одну команду назад следует нажать комбинацию клавиш <Ctrl>+<X>.

Если программа, запускаемая любым способом из DOS Navigator, выводит какие-то данные на экран, то чтение их будет невозможно из-за присутствия на экране панелей. Для того чтобы убрать с экрана обе панели, требуется одновременно нажать <Ctrl>+<O>. Для восстановления панелей достаточно опять нажать <Ctrl>+<O>. Удаление (восстановление) только левой панели производится одновременным нажатием <Ctrl>+<F1>, а правой – <Ctrl>+<F2>.

Лабораторная работа 7

Лабораторная работа состоит из следующих двух этапов.

1. Пользуясь командами DOS, выполнить следующие шаги.

Ш а г 1. Создать каталог *PROBA* в корневом каталоге логического диска *c*:.

Ш а г 2. Создать дерево каталогов с корнем *PROBA*. Структура этого дерева изображена на рис. 2.4.

Ш а г 3. В каталоге 133 создать текстовый файл, содержащий несколько слов русского или английского языка.

Ш а г 4. Переименовать файл, полученный на предыдущем шаге.

Ш а г 5. Одной командой выполнить копирование ранее полученного файла в каталог 3.

Ш а г 6. Вывести на экран содержимое текстового файла, полученного на шаге 5.

Ш а г 7. Выполнить уничтожение обоих текстовых файлов.

Ш а г 8. Поместить в каталог 2, а затем выполнить командный файл, содержащий команды, перечисленные в шагах 3–7.

Ш а г 9. Выполнить уничтожение всех созданных файлов и каталогов.

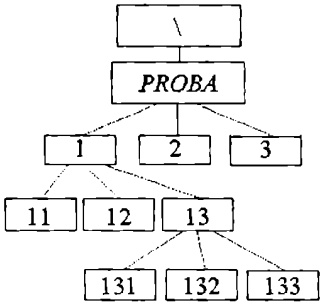

Рис. 2.4. Структура дерева каталогов

2. Выполнить перечисленные выше шаги 1–9, используя команды DOS Navigator.

☑ **Примечание**. Защита лабораторной работы производится путем демонстрации выполнения (наизусть) перечисленных выше этапов. По указанию преподавателя выполняются или все шаги обоих этапов, или часть этих шагов.

10. Простые программы на ассемблере

10.1. Общая структура простых ассемблерных программ

Существуют несколько языков ассемблера (и соответствующих трансляторов), ориентированных на получение машинных программ для *Intel*-процессоров. Из них нами выбран язык ассемблера и одноименный транслятор *NASM*.

Основным отличием программы на любом языке ассемблера от соответствующей машинной программы, которую мы вводим в ЭВМ с помощью Debug, является наличие псевдооператоров. В отличие от исполнительного оператора, который преобразуется транслятором-ассемблером в одну машинную команду, псевдооператор ни в какие машинные команды не транслируется. Такой оператор представляет собой указание транслятору и нигде, кроме самого транслятора, не используется.

Общая структура простых ассемблерных программ, которая будет для нас достаточна на ближайшее время, имеет вид

```
org   100h
. . . . . .
mov   ax,4c00h
int   21h
```

Посмотрим внимательно на данную структуру. По горизонтали она разделена на две части. Левая (пока пустая) часть предназначена для записи идентификаторов (меток) программных объектов, к которым относятся виртуальные сегменты, процедуры, исполнительные операторы и элементы данных. В правой части листинга находятся псевдооператоры и исполнительные операторы, а также операнды этих операторов. Идентификатор отделяется от соответствующего оператора минимум одним пробелом.

Простейшая программа состоит всего из одного виртуального сегмента кода, в котором находятся команды программы. При этом псевдооператор «*org 100h*» сообщает транслятору о том, что самый первый исполнительный оператор нашей программы должен быть помещен в выделенный программе сегмент ОП со смещением *100h* относи-

тельно начала сегмента. Мы и раньше использовали это смещение, вводя машинные программы с помощью Debug. Следует обратить внимание на символ *h* после шестнадцатеричного числа 100. Использование этого символа после шестнадцатеричных чисел обязательно. Так как транслятор-ассемблер в отличие от Debug «обычной» считает не шестнадцатеричную, а десятичную систему счисления.

В конце программы вместо хорошо знакомого нам исполнительного оператора «*int 20h*» мы будем записывать два оператора «*mov ax,4c00h*» и «*int 21h*». Функция этих двух операторов та же самая – обеспечение возврата из программы туда, откуда она была запущена. Но их применение позволяет не только вернуть управление в главную программу, но и передать в нее код завершения. Это имеет существенное значение при запуске более сложных программ.

10.2. Пример программы на ассемблере

В качестве примера запишем на ассемблере ту программу *Writestr*, которую мы создали в п. 4.3 с помощью Debug. Напомним текст машинной программы:

```
100  mov  ah,02
102  mov  dl,2a
104  int 21
106  int 20
```

Соответствующий текст программы на ассемблере:

```
org   100h
mov   ah, 2h
mov   dl, 2ah
int   21h
mov   ax, 4c00h
int   21h
```

Следует отметить, что если в программе есть шестнадцатеричные числа типа *ACh*, то чтобы транслятор не запутался с ними (он может принять их за имена), всякое шестнадцатеричное число, начинающееся с буквы, следует предварять нулем. Например: *0ACh*.

10.3. Подготовка программы к выполнению

Текст программы на ассемблере или на любом другом языке программирования называется ***исходной программой***. Для того чтобы преобразовать этот текст в машинную программу, нам нужна помощь не только со стороны транслятора, но и от некоторых других системных программ.

Во-первых, с помощью текстового редактора мы получаем исходный файл программы. Имя исходного файла должно иметь расширение *asm*. Можно использовать любой редактор, который выдает результирующий текст в коде ASCII. Примерами такого редактора является *Edit*, запускаемый из командной строки DOS, или текстовый редактор в DOS Navigator. *С о з д а й т е* исходный файл *Writestr.asm*, поместив в него приведенную выше программу на ассемблере. Убедитесь, что это именно ASCII-файл. Для этого, находясь в DOS, напечатайте:

```
C:\ > TYPE  Writestr.asm
```

Вы должны увидеть тот же текст, который ввели в текстовом редакторе. Если вы увидите в вашей программе странные символы, то для ввода текста программ следует

использовать другой текстовый редактор. Теперь давайте начнем ассемблировать программу *Writestr*:

```
C:\ > nasm  Writestr.asm –o Writestr.com
```

Ответного сообщения транслятора в случае успешной трансляции не будет:

```
C:\ >
```

В результате транслятор-ассемблер создал файл, называющийся *Writestr.com*, который вы найдете на диске. Конечная часть команды «*-o Writestr.com*» используется для задания имени результирующему (целевому) файлу (*Writestr.com*). При отсутствии этой части результирующий файл получит имя *Writestr*.

Напечатайте «*Writestr.com*», чтобы запустить *com*-файл и убедитесь, что ваша программа функционирует правильно (напоминаем, что она должна вывести звездочку на экран).

Теперь введем созданный *com*-файл в Debug и разассемблируем его, чтобы увидеть получившуюся машинную программу:

```
C:\ > Debug   Writestr.com
_U
1593:0100  B402      MOV   AH, 02
1593:0102  B22A      MOV   DL, 2A
1593:0104  CD21      INT   21
1593:0106  B8004C    MOV   AX,4C00
1593:0109  CD21      INT   21
```

Получили именно то, что и ожидалось получить.

10.4. Комментарии

Программа на ассемблере может содержать любые сообщения, информирующие программиста о содержании текста программы. Такое сообщение называется ***комментарием***. Комментарии могут находиться на любых строках программы. На каждой строке, где есть комментарий, ему предшествует точка с запятой (;). Подобно псевдооператорам комментарии не транслируются ни в какие машинные команды. Но в отличие от псевдооператоров они не интересуют и сам транслятор, существуя лишь для человека, который читает программу.

Добавим комментарии в записанную ранее программу на ассемблере:

```
org  100h

;
;       Вывод звездочки на экран
;       ------------------------
;

        mov   ah,2h     ; Функция вывода символа
        mov   dl,2ah    ; Символ * в DL
        int   21h       ; Вывод символа
        mov   ax,4c00   ; В AH функция (4Ch), а в AL – подфункция (00h)
        int   21h       ; Возврат в DOS
```

Теперь можно легко понять смысл программы. Обратите внимание на начальные комментарии. Каждая программа или подпрограмма (процедура) должна обязательно иметь вводные комментарии, которые содержат:

1) Словесное описание выполняемых программой (процедурой) функций.

2) Перечень и способ передачи каждого входного и выходного параметра процедуры.

3) Перечень процедур, вызываемых в данной программе (процедуре).

4) Перечень переменных (областей памяти), которые используются процедурой, с указанием для каждой переменной способа ее использования. Допустимые способы использования: чтение; запись; чтение-запись.

Для приведенной выше простой программы вводный комментарий содержит лишь описание выполняемых программой функций.

Что касается текущих комментариев, поясняющих назначение отдельных операторов программы и их групп, то к ним нет жестких требований. Желательно, чтобы были прокомментированы основные управляющие структуры программы (цепочки, ветвления и циклы). Кроме того, должны быть прокомментированы вызовы подпрограмм, т. е. операторы *call* и *int*. Каждый такой оператор инициирует десятки или сотни машинных команд и поэтому заслуживает пояснения.

Следует помнить, что исходная программа без комментариев не имеет какой-либо коммерческой ценности. Это просто-напросто черновик автора программы. Более того, по истечении нескольких месяцев даже автору требуются значительные усилия на то, чтобы понять смысл своей программы.

10.5. Метки

При построении нами машинных программ с помощью Debug команды переходов содержали адреса тех команд программы, на которые переход делался. Допустим, что мы решили добавить в программу новые команды. В этом случае от нас требуется изменить многие ранее записанные адреса переходов, что чрезвычайно неудобно. При написании программы на ассемблере вместо адресов перехода мы используем метки тех операторов, на которые делается переход. То же самое относится и к процедурам: первые операторы процедур мы помечаем метками, которые теперь являются именами соответствующих процедур и используются в операторах вызова этих процедур. Рассмотрим правило записи меток.

Максимальная длина метки – 31 символ. Это могут быть следующие символы:

1) латинские буквы – от *A* до *Z* и от *a* до *z*;

2) цифры – от 0 до 9;

3) специальные символы – знак вопроса «?»; знак «@»; точка «.» ; подчеркивание «_»; доллар «*$*».

Первым символом в метке должна быть буква или специальный символ. Точка может использоваться в метке только в качестве первого символа. В отличие от операционных систем Windows, DOS, а также многих других программ, транслятор-ассемблер *NASM* различает заглавные и строчные буквы в именах меток. Для него это совершенно разные символы. Примеры меток: *count*, *Page25*, *$E10*. Весьма желательно, чтобы метки поясняли смысл программы, а не выбирались отвлеченно.

Метки бывают локальными и глобальными. ***Глобальная метка*** может использоваться во всем исходном файле. Имя такой метки не может начинаться с точки. Имя ***локальной метки*** начинается с точки. Областью действия такой метки является фрагмент исходной программы, заключенный между двумя ближайшими глобальными метками. Вне этого фрагмента может быть задана другая локальная метка с точно таким же именем.

Особенно полезно применение локальных меток внутри процедур. Дело в том, что в разных процедурах часто используются идентичные метки. В процессе сборки исходной программы транслятором (такая сборка производится при обработке рассматриваемого позже псевдооператора *%include*) он выведет сообщения об ошибках только в том случае, если

идентичные метки являются глобальными. Если эти метки оформлены как локальные, то никаких ошибок не будет, так как это совершенно различные метки. Заметим, что использование идентичных меток внутри процедур очень удобно для программиста, позволяя присваивать схожим фрагментам процедур одинаковые имена. Примером является использование метки «*.Exit*» для заключительного фрагмента процедуры.

Кроме того, удобно давать локальные имена тем данным, которые используются только внутри конкретной процедуры. Например, к таким данным относятся тексты сообщений, выводимых пользователю операторами процедуры. В тексте программы мы будем размещать такие локальные данные процедуры сразу же за ее последним исполнительным оператором *ret*.

Некоторые из буквенных слов зарезервированы транслятором-ассемблером и не могут быть использованы в качестве меток. Сюда относятся имена регистров (например, *AX*), мнемоники исполнительных команд (например, *add*), указатель на точку входа (*..start*), и псевдооператоры (например, «*segment*»).

10.6. Еще один пример программы

Ранее в п. 7.4 мы создали небольшую программу, которая выводила на экран буквы от *A* до *J*. Вот эта машинная программа:

```
100   mov  dl,41
102   mov  cx,000a
105   call 0200
108   loop 0105
10A   int  20
200   mov  ah,02
202   int  21
204   inc  dl
206   ret
```

Перепишем теперь эту программу на языке ассемблера:

```
    org   100h
;
;   Вывод 10 символов от A до J
;   ---------------------------
;   Вызовы:   Write_char
;
Start:
        mov     dl,'A'          ; В DL код буквы A
        mov     cx,10           ; В счетчике символов -10
.Loop:  call    Write_char      ; Вывод символа
        loop    .Loop           ; Переход к следующему символу
        mov     ax, 4c00h       ; Функция и подфункция завершения
        int     21h             ; Возврат в DOS
;
;       Вывод символа на экран и увеличение кода символа на 1
;       -----------------------------------------------------
;   Входы:    DL содержит код символа
;   Выходы:   DL содержит код нового символа
```

```
; Вывод символа на экран (в DL)
Write_char:
        push    ax
        mov     ah,02   ; Функция вывода символа
        int     21h     ; Вывод символа на экран
        inc     dl      ; Код следующего символа
        pop     ax
        ret             ; Возврат из процедуры
```

Обратите внимание, что в первом операторе программы второй операнд представляет собой символ, заключенный в кавычки (кавычки могут быть как одиночными, так и двойными). Это означает, что операнд представляет собой код ASCII-символа, заключенного в кавычки.

Отметим попутно еще одну весьма полезную деталь ассемблера. Данный язык позволяет записывать в качестве операнда арифметическое выражение, элементами которого являются константы. Например, оператор «*mov dl,'A'-10*» помещает в регистр *DL* код ASCII-символа «*A*», уменьшенный на 10.

В в е д и т е эту программу в файл *Printaj.asm* и получите файл *Printaj.com*. Затем выполните программу. Добившись правильной ее работы, используйте Debug для разассемблирования программы. Посмотрите, как транслятор распределил память для основной программы и для процедуры. Если мы раньше перестраховывались, располагая процедуру подальше (по адресу *200h*), то теперь транслятор расходует память экономно, не оставляя промежутков между процедурами.

10.7. Вывод на экран двузначного шестнадцатеричного числа

В гл. 6 подобная программа была разработана нами с помощью Debug. Теперь мы запишем ее на языке ассемблера, добавив небольшую процедуру, которая выводит один символ на экран. Выделение этой, на первый взгляд, лишней процедуры, обусловлено желанием в дальнейшем вносить изменения в операцию вывода символа, совершенно не влияя на остальную часть программы.

На рис. 2.5 приведено ***дерево подпрограмм*** для данной программы. Эта структура показывает, какие подпрограммы (процедуры) входят в состав программы и как эти процедуры связаны между собой по управлению. При этом самой верхней (главной процедурой) является *Test_write_byte_hex*. Данная процедура предназначена для тестирования процедуры *Write_byte_hex*, выполняющей вывод на экран двузначного шестнадцатеричного числа, содержащегося в байте.

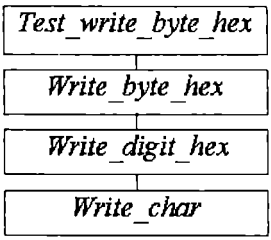

Рис. 2.5. Дерево подпрограмм для программы вывода двузначного шестнадцатеричного числа

В свою очередь, процедура *Write_byte_hex* вызывает (инициирует) процедуру *Write_digit_hex*, выполняющую вывод на экран шестнадцатеричной цифры. В ходе своей работы данная процедура вызывает процедуру *Write_char*, выполняющую вывод символа на экран.

В гл. 6 приведены блок-схемы для процедур *Write_byte_hex* и *Write_digit_hex*. В следующей программе тексты этих процедур опущены:

```
        org100h
;
;       Тестирование процедуры  Write_byte_hex
;       ---------------------------------------
;  Вызовы:  Write_byte_hex
;
Test_write_byte_hex:
        mov     dl,3fh                  ; Тестировать с 3Fh
        call    Write_byte_hex          ; Вывод числа
        mov     ax, 4c00h
        int     21h                     ; Возврат в  DOS
;
;       Вывод  двузначного шестнадцатеричного числа
;       -------------------------------------------
;  Входы:  DL  содержит выводимое число
;  Вызовы:   Write_digit_hex
;
Write_byte_hex:
        . .                             ; Тело процедуры Write_byte_hex
      ret
;
;       Вывод шестнадцатеричной цифры
;       -----------------------------
;  Входы: BL содержит шестнадцатеричную цифру
;  Вызовы: Write_char
Write_digit_hex:
        . .                             ; Тело процедуры Write_digit_hex
        ret
;
;               Вывод символа на экран
;               ----------------------
;  Входы:  DL  содержит код символа, выводимый на экран
;
Write_char:
        push    ax
        mov     ah,2                    ; Функция вывода символа
        int     21h                     ; Вывод символа на экран
        pop     ax
        ret
```

Лабораторная работа 8

Напишите на ассемблере и поместите в файл *Video_io.asm* программу вывода двузначного шестнадцатеричного числа. Все процедуры исходной программы должны содержать вводные и текущие комментарии.

Получите файл *Video_io.com* и, используя Debug, тщательно протестируйте программу, меняя *3Fh* по адресу *101h* на каждое из граничных условий, которые использовались ранее в гл. 6 для проверки программы, выполняющей те же функции.

Обязательно сохраните файл *Video_io.asm*. На последующих работах мы будем добавлять в него новые процедуры. Пользуясь этими процедурами как «кирпичиками», мы сможем создавать достаточно сложные программы.

☑ **Примечание.** Для отладки вашей программы удобно использовать подход, называемый «снизу-вверх». Согласно ему сначала отлаживаются процедуры, расположенные внизу дерева подпрограмм. После того, как эти процедуры отлажены, отлаживаются процедуры, их вызывающие, и т. д.

Реализация данного подхода предполагает первоначальную корректировку процедуры *Test_write_byte_hex* так, чтобы из нее вызывалась не процедура *Write_byte_hex*, а процедура *Write_digit_hex*. После того как *Write_digit_hex* будет отлажена, *Test_write_byte_hex* восстанавливается, и программа отлаживается целиком.

11. Вывод на экран десятичных и шестнадцатеричных чисел

В процессе разработки практически важных процедур, выполняющих вывод на экран десятичных и шестнадцатеричных чисел, производится изучение нескольких новых исполнительных операторов ассемблера, расширяются навыки работы со стеком, а также производится знакомство с представлением исходной программы в виде нескольких файлов.

11.1. Получение алгоритма

Назовем процедуру, выполняющую вывод на экран десятичного представления слова данных, содержащего двоичное число без знака, как *Write_word_dec*. Пусть это слово данных передается на вход процедуры в регистре *DX*. Обсудим алгоритм процедуры *Write_word_dec*.

Предполагаемый результат работы процедуры состоит в том, что сначала на экран будет выведена старшая десятичная цифра числа, затем вторая слева цифра и т. д. Для этого следует иметь десятичное представление числа, хранящегося у нас в двоичном виде. Вспомним, что при делении числа на 10 остаток есть младшая десятичная цифра числа. Если полученное частное разделим на 10, то получим вторую справа десятичную цифру. Подобное деление можно продолжать до тех пор, пока очередное частное не будет меньше 10 и, следовательно, оно и будет равно старшей цифре числа. Если бы порядок получения десятичных цифр из числа совпал бы с порядком их вывода на экран, то задача была бы уже решена. Но у нас эти порядки противоположны. Поэтому мы опять обратимся за помощью к стеку. Вспомним его свойство «кто пришел первым, тот уйдет

последним». Поэтому, если мы поместим в стек младшую десятичную цифру числа (она получена первой), то она будет извлечена из стека для вывода на экран последней. Блок-схема соответствующего алгоритма приведена на рис. 2.6.

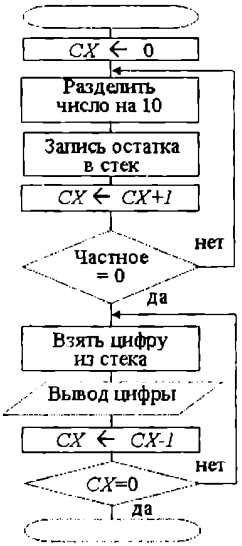

Рис. 2.6. Алгоритм вывода десятичного числа: *CX* – счетчик десятичных цифр в числе

11.2. Дерево подпрограмм

Кроме самой процедуры *Write_word_dec*, выполняющей вывод на экран десятичного представления слова данных, наша программа включает и другие процедуры (рис. 2.7). Во-первых, это главная подпрограмма *Test_write_word_dec*, используемая для тестирования *Write_word_dec*. Во-вторых, это процедура вывода шестнадцатеричной цифры *Write_digit_hex* (она пригодна и для вывода десятичной цифры), а также процедура вывода символа *Write_char*.

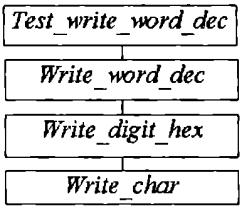

Рис. 2.7. Дерево подпрограмм для программы вывода десятичного числа

Главная подпрограмма *Test_write_word_dec* тестирует процедуру *Write_word_dec* с помощью числа 12345 (которое транслятор переводит в слово 3039h):

```
Test_write_word_dec:
        mov     dx,12345
        call    Write_word_dec
        mov     ax, 4c00h
        int     21h                 ; Возврат в DOS
```

11.3. Запись на ассемблере

При кодировании процедуры *Write_word_dec* можно использовать два следующих приема программирования. Хотя выгода от их применения небольшая, они широко используются на практике.

Целью первого приема является обнуление ячейки (слова или байта). Для этого используется логическая команда ***xor*** – «исключающее ИЛИ». Например, команда «*xor ax,ax*» обнуляет регистр *AX*. Она сравнивает оба операнда побитно, и если один из битов равен 1, то и в соответствующий бит результата записывается 1. Если оба сравниваемых бита содержат 0 или оба содержат 1, то в результирующий бит записывается 0. Например:

```
      1011 0101
XOR   1011 0101
      0000 0000
```

Второй прием используется для того, чтобы проверить на равенство *0* слово или байт. Вместо команды «*cmp ax*,0» можно записать команду «*or ax,ax*», используя далее или команду условного перехода ***jne*** (от «*jump if not equal*» – перейти, если не равно) или ***jnz*** (перейти, если не нуль).

Логическая команда ***or*** («*ИЛИ*») побитно сравнивает оба операнда и записывает 1 в бит результата тогда, когда хотя бы в одном из двух сравниваемых битов есть 1. Данная команда устанавливает флаг нуля только тогда, когда все биты результата содержат 0.

Операция *OR*, выполненная по отношению к одному и тому же числу, дает в результате это же число:

```
      1011  0101
OR    1011  0101
      1011  0101
```

Команда *or* также полезна при установке одного бита в 1. Например, мы можем установить бит 3:

```
      1011  0101
OR    0000  1000
      1011  1101
```

В н е с и т е изменения в полученный на предыдущей работе исходный файл *Video_io.asm*. Во-первых, с помощью текстового редактора удалите процедуру *Test_write_byte_hex* и на ее место запишите тестовую процедуру *Test_write_word_dec*.

Во-вторых, получите текст на ассемблере процедуры *Write_word_dec*, выполняющей вывод на экран содержимого 2-байтового регистра *DX* в виде десятичного числа. При этом для кодирования этапов алгоритма «*CX* ← 0» и «Частное = 0» следует использовать описанные выше два приема. Процедуру *Write_word_dec* добавьте в конец файла *Video_io.asm*.

Внеся изменения, *п р о д е л а й т е* с *Video_io.asm* все необходимые шаги, чтобы получить *com*-файл. После этого протестируйте программу. В случае ошибки проверьте исходный файл. Если это не поможет, то для поиска ошибки используйте Debug.

Выполняя тестирование, будьте осторожны при проверке граничных условий. Первое граничное условие 0 не представляет трудности. Другое граничное условие – 65535 (*FFFFh*), его вам лучше проверять с помощью Debug. *З а г р у з и т е Video_io.com* в Debug, напечатав «Debug *Video_io.com*», и замените 12345 (*3039h*) по адресу *101h* и *102h* на 65535 (*FFFFh*). Напомним, что *Write_word_dec* работает с беззнаковыми числами.

11.4. Многофайловая исходная программа

Для выполнения тестирования процедуры *Write_word_dec* нам пришлось одну тестовую процедуру в файле *Video_io.asm* заменить другой, которая после завершения тестирования также больше не нужна. Очень неудобно вносить изменения в файл с готовыми (или почти готовыми процедурами) с целью добавления в него процедур временного назначения. Выход заключается в размещении тестирующей процедуры в отдельном исходном файле. Когда данная процедура становится ненужной, соответствующий файл уничтожается. Другой причиной размещения исходной программы в нескольких файлах является большой объем программы.

Далее будем понимать под ***файловой структурой исходной программы*** перечень файлов, содержащих текст программы на языке программирования. А также перечень процедур, входящих в состав каждого файла. Кроме процедур файлы могут содержать ***переменные*** – области данных, которым в исходной программе присвоены символьные имена. Примеры применения переменных встретятся нам в последующих работах.

Следует отметить, что в отличие от дерева подпрограмм файловая структура не влияет на машинную программу. Данная структура создается для исходной программы с целью обеспечения удобства в программировании. Нескольким программам, имеющим разные файловые структуры, может соответствовать одна и та же результирующая машинная программа.

При выполнении данной работы мы добавили в файл *Video_io.asm* короткую тестовую процедуру *Test_write_word_dec*. Теперь уберем ее оттуда и поместим в собственный отдельный файл *Test.asm*. В результате получим файловую структуру программы, приведенную на рис. 2.8.

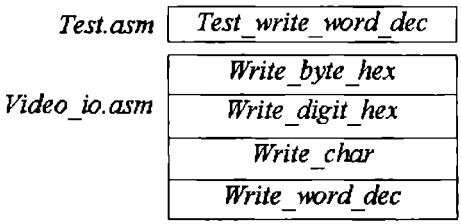

Рис. 2.8. Файловая структура исходной программы

Для того чтобы транслятор-ассемблер объединил несколько исходных файлов в единый текст исходной программы, каждый «родительский» исходный файл должен содержать псевдооператоры ***%include*** («включить»), в качестве операндов которых записаны имена «дочерних» файлов, заключенные в кавычки. Встретив оператор *%include*, транслятор заменяет его на текст «дочернего» исходного файла, указанного в этом операторе.

Рассмотрим эти псевдооператоры на примере файла *Test.asm*:

```
        org         100h
Test_write_word_dec:
        mov         dx,12345
        call        Write_word_dec
        mov         ax, 4c00h
        int         21h                 ; Возврат в DOS
        %include    'Video_io.asm'      ; Включение файла Video_io.asm
```

З а п и ш и т е файл *Test.asm* на диск и внесите в файл *Video_io.asm* такие изменения:

1) удалите из *Video_io.asm* процедуру *Test_write_word_dec*, так как мы поместили ее в файл *Test.asm*;

2) удалите из *Video_io.asm* выражение «*org 100h*», так как мы перенесли его в файл *Test.asm*, который теперь содержит главную процедуру программы.

Лабораторная работа 9

Требуется написать и отладить две различные процедуры вывода на экран шестнадцатеричного числа, содержащегося в одном слове (в двух байтах). Первая из этих процедур использует тот же алгоритм, что и приведенная выше процедура вывода десятичных чисел.

Вторая процедура *Write_word_hex* гораздо короче. Она выводит шестнадцатеричное содержимое слова путем двух вызовов процедуры *Write_byte_hex*, полученной на предыдущей лабораторной работе. При первом вызове на экран выводится шестнадцатеричное содержимое старшего байта слова, а при втором – младшего байта. Данная процедура потребуется нам на последующих лабораторных работах для вывода на экран 4-значных адресов памяти. При этом в отличие от первой процедуры каждый адрес будет отображен на экране в виде ровно четырех символов.

Для обеих разрабатываемых процедур требуется, чтобы тестирующая программа и сама процедура находились в разных исходных файлах (*Test.asm* и *Video_io.asm*).

12. Дампирование памяти

Перейдем к разработке более сложной программы на ассемблере, выполняющей дампирование (вывод на экран) содержимого памяти подобно тому, как это делает Debug при выполнении команды *D*. В процессе этой разработки производится изучение новых приемов программирования на ассемблере, к которым относятся применение относительной регистровой адресации для обработки последовательности однотипных величин, а также первоначальное знакомство с системными программами BIOS.

12.1. Дампирование 16 байт

Разработку программы, выполняющей дампирование, целесообразно начать с создания процедуры, выполняющей вывод на экран содержимого 16 байт. Именно столько информации удобно разместить на одной строке экрана. Назовем данную процедуру *Disp_line*. Для каждого из 16 байт она выводит на экран соответствующее шестнадцатеричное представление, разделяя два соседних числа пробелами.

Ниже приведен файл *Disp_sec.asm*, который содержит начальную версию процедуры *Disp_line*:

```
        org  100h
        jmp     Disp_line
;       Данные программы
Sector  db      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h   ; Образец
        db      18h, 19h, 1Ah, 1Bh,1Ch,1Dh,1Eh,1Fh    ;  текста
; ---------------------------------------------------
```

```
;   Процедура дампирует 16 байт памяти в одну строку
;                   шестнадцатеричных чисел
;       ---------------------------------------------------------
; Вызовы:  Write_hex,  Write_char
; Чтение:  Sector
;
Disp_line:
        xor     bx,bx                   ; Обнуление BX
        mov     cx,16                   ; Счетчик байтов
.M:     mov  dl, [Sector+bx]            ; Получить 1 байт
        call    Write_byte_hex          ; Вывод шестнадцатеричного числа
        mov     DL, ' '                 ; Вывод на экран
        call    Write_char              ;  пробела
        inc     bx                      ; Возврат за следующим
        loop    .M                      ;  байтом
        mov     ax, 4c00h
        int     21h                     ; Возврат в DOS
        %include 'Video_io.asm'         ; Подсоединение процедур
```

В начале программы находится оператор безусловного перехода, осуществляющий переход через область данных, расположенную после оператора перехода, на начало области исполнительных операторов программы. Для исполнительных операторов новым является использование относительной регистровой адресации в операторе:

```
.M:     mov  dl, [Sector+bx]
```

Байт программы, отмеченный меткой *Sector*, имеет смещение *259 (100h+3)* относительно самого первого байта сегмента памяти, выделенного программе (команда *jmp* имеет длину три байта). Тогда при выполнении на ЦП машинной команды *mov*, соответствующей записанному оператору, реальный адрес второго операнда будет определен по формуле

R = (DS) * 16 + 259 + (BX).

Задав нулевое содержимое регистра *BX*: (*BX*) = 0, мы получим адрес байта с меткой *Sector*, в который транслятором было помещено число *10h*. Меняя содержимое регистра *BX* от 0 до 15, мы можем с помощью приведенного выше оператора *mov* записать в регистр *DL* содержимое любого из 16 байт, проинициализированных по нашей просьбе транслятором.

З а п и ш и т е файл *Disp_sec.asm*, а затем получите файл *Disp_sec.com*. Если после запуска программы вы не увидите:

10 11 12 13 14 15 16 17 18 19 IA 1B 1C 1D 1E 1F,

то вернитесь назад и найдите ошибку.

12.2. Дампирование 256 байт памяти

После того как мы сделали первую версию процедуры вывода на экран шестнадцатеричного представления 16 байт памяти, перейдем к изготовлению программы, выполняющей дамп 256 байт памяти.

Число 256 обусловлено следующим. Во-первых, дальнейшей целью наших лабораторных работ является создание программы, позволяющей выполнять редактирование текстовой информации, находящейся в сегменте памяти. Во-вторых, объем сегмента равен 65536 байт и 256 есть наибольшее число, которое одновременно отвечает двум требованиям: 1) является делителем числа 65536; 2) содержимое 256 байт одновременно умещается на экране. Далее будем называть такой объем памяти ***сектором***. Поэтому назовем процедуру, выполняющую дамп 256 байт, как *Disp_sector*.

До сих пор мы делали вывод символов, которые размещались на одной строке экрана. При выполнении дампа 256 байт нам потребуются 16 строк. Поэтому потребуется выполнять переход с одной строки экрана на следующую строку. Для осуществления такого перехода достаточно «вывести» на экран два управляющих символа кода ASCII. Первый символ имеет код *0Dh* и называется «возврат каретки». В результате его «вывода» последующий вывод на экран начнется с самой левой позиции строки. Второй управляющий символ имеет код *0Ah* и называется «перевод строки». Его «вывод» и реализует перевод строки экрана.

Назовем процедуру, выполняющую перевод строки как *Send_crlf (crlf* означает «*Carriage Return —Line Feed*» – «Возврат каретки – Перевод строки»). Текст файла *Cursor.asm*, содержащего эту процедуру:

```
cr   equ  13    ; Возврат каретки
lf   equ  10    ; Перевод строки
;
;          Перевод строки экрана
;          ---------------------
;
Send_crlf:
      push   ax
      push   dx
      mov    ah,2       ; Функция вывода
      mov    dl,cr      ; Выводимый символ
      int    21h        ; Вывод символа
      mov    dl,lf
      int    21h        ; –"–
      pop    dx
      pop    ax
      ret
```

Псевдооператор ***equ*** указывает транслятору, что имя *cr* эквивалентно числу 13, а имя *lf* эквивалентно числу 10. Поэтому везде в программе, где транслятор встретит данное имя, он заменит его указанным числом. Применение данного псевдооператора позволяет программисту записывать в программе вместо чисел более удобные их символьные обозначения. Например, вместо 13 записывать *cr*.

На рис. 2.9 приведены дерево подпрограмм и файловая структура программы вывода на экран 256 байт из переменной *Sector*. Переменная *Address* изображена пунктиром, так как в первой версии программы она не используется.

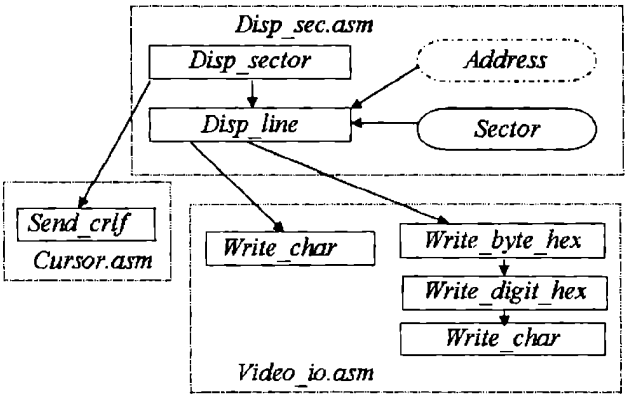

Рис. 2.9. Дерево подпрограмм и файловая структура программы вывода на экран сектора

В в е д и т е файл *Cursor.asm* и скорректируйте файл *Disp_sec.asm* в соответствии с его новым вариантом:

```
        org     100h
        jmp     Disp_sector
;   Данные программы
Sector  times   16    db  10h       ; Первая строка байтов
        times   16    db  11h
        . . . . . . . . . . . . . . . . . . . .
        times   16    db  1Fh       ; Последняя строка байтов
;-----------------------------------------------------------
;         Отображает на экран сектор (256 байт)
;         -------------------------------------
;   Вызовы:   Disp_line,  Send_crlf
;
Disp_sector:
        xor     dx, dx              ; Начало  Sector
        mov     cx,16               ; Число строк 16
.M:     call    Disp_line           ; Вывод строки
        call    Send_crlf           ; Перевод строки
        add     dx,16               ; Номер следующего байта
        loop    .M                  ; Проверка числа строк
        mov     ax, 4c00h
        int     21h                 ; Возврат в DOS
;
; Процедура дампирует 16 байт памяти в одну строку шестнадцат. чисел
; ------------------------------------------------------------------
; Входы: DX – номер первого байта строки в  Sector
; Вызовы :   Write_char,   Write_hex
; Читается :  Sector
;
Disp_line:
        push    bx
        push    cx
        push    dx
```

```
        mov     bx,dx           ; В BX номер первого байта
        mov     cx, 16          ; Счетчик байтов
.M:     mov     dl, [Sector+bx] ; Получить 1 байт
        call    Write_byte_hex  ; Вывод шестнад. числа
        mov     dl, ' '         ; Вывод на экран
        call    Write_char      ;  пробела
        inc     bx              ; Возврат за
        loop    .M              ;  следующим байтом
        pop     dx
        pop     cx
        pop     bx
        ret
        %include 'Video_io.asm' ; Подсоединение процедур
        %include 'Cursor.asm'   ;  —"—
```

Обратите внимание на изменения, которые мы внесли в процедуру *Disp_line*. Теперь она имеет входной параметр, передаваемый в регистре *DX*. Это номер байта в поле *Sector*, начиная с которого следует вывести на экран 16 байт. Есть и другие изменения в данной процедуре, не требующие пояснений.

Выполните ассемблирование файла *Disp_sec*. Выполнив программу, вы должны получить изображение на экране:

```
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
                        .  .  .
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F
```

12.3. Очистка экрана

Прежде чем вывести на экран сектор, наша программа должна позаботиться об очистке экрана. Для выполнения этой операции мы обратимся за помощью к системной программе, называемой BIOS.

Операционная система, например DOS, использует подпрограммы BIOS для выполнения своих операций обмена с внешними устройствами. Разработчик прикладной программы может применять подпрограммы BIOS точно так же, как и подпрограммы DOS, т. е. с помощью команд *int*. Подпрограммы BIOS применяются в двух случаях:

1) соответствующая функция DOS отсутствует;

2) требуется повысить скорость выполнения функции.

Для очистки экрана будем использовать функцию BIOS – «Прокрутка экрана вверх». Она вызывается оператором «*int 10h*» (функция 6). Данная функция требует задания следующих входных параметров:

(*AL*) – число строк, которые должны быть стерты внизу окна. Обычная прокрутка стирает одну строку. Нуль означает, что нужно стереть все окно;

(*CH,CL*) – строка и столбец левого верхнего угла окна;

(*DH,DL*) – строка и столбец правого нижнего угла окна;

(*BH*) – атрибуты (например, цвет), используемые для строк, окна.

Координаты правого нижнего угла (содержимое регистров *DH* и *DL*) определяются размерами используемого экрана. Эти координаты нетрудно подсчитать, открыв окно текстового редактора, запущенного из DOS или из DOS Navigator.

Таким образом, функция номер 6 десятого прерывания требует довольно много входной информации, даже если все сводится только к очищению экрана. Отметим, что она обладает мощными способностями: может очистить любую прямоугольную часть экрана – окно (*Window*).

После того, как экран очищен, необходимо установить курсор в левый верхний угол экрана. Это делается потому, что курсор указывает на ту позицию экрана, в которой будет выведен следующий символ. Поэтому если мы хотим, чтобы строка символов, выводимая на чистый экран, размещалась в его начале, то обязаны выполнить первоначальную установку курсора.

Для установки курсора используется системный вызов «*int 10h*» (функция 2, подфункция 0). Этот системный вызов требует предварительной записи в регистры:

DL – координаты курсора на экране по горизонтали;

DH – координаты курсора по вертикали;

BX – номер видеостраницы. Далее везде будет использоваться видеостраница 0. Текст процедуры *Clear_screen*, выполняющей очистку экрана:

```
;               Очистка экрана
;               --------------
Clear_screen:
        push  ax
        push  bx
        push  cx
        push  dx
        xor   al, al        ; Очистить все окно
        xor   cx, cx        ; Верхний левый угол в (0,0)
        mov   dh, 49        ; Нижняя строка экрана -49
        mov   dl, 79        ; Правая граница в 79 столбце
        mov   bh, 7         ; Применить нормальные атрибуты
        mov   ah, 6         ; Очистить
        int   10h           ;   окно
        xor   dl, dl        ; Нулевые координаты
        xor   dh, dh        ;  курсора
        xor   bx, bx        ; Нулевая видеостраница
        xor   al, al        ; Установка
        mov   ah, 2         ;  курсора
        int   10h           ;  в заданную точку
        pop   dx
        pop   cx
        pop   bx
        pop   ax
        ret
```

З а п и ш и т е процедуру *Clear_screen* в файл *Cursor.asm* и протестируйте ее. Затем скорректируйте файл *Disp_sec.asm*, записав в начале процедуры *Disp_sector* оператор вызова процедуры очистки экрана.

Запустив программу *Disp_sec.com*, можно убедиться, что экран очищается на весьма короткое время. Это обусловлено тем, что после завершения *Disp_sec.com* управление

возвращается в DOS, которая восстанавливает на экране свою информацию. Для избежания этого в конце процедуры *Disp_sector* поместите оператор, выполняющий ввод символа с клавиатуры. Наличие такого оператора позволит наблюдать на экране неискаженное изображение сектора до тех пор, пока не будет нажата любая клавиша. Впоследствии у нас отпадет потребность в подобном «торможении» нашей программы.

Лабораторная работа 10

Требуется скорректировать программу (и соответствующие файлы) так, чтобы экранная строка дампа содержала бы сначала 4-позиционный шестнадцатеричный адрес первого байта строки, далее один пробел, после которого шестнадцатеричное представление 16 байт памяти. Далее следует еще один пробел, после которого следует символьное представление этих же 16 байт (между символами пробелов нет). При этом требуется скорректировать процедуру *Write_char* так, чтобы вместо любого управляющего символа (код ASCII от *00h* до *1Fh* включительно) должен выводиться символ «.». Пример экранной строки:

A17F 41 42 43 44 45 46 47 00 20 1F 80 81 82 83 84 85 ABCDEFG. .АБВГДЕ

☑ **Примечания:**

1. Переменная-слово *Address* содержит начальный адрес-смещение 256-байтовой области (см. рис. 2.9). Поэтому начальный адрес строки рассчитывается в процедуре *Disp_line* путем суммирования содержимого *Address* с содержимым регистра *DX*. Это суммирование выполняет оператор:

```
add   dx, [Address]
```

Обратите внимание, что имя переменной *Address* заключено в квадратные скобки. При отсутствии этих скобок к содержимому регистра *dx* было бы прибавлено не содержимое переменной, а ее адрес-смещение.

Не забудьте определить переменную-слово *Address* в начале файла *Disp_sec.asm*, задав ей первоначальное значение, например 100*h*. Для этого используется псевдооператор резервирования слова памяти ***dw***:

```
Address   dw  100h
```

В последующих работах будут созданы процедуры, которые загружают в переменную *Sector* редактируемый фрагмент ОП, а в переменную *Address* – начальный адрес (внутрисегментное смещение) этого фрагмента.

2. Перед оператором с меткой «*.M*» в *Disp_line* запишите оператор «*push bx*», который сохранит в стеке номер первого байта выводимой строки. Этот номер пригодится при выводе на экран символов (для его получения не забудьте записать оператор «*pop bx*»).

13. Переписка сектора памяти

Разработанная в предыдущем разделе программа выполняет вывод на экран единственного 256-байтового сектора, заполняемого транслятором до начала выполнения программы. Теперь пришла пора заняться выводом на экран любого сектора, входящего в состав того сегмента ОП, который выделен нашей программе. Напомним, что размер одного сегмента ОП – 64 K, что соответствует 256 секторам по 256 байт.

13.1. Функции переписки сектора

В полученной ранее программе мы использовали для хранения редактируемого сектора (256 байт) область (переменную) *Sector*. В принципе можно было бы не использовать эту область, а напрямую работать с требуемым сектором памяти. Но в этом случае любая ошибка в редактировании мгновенно отразилась бы на оригинале. Поэтому мы будем копировать текущий сектор памяти в область *Sector*, а после выполнения редактирования осуществим обратное копирование.

На рис. 2.10 приведено дерево подпрограмм для программы, выполняющей различные виды функций переписки сектора. Набор этих функций определяется потребностями пользователя разрабатываемой программы. Каждая из функций программно реализуется одной из следующих процедур:

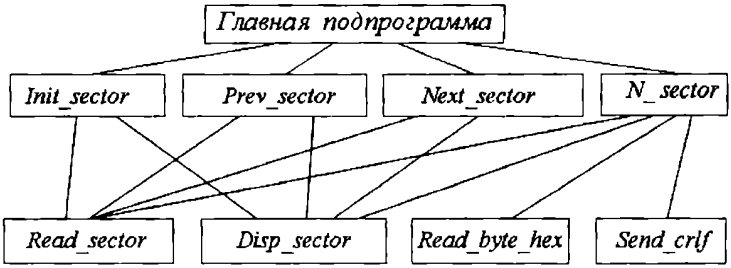

Рис. 2.10. Дерево подпрограмм для программы, выполняющей функции переписки сектора

Init_sector помещает в переменную *Sector* начальный сектор сегмента памяти нашей программы. Кроме того, эта процедура помещает *0* в переменную *Address*;

Prev_sector помещает в *Sector* предыдущий сектор памяти, а в переменную *Address* записывает начальный адрес этого сектора;

Next_sector помещает в переменные *Sector* и *Address* соответственно содержимое и начальный адрес следующего сектора памяти;

N_sector загружает в область *Sector* *N*-й сектор сегмента программы. При этом значение *N* ($0 \le N \le 255$), умноженное на 256, записывается в переменную *Address*.

13.2. Чтение сектора

Как видно из рис. 2.10, все процедуры, выполняющие функции по переписке сектора, пользуются услугами процедуры *Read_sector*, которая записывает в переменную *Sector* содержимое сектора памяти, начальный адрес которого в переменной *Address*. Процедура *Read_sector* не имеет параметров.

При написании этой процедуры мы могли бы ограничиться уже известными нам операторами, но мы привлечем для этого ряд новых операторов:

```
; Копирует  в переменную Sector содержимое сектора памяти, начальный
;                  адрес которого в переменной Address
;       ---------------------------------------------------
;
;
Read_sector:
        push    cx
        push    si
```

```
        push    di
        pushf                   ; Сохранить флаг направления  DF
        mov     si, [Address]   ; Откуда копировать
        mov     di, Sector      ; Куда копировать
        cld                     ; Сбросить флаг DF
        mov     cx, 256         ; В счетчике число байт
rep     movsb                   ; Пересылка цепочки байтов
        popf                    ; Восстановить флаг  DF
        pop     di
        pop     si
        pop     cx
        ret
```

Строковый (цепочечный) оператор ***movsb*** занимает центральное место в процедуре *Read_sector*. Взятый без префикса ***rep***, он выполняет:

1) пересылку байта из ячейки памяти с адресом в регистре *SI* в ячейку памяти, адрес которой находится в регистре *DI*;

2) изменяет на единицу адреса в регистрах *SI* и *DI*.

Направление изменения (увеличение или уменьшение) адресов в *SI* и *DI* определяется значением флага направления *DF* в регистре *FLAGS*. Если *DF*=0, то адреса увеличиваются, а если *DF*=1, то уменьшаются. Сброс флага *DF* выполняет специальный оператор ***cld***, а установку – оператор ***std***. Так как флаг направления может использоваться и в процедуре, вызывающей нашу процедуру, то целесообразно сохранить его перед изменением в стеке, а затем восстановить его оттуда. Для этого используются операторы ***pushf*** и ***popf***, выполняющие запись в стек и извлечение оттуда регистра флагов.

Префикс повторения команды ***rep*** многократно усиливает мощность строковой команды. При этом оператор *movsb* выполняется столько раз, каково содержимое регистра *CX*. При каждом выполнении содержимое *CX* уменьшается на единицу. Как только это содержимое станет равным нулю, то выполняется следующий за *movsb* оператор.

Перед выполнением *movsb* необходимо загрузить в регистры *SI* и *DI* начальные адреса секторов памяти. Для этого можно использовать оператор *mov*. Например, в результате оператора «*mov si,Sector*» адрес самого первого байта области *Sector* будет помещен в регистр *SI*.

Запишите процедуру *Read_sector* в файл *Disp_sec.asm*.

13.3. Алгоритмы процедур

На рис. 2.11 приведён алгоритм процедуры *Init_sector*, а на рис. 2.12 алгоритм процедуры *Prev_sector*. Что касается алгоритма процедуры *Next_sector*, то он очень похож на алгоритм *Prev_sector*. Отличие состоит в том, что номер текущего сектора *N* сравнивается не с 0, а с числом *FFh*.

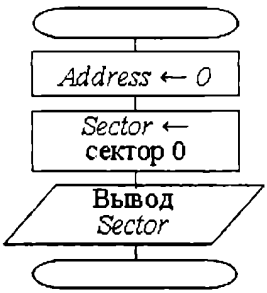

Рис. 2.11. Алгоритм процедуры *Init_sector*

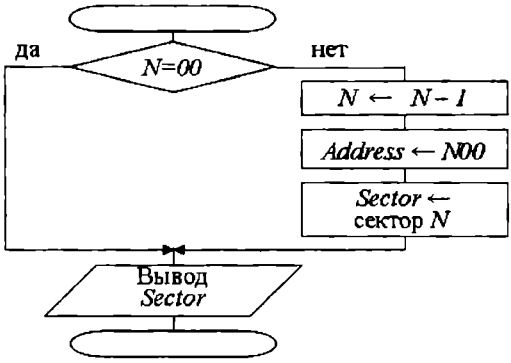

Рис. 2.12. Алгоритм процедуры *Prev_sector*. *N* – номер текущего сектора

На рис. 2.13 приведен алгоритм процедуры *N_sector*. Реализация этапа «Ввод *N*» осуществляется путём вызова процедуры *Read_byte_hex*, выполняющей ввод с клавиатуры двузначного шестнадцатеричного числа. *Read_byte_hex*, в свою очередь, вызывает процедуру *Read_digit_hex*, выполняющую ввод шестнадцатеричной цифры. Алгоритмы обеих процедур были рассмотрены нами ранее в п. 8.3. *З а п и ш и т е* тексты процедур *Read_byte_hex* и *Read_digit_hex* в новый файл *Kbd_io.asm*.

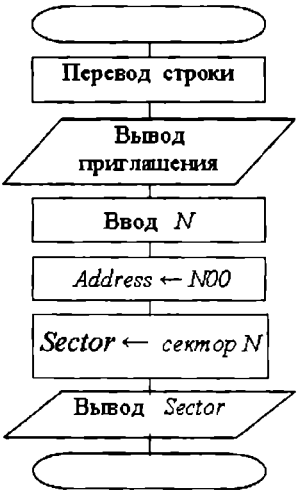

Рис. 2.13. Алгоритм процедуры *N_sector*.
N – номер сектора (2-значное шестнадцатеричное число)

З а п и ш и т е тексты процедур *Init_sector*, *Prev_sector*, *Next_sector*, *N_sector* в файл *Disp_sec.asm*.

13.4. Макрооператоры

Макрооператор – оператор, заменяющий в программе на ассемблере последовательность обычных операторов (исполнительных операторов и псевдооператоров), т. е. вместо того, чтобы записывать в программе несколько операторов, мы помещаем в нее единственный макрооператор. Структура макрооператора:

<имя макрооператора> [список фактических параметров]

Пример макрооператора:

```
Out_str  Buff
```

Здесь макрооператор *Out_str* предназначен для вывода на экран строки символов, расположенной в области памяти с именем *Buff*. В этом примере всего один фактический параметр (*Buff*). Если таких параметров несколько, то они разделяются запятыми.

В начале своей работы транслятор-ассемблер выполняет для каждого макрооператора ***макрорасширение*** – заменяет макрооператор на соответствующую совокупность операторов ассемблера. После завершения макрорасширений текст программы не содержит макрооператоров и представляет собой обычную ассемблерную программу.

Для того чтобы транслятор мог выполнить макрорасширение, имя макрооператора должно быть определено в ***макроопределении***. Структура макроопределения:

```
%macro   <имя макрооператора> [количество формальных параметров]
    . . . . . . . . . . . . . . . . ; Тело макроопределения
%endmacro
```

Пример макроопределения:

```
 %macro  Out_str  1
;  Вывод строки на экран
;  ---------------------
;  Вход:  %1 – адрес-смещение выводимой строки
       push     ax
       push     dx
       mov      ah, 09h
       mov      dx, %1
       int      21h
       pop      dx
       pop      ax
%endmacro
```

При выполнении макрорасширения для *Out_str* транслятор заменит этот макрооператор на тело макроопределения, причем он заменит формальный параметр *Str* в теле макроопределения на фактический параметр *Buff*. (В примере такая замена выполняется в единственном операторе *mov*.)

Следует заметить, что в качестве фактического параметра макрооператора нельзя использовать регистр, в который производится запись внутри макроопределения. Например, в макроопределении для *Out_str* выполняется запись в регистр *AH*. Если в качестве единственного фактического параметра для *Out_str* задать регистр *AX*, то произойдет ошибка, так как прежде, чем фактический параметр *AX* будет использоваться внутри макроопределения, величина этого параметра будет изменена.

В качестве примера макрооператора с несколькими параметрами приведем макрооператор *Push_5*, выполняющий запись в стек содержимого пяти регистров. Соответствующее макроопределение:

```
%macro      Push_5  5
            push    %1
            push    %2
            push    %3
```

```
        push    %4
        push    %5
%endmacro  Push_5
```

Областью применения макрооператора *Push_5* является сохранение содержимого регистров в начале процедур. Вместо того, чтобы записывать пять операторов *push*, в программу достаточно поместить всего один макрооператор вида: «*Push ax, bx, cx, dx, es*». Естественно, что для восстановления содержимого регистров в конце процедуры желательно использовать другой макрооператор – *Pop_5*. Нетрудно сделать соответствующее макроопределение таким, чтобы регистры-параметры записывались не в обратном, а в прямом направлении. Естественно, что кроме макрооператоров *Push_5* и *Pop_5* желательно иметь макрооператоры для сохранения и восстановления другого числа регистров.

Где расположить макроопределение? Здесь можно использовать два варианта. В первом из них макроопределение располагается в начале того же файла, где оно будет использоваться. Недостаток: в каждом файле, где предполагается использовать аналогичный макрооператор, его придется описывать заново. Во втором варианте макроопределение выносится в отдельный файл (текстовый ASCII-файл, полученный с помощью любого текстового редактора). А в начале каждого ассемблерного файла, где предполагается использовать макроопределение, записывается псевдооператор *%include* с указанием имени включаемого файла. Например:

```
%include    «Outstr.inc»
```

Обычно в один включаемый файл помещают несколько макроопределений и, возможно, описания структур (которые фактически являются разновидностью макроопределений). При выполнении транслятором псевдооператора *%include* весь включаемый файл подсоединяется к тексту программы, содержащей *%include*, но на текст машинной программы наличие этого файла не влияет. После выполнения макрорасширений транслятор уничтожит подсоединенный фрагмент исходной программы.

Полезно выполнить сравнение макроопределений с подпрограммами. Оба эти типа модулей позволяют программисту инициировать выполнение многих операторов с помощью единственного оператора (*call* или *int* для подпрограммы и макрооператор для макроопределения). Но механизм их реализации совершенно различен. В то время как тело макроопределения будет записано транслятором всюду, где он встретит соответствующий макрооператор, тело подпрограммы находится в памяти в единственном экземпляре, а операторы *call* (или *int*) выполняют передачу управления этому экземпляру. Отсюда можно сделать более обоснованным выбор между этими типами модулей.

Так как каждое выполнение подпрограммы связано с «накладными расходами» в виде двух машинных команд (*call* и *ret*, или *int* и *iret*), то вклад этих команд в продолжительность выполнения тем больше, чем меньше подпрограмма. С другой стороны, чем длиннее тело макроопределения, тем больше затраты памяти на его «тиражирование». Следовательно, короткие повторяющиеся последовательности операторов лучше оформлять в виде макроопределений, а длинные – в виде подпрограмм.

Лабораторная работа 11

В ы п о л н и т е отладку процедур, предназначенных для переписки сектора. Для этого в начале файла *Disp_sec.asm* временно поместите тестовую процедуру *Imit*:

```
Imit:
    call    Init_sector
    mov     ah,1                ; Ожидание нажатия
```

```
int   21h              ; клавиши
call  N_sector
mov   ah,1             ;      –"–
int   21h
call  Prev_sector
mov   ah,1             ;      –"–
int   21h
call  N_sector
mov   ah,1             ;      –"–
int   21h
call  Next_sector
mov   ah,1             ;      –"–
int   21h
mov   ax, 4c00h
int   21h              ; Возврат в DOS
```

Применение двойного вызова процедуры *N_sector* преследует цель создать требуемые условия для тестирования процедур *Prev_sector* и *Next_sector*, так как позволяет сделать текущим любой интересующий нас сектор. Особое внимание следует уделить граничным условиям – первому и последнему секторам.

☑ **Примечание.** В программе требуется использовать не менее двух различных макрооператоров.

14. Диспетчер команд

В данной главе решается задача разработки диспетчера – программного модуля, обеспечивающего диалог с пользователем и выполняющего координацию работы других модулей программы (редактора). Применяемый при этом подход может быть использован для построения диспетчеров в других программных системах.

14.1. Ввод команд

Почти любая полноценная программа является интерактивной, т. е. способной вести диалог со своим пользователем. Естественно, что наш редактор информации также должен быть интерактивным. Он должен воспринимать команды пользователя, набираемые на клавиатуре и выводить на экран ответные сообщения. При этом принято называть сообщения пользователя (например, команды) ***входными сообщениями***, а сообщения программы – ***выходными***.

Распознавание команд пользователя поручим программному модулю (процедуре), называемому ***диспетчером команд***. Данный модуль занимает центральное место в программной системе и координирует работу других модулей. Допустим, что наш редактор выполняет всего пять команд, каждой из которых соответствует своя управляющая клавиша:

<F1> – вывести на экран начальный сектор (256 байт) ОП;
<F3> – вывести на экран следующий сектор ОП;
<F5> – вывести на экран предыдущий сектор ОП;
<F6> – вывести на экран сектор N, где $0 \le N \le 256$;
<F10> – закончить работу редактора.

Клавиши <F2> и <F4> оставлены «в резерве» для реализации нами в будущем функции редактирования.

Как говорилось ранее, каждому выводимому на экран символу соответствует свой код ASCII. Но каждому из этих символов соответствует также свой ***скан-код*** BIOS. Добавление слова BIOS требуется потому, что, вообще то, говоря, существуют несколько скан-кодов. В данном случае речь идет о том скан-коде клавиши, который возвращается из BIOS в вызвавшую его прикладную программу. Далее, говоря «скан-код», мы будем подразумевать скан-код BIOS.

Грубо говоря, скан-код – номер нажатой клавиши. Но любой пользователь знает, что большинству клавиш соответствуют не один, а два или более символов. Поэтому один и тот же скан-код может соответствовать нескольким символам. Например, скан-код *16h* соответствует четырем символам: «*и*», «*U*», «*г*», «*Г*». Поэтому для идентификации алфавитно-цифровых символов в обрабатывающих программах скан-коды не используются. Другое дело – управляющие символы, так как каждому из них соответствует своя отдельная клавиша.

Для того чтобы получить одновременно и ASCII-код клавиши, и ее скан-код, следует использовать системный вызов BIOS – «***int 16h***», функция ***0***, выполняющий ввод символа с клавиатуры. После выполнения данного вызова в регистре *AL* содержится ASCII-код нажатой клавиши, а в регистре *AH* – ее скан-код.

Что касается управляющих клавиш, то они используются не для отображения на экране, а для управления работой программ. Многим из этих клавиш, в том числе и для *<Fi>*, соответствует один и тот же ASCII-код – *00h*. Поэтому различить такие клавиши между собой можно лишь по их скан-коду:

3Bh – <F1>, 3Ch – <F2>, 3Dh – <F3>, 3Eh – <F4>,
3Fh – <F5>, 40h – <F6>, 41h – <F7>, 42h – <F8>,
43h – <F9>, 44h – <F10>.

Ниже приведен текст процедуры *Read_byte*, выполняющей ввод с клавиатуры любого символа – обычного или управляющего. При этом код ASCII символа возвращается в регистре *AL*, а в регистре *AH* возвращается соответствующий скан-код. Обратите внимание, что вывод «эха» символа не производится.

```
;       Считывает код символа с клавиатуры
;       ----------------------------------
;  Выход:  AL – код ASCII символа (0 – управляющий символ)
;          AH – скан-код символа
Read_byte:
   mov  ah,0   ; Ввод символа
   int 16h     ;  без эха
   ret
```

П о м е с т и т е процедуру *Read_byte* в начало файла *Kbd_io.asm*, временно записав перед ней оператор «*org 100h*», а затем выполните ее отладку, используя Debug. Для этого получите файл *Kbd_io.com* и наберите команду DOS:

```
DEBUG   Kbd_io.com
```

Для запуска программы используйте команду Debug: «*G i*», где *i* – адрес в листинге команды *ret*. Нажимая различные клавиши, проверьте правильность заполнения процедурой регистра *AX*.

14.2. Алгоритм диспетчера

На рис. 2.14 приведена блок-схема главной подпрограммы (процедуры) *Dispatcher*, выполняющей совместно с рассматриваемой далее процедурой *Command* функции диспетчера команд. Для того чтобы обеспечить структурность алгоритма, мы, как и в одном из предыдущих разделов, используем флаг переноса *CF* и операции над ним.

Кодирование процедуры *Dispatcher* не представляет особого труда. Многие этапы ее алгоритма реализуются путем вызова ранее разработанных процедур.

На этапе «Вывод справки» в качестве самой первой строки на экран выводится справочная строка, поясняющая назначение управляющих клавиш (аналог справочной строки в DOS Navigator). Возможный вид такой строки:

F1-начало| F2-запись| F3-след.| F4-редакт.| F5-предыд.| F6-произв.| F10-завер.

Этап «Вывод приглашения» реализуется путем вывода на экран строки символов или, даже, всего одного-двух символов, однозначно указывающих на то, что редактор ожидает команд пользователя. Если приглашение представляет собой строку символов, то эту строку необходимо определить где-то в программе с помощью псевдооператора *db* (определение байтов). Пример такого определения:

```
.Msg    db   0ah, 0dh,   'Пожалуйста, наберите команду',0ah, 0dh, '$'
```

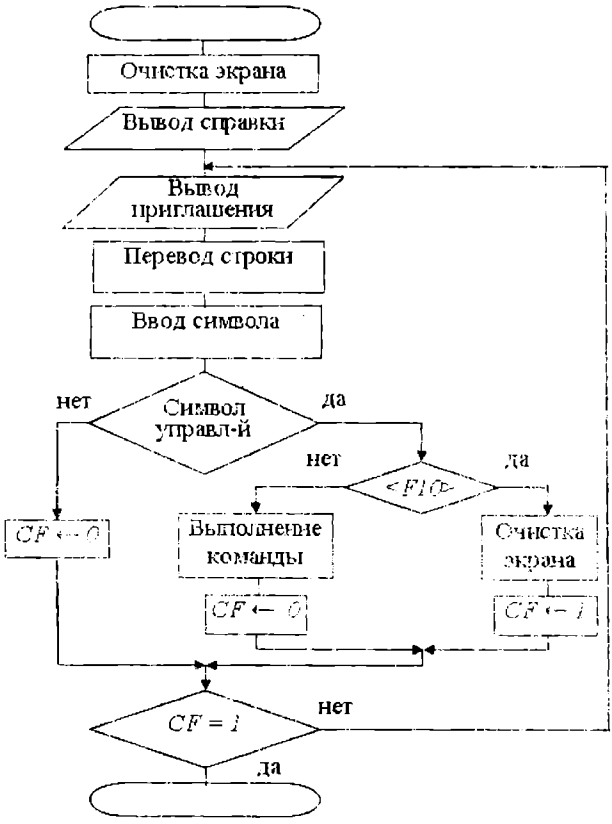

Рис. 2.14. Алгоритм процедуры *Dispatcher*:
CF – признак завершения (*0* – продолжить, *1* – окончить работу)

Обратите внимание, что для удобства чтения сообщения пользователем символы перевода строки (*0ah*) и возврата курсора (*0dh*) записаны не только в конец, но и в начало строки. В качестве последнего символа строки записан символ-ограничитель – *$*. Так как данному сообщению присвоено локальное имя (начинается с точки), то это имя должно использоваться в программе не далее ближайшего места определения глобальной метки. Напомним, что глобальная метка (глобальное имя) не начинается с точки и может использоваться в любом месте программы. Что касается размещения записанного выше сообщения, то его удобно выполнить сразу же за оператором *ret*, завершающим процедуру *Dispatcher*.

Рассмотрим реализацию этапа «Выполнение команды».

14.3. Выполнение команды

Этап «Выполнение команды» реализуется путем вызова процедуры *Command*. Данная процедура получает в регистре *AH* скан-код, соответствующий команде, и в зависимости от этого кода вызывает процедуру, выполняющую заданное командой действие.

На рис. 2.15 приведена блок-схема процедуры *Command*. Центральной особенностью ее алгоритма является использование таблицы переходов. В данной таблице для каждой разрешенной команды пользователя отводятся три байта. В первом байте находится скан-код, соответствующий команде, а в двух других байтах – начальный адрес (внутрисегментное смещение) соответствующей процедуры. В последнем байте таблицы находится *0ffh*.

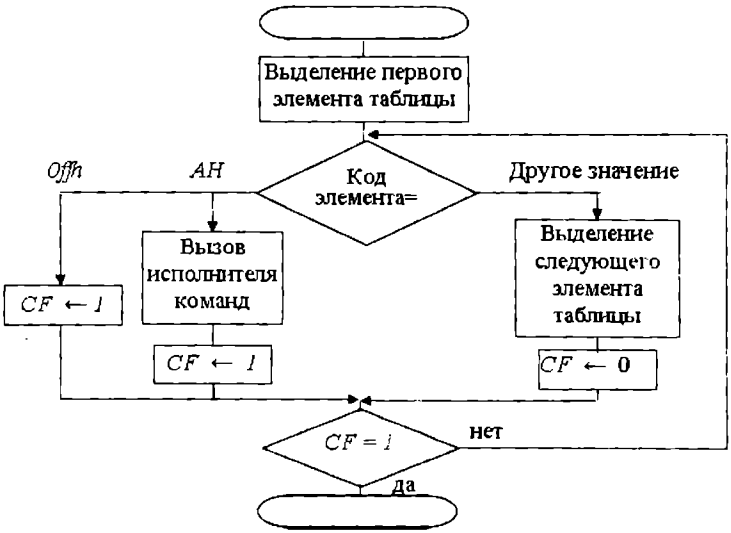

Рис. 2.15. Алгоритм процедуры *Command*:
AH – код команды; *CF* – признак завершения (*0* – продолжить, *1* – завершить работу)

Процедура *Command* совместно с главной процедурой *Dispatcher* обеспечивают выполнение алгоритма диспетчера команд. Поэтому обе процедуры, а также таблицу переходов .Tab*le* мы поместим в один и тот же файл *Dispatch.asm*. Заключительный фрагмент этого файла содержит модули *Command* и .Tab*le*:

```
    org 100h
;       Координирует выполнение модулей редактора
;       ----------------------------------------
;
Dispatcher:
.  .  .  .  .  .  .  .  .                  ; Тело главной процедуры
            mov   ax, 4c00h
            int   21h                      ; Возврат в DOS
;
;         Интерпретатор команд
;         --------------------
;    Входы:  AH – скан-код, соответствующий команде
;    Чтение: .Table – таблица переходов
;
Command:
            push  bx
            mov   bx, .Table       ; BX ← адрес таблицы
.M1:        cmp   byte[bx], 0ffh   ; Конец таблицы ?
            je    .Exit            ; Да, кода  нет в таблице
            cmp   ah,[bx]          ; Это вход в таблицу?
            je    .Dispatch        ; Да, выполнить команду
            add   bx,3             ; Нет, переход к
            clc                    ;   следующему
            jmp   .M2              ;   элементу таблицы
.Dispatch:  inc   bx               ; Адрес процедуры
            call  [bx]             ; Вызов процедуры
.Exit:      stc                    ;  CF ← 1
.M2:        jnc   .M1              ; Повторение для нового элемента таблицы
            pop   bx
            ret
;  Таблица содержит скан-коды управляющих клавиш и адреса
;  процедур, выполняющих соответствующие команды
.Table   db      3bh               ; <F1>
         dw      Init_sector
         db      3dh               ; <F3>
         dw      Next_sector
         db      3fh               ; <F5>
         dw      Prev_sector
         db      40h               ; <F6>
         dw      N_sector
         db      0ffh              ; Конец таблицы
```

Рассмотрим особенности этих модулей. Во-первых, сравним между собой два оператора: «*cmp byte [bx], 0*» и «*cmp dl, [bx]*». Оба этих оператора выполняют сравнение заданной величины (число 0 или содержимое регистра *dl*) с содержимым той ячейки памяти, адрес которой находится в регистре *bx*. В данном регистре находится адрес-смещение младшего байта ячейки памяти, но не задана длина этой ячейки. Об этой длине

транслятор может узнать или из анализа второго операнда команды, или если задан дополнительный псевдооператор ***byte***, ***word*** или ***dword***. В первой из двух записанных выше команд псевдооператор *byte* сообщает транслятору, что сравниваются байты. Аналогично, команда «*cmp word [bx], 0*» будет сравнивать слова. Во второй из приведенных выше команд дополнительный псевдооператор излишен, так как о байтовой длине ячейки памяти сообщает другой операнд команды – регистр *dl*.

Для того чтобы получить адреса процедур, вызываемых диспетчером и содержащихся в таблице, мы записали имя каждой процедуры в качестве операнда псевдооператора *dw*, выполняющего запрос к транслятору с просьбой выполнить резервирование слова ОП и поместить в это слово памяти требуемое значение. В данном случае таким требуемым значением является адрес-смещение первого байта процедуры. Транслятор выполнит нашу просьбу правильно, так как имя процедуры записано без квадратных скобок, что и означает ее адрес.

Применение для реализации диспетчера команд таблицы переходов значительно увеличивает его гибкость, т. е. пригодность к модификации. В самом деле, мы легко можем добавить в нашу программу новые команды, набираемые на клавиатуре – достаточно записать процедуру, выполняющую новую команду, в подходящий файл, и поместить новую точку входа в .Tab*le*.

Лабораторная работа 12

Запишите процедуры диспетчера в файл *Redaktor asm* (главный исходный файл) и выполните отладку программы. Отлаженная программа должна правильно реагировать на нажатие любой клавиши.

☑ **Примечание**. Для того, чтобы справочная строка (первая строка экрана) не исчезала после каждого вывода сектора на экран, необходимо внести изменения в процедуру *Disp_sector* (в файле *Disp_sector*). Суть изменения в том, что после очистки экрана необходимо выполнить вывод на экран справочной строки. Далее рекомендуется выполнить пропуск не одной, а трех строк.

15. Редактирование информации в ОП

15.1. Алгоритм редактирования сектора

Редактирование сектора выполняет процедура *Edit_sector*, инициируемая диспетчером при нажатии клавиши <F4>. После того, как процедура *Edit_sector* запущена, она, во-первых, выполняет обработку своих команд. Такими командами являются клавиши <F4> и <Esc>, выполняющие завершение функции редактирования, клавиша <F2> – запись отредактированного сектора в ОП, а также клавиши, выполняющие управление курсором:

→ – переместить курсор на одну позицию вправо (скан-код – *4dh*);
← – переместить курсор на одну позицию влево (скан-код – *4bh*);
↑ – переместить курсор на одну позицию вверх (скан-код – *48h*);
↓ – переместить курсор на одну позицию вниз (скан-код – *50h*).

Во-вторых, процедура *Edit_sector* может выполнить редактирование любых четырех битов (тетрады) переменной *Sector*. Для этого она вводит с клавиатуры двоичный код шестнадцатеричной цифры и помещает его в качестве той тетрады переменной *Sector*, на которую в данный момент времени указывает курсор монитора (экрана).

Алгоритм процедуры *Edit_sector* приведен на рис. 2.16. Этап «Вывод *Sector*» реализуется путем вызова процедуры *Disp_sector*. Для того чтобы выполнить этап «Инициализация курсора», обнуляются две переменные, используемые для управления внутри процедуры *Edit_sector*:

B (длиной 1 байт) – номер редактируемого байта в глобальной переменной *Sector*.

Nt (длиной 1 байт) – номер (0 или 1) редактируемой тетрады в байте. Номер 0 соответствует старшей тетраде байта, а 1 – младшей тетраде.

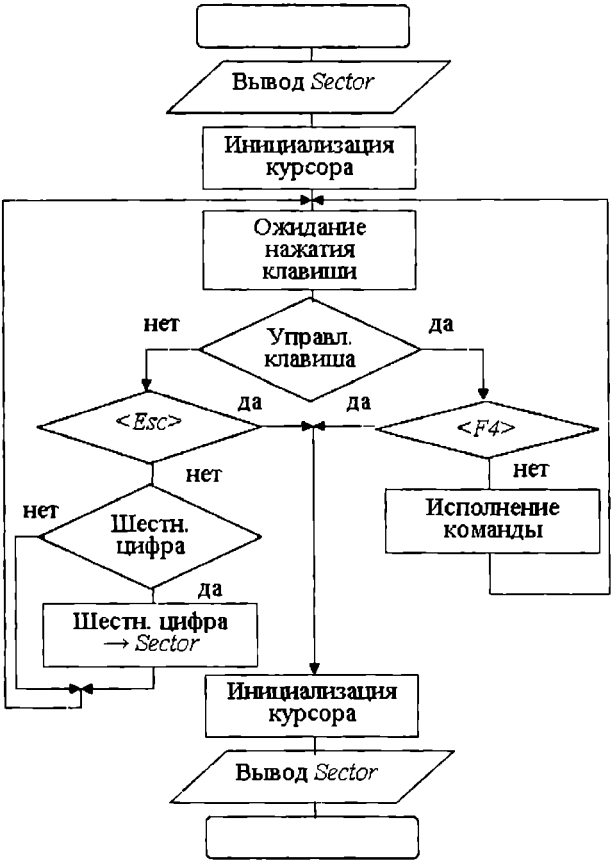

Рис. 2.16. Алгоритм процедуры *Edit_sector*

При этом заметим, что шестнадцатеричная цифра, соответствующая старшей тетраде (*Nt = 0*), на экране выводится левее, а цифра, соответствующая младшей тетраде (*Nt = 1*) – правее.

Далее вызывается процедура *Set_cursor*, выполняющая установку курсора в ту позицию экрана, которая соответствует переменным *B* и *Nt*.

Этап «Ожидание нажатия клавиши» реализуется с помощью известного нам системного вызова BIOS – «*int 16h*», функция 0. Напомним, что данный вызов возвращает в регистре *AL* код ASCII нажатой клавиши, а в *AH* – скан-код клавиши. Определение управляющего символа производится по наличию у клавиши нулевого ASCII-кода.

Если управляющая клавиша не является клавишей <F4>, то производится исполнение соответствующей команды путем вызова процедуры *Edit_command*, рассматриваемой ниже. Что касается клавиши <Esc>, то она не отнесена к управляющим клавишам, так как ее ASCII-код ненулевой – *1Bh*. Так как скан-код клавиши <Esc> – *01h*, то для проверки нажатия клавиши <Esc> следует сравнить содержимое регистра *AX* с числом *011bh*.

Для проверки того, что введенный символ является шестнадцатеричной цифрой, можно воспользоваться слегка переделанной процедурой ввода шестнадцатеричной цифры *Read_digit_hex*, разработанной ранее. Переделанная процедура должна не сама вводить символ с клавиатуры, а получать его ASCII-код в качестве входного параметра (в регистре *AL*). Выходные параметры процедуры: флаг *CF* – признак шестнадцатеричной цифры (0 – цифра, 1 – нет), *DL* – содержит саму цифру. Естественно, что следует изменить название процедуры, например, на *Test_digit_hex*.

Что касается записи шестнадцатеричной цифры в переменную *Sector*, то для этого требуется разработать новую процедуру – *Write_digit_sector*.

Последними двумя этапами процедуры *Edit_sector* являются инициализация курсора и вывод переменной *Sector* на экран, для реализации которых используются соответственно процедуры *Set_cursor* и *Disp_sector*. Может показаться странным инициализация курсора в конце алгоритма редактирования сектора, так как другие наши процедуры курсор не используют. Наличие данного этапа обусловлено следующим. Вспомним, что курсор указывает на ту позицию экрана, в которой будет выведен следующий символ. Поэтому вывод на экран строки символов, выполняемый процедурой *Disp_sector*, также начинается с позиции, отмеченной курсором. Если не установить курсор в верхний левый угол изображения, то часть этого изображения не будет показана. Можете проверить это практически.

Получите исходные тексты процедур *Edit_sector* и *Test_digit_hex*, поместив их в файл *Edit_sec.asm*.

15.2. Установка курсора

Установку курсора в требуемую позицию на экране выполняет процедура *Set_cursor*.

Данная процедура преобразует одну пару чисел (*B*, *Nt*) в другую пару чисел ((*DL*), (*DH*)), где *B* – номер байта в переменной *Sector*, *Nt* – номер тетрады в байте, (*DL*) – координата курсора на экране по горизонтали, (*DH*) – координата курсора на экране по вертикали. В завершении процедура обращается к BIOS с целью установить курсор в точку экрана с заданными координатами, используя знакомый нам системный вызов «*int 10h*» (функция 2, подфункция 0).

Алгоритм процедуры *Set_cursor* приведен на рис. 2.17. На первом этапе алгоритма в регистр *DH* загружается номер строки дампа сектора (не путать со строкой экрана, номер которой еще предстоит получить). Для этого номер байта в секторе *B* делится на число байтов в строке, т. е. на число 16. Такое деление рекомендуется выполнить путем сдвига содержимого регистра на четыре бита вправо.

На втором этапе в *DL* загружается номер байта в строке. Этот номер есть разность между номером байта в переменной *Sector* (переменная *B*) и таким же номером последнего байта в последней полностью заполненной строке. Последний номер нетрудно получить сдвигом влево на четыре бита полученного на первом этапе номера строки дампа.

Далее производится преобразование номера байта в строке дампа в номер символа на строке экрана. Для этого на следующем этапе алгоритма полученный ранее номер в строке дампа умножается на три (два символа есть шестнадцатеричные цифры для двух тетрад байта, а один символ – пробел). Далее к полученному номеру прибавляется величина *Nt* для того, чтобы курсор указывал на требуемую тетраду байта.

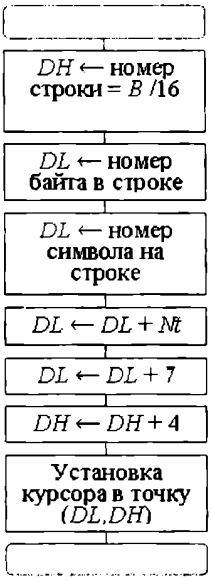

Рис. 2.17. Алгоритм процедуры *Set_cursor*: *DL* – координата курсора на экране по горизонтали; *DH* – координата курсора на экране по вертикали; *B* – номер байта в переменной *Sector*; *Nt* – номер тетрады в байте (0 – старшая, 1 – младшая)

Далее положение курсора сдвигается ближе к центру экрана. Для этого к *DL* прибавляется 7, а к *DH* 4. Первое из этих прибавлений позволяет «перепрыгнуть» через изображение адреса, а второе – учитывает наличие на экране справочной строки, а также наличие трех пустых строк. Завершает работу по перемещению курсора соответствующий системный вызов.

П о л у ч и т е исходный текст процедуры *Set_cursor* и добавьте его в файл *Cursor.asm*.

15.3. Выполнение команды

Для выполнения любой из четырех своих команд, выполняющих управление курсором, а также для записи сектора в ОП, процедура *Edit_sector* вызывает свою дочернюю процедуру *Edit_command*. Алгоритм данной процедуры аналогичен алгоритму ранее разработанного интерпретатора команд – *Command*. Более того, идентичны тексты этих процедур. Идентичны и структуры используемых таблиц переходов. Естественно, что эти таблицы различаются числом строк, а также именами вызываемых процедур.

Имена процедур в новой таблице переходов: *Cur_right* (сдвиг курсора вправо), *Cur_left* (сдвиг курсора влево), *Cur_top* (сдвиг курсора вверх), *Cur_down* (сдвиг курсора вниз), *Write_sector* (сохранение содержимого переменной *Sector* в памяти).

Вот тексты двух из четырех процедур, выполняющих сдвиг курсора:

```
;  Перемещение курсора на одну позицию влево
;
Cur_left:
```

```
        cmp   [Nt], byte 0
        je    .M1
        dec   byte [Nt]
        jmp   .Exit
.M1:    inc   byte [Nt]
        dec   byte [B]
.Exit:  call  Set_cursor
        ret
;  Перемещение курсора на одну позицию вверх
;
Cur_top:
        sub   [B], byte 16
        call  Set_cursor
        ret
```

Тексты двух других процедур *Cur_right* и *Cur_down* очень похожи, соответственно, на тексты процедур *Cur_left* и *Cur_top*.

Обратите внимание, что в данных процедурах нет проверок достижения переменной *B* своих крайних значений 0 и 255. Это объясняется тем, что длина данной переменной 1 байт, и поэтому при попытке уменьшить значение 0 в переменную будет записано число 255. А при попытке увеличить значение 255 в переменную будет записано число 0. Подобные изменения переменной *B* приведут к правильным перемещениям курсора на экране.

Что касается процедуры *Write_sector*, то она не имеет параметров. Данная процедура переписывает содержимое переменной *Sector* в область памяти, начальный адрес которой находится в переменной *Address*. Текст этой процедуры очень похож на текст ранее полученной процедуры *Read_sector*.

Получите исходные тексты процедур *Edit_command* и *Write_sector*, поместив первую из них в файл *Edit_sec.asm*, а вторую – в файл *Disp_sec.asm*. Исходные тексты процедур *Cur_right, Cur_left, Cur_top, Cur_down* добавьте в файл *Cursor.asm*.

15.4. Запись шестнадцатеричной цифры

Алгоритм процедуры *Write_digit_sector*, выполняющей запись в переменную *Sector* шестнадцатеричной цифры, приведен на рис. 2.18. Данная процедура имеет единственный входной параметр – в регистре *DL* передается величина шестнадцатеричной цифры, которая должна быть записана в позицию переменной *Sector*, на которую указывает курсор.

На первом этапе алгоритма проверяется на равенство нулю номер тетрады *Nt*. Если *Nt = 0*, то это значит, что редактируется старшая тетрада байта (соответствующая шестнадцатеричная цифра расположена на экране слева). В этом случае данная тетрада сначала обнуляется (с помощью операции логического умножения), а затем в нее помещается двоичный код шестнадцатеричной цифры *Z* (с помощью операции логического сложения).

Если *Nt = 1*, то редактируется младшая тетрада байта, которая сначала обнуляется, а затем в нее помещается двоичный код шестнадцатеричной цифры *Z*. Для кодирования данных двух этапов используются операторы сдвига, а также операторы логического умножения *and* и логического сложения *or*.

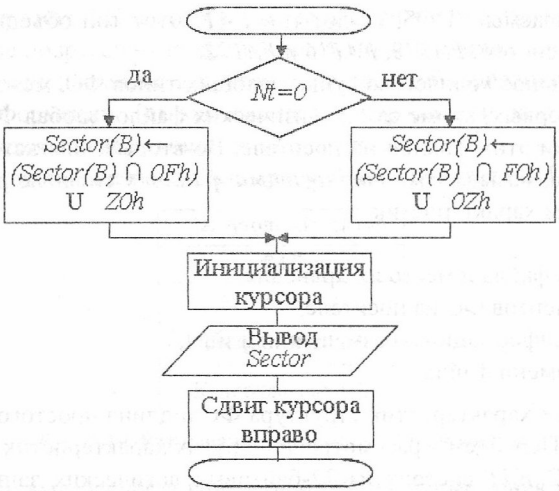

Рис. 2.18. Алгоритм процедуры *Write_digit_sector*:
B – номер байта в переменной *Sector*; *Nt* – номер тетрады (0 или 1) в редактируемом байте;
∩ и U – операции логического умножения и сложения; *Z* – шестнадцатеричная цифра

Назначение последних трех этапов алгоритма достаточно очевидно и не требует пояснений. *П о л у ч и т е* исходный текст процедуры *Write_digit_sector* и добавьте его в файл *Edit_sec.asm*.

Лабораторная работа 13

Требуется написать и отладить описанные выше процедуры редактирования сектора. Процедура *Edit_sector* должна вызываться из диспетчера команд при нажатии клавиши <F4>.

Заметим, что редактор информации, полученный в результате выполнения данной и предыдущих лабораторных работ, является базовым. Несмотря на то, что на последующих работах этот вариант редактора получит существенное развитие, исходные тексты базового варианта обязательно должны быть сохранены. Это обусловлено тем, что впоследствии текст базового варианта будет использоваться в качестве основы при разработке программ, выполняемых не только под управлением DOS, но и автономно, т. е. без помощи ОС.

16. Операции с файлами

16.1. Файловая система FAT

Физический файл существует в ВС не обособленно, а как часть более крупной информационной единицы, называемой ***файловой системой*** (**ФС**). Эту систему образуют файлы и таблицы, расположенные на конкретном носителе ВП или расположенные на его части (в разделе или на логическом диске). Иногда в понятие файловой системы включают и совокупность подпрограмм ОС, выполняющих операции с файлами. Но мы, для избежания путаницы, этого делать не будем. Например, файловые системы, поддер-

живаемые (обслуживаемые) DOS, имеют тип ***FAT***. Этот тип объединяет три родственных типа файловых систем: *FAT12*, *FAT16* и *FAT32*.

Несмотря на большое количество существующих типов ФС, между этими системами много общего. Во-первых, кроме самих физических файлов любая ФС содержит информацию о размещении этих файлов на носителе. Во-вторых, для каждого файла ФС содержит его свойства, называемые ***атрибутами файла***. Основные различия между ФС сводятся к различиям характеристик:

- структура ФС;
- состав атрибутов файла и место их хранения;
- размещение элементов ФС на носителе;
- место хранения информации о размещении файла;
- длина простого имени файла.

Из перечисленных характеристик структура ФС и длина простого имени файла были рассмотрены ранее. Перейдем к рассмотрению других характеристик.

Файл-каталог ФС *FAT* состоит из 32-байтовых логических записей. В *FAT12* и в *FAT16* одна такая запись соответствует одному файлу. В ней находятся: простое имя файла (собственно имя и расширение имени); номер первого кластера логического диска, занимаемого данным файлом; атрибуты файла. В *FAT32* каждому файлу соответствуют несколько 32-байтовых записей каталога. Структура первой из них аналогична записи каталога в *FAT12* и в *FAT16*, а в последующих записях содержится длинное (до 255 байт) простое имя файла. Таким образом, один и тот же файл может обрабатываться как системными вызовами, предназначенными для работы с короткими, так и системными вызовами для работы с длинными именами файлов.

Любая ФС содержит для каждого из своих файлов его атрибуты. Они используются самой ОС, а также пользователями и обрабатывающими программами. Наименьшее число атрибутов имеют файлы в ФС, поддерживаемых однопользовательскими однопрограммными системами. Например, в *FAT* файл имеет атрибуты:

- дата и время последней модификации файла;
- текущий размер файла;
- флаг «каталог»;
- флаг «только для чтения»;
- флаг «скрытый файл»;
- флаг «системный файл»;
- флаг «архивный файл».

Атрибуты файла могут находиться или в записи каталога, соответствующей файлу, или в отдельной таблице (в части таблицы). Как уже говорилось, в *FAT* атрибуты файла содержатся в 32-байтовой записи каталога, соответствующей этому файлу.

Разные ФС различным образом распределяют пространство носителя ВП (или его раздела) между своими элементами (файлами и таблицами).

Первоначальное распределение пространства носителя производится во время ***логического форматирования носителя***. В отличие от физического форматирования, которое не зависит не только от используемой ФС, но и от типа ОС, логическое форматирование носителя (или его раздела) ориентировано на применение конкретной ФС. В результате такого форматирования на носителе создаются корневой каталог и таблицы ФС. Кроме того, возможно, на носитель записывается ***начальный загрузчик*** – программа, предназначенная для загрузки используемой ОС в ОП (в начале работы или после сбоя).

В зависимости от типа ФС некоторые из ее элементов дублируются для повышения надежности: в случае повреждения одной области носителя может использоваться копия этой информации, расположенная в другой области. (Дублирование не спасает от программной порчи информации. Так как изменение информации на носителе программой синхронно выполняется для всех копий этой информации.)

На рис. 2.19 приведено распределение пространства носителя ВП для ФС типа *FAT*.

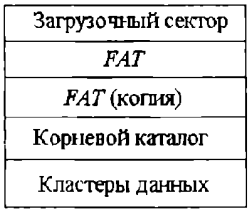

Рис. 2.19. Размещение системы *FAT* на носителе

Пространство ВП, выделенное в конкретной ФС для размещения файлов (в том числе, для размещения каталогов), может распределяться между ними двумя основными способами: 1) непрерывными разделами; 2) в виде совокупности блоков памяти, необязательно соседних друг с другом. Непрерывное распределение имеет серьезные недостатки:

- Вся память назначается файлу ***статически*** – в момент его создания. Для многих файлов нельзя заранее предсказать их будущую длину и поэтому память для них должна выделяться ***динамически*** – по мере надобности во время выполнения операций записи в файл.
- Имеет место ***внешняя фрагментация памяти*** – наличие в ВП пустых участков, полученных в результате удаления ненужных файлов. Несмотря на значительную суммарную длину этих участков, отсутствуют непрерывные разделы, позволяющие размещать достаточно большие файлы.

Перечисленные недостатки привели к тому, что в настоящее время непрерывное распределение почти не используется для ФС на магнитных дисках. Оно применяется лишь для магнитных лент и лазерных дисков, так как пространство этих носителей распределяется между файлами только статически. (А для лазерных дисков отсутствует и внешняя фрагментация памяти.)

Что касается файлов на магнитных дисках, то память для них назначается динамически блоками, в качестве которых используются ***кластеры***. Размер кластера для носителя (раздела) фиксируется при выполнении логического форматирования и составляет от 512 байт (1 сектор) до 64 Кбайт (128 секторов). Размер кластера зависит как от размера носителя (раздела), так и от типа ФС.

Например, в *FAT12*, *FAT16* и *FAT32* для кодирования номера кластера используются соответственно 12, 16 и 32 бита (отсюда различия в названиях ФС). Поэтому максимальный номер кластера в *FAT12* равен 4095, в *FAT16* – 65535, в *FAT32* – 4*Г*. Размер кластера для каждого носителя (раздела) выбирается с таким расчетом, чтобы произведение размера кластера на его максимальный номер не было меньше, чем размер носителя (раздела). Иначе на носителе будут неадресуемые кластеры.

Другим фактором, влияющим на выбор размера кластера, является наличие ***внутренней фрагментации*** – из всех кластеров ВП, выделенных файлу, половина одного кластера (последнего), в среднем, не используется. Поэтому, чем меньше размер кластера, тем меньше суммарный объем неиспользуемого пространства ВП.

Основным следствием из двух названных факторов является то, что *FAT12* используется для небольших дисков (не более 16 Мбайт), что позволяет не делать кластеры более 4 Кбайт. *FAT16* целесообразна для дисков не более 512 Мбайт. Для больших дисков целесообразна *FAT32*, которая позволяет для дисков емкостью до 8 Гбайт ограничиться размером кластера 4 Кбайта. Для больших дисков приходится применять размер кластера 8, 16 или 32 Кбайта.

При размещении файла в непрерывном пространстве ВП информация об его местоположении сводится к двум численным параметрам: номер первого кластера файла и длина файла (в кластерах). Но при размещении файла в разрывном пространстве надо знать, где находится каждый его кластер. В различных ФС такая информация содержится:

- в записи каталога;
- в самих кластерах файла;
- в специальной таблице.

Первый из этих способов предполагает, что запись каталога, соответствующая файлу, содержит перечень номеров всех его кластеров. Недостаток очевиден: размер записи каталога должен зависить от размера файла, что очень неудобно. Поэтому в записи каталога обычно хранится только номер самого первого кластера файла или указатель на место, где этот номер хранится.

Второй способ предполагает, что предыдущий кластер файла содержит номер следующего кластера. В результате все кластеры файла связаны в единый список. Считав из каталога номер первого кластера файла, по цепочке нетрудно прочитать номера всех остальных его кластеров. Основным недостатком данного способа является то, что для доступа к кластеру, расположенному в середине файла, необходимо прочитать все предыдущие кластеры.

Третий способ предполагает, что номера кластеров файла содержатся в специальной таблице. Данный способ отличается от предыдущего тем, что поиск нужного кластера ОС производит не путем считывания многих кластеров носителя, а путем просмотра содержимого одного и того же кластера (в котором находится таблица).

Реализация такого метода в DOS основана на использовании таблицы размещения файлов *FAT* (*File Allocation Table*), название которой используется в качестве названия всей ФС. Таблица *FAT* имеет столько 12-, 16- или 32-битовых элементов, сколько кластеров носителя могут распределяться между файлами. Иными словами, *FAT* представляет собой уменьшенную модель распределяемой части носителя. Ее наличие позволяет размещать файл в разрывной области ВП. Для этого каждому файлу ставится в соответствие вспомогательный линейный связанный список, построенный из элементов таблицы *FAT*.

Вспомним, что одно из полей записи каталога, описывающей файл, содержит его начальный адрес. Этот адрес представляет собой номер первого кластера файла и, следовательно, номер соответствующего элемента в таблице *FAT*. Содержимым этого элемента является номер следующего элемента связанного списка, который совпадает с номером следующего кластера файла. Если элемент *FAT*-таблицы соответствует последнему кластеру файла, то он содержит специальное число (*FFFh*, *FFFFh* или *FFFFFFFFh*).

16.2. Создание и открытие файла

Работа с любым файлом начинается с его создания. Оно выполняется ОС по соответствующей просьбе со стороны программы. Во время ***создания файла*** добавляется новая запись в «родительский» каталог, заданный программой. В эту запись помещается простое имя файла, а также другая информация, соответствующая типу используемой ФС. Например, при включении файла в систему *FAT*, в запись каталога помещаются атрибуты нового файла. (Многие другие файловые системы размещают атрибуты файла не в каталоге, а в своих таблицах.)

Кроме того, при создании файла в ФС со статическим распределением ВП файлу выделяется необходимое пространство на носителе. В ФС с динамическим распределением памяти (к которым относится и *FAT*) память файлу при его создании не выделяется, а его длина принимается нулевой.

Создание файла является необходимым, но недостаточным условием выполнения информационного обмена с ним. Для этого требуется выполнить также ***открытие файла***. Во время этой операции происходит связывание реального физического файла на носителе ВП (или файла-устройства ввода/вывода) с соответствующим логическим файлом.

Программа, по запросу которой ОС выполняет открытие файла, должна сообщить в системном запросе не только имя открываемого физического файла, но и тип последующей работы с ним (чтение или запись). В качестве результата открытия файла в программу возвращается логический номер файла. Рассмотрим теперь внутреннюю работу ОС при открытии файла.

Во-первых, во время открытия файла ОС создает для работы с ним столько буферов, сколько типов операций информационного обмена с ним будут выполняться. Поэтому при открытии файла и на чтение, и на запись, ему выделяются два буфера, а при открытии только на чтение – один. Размер системных буферов кратен одному сектору и обычно равен одному кластеру диска.

Во-вторых, ОС помещает ***БУФ*** (***блок управления файлом***) в свою ***таблицу открытых файлов***. БУФ содержит атрибуты файла (переписываются из каталога), а также указатели на системные буферы для работы с ним.

В-третьих, ОС просматривает ***таблицу логических номеров файлов*** для данной программы и выбирает из этой таблицы первый свободный элемент (содержит *FFh*). В этот элемент помещается номер соответствующего БУФ в таблице открытых файлов. После этого ОС возвращает управление в программу, передав ей в качестве логического номера файла номер элемента в таблице логических номеров. Таким образом, в результате открытия файла создается путь от логического номера файла в программе до физического файла на носителе (рис. 2.20).

Для программы, выполняемой в среде DOS, таблица логических номеров файлов содержится в *PSP* программы, начиная с адреса-смещения *18h*. Размер этой таблицы равен 20 байтам (по одному байту на номер файла). Поэтому общее число файлов, открытых в программе, не может превышать 20.

Из этих номеров первые пять DOS использует сама, открывая для программы (до начала ее выполнения) следующие файлы:

0 – стандартное устройство ввода (обычно клавиатура);

1 – стандартное устройство вывода (обычно экран);

2 – устройство вывода сообщений об ошибках (экран);
3 – последовательный порт (обычно *COM1*);
4 – параллельный порт (обычно *LPT1*).

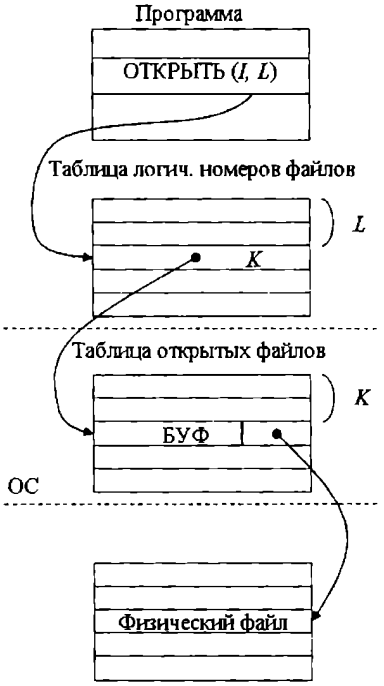

Рис. 2.20. Результат операции открытия файла:
I – имя физического файла; *L* – логический номер файла; *K* – номер БУФ

Нетрудно заметить, что все перечисленные файлы являются файлами-устройствами. Приняв их во внимание, максимальное число файлов, открытых самой программой, не может превышать число 15. При этом следует учесть, что программа наследует от родительской программы все файлы, открытые ею к моменту создания новой программы.

Другим ограничителем числа открытых файлов является строка «*FILES=N*» в командном файле конфигурации системы *Config.sys* (для MS-DOS) или *Fdconfig.sys* (для FreeDOS). Число *N* задает предельное суммарное число файлов, открытых во всей системе. Именно столько строк имеет таблица открытых файлов. Последнее ограничение нетрудно скорректировать, используя любой текстовый редактор.

В отличие от таблицы логических номеров таблица открытых файлов является внутренней таблицей DOS, к которой прикладные программы не имеют непосредственного доступа. Работу с этой таблицей выполняют только подпрограммы ОС, инициируемые в результате системных вызовов. Для создания и открытия файлов используются следующие системные вызовы DOS: «*int 21h*» (функция *3Ch*) и «*int 21h*» (функция *3Dh*).

int 21h (функция ***3Ch***) – создание и открытие файла. Данный вызов создает и открывает новый файл. Если указанный файл уже существует, то он усекается до нулевой длины. Перед применением вызова программа помещает в какую-то область ОП символьную строку, представляющую собой имя создаваемого файла. Эта строка должна заканчиваться нулевым байтом. При этом имя файла может быть задано в любой из до-

пустимых форм: 1) имя-путь, предваряемое именем логического диска; 2) имя-путь; 3) относительное имя файла; 4) простое имя.

В регистры *DS* и *DX* программа должна поместить соответственно адрес-сегмент и адрес-смещение области памяти, содержащей имя файла. В регистр *CX* должна быть помещена битовая строка, задающая особые свойства создаваемого файла. При этом установка каждого бита задает одно особое свойство (атрибут) файла. Например, «бит 1 = 1» требует, чтобы файл был скрытым (его имя не должно выводиться на экран при выполнении утилиты *DIR*). Если создается обыкновенный файл, то все биты-атрибуты должны быть нулевыми.

При успешном завершении системного вызова в программу возвращается флаг *CF=0*. При этом в регистре *AX* возвращается логический номер файла.

int 21h (функция ***3Dh***) – открытие существующего файла. Перед применением вызова программа задает в регистрах *DS* и *DX* адрес имени открываемого файла (аналогично функции *3Ch*). В регистр *AL* она помещает ***режим доступа к файлу***. Младшие биты 0–1 режима доступа задают допустимые операции с открытым файлом:

00 – открыть для чтения;

01 – открыть для записи;

10 – открыть для чтения и для записи.

При успешном завершении системного вызова в программу возвращается флаг *CF=0*, а в регистре *AX* передается логический номер файла.

16.3. Операции чтения и записи

После того как программа открыла файл, она может выполнять информационный обмен с ним, используя операции чтения и записи. Выполняя эти операции с помощью соответствующих системных вызовов, программа рассматривает файл как строку байтов. При этом все байты файла пронумерованы, начиная с нуля. Для того чтобы ОС «знала», к какому месту файла относится следующая команда чтения или записи, используется переменная, называемая ***указателем файла***. Она содержит номер байта файла, начиная с которого файл будет обрабатываться следующей командой.

Сразу после открытия файла его указатель содержит 0. Используя специальный системный вызов, программа может записать в эту переменную любое другое значение, что позволяет выполнять информационный обмен с внутренней частью файла напрямую, не просматривая его предыдущих байтов. В результате завершения операции чтения или записи указатель файла содержит номер байта, расположенного сразу же за строкой, обработанной командой. Поэтому, если программа выполняет последовательное чтение (запись) всего файла, отдельная установка его указателя не требуется.

Каждому открытому файлу всегда соответствует только один указатель, несмотря на то, что файл может быть одновременно открыт в нескольких программах. (Однопрограммная система допускает одновременное нахождение в ОП нескольких программ, одни из которых находятся между собой в отношении «предок–потомок», а другие являются резидентными.) Наследование открытых файлов дочерней программой, а также наличие единых указателей файлов делают файлы удобным средством информационного обмена между родительской и дочерней программами в том случае, если объем передаваемой информации достаточно велик.

При выполнении операции чтения (записи) выполняется копирование строки байтов из файла (из прикладного буфера) в прикладной буфер (в файл). ***Прикладной буфер*** –

область памяти в пространстве данных программы. При этом фактический информационный обмен выполняется, с одной стороны, между файлом на носителе и системным буфером, а с другой – между системным и прикладным буферами. Поэтому очередная операция чтения (записи) необязательно сопровождается фактическим информационным обменом с файлом. Подобный обмен отсутствует, если имеющееся содержимое системного буфера соответствует тому месту файла, которое указано в команде чтения (записи).

Следующие системные вызовы DOS выполняют установку указателя файла, а также используются для информационного обмена с ним: «*int 21h*» (функция *42h*), «*int 21h*» (функция *3Fh*) и «*int 21h*» (функция *40h*).

int 21h (функция ***42h***) – установить указатель файла. Перед применением вызова программа помещает в регистр *BX* логический номер файла, а в регистр *AL* тип требуемого перемещения указателя файла:

0 – перемещение относительно начала файла;

1 – перемещение относительно текущего положения указателя;

2 – перемещение относительно конца файла.

Величина перемещения (в байтах) представляет собой число со знаком, которое программа должна поместить в пару регистров: *CX* (старшая часть величины перемещения) и *DX* (младшая часть).

При успешном завершении системного вызова в программу возвращается флаг *CF=0*, а в регистрах *DX* и *AX* возвращается новое значение указателя файла (число байтов от начала файла).

☑ **Примечание.** В результате выполнения данного вызова указатель может быть установлен не только внутри, но и за пределами файла. Если он принял отрицательное значение, то первая же операция чтения/записи закончится с ошибкой. Если же его величина положительна, но превосходит длину файла, то очередная операция записи увеличит размер файла.

Этот вызов часто используется для определения длины файла: достаточно выполнить вызов с *CX=0*, *DX=0*, *AL=2*. Тогда в *DX:AX* будет возвращена длина файла в байтах.

int 21h (функция ***3Fh***) – выполнить чтение из файла. Чтение начинается с той позиции, на которую ранее был установлен указатель файла. Перед применением вызова программа помещает в регистр *BX* логический номер файла, в регистр *CX* – число байтов, читаемых из файла, а в регистры *DS* и *DX* соответственно адрес-сегмент и адрес-смещение начала прикладного буфера, в который должно быть выполнено чтение.

При успешном завершении системного вызова в программу возвращается флаг *CF=0*, а в регистре *AX* возвращается число фактически считанных байтов. Указатель файла установлен на первый несчитанный байт.

☑ **Примечание.** Если в результате выполнения данного вызова число фактически считанных байтов (в *AX*) меньше, чем число заказанных байтов (в *CX*), то, следовательно, был достигнут конец файла.

int 21h (функция ***40h***) – выполнить запись в файл. Запись начинается с той позиции, на которую ранее был установлен указатель файла. Перед применением вызова программа помещает в регистр *BX* логический номер файла, в регистр *CX* – число байтов, записываемых в файл, а в регистры *DS* и *DX* – соответственно адрес-сегмент и адрес-смещение прикладного буфера, содержимое которого должно быть записано в файл.

При успешном завершении системного вызова в программу возвращается флаг *CF=0*, а в регистре *AX* возвращается число фактически записанных байтов. Указатель файла установлен на байт, следующий за последним записанным байтом.

16.4. Закрытие и уничтожение файла

После того как программа завершила работу с файлом, она может его закрыть. ***Закрытие файла*** – операция, обратная открытию файла. В результате ее выполнения освобождаются ресурсы, выделенные для того, чтобы конкретная программа могла работать с конкретным файлом. Сюда относятся системный буфер (буферы), а также логический номер файла. Если закрываемый файл был открыт ранее на запись, то при его закрытии, во-первых, выполняется копирование системного буфера записи в файл на диске. Во-вторых, корректируются атрибуты файла: дата и время последней модификации, размер файла.

В принципе, все файлы, открытые программой, закрываются при ее завершении с помощью системных вызовов, выполняющих возврат в ОС (а точнее – в родительскую программу). Но следующие два фактора способствуют тому, чтобы программа сама выполняла закрытие файла. Во-первых, так как суммарное число открытых файлов ограничено, то освобождение логического номера файла в результате его закрытия позволяет использовать этот номер для другого файла. Во-вторых, так как при закрытии файла производится сброс буфера записи на диск, то это гарантирует правильность данных в файле при аварийном завершении программы.

Если файл больше не нужен не только для текущего выполнения программы, но и для последующего использования, его следует удалить. ***Удаление файла*** – операция, обратная созданию файла. В ее результате удаляется запись в родительском каталоге, а также освобождается пространство на носителе ВП, выделенное ранее для файла. Заметим, что удаляемый файл обязательно должен быть закрыт. (В противном случае возможны операции информационного обмена с несуществующим файлом.)

Применительно к *FAT* удаление файла выполняется следующим образом. Во-первых, первый символ имени файла в родительском каталоге заменяется кодом *E5h*. Во-вторых, все элементы таблицы *FAT*, соответствующие кластерам файла, помечаются как свободные. Что касается самих кластеров файла, то их содержимое не меняется до тех пор, пока данный кластер не будет распределен другому файлу.

Следующие системные вызовы DOS используются для закрытия и удаления файлов: «*int 21h*» (функция *3Eh*) и «*int 21h*» (функция *41h*).

int 21h (функция ***3Eh***) – закрытие файла. Перед применением вызова программа помещает в регистр *BX* логический номер закрываемого файла. При успешном завершении системного вызова в программу возвращается флаг *CF=0*.

int 21h (функция ***41h***) – удаление файла. Перед применением вызова программа помещает в регистры *DS* и *DX* соответственно адрес-сегмент и адрес-смещение имени файла, которое задается аналогично функции *3Ch*. При успешном завершении системного вызова в программу возвращается флаг *CF=0*.

16.5. Пример программы

Следующая программа выполняет копирование файла. Имя копируемого файла и имя файла-копии программа запрашивает у пользователя сама.

```
;       Программа копирует файл
;       -----------------------
;   Имя копируемого файла и имя копии вводятся с клавиатуры
;
cr          equ     0dh                 ; Код ASCII возврата каретки
lf          equ     0ah                 ; Код ASCII перевода строки
            org     100h
Start:      jmp     Begin               ; Переход через данные
;           Данные программы
Msg1        db          'Введите имя копируемого файла', cr, lf, '$'
Msg2        db          'Копируемый файл отсутствует', cr, lf, '$'
Msg3        db          'Введите имя файла-копии', cr, lf, '$'
Msg4        db          'Ошибка создания файла', cr, lf, '$'
Msg5        db          'Ошибка чтения файла', cr, lf, '$'
Msg6        db          'Ошибка записи в файл', cr, lf, '$'
Msg7        db          'Копирование выполнено', cr, lf, '$'
Lname0      db      81                  ; Максимальная длина имени файла
Lname       db      0                   ; Фактическая длина имени файла
Namefile    times   81      db 0        ; Здесь размещается имя файла
Bufer       times   512     db 0        ; Буфер для копирования
Lognum1     dw      0                   ; Логический номер файла 1
Lognum2     dw      0                   ; Логический номер файла 2
;       Команды программы
Begin:
;   Запрос и открытие копируемого файла
        mov     dx, Msg1                ; Вывод сообщения с просьбой
        mov     ah, 9                   ;   ввести имя
        int     21h                     ;   копируемого файла
        mov     ah, 0ah                 ; Функция ввода строки
        mov     dx, Lname0              ; DX ← адрес-смещение буфера ввода
        int     21h                     ; Ввод строки
        xor     bx, bx                  ; BX ← 0
        mov     bl, [Lname]             ; BX ← длина имени файла
        mov     [Namefile+bx], byte 0   ; Запись нулевого байта
        mov     ah, 3dh                 ; Открытие
        mov     dx, Namefile            ;   копируемого
        mov     al, 0                   ;   файла
        int     21h                     ;   на чтение
        jnc     .M1                     ; Если успешно
        mov     dx, Msg2                ; Вывод сообщения
        mov     ah, 9                   ;   об отсутствии
        int     21h                     ;   копируемого файла
        jmp     .Exit                   ; На завершение программы
.M1:    mov     [Lognum1], ax           ; Сохранение логич. номера файла
    Создание и открытие файла-копии
        mov     dx, Msg3                ; Вывод сообщения с просьбой
```

```
        mov   ah, 9                    ;   ввести имя
        int   21h                      ;   файла-копии
        mov   ah, 0ah                  ; Функция ввода строки
        mov   dx, Lname0               ; DX ← адрес-смещение буфера ввода
        int   21h                      ; Ввод строки
        xor   bx, bx                   ; BX ← 0
        mov   bl, [Lname]              ; BX ← длина имени файла
        mov   [Namefile+bx], byte 0    ; Запись нулевого байта
        mov   ah, 3ch                  ; Создание
        mov   dx, Namefile             ;   и
        mov   cx, 0                    ;   открытие
        int   21h                      ;   файла-копии
        jnc   .M2                      ; Если успешно
        mov   dx, Msg4                 ; Вывод сообщения
        mov   ah, 9                    ;   об ошибке
        int   21h                      ;   создания файла
        jmp   .Exit                    ; На завершение программы
.M2:    mov   [Lognum2], ax            ; Сохранение логич. номера файла
;   Копирование файла
.M3:    mov   ah, 3fh                  ; Функция чтения из файла
        mov   bx, [Lognum1]            ; BX ← логич. номер читаемого файла
        mov   cx, 512                  ; CX ← число читаемых байтов
        mov   dx, Bufer                ; DX ← адрес-смещение буфера
        int   21h                      ; Чтение из файла в буфер
        jnc   .M4                      ; Если успешно
        mov   dx, Msg5                 ; Вывод сообщения
        mov   ah, 9                    ;   об ошибке
        int   21h                      ;   чтения файла
        jmp   .Exit                    ; На завершение программы
.M4:    mov   bx, [Lognum2]            ; BX ← логич. номер файла-копии
        mov   cx, ax                   ; CX ← число записываемых байтов
        mov   ah, 40h                  ; Функция записи в файл
        mov   dx, Bufer                ; DX ← адрес-смещение буфера
        int   21h                      ; Запись из буфера в файл
        jnc   .M5                      ; Если успешно
        mov   dx, Msg6                 ;  Вывод сообщения
        mov   ah, 9                    ;   об ошибке
        int   21h                      ;   записи в файл
        jmp   .Exit                    ; На завершение программы
.M5:    cmp   ax, 512                  ; Конец файла?
        jz    .M3                      ; Если нет
;   Завершение программы
        mov   dx, Msg7                 ; Вывод сообщения
        mov   ah, 9                    ;   о  завершении
        int   21h                      ;   копирования
.Exit:  mov   ax, 4c00h                ; Возврат в DOS с
        int   21h                      ;   кодом завершения 0
```

Данная программа сначала запрашивает у пользователя имя копируемого файла, используя для ввода символьной строки с клавиатуры системный вызов DOS – «***int 21h***» (функция ***0Ah***) – ввод с клавиатуры символьной строки. При этом в качестве последнего символа строки вводится символ *0Dh* (возврат каретки). Этот символ помещается во вводимую строку драйвером клавиатуры при нажатии клавиши <Enter>.

Перед применением вызова программа помещает в регистры *DS* и *DX* соответственно адрес-сегмент и адрес-смещение для младшего байта буфера, в который должна быть введена строка. Данный буфер имеет три поля:

1. Младший байт содержит максимальное число вводимых символов. Запись этого числа выполняет сама программа перед применением данного вызова;

2. Второй байт содержит число фактически введенных символов. Запись в этот байт выполняет DOS;

3. Остальные байты буфера содержат введенную строку, заканчивающуюся кодом *0Dh*.

Далее программа открывает копируемый файл, если он существует. Иначе соответствующее сообщение выводится на экран пользователю. В случае успешного открытия файла его логический номер, полученный в результате открытия, сохраняется для последующего чтения из файла. Далее создается новый файл, имя которого вводится с клавиатуры. Одновременно этот файл открывается для последующей записи в него.

Собственно копирование выполняется циклически. За одну итерацию цикла одна логическая запись (512 байт) считывается из исходного файла в буфер программы *Bufer*, а затем переписывается из него в выходной файл. После этого число фактически записанных байтов сравнивается с числом 512. При неравенстве этих чисел делается вывод о достижении конца файла, и копирование прекращается.

Обратим внимание, что программа не выполняет закрытие файлов. Это объясняется, во-первых, небольшим количеством файлов, открываемых в программе. Во-вторых, вскоре после последней операции записи в файл программа завершается.

Заслуживает внимания оператор:

```
mov  [Namefile+bx], byte 0
```

Данный оператор выполняет загрузку числа 0, заданного в качестве второго операнда, в ячейку ОП, являющейся первым операндом. Адрес-смещение этой ячейки задан выражением в квадратных скобках и представляет собой сумму адреса-смещения байта, имеющего имя *Namefile* и содержимого регистра *BX*. Если бы квадратных скобок не было, то транслятор интерпретировал бы первый операнд не как содержимое ячейки ОП, а как ее адрес.

Для уточнения второго операнда в записанном операторе используется вспомогательный псевдооператор ***byte*** (байт). При отсутствии такого вспомогательного оператора транслятор не знает, какова длина пересылаемого данного – байт, слово или двойное слово. Оператор *byte* или ***word*** (слово), или ***dword*** (двойное слово) будет лишним, если хотя бы один из операндов есть регистр, так как длина регистра всегда известна транслятору.

Рассмотренная программа пригодна для копирования любых обычных (не каталогов) файлов. Например, она может копировать себя. Допустим, что мы поместили загрузочный модуль программы в файл *copir.com*, тогда выполнив копирование этого файла в файл с именем, например, *copir1.com,* получим программу, которая также может выполнять копирование (проверьте это).

16.6. Другие операции с файлами

Рассмотренные выше системные вызовы для работы с файлами являются основными, но не единственными Кратко перечислим некоторые другие функции системного вызова «*int 21h*», используемые в DOS для работы с обычными файлами (не каталогами), имеющими короткие двенадцатибайтовые имена:

43h – получить или установить атрибуты файла;

56h – переименовать файл;

57h – получить или установить дату и время создания файла.

Кроме того, DOS предоставляет отдельные системные вызовы для работы с логическими дисками и каталогами:

0Eh – установить текущий логический диск;

19h – получить текущий логический диск;

39h – создать каталог;

3Ah – удалить каталог;

3Bh – установить текущий каталог;

47h – получить текущий каталог.

DOS современных версий предоставляют также системные вызовы для работы с файлами в *FAT32*, используя их длинные имена. Так как эти же файлы имеют и короткие двенадцатибайтовые имена, то с ними могут работать и ранее рассмотренные системные вызовы.

16.7. Модернизация редактора для работы с файлами

16.7.1. Алгоритм диспетчера

Сейчас мы приступим к модернизации разработанного ранее редактора информации так, чтобы он был пригоден для редактирования не только информации в сегменте ОП, занимаемого программой, но и для редактирования файлов на магнитных дисках.

Модернизацию редактора мы начнем с центрального модуля, т. е. с диспетчера. Его новый алгоритм приведен на рис. 2.21. Как видно из этого рисунка, существенное развитие получила начальная фаза выполнения диспетчера, после успешного завершения которой должен выполняться прежний алгоритм диспетчера. Напомним, что прежний алгоритм диспетчера решает задачу распознавания команды пользователя в виде нажатия одной из клавиш <Fi> и инициирования того модуля редактора, который выполняет данную команду.

Начальная фаза нового диспетчера начинает с того, что опрашивает пользователя о его желании редактировать какой-то файл на диске. В случае его отказа сбрасывается в ноль признак работы с файлом – глобальная переменная длиной 1 байт с именем *Pf*. А далее выполняется прежний алгоритм диспетчера. В случае согласия пользователя редактировать файл ему выдается просьба ввести имя этого, уже существующего файла. Имя файла записывается в качестве содержимого переменной *Namefile*.

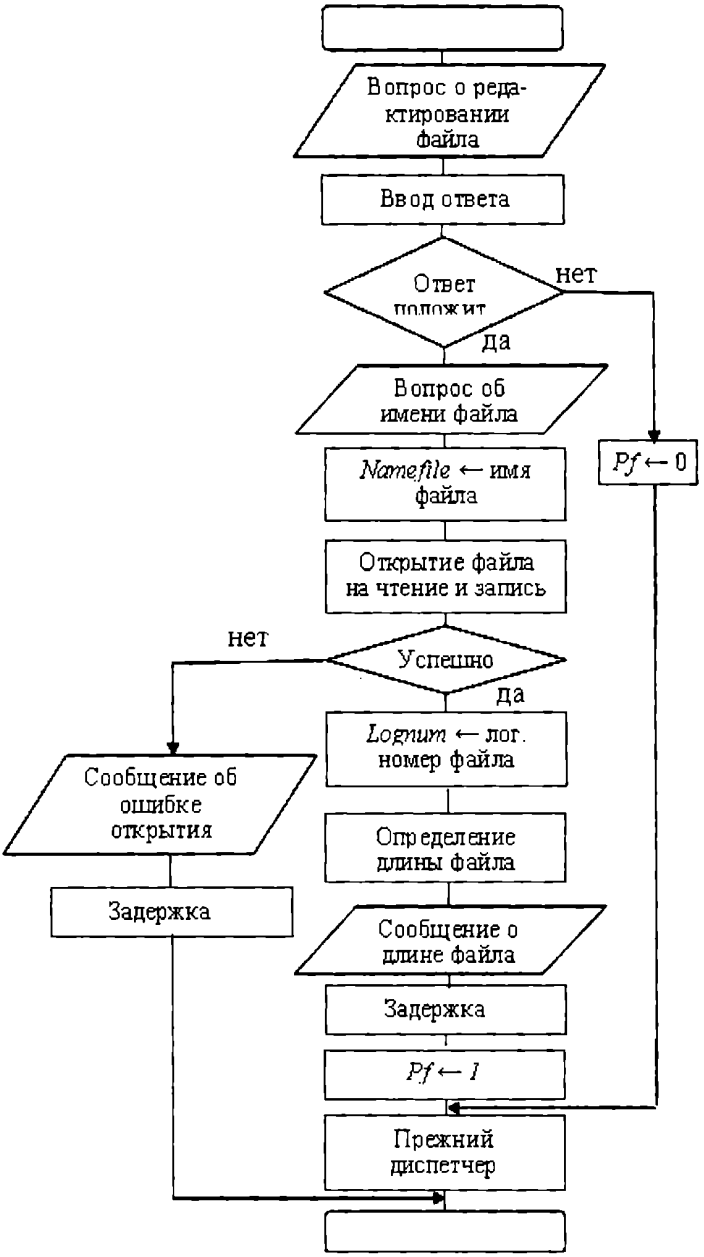

Рис. 2.21. Алгоритм диспетчера для работы с файлами

Возможная реализация первых двух этапов алгоритма:

```
call    Clear_screen    ; Очистка экрана
mov     dx, Msg         ; DX ← адрес строки сообщения
mov     ah, 9           ; Вывод строки
```

```
int     21h         ;  на экран
mov     ah, 1       ; Ввод символа
int     21h         ;  с клавиатуры
or      al, 20h     ; Перевод символа на нижний регистр
cmp     al, 'y'     ; Введен символ 'y' или 'Y' ?
jnz     .M          ; Если нет
```

В случае согласия с вопросом программы пользователь должен ввести «*y*» или «*Y*». Так как коды этих двух символов (*79h* и *59h*) отличаются только одним битом 5, то установка этого бита командой *or* преобразует код символа «*Y*» в код символа «*y*». Напомним, что команда логического сложения *or* сравнивает оба операнда побитно. Если хотя бы один из сравниваемых битов содержит «1», то в результирующий бит также записывается «1». Результат сравнения (слово или байт) записывается на место первого операнда.

Далее производится открытие указанного пользователем файла на чтение и на запись. Если открытие произошло успешно, то логический номер файла записывается в переменную (слово) *Lognum*. Далее определяется значение переменной (слово) *Lf* – длина открытого файла в секторах по 256 байт. Эта величина находится в два этапа. На первом этапе определяется длина файла в байтах (используя системный вызов установки указателя файла). На втором этапе длина файла в байтах (находится в регистрах *DX* и *AX*) делится на длину сектора (256 байтов). Для того чтобы был учтен и последний сектор файла, перед делением необходимо длину файла в байтах увеличить на число 255. Возможная реализация данного этапа:

```
xor     cx, cx      ; Величина перемещения указателя
xor     dx, dx      ;  файла – 0
mov     bx, [Lognum] ; Установка
mov     al, 2       ;  указателя
mov     ah, 42h     ;  на конец
int     21h         ;  файла
add     ax, 255     ; Сумма младшей части длины и 255
adc     dx, 0       ; Сумма старшей части длины и (CF)
mov     bx, 256     ; BX ← длина сектора
div     bx          ; Деление (DX:AX) на BX
mov     [Lf], ax    ; Lf ← длина файла в секторах
```

Обратите внимание на использование оператора ***adc*** – суммирование с флагом переноса *CF*. Он применяется для того, чтобы учесть возможный перенос при суммировании младшей части длины файла (в *AX*) и числа 255. В результате единица переноса (*CF = 1*) суммируется со старшей частью длины.

После того, как на экран будет выведено сообщение о длине файла в секторах (по 256 байт), выполняется задержка выполнения программы (с помощью ввода символа с клавиатуры). Подобная задержка позволяет увидеть сообщение программы прежде, чем будет выполнена очистка экрана.

После этого в переменную *Pf* записывается значение 1. Далее выполняется прежний алгоритм диспетчера.

16.7.2. Алгоритмы процедур

Естественно, что переделка редактора с целью обработки файлов не сводится лишь к модернизации одного диспетчера команд. Для этого необходимо также внести измене-

ния в алгоритмы двух процедур – *Read_sector* и *Write_sector*. Напомним, что процедура *Read_sector* копирует в переменную *Sector* содержимое сектора памяти, начальный адрес которого находится в переменной *Address*. Процедура *Write_sector*, наоборот, копирует содержимое переменной *Sector* в область памяти, начальный адрес которой в переменной *Address*. Ранее процедуры *Read_sector* и *Write_sector* использовали в качестве области памяти сегмент ОП, принадлежащий самой выполняемой программе. Теперь в зависимости от значения переменной *Pf* они должны работать не только с сегментом ОП, но и с файлом на магнитном диске. При этом независимо от типа используемой памяти, начальный адрес копируемого сектора относительно начала памяти по-прежнему содержится в переменной *Address*.

Алгоритмы модифицированных процедур *Read_sector* и *Write_sector* представлены на рис. 2.22 и рис. 2.23. Выполнение этих процедур начинается с проверки значения переменной *Pf*. Если это значение равно 0, то выполняются прежние алгоритмы этих процедур, предназначенные для работы с сегментами ОП. Если же содержимое *Pf* равно единице, то процедура *Read_sector* выполняет ввод в переменную *Sector* заданного сектора файла. А процедура *Write_sector* выводит содержимое переменной *Sector* на диск в качестве содержимого сектора файла.

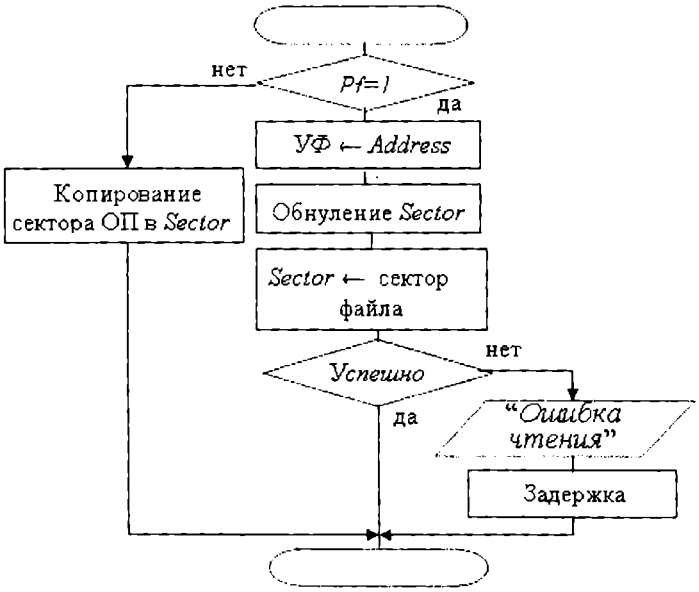

Рис. 2.22. Алгоритм процедуры *Read_sector*. *УФ* – указатель файла

Если при выполнении процедуры *Read_sector* в переменную *Sector* был считан последний сектор файла, то, скорее всего, этот фрагмент файла имел длину менее 256 байт. Если после редактирования содержимого *Sector* была выполнена запись этой информации в файл (вызов *Write_sector* в результате нажатия <F2>), то длина файла возрастет и будет кратна 256 байтам.

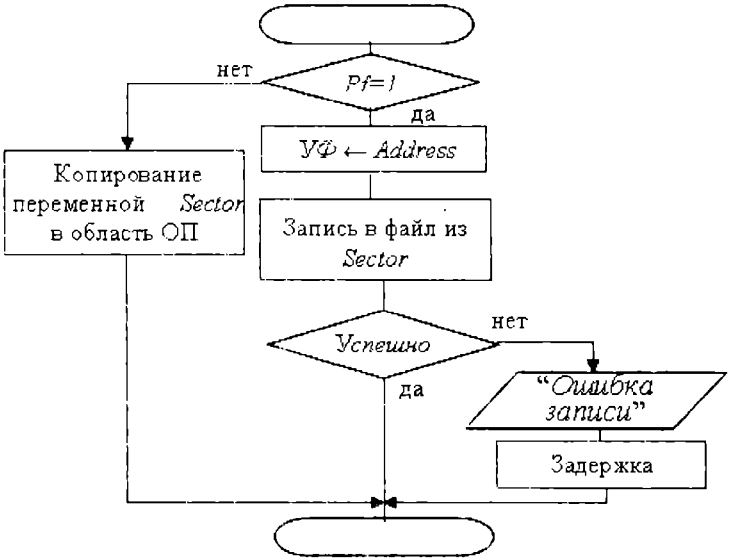

Рис. 2.23. Алгоритм модифицированной процедуры *Write_sector* *УФ* – указатель файла

Еще более интересный эффект имеет место, если содержимое *Sector* записывается в файл после выполнения процедуры *N_sector*, установившей величину *N*, превышающую первоначальную длину файла. В этом случае новая длина файла составит *N+1* секторов по 256 байт. При этом все новые байты файла будут содержать нули, за исключением байтов, записанных в ходе редактирования. Подобное свойство является следствием рассмотренного ранее явления изменения длины файла в результате установки указателя файла в позицию, превышающую конец файла.

Лабораторная работа 14

В ходе выполнения данной лабораторной работы требуется осуществить изменения в редакторе так, чтобы он позволял выполнять редактирование информации не только в сегменте ОП, но и информации, содержащейся в файле на магнитном диске.

17. Раздельная трансляция программы

17.1. Префикс программного сегмента

Что касается машинной программы, записанной загрузчиком в ОП, то кроме собственно машинных команд программы и обрабатываемых ею данных она содержит вспомогательную информацию, называемую ***префиксом программного сегмента*** (***PSP***). Длина *PSP* 256 (*100h*) байт, и эта структура данных находится в начале любой машинной программы, вне зависимости от того, получена ли программа загрузкой *com-* или *exe*-файла. При этом ни *com-*, ни *exe*-файл *PSP* не содержит. Информация в *PSP* записывается загрузчиком и используется ОС при выполнении запросов со стороны программы, а также может быть использована самой программой. Структура *PSP* на рис. 2.24.

Относит. адрес (16)	
00h	Команда *int 20h*
02h	Верхняя граница памяти программы
04h	Зарезервировано
05h	Вызов диспетчера функций
0Ah	Векторы прерываний *22h* – *24h*
16h	Адрес-сегмент *PSP* родительской программы
18h	Таблица логич. номеров файлов данных
2Ch	Адрес-сегмент блока окружения
2Eh	Зарезервировано
5Ch	2 блока управления файлами
80h	Хвост команды

Рис. 2.24. Структура *PSP*

Для наглядного знакомства со структурой *PSP* удобно использовать отладчик Debug. Допустим, что мы ранее получили загрузочный модуль программы с именем *Prob.com*. Тогда для анализа *PSP* вызовем Debug командой DOS: «Debug *Prob.com parametr*», где вместо слова *parametr* может быть записана любая символьная строка (в коде ASCII), содержащая входные параметры нашей прикладной программы. Эти параметры совершенно не интересуют ни Debug, ни тем более DOS, а используются лишь для передачи в программу какой-то исходной информации. (Кстати, если рассматривать в качестве программы сам Debug, то в качестве его параметра выступает имя исследуемой программы, т. е. *Prob.com*.)

После того как мы запустили Debug описанным выше способом, с помощью команды Debug – «*d 0*» получим на экране дамп начальной части *PSP*. В результате на экран будут выведены первые 128 байт *PSP*. Для вывода на экран каждых следующих 128 байт достаточно просто ввести команду *d*.

В первых двух байтах *PSP* находится код машинной команды «*int 20h*». (Чтобы убедиться в этом, введем команду Debug – «*u 0*».) Команда программного прерывания выполняет возврат в ОС при завершении программы. Сама же эта команда получает управление следующим образом.

Во-первых, сразу после загрузки программы в память загрузчик помещает в ее стек 16-битовое нулевое слово (*0000h*). Во-вторых, если в конце главной подпрограммы прикладной программы стоит команда *ret*, то при попадании этой команды на ЦП, в качестве адреса возврата из стека будет выбран 0 и, следовательно, следующей исполняемой командой будет «*int 20h*». Естественно, что это не единственный способ возвращения из программы в ОС. Например, вместо *ret* в главной подпрограмме можно записать оператор «*int 20h*». В этом случае управление из программы возвращается в ОС непосредственно, минуя *PSP*. Наиболее предпочтительный способ возврата – запись в конце главной подпрограммы команды «*int 21h*» (функция *4Ch*). Он позволяет возвращать из про-

граммы в ОС код возврата (в регистре *AL*), информирующий операционную систему о причине завершения программы.

В следующих двух байтах *PSP* находится верхняя граница (адрес-сегмент) ОП. Обычно она равна величине *A000h*. В дальнейшем мы еще вернемся к этой величине, когда будем рассматривать распределение ОП.

В 5 байтах *PSP*, начиная с *05h*, находится вызов диспетчера функций DOS. Данный вызов представляет собой команду дальнего вызова процедуры. С помощью команды Debug – «*u 5*» мы можем получить на экране, например, следующее: «*call F01D:FEF0*», где *F01D* – адрес-сегмент, а *FEF0* – адрес-смещение первой команды диспетчера функций DOS. (Соответствующий реальный адрес интересен тем, что он находится за пределами первого мегабайта. Мы вернемся к нему в вопросе о распределении памяти.)

Для того чтобы обратиться из программы к DOS (с целью получения помощи), достаточно поместить в нее команду «*call 5*». В результате следующей командой, исполняемой на ЦП, будет команда вызова диспетчера функций. (Данный способ обращения к DOS менее предпочтителен по сравнению с командой программного прерывания «*int 21h*».)

В 12 байтах *PSP*, начиная с *0Ah*, содержатся копии векторов прерываний с номерами *22h*, *23h*, *24h*. Эти прерывания DOS использует для управления выполнением программы. При этом прерывание *22h* используется для вызова той подпрограммы, которая должна получить управление при завершении прикладной программы. Прерывание *23h* происходит при нажатии клавиш <Ctrl>+<C>. (DOS использует для обработки этой комбинации специальный обработчик.) Причинами прерывания *24h* являются аппаратные ошибки при работе с ПУ. Некоторые из этих ошибок: 1) попытка записи на защищенный диск; 2) нет бумаги на принтере; 3) неизвестное устройство. Хранение копий векторов перечисленных прерываний обусловлено тем, что в случае, если прикладная программа изменяет содержимое векторов (с целью выполнить свою обработку прерываний), то после ее завершения DOS восстанавливает прежнее содержимое векторов, используя копии из *PSP*.

По адресу *16h* находится адрес-сегмент *PSP* родительской программы. Назначение этого поля будет понятно позже. Следующее поле *PSP* начинается по адресу *18h* и имеет длину 20 байт. В нем размещена таблица логических номеров файлов, которая была рассмотрена в предыдущем разделе.

Начиная с ячейки *2Ch*, два байта *PSP* содержат адрес-сегмент блока окружения программы. ***Блок окружения*** – совокупность переменных окружения. ***Переменная окружения*** – символьная строка в коде ASCII, имеющая структуру:

<имя переменной> = <первое значение>; ... ;<последнее значение>*00h*

После последней переменной окружения записывается не один, а два нулевых байта (*0000h*). Пример переменной окружения:

PATH = C:\Windows ; C:\Windows \COMMAND00h

Данная переменная задает те каталоги (директории), в которых могут находиться используемые программой файлы. В том случае, если на вход программы поступает не имя-путь файла, а лишь завершающая часть этого имени, то для получения имени-пути программа должна последовательно просмотреть заданные значения переменной окружения *PATH*. В случае обнаружения в очередном каталоге искомого файла, имя-путь каталога соединяется с именем файла в единое имя-путь файла.

Допустим, что с помощью команды Debug – «*d 0*» мы прочитали значение адреса-сегмента блока окружения, равное 2065. Тогда для получения на экране содержимого блока окружения следует воспользоваться командой «*d 2065:0*». Первоначальное содержимое своего блока окружения программа получает «в наследство» от той программы, которая создает данную программу, используя системный вызов *EXEC* (*int 21h*, функция *4Bh*). Данный вызов будет рассматриваться позже. А сейчас лишь заметим, что программа может не только читать свои переменные окружения, но и вносить в них изменения.

По адресу *5Ch* находятся два блока управления файлами, используемые при работе с линейными файловыми структурами. Так как эти структуры сейчас полностью вытеснены иерархическими файловыми структурами, соответствующее поле *PSP* может использоваться для хранения любой другой информации. Например, учитывая, что подобно блоку окружения данные блоки управления передаются «по наследству», родительская программа может помещать в них исходные данные для дочерней программы. Другой способ передачи исходных данных в программу заключается в использовании «хвоста» команды.

Хвост команды – остаток команды ОС (после имени программы), запустившей прикладную программу. Иными словами, хвост содержит параметры команды, а также пробелы, которые были набраны в командной строке. В приведенном выше примере хвостом является слово *parametr*, предваряемое одним пробелом. Для размещения хвоста используется поле в *PSP*, начиная с ячейки *81h*. Длина хвоста содержится в байте *80h*. (Убедитесь в этом, используя Debug.)

Пример. Следующая простая *com*-программа выводит на экран свои параметры (хвост команды).

```
;
;       Программа выводит на экран свои параметры
;       ------------------------------------------
        absolute 80h            ; Следующий байт имеет адрес-смещение 80h
Lparam  resb  1                 ; Длина хвоста команды
Param   resb  1                 ; Первый байт хвоста
        segment    code         ; Виртуальный сегмент кода
        org   100h
Start:  xor   cx, cx            ; Обнуление CX
        mov   cl, [Lparam]      ; CL ← Длина хвоста
        cmp   cx, 0             ; Хвост отсутствует ?
        jz    .Exit             ; Если да
        mov   ah, 0eh           ; 0Eh – функция вывода символа
        mov   bh, 0             ; Нулевая страница экрана
        xor   si, si            ; SI ← 0
.Next:  mov   al, [Param+si]    ; Вывод следующего
        int   10h               ;   символа
        inc   si
        loop  .Next             ; Повторить для след. символа
.Exit:  mov   ax, 4c00h         ; Возврат в DOS
        int   21h               ;   с кодом завершения 0
```

В начале данной программы псевдооператор ***absolute*** создает виртуальный сегмент данных. При этом параметр *80h* задает его расположение относительно начала сегмента памяти программы. Термин ***виртуальный сегмент исходной программы*** означает, что

данный сегмент создается в исходной программе явно, с использованием специально предназначенных для этого псевдооператоров *absolute* или *segment*. В результате трансляции каждый виртуальный сегмент исходной программы преобразуется в ***виртуальный сегмент машинной программы***. Здесь прилагательное «виртуальный» означает отсутствие каких-то особых машинных команд, выделяющих данный сегмент. Напомним, что псевдооператоры, в том числе, *absolute* и *segment* ни в какие машинные команды не транслируются. Далее термин «виртуальный сегмент» означает, в зависимости от контекста, или виртуальный сегмент исходной программы, или виртуальный сегмент машинной программы.

В отличие от виртуальных сегментов, логические сегменты машинной программы более «осязаемы». ***Логический сегмент*** – участок машинной программы, начальный параграф которого загружен в один из сегментных регистров (*CS, DS, ES, SS*). Если на начало логического сегмента указывает *CS*, то это – сегмент кода, если *DS* – основной сегмент данных, если *ES* – дополнительный сегмент данных, *SS* – сегмент стека. В отличие от начала логического сегмента, окончание этого сегмента неизвестно центральному процессору. Известна лишь предельная длина логического сегмента – 64 К.

Виртуальный сегмент данных в рассматриваемой программе включает данные, объявленные с помощью псевдооператоров ***resb***. Данный псевдооператор резервирует последовательность байтов ОП, присваиваивая первому байту заданное символьное имя (в примере это *Lparam* или *Param*). Число резервируемых байтов задается параметром оператора *resb*. В примере это число равно 1. Отличием псевдооператора *resb* от другого псевдооператора резервирования байтов *db* в том, что транслятор не производит инициализацию (заполнение) выделенных байтов. Аналогичное отличие имеет оператор резервирования слов ***resw*** от оператора *dw*.

Виртуальный сегмент, начинающийся с оператора *absolute*, закончится тогда, когда в программе начнется следущий виртуальный сегмент. Его начало задается или другим оператором *absolute*, или оператором *segment*. Этот псевдооператор будет рассмотрен нами позже, при рассмотрении *exe*-программы. Пока лишь заметим, что если расположить наш виртуальный сегмент данных в конце программы, то оператор *segment* становится необязательным.

Выполните два варианта приведенной выше программы – с использованием оператора *segment* и без него.

Заметим, что в данной программе для вывода символа на экран используется системный вызов BIOS – ***int 10h*** (функция ***0Eh***). Предварительно программа помещает код выводимого символа в регистр *AL*.

17.2. Получение прикладной программы

В п. 9.1 отмечалось, что преобразование виртуальной прикладной программы, записанной на языке программирования, в реальную машинную программу выполняет системная обрабатывающая программа, называемая лингвистическим процессором. На самом деле для такого преобразования требуются три лингвистических процессора – транслятор, редактор связей и загрузчик. В принципе, их можно объединить в единую программу, которая, естественно, также будет лингвистическим процессором. Именно так устроены современные ***среды программирования***, позволяющие «выполнять» виртуальные программы. Кроме перечисленных программ они могут включать также текстовый редактор для набора программ, а также отладчик. Вне зависимости от того, ис-

пользуются ли перечисленные лингвистические процессоры и утилиты по отдельности, или в виде модулей среды программирования, общая схема получения и выполнения прикладной программы остается прежней (рис. 2.25).

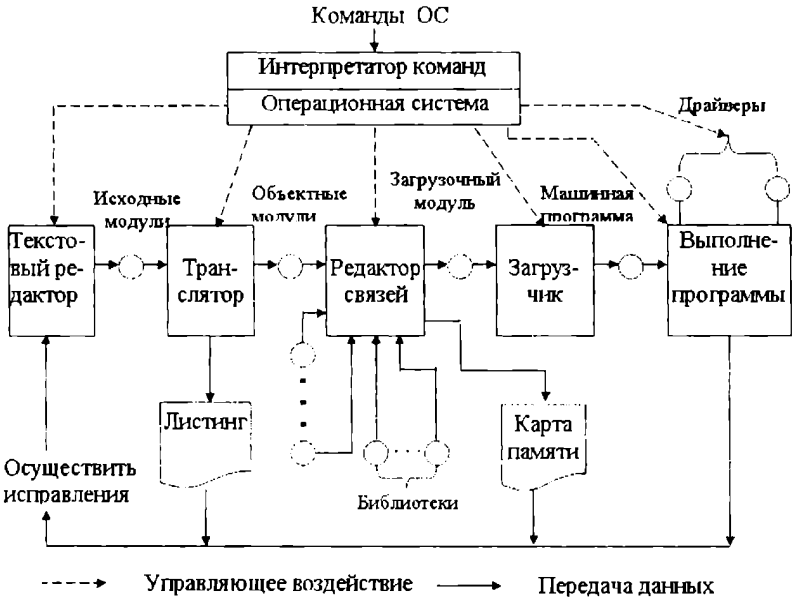

Рис. 2.25. Общая схема создания и выполнения программы

Преобразование файлов, образующих исходную программу в машинную программу, схематично показано на рис. 2.26.

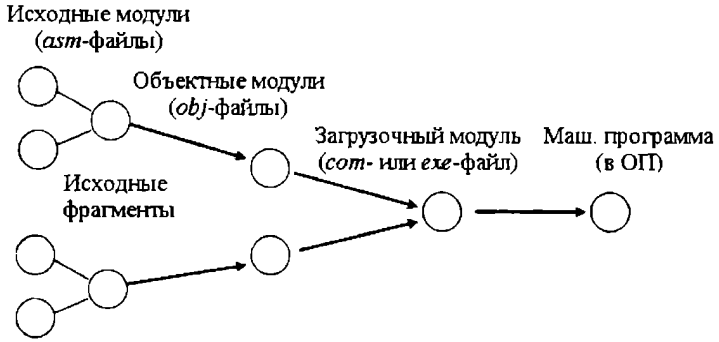

Рис. 2.26. Преобразование файлов программы

На первом этапе в диалоге с текстовым редактором программист вводит в память ВС текст исходной программы. Текст исходной программы состоит из одного или нескольких фрагментов исходной программы. В свою очередь, ***фрагмент исходной программы*** состоит из одного или нескольких текстовых файлов, называемых ***исходными модулями***. Эти исходные модули связываются в единый фрагмент исходной программы с помощью операторов *%include*. Программист сам определяет, как разбить свою исходную программу на модули и фрагменты. Более того, он может программировать свои исходные фрагменты на разных языках.

Исходный фрагмент является входными данными для транслятора того языка, на котором написан данный фрагмент исходной программы. В результате одного выполнения транслятора исходный фрагмент преобразуется в ***объектный программный модуль***.

Все полученные до сих пор нами программы были небольшими. При этом каждая исходная программа представляла собой единственный исходный фрагмент, обрабатываемый транслятором ассемблера за одно его выполнение. Получаемый при этом транслятором объектный модуль был нам не нужен, и поэтому мы просили транслятор о выдаче готового загрузочного модуля в виде *com*-файла. Такая просьба не вызывала затруднений у транслятора, так как единственный объектный модуль программы не сильно отличается от ее загрузочного модуля. Другое дело, если исходная программа состоит из нескольких фрагментов, каждый из которых требует, естественно, своего выполнения транслятора. В этом случае выдача загрузочного модуля транслятором в принципе невозможна по следующим причинам.

При обработке транслятором фрагмента исходной программы частично решается задача получения последовательности машинных команд, отображающих алгоритм исходного фрагмента. Транслятор окончательно записывает коды операций машинных команд, а также проставляет в эти команды номера используемых регистров. Что касается символьных имен, то они обрабатываются транслятором по-разному.

Рассмотрим преобразование адресов транслятором. ***Адрес*** – место в ОП, которое займет соответствующий программный объект – команда или данное. Любой язык программирования позволяет программисту использовать в программе не адреса, а их заменители – ***символьные имена*** (***метки***). Основные типы меток: 1) метки операторов; 2) имена переменных; 3) имена процедур (имя процедуры заменяет для программиста адрес, по которому первая команда процедуры находится в ОП).

Все символьные имена в исходном фрагменте делятся на внешние и внутренние. Внутреннее символьное имя определено и используется в том фрагменте исходной программы, который «видим» транслятором в конкретном его выполнении. Такой «видимый» фрагмент исходной программы образуют главный исходный модуль (исходный файл), указанный в качестве операнда команды ОС, запускающей транслятор, а также исходные модули, заданные с помощью операторов *%include* как в главном исходном файле, так и в связанных с ним с помощью операторов *%include* файлах.

Все внешние метки делятся на входные и выходные. ***Внешние выходные метки*** определены внутри данного фрагмента исходной программы, «видимого» транслятором, а используются вне его: 1) имена процедур, входящих в состав данного исходного фрагмента, но которые могут вызываться в других фрагментах исходной программы; 2) имена переменных, которые определены в данном исходном фрагменте, а используются вне его. ***Внешние входные метки*** определены вне данного исходного фрагмента, а используются в нем: 1) имена процедур, не входящих в данный фрагмент программы, но используемых в нем; 2) имена переменных, которые используются в исходном фрагменте, но определены вне его.

При получении объектного модуля, вместо внутренних и внешних выходных меток транслятор проставляет в те машинные команды, которые используют соответствующие адреса, или смещения относительно текущего содержимого указателя команд *IP*, или смещения относительно начала сегмента ОП, в котором находятся соответствующие программные объекты. В первом из этих вариантов расчет смещения относительно начала сегмента памяти производится в момент выполнения данной машинной команды на ЦП. Что касается внешних входных меток, то обработать их транслятор не может.

Он ничего не знает о размещении соответствующих программных объектов в памяти, так как имеет в своем распоряжении тот фрагмент исходной программы, в котором этих объектов нет. Дальнейшее преобразование программы выполняет системная программа, называемая редактором связей.

Редактор связей (компоновщик) связывает («сшивает») все объектные модули программы в единый ***загрузочный модуль***. При получении загрузочного модуля редактор связей записывает объектные модули один за другим в ОП. Поэтому он «знает», где расположен в памяти каждый программный объект. Следовательно, он может заменить все внешние метки, оставленные транслятором, на соответствующие численные адреса (адреса-сегменты и адреса-смещения). В конце своей работы редактор связей записывает загрузочный модуль в файл на магнитном диске и выдает программисту сообщение, называемое ***картой памяти***. Эта карта описывает распределение памяти машинной программе.

В операционной системе DOS допускается существование двух типов загрузочных модулей – типа *com* и типа *exe* (тип модуля совпадает с расширением имени файла, содержащего загрузочный модуль). Загрузочный модуль типа ***com*** применяется для создания небольших машинных программ (объемом не более 64 К), а типа ***exe*** – для создания любых программ. Основное различие между этими загрузочными модулями заключается в том, что ***com–файл*** содержит готовую машинную программу, не требующую «доводки» при загрузке в память, а ***exe–файл*** содержит «заготовку» машинной программы, а также служебную информацию, используемую загрузчиком для настройки программы.

Как говорилось в п. 9.1, пользователь осуществляет запуск прикладных и системных обрабатывающих программ с помощью интерпретатора команд ОС. (ИК для DOS существует в виде отдельного загрузочного модуля *Command.com*.) В диалоге с ним пользователь сообщает имя загрузочного модуля требуемой программы. В ответ интерпретатор команд вызывает системную программу – ***загрузчик***, передав ему имя загрузочного модуля.

Загрузчик переписывает загрузочный модуль из ВП в ОП и, возможно, настраивает некоторые адреса в полях машинных команд. ***Настройка*** – изменение адреса в зависимости от фактического расположения машинной программы в ОП. Нетрудно предположить, что при загрузке *exe*-файла настройка производится, а для *com*-файла – не производится.

17.3. Программа типа COM

Такая программа получается в результате загрузки *com*-файла, которая сводится к простому переписыванию содержимого этого файла с диска в ОП, а также к записи в начале сегмента памяти, занимаемого программой, ее *PSP*.

На рис. 2.27 приведено содержимое сегментных регистров и регистров-указателей *IP* и *SP* после загрузки *com*-программы в момент передачи ей управления. (Собственно передача управления заключается в записи в *CS* и *IP* тех значений, которые соответствуют первой исполняемой команде программы.) В этот момент все сегментные регистры содержат одно и то же – начальный адрес (а точнее – номер начального параграфа) области ОП, в которую загружена программа. Так как программа начинается с *PSP*, то, следовательно, номер начального параграфа *PSP* и содержится в сегментных регистрах *CS*, *DS*, *ES* и *SS*.

Первоначальное содержимое указателя команд *IP* задает адрес-смещение первой исполняемой команды программы. Для *com*-программ это всегда число *100h*, так как первая исполняемая команда начинается в памяти сразу же после завершения *PSP*. Что касается указателя стека *SP*, то он содержит адрес-смещение вершины стека, в который при загрузке программы уже было помещено одно нулевое слово. По мере того, как в стек будут записываться новые слова данных, его вершина будет приближаться к области памяти, занимаемой командами и данными программы. В случае наложения стека на другие данные программы, дальнейшее ее продолжение будет или ошибочным, или вообще невозможным.

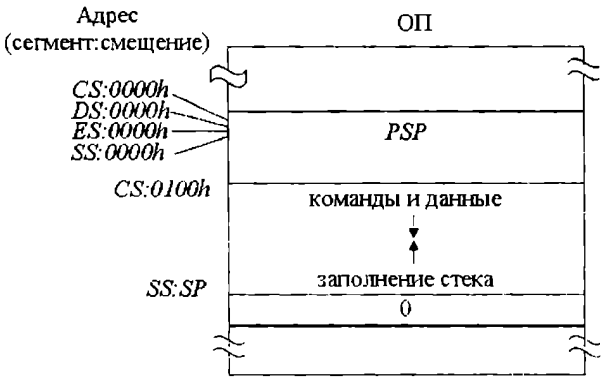

Рис. 2.27. Результат загрузки *com*-программы

Для получения загрузочного модуля типа *com* можно применить два способа. Первый из этих способов, который мы использовали до сих пор во всех своих программах, заключается в том, что вся работа по получению *com*-файла поручается программе *NASM*. При этом исходная программа на ассемблере (*asm*-файл) имеет следующие особенности:

первому исполнительному оператору программы предшествует псевдооператор «*org 100h*», который обеспечивает требуемое смещение соответствующей машинной команды относительно начала программы;

исходная программа может не иметь виртуальных сегментов. Отсутствие виртуальных, т. е. явно определенных в исходной программе сегментов, никак не влияет на адресацию данных в машинной программе;

исходная программа не должна содержать исполнительных операторов, выполняющих действия с сегментными адресами. (Такие адреса должны настраиваться загрузчиком в зависимости от размещения машинной программы в ОП, а настройка *com*-файла не производится.) Например, недопустимы следующие операторы:

```
mov     ax, _Data          ; _Data – сегмент данных
mov     si, seg  Buff      ; si ← адрес-сегмент переменной Buff
```

Пример исходной программы, ориентированной на получение загрузочного модуля типа *com*:

```
;       Программа выводит сообщение «Здравствуйте» на экран
;       ----------------------------------------------------------------
;
cr      equ     0dh                     ; Код ASCII возврата каретки
lf      equ     0ah                     ; Код ASCII перевода строки
```

```
        org     100h
        jmp     Begin                   ; Переход через данные
Msg     db      'Здравствуйте', cr, lf, '$'  ; Выводимое сообщение
Begin:  mov     dx, Msg                 ; DX ← адрес сообщения
        mov     ah, 9                   ; Вывод строки
        int     21h                     ;  на экран
        mov     ax, 4c00h               ; Возврат в DOS с
        int     21h                     ;  кодом завершения 0
```

Данные, обрабатываемые программой, размещены в сегменте памяти, занимаемой программой, так, чтобы они не могли попасть на ЦП в качестве исполняемых команд, что неизбежно привело бы к сбою в работе процессора. Считается хорошим тоном при написании *com*-программы на ассемблере располагать ее данные в начале программы, так как это упрощает трансляцию. Первый исполнительный оператор *jmp* «перепрыгивает» через эти данные на исполнительную часть программы. Сказанное относится к глобальным данным, используемым в любом месте программы. Локальные данные, используемые только внутри конкретной процедуры, лучше располагать сразу же за текстом этой процедуры, т. е. после последней команды процедуры *ret*. Удобно присваивать таким данным локальные имена (начинаются с точки).

Для вывода строки на экран программа использует системный вызов «*int 21h*» (функция 9). Перед применением вызова программа помещает в регистры *DS* и *DX* соответственно адрес-сегмент и адрес-смещение для младшего байта выводимой строки. В нашей программе *DS* уже содержит требуемый адрес (это сделал загрузчик) и поэтому не загружается. Выводимая строка обязательно должна заканчиваться символом «*$*». Сама строка может содержать любые другие символы, в том числе и управляющие. В программе используются управляющие символы *0dh* (возврат каретки) и *0ah* (перевод строки).

Второй метод получения *com*-файла предполагает использование для этого не только транслятора (*NASM*) но и редактора связей (*ALINK*). При этом транслятор обеспечивает преобразование исходного фрагмента в объектный модуль (*obj*-файл), а редактор связей получает из объектного модуля загрузочный модуль (*com*-файл). Реализация данного метода предполагает внесение существенных изменений в исходную программу. Особенности данных изменений опять рассмотрим на примере «вежливой» программы, выводящей «Здравствуйте» на экран. Ее текст:

```
;       Программа выводит сообщение «Здравствуйте» на экран
;       ----------------------------------------------------
;
cr      equ     0dh                     ; Код ASCII возврата каретки
lf      equ     0ah                     ; Код ASCII перевода строки
        segment code
        resb    100h
..start:
        jmp     Begin                   ; Переход через данные
Msg     db      'Здравствуйте', cr, lf, '$'  ; Выводимое сообщение
Begin:  mov     dx, Msg                 ; DX ← адрес сообщения
        mov     ah, 9                   ; Вывод строки
        int     21h                     ;  на экран
```

```
mov     ax, 4c00h          ; Возврат в DOS с
int     21h                ;   кодом завершения 0
```

Во-первых, обратим внимание на наличие виртуального сегмента кода. Напомним, что каждый виртуальный сегмент начинается с рассмотренного ранее псевдооператора *absolute* или с псевдооператора ***segment***. Параметром оператора *segment* является тип сегмента – ***code***. Это слово «*code*» может использоваться в программе в качестве имени (метки) этого сегмента.

Кроме сегмента *code* в программе возможно наличие сегментов других типов – ***data*** (данные) и ***stack*** (стек). Отсутствие других виртуальных сегментов в рассматриваемой исходной программе обусловлено тем, что она ориентирована на получение машинной *com*-программы. Поэтому наличие других виртуальных сегментов является излишним, так как после загрузки *com*-программы загрузчик копирует содержимое регистра *CS* в качестве содержимого других сегментных регистров – *DS*, *ES* и *SS*.

Во-вторых, обратим внимание на отсутствие псевдооператора «*org*». Данный оператор нельзя использовать в исходных программах, предназначенных для получения объектных модулей. Вместо него в начале виртуального сегмента может использоваться знакомый нам псевдооператор *resb*. Данный оператор задает адрес-смещение для самой первой команды сегмента кода внутри сегмента ОП, выделенного машинной программе. Обратите внимание на порядок записи операторов *segment* и *resb*. Если их поменять местами, то сегмент кода будет начинаться не от начала сегмента памяти, а со смещением *100h*, что не соответствует размещению логических сегментов для *com*-программы (см. рис. 2.27).

В-третьих, первый исполнительный оператор виртуального сегмента кода имеет стандартную метку входа в программу «***..start***:».

Для получения объектного модуля программы с помощью транслятора *NASM* необходимо использовать при вызове этой программы дополнительный параметр «***-f obj***», уточняющий тип целевого файла. Допустим, что приведенная выше программа записана в исходный файл *test.asm*. Тогда для получения объектного файла можно воспользоваться командой DOS:

```
nasm  test.asm  -f obj  -o test.obj
```

В данном примере первый параметр команды задает имя исходного файла, второй параметр -- тип целевого файла, а третий – имя целевого файла. Последняя часть команды в данном примере является излишней, так как имя целевого файла аналогично (за исключением расширения имени) имени исходного файла. Поэтому последнюю команду можно заменить следующей:

```
nasm  test.asm  -f obj
```

Преобразование одного или нескольких объектных модулей в загрузочный модуль (*com*- или *exe*-файл) выполняет редактор связей *ALINK*. В качестве одного или нескольких параметров команды запуска *ALINK* задается имя одного или нескольких объектных модулей. В качестве следующего параметра – тип целевого файла (загрузочного модуля). А в качестве последнего параметра – имя загрузочного модуля. Пример команды запуска *ALINK*:

```
alink  test.obj  -oCOM  -o  test.com
```

Обратите внимание, что второй параметр записан слитно, а тип файла (*COM*) записан большими буквами. Сокращенная запись этой же команды:

```
alink  test.obj  -oCOM
```

Получите com-файл для приведенной выше «вежливой» программы, а затем выполните ее.

17.4. Программа типа EXE

В отличие от *com*-программы, загружаемой по умолчанию в один стандартный сегмент ОП объемом 64 К, программа типа *exe* может занимать несколько таких сегментов. Следствием этого является наличие в программе команд, выполняющих запись адресов-сегментов в сегментные регистры. Эти адреса требуют своей настройки во время загрузки программы. В результате, получение *exe*-программы требует не простого копирования соответствующего файла в ОП, а предполагает преобразование этого файла загрузчиком. Поэтому загрузчик не только записывает *PSP* программы (это он делает и для *com*-файла), но и настраивает некоторые адреса в полях машинных команд.

В начале *exe*-файла находится ***заголовок***, размер которого кратен 512 байтам, содержащий управляющую информацию, используемую загрузчиком для получения *exe*-программы. Свою работу по созданию *exe*-файла редактор связей начинает с того, что последовательно записывает в свою память код (команды), данные и стек будущей программы. Порядок расположения этих частей программы, в отличие от *com*-файла, может быть любым. Их содержимое редактор связей считывает из доступных ему объектных модулей программы. Выполняя размещение программы в памяти, редактор связей обращается с ее адресами так, как будто программа загружена в ОП, начиная с условного адреса *00000h*. Поэтому считается, что часть программы (код, данные, или стек), размещаемая первой, имеет начальный логический адрес *0000h:0000h*. Если, например, вторая часть программы может быть загружена, начиная с условного реального адреса *000E7h*, то редактор связей начнет загрузку этой части программы с ближайшей границы параграфа – с условного реального адреса *000F0*. Этому условному реальному адресу соответствует начальный логический адрес *000Fh:0000h*.

Применение условных адресов позволяет редактору связей выполнить запись численных значений для всех внешних адресов, оставшихся не проставленными после трансляции. Исключение составляют команды, выполняющие запись в регистры значений адресов-сегментов. К таким командам относятся, во-первых, команды дальних переходов *jmp* и *call* (они выполняют запись адреса-сегмента в регистр *CS*), а во-вторых, команды *mov*, выполняющие запись адресов-сегментов в регистры данных (из регистров данных другие команды *mov* переписывают значения в регистры сегментов *DS*, *ES*, *SS*). Например, следующие два оператора исходной программы выполняют запись в регистр *DS* того адреса-сегмента, который соответствует точке исходной программы с меткой *Msg*:

```
mov  dx, seg Msg
mov  ds, dx
```

В результате выполнения этих двух команд в сегментный регистр *DS* будет помещен адрес-сегмент элемента программы с именем *Msg*. Так как сегментный регистр *DS* «указывает» на начало основного логического сегмента данных, то выполнение этих двух операторов приводит к «загрузке» виртуального сегмента, которому принадлежит элемент программы с меткой *Msg* (элемент данных или оператор) в основной сегмент данных. Термин «логический» сдесь опущен без потери смысла. Следует заметить, что первый оператор *mov* транслируется в трех, а второй – в двухбайтовую машинную команду.

При этом второй и третий байты первой команды используются для размещения адреса-сегмента, загружаемого в регистр *DX*.

Так как адреса-сегменты зависят от фактического размещения будущей программы в ОП, которое редактор связей «не знает», то он не может проставить численные значения адресов-сегментов в поля машинных команд. Тем не менее, он выполняет значительную подготовку для будущей простановки этих значений загрузчиком. Для этого, во-первых, редактор связей помещает в эти поля условные адреса-сегменты, полученные в предположении, что программа загружается в память, начиная с нулевого адреса.

Во-вторых, редактор связей создает ***таблицу настройки***, занимающую почти весь объем заголовка *exe*-файла. Эта таблица состоит из четырехбайтовых полей, каждое из которых соответствует одному адресу-сегменту, используемому какой-то командой программы, и который требует настройки в зависимости от фактического размещения программы в памяти. В качестве содержимого этого поля редактор связей записывает условный логический адрес (сегмент и смещение) той ячейки (двух байтов) программы, которая содержит адрес-сегмент, требующий настройки.

Сама настройка адресов-сегментов программы производится загрузчиком сразу же после размещения программы (в том числе и *PSP*) в ОП. Для этого загрузчик последовательно просматривает таблицу настройки, считывая из нее указатель на очередную последовательность из двух байтов программы, требующую настройки. А затем он прибавляет к их содержимому (то есть к условному адресу-сегменту) номер того параграфа ОП, начиная с которого программа загружена фактически.

После того, как программа настроена, загрузчик передает ей управление. На рис. 2.28 приведено содержимое сегментных регистров и указателя стека *SP* в момент передачи управления *exe*-программе. В этот момент сегментные регистры данных *DS* и *ES* содержат номер начального параграфа *PSP*. Благодаря этому программа может использовать полезную информацию, содержащуюся в *PSP*. (Запомнив где-то первоначальное значение этих регистров, программа обычно записывает в них новое значение, указывающее на начало области данных программы.) Что касается *CS*, то в него загрузчик помещает начальный номер параграфа не *PSP*, а области кода программы. Аналогично содержимое *SS* указывает на границу области стека программы.

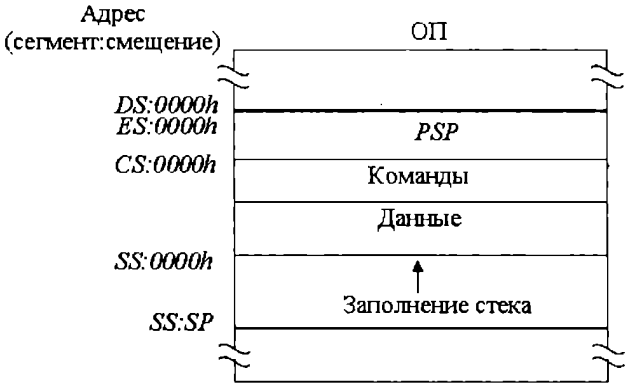

Рис. 2.28. Результат загрузки *exe*-программы

Исходная программа на ассемблере, ориентированная на получение *exe*-программы, обязательно имеет кроме виртуальных сегментов кода и данных виртуальный сегмент

стека. Порядок записи этих сегментов в программе значения не имеет. В качестве точки входа в программу может быть задан любой исполнительный оператор программы. Для этого он должен иметь в исходной программе стандартную метку «*..start*:».

Пример. Приведем исходный текст «вежливой» программы, ориентированной на получение загрузочного модуля типа *exe*:

```
;       Программа выводит сообщение «Здравствуйте» на экран
;       ----------------------------------------------------
;
cr      equ     0dh             ; Код ASCII возврата каретки
lf      equ     0ah             ; Код ASCII перевода строки
        segment stack           ; Сегмент
        times   64 db 0         ;   стека
        segment code            ; Сегмент кода
..start:
        mov     ax, data        ; Сделаем сегмент данных
        mov     ds, ax          ;   адресуемым
        mov     dx, Msg         ; DX ← адрес сообщения
        mov     ah, 9           ; Вывод строки
        int     21h             ;   на экран
        mov     ax, 4c00h       ; Возврат в DOS с
        int     21h             ;   кодом завершения 0
        segment data            ; Сегмент данных
Msg     db  'Здравствуйте', cr, lf, '$'  ; Выводимое сообщение
```

В данной исходной программе три виртуальных сегмента – сегмент кода *code*, сегмент данных *data* и сегмент стека *stack*. Параметр *code*, *data* или *stack* оператора *segment* означает тип виртуального сегмента – код, данные или стек. Одновременно единственный параметр оператора *segment* задает и имя своего виртуального сегмента. Если виртуальному сегменту требуется присвоить какое-то другое имя, то это имя записывается в качестве дополнительного параметра оператора *segment*. Причем этот параметр записывается первым. Например, следующий оператор присваивает сегменту кода имя «*Program*»:

```
segment Program code
```

Обратите внимание, что в начале сегмента кода в регистр *DS* загружается адрес-сегмент, соответствующий началу виртуального сегмента данных. Благодаря этому фактическое размещение данных в памяти никак не зависит ни от размещения в ней *PSP*, ни от размещения в памяти других сегментов программы. (Напомним, что для *com*-программы адрес-смещение данных вычисляется относительно начала *PSP*.) Обратите внимание, что запись значения в сегментный регистр *DS* производится не непосредственно, а через рабочий регистр *AX*.

Еще одним отличием исходных *exe*- и *com*-программ является то, что в *exe*-программе никак не задается расположение сегмента кода в памяти. Это полностью остается на усмотрение редактора связей и загрузчика.

Допустим, что записанная выше исходная *exe*-программа имеет имя *test1.asm*. Тогда для получения соответствующего объектного модуля можно использовать следующую команду DOS:

```
nasm  test1.asm  -f obj  -o test1.obj
```

Сокращенная форма записи этой же команды:

```
nasm  test1.asm  -f obj
```

Команда DOS для получения exe-файла:

```
alink  test1.obj  -oEXE  -o test1.exe
```

Этот же результат обеспечивает команда:

```
alink  test1.obj  -oEXE
```

Получите exe-файл для приведенной выше программы, а затем выполните его.

17.5. АДРЕСАЦИЯ ПАМЯТИ В ПРОГРАММЕ ТИПА EXE

Многие исполнительные операторы программы и соответствующие им машинные команды выполняют действия с содержимым ячеек ОП. Напомним, что при выполнении машинной команды ЦП преобразует логический адрес ячейки ОП в соответствующий реальный адрес. При этом логический адрес есть пара чисел: (адрес-сегмент; адрес-смещение), а реальный адрес – номер ячейки в ОП. Адрес-смещение ЦП считывает или из поля машинной команды, или из того регистра, который указан в этой команде. Что касается адреса-сегмента, то он задается в команде не явно, а «по умолчанию». При написании *exe*-программы очень важно понять принцип действий ЦП по определению адреса-сегмента.

Если машинная команда имеет дело с одной ячейкой ОП (байтом или словом), то «по умолчанию» ЦП всегда получает адрес-сегмент ячейки ОП из сегментного регистра *DS*. Примеры соответствующих исполнительных операторов:

```
add   ax, [Mem]   ; AX ← содержимое слова данных с именем Mem
mov   dx, Mem     ; DX ← адрес-смещение ячейки с именем Mem
```

При обработке транслятором каждого из этих двух исполнительных операторов, в поле соответствующей машинной команды будет записано смещение относительно начала того виртуального сегмента (*data*, *code* или *absolute*), в тексте которого находится определение слова данных с именем *Mem*. (Для такого определения в исходной программе может использоваться, например, оператор «*Mem dw 0*»).

Не только машинные команды, но и системные подпрограммы, вызываемые с помощью системных вызовов (то есть с помощью команд *int*) обычно получают свои входные данные в том логическом сегменте, на который указывает регистр *DS*. Например, хорошо знакомый нам системный вызов DOS – «*int 21h*» (функция 9) выводит на экран ту строку символов, первый байт которой имеет логический адрес (*DS:DX*).

Если машинная команда имеет дело сразу с двумя ячейками ОП (байтами или словами), то первую из ячеек она ищет в том логическом сегменте, на который указывает *DS*, а вторую – *ES*. (Напомним, что *ES* – регистр дополнительного сегмента данных.) Например, в п. 13.2 приведено сокращенное описание команды *movsb*, выполняющей копирование строки байтов. Сущность сокращения заключается в том, что предполагалось, что исходная строка имеет адрес-смещение – (*SI*), а строка-копия – (*DI*), и при этом ничего не говорилось об адресах-сегментах строк. Подобное сокращенное описание команды *movsb* пригодно лишь для *com*-программы. Напомним, что для такой программы все четыре логических сегмента совпадают вследствие совпадения содержимого всех сегментных регистров – *CS, DS, ES* и *SS*. При этом не имеет значения, находятся ли исходная строка и строка-копия в одном или в разных виртуальных сегментах, так как во время

выполнения программы они оказываются в одном и том же логическом сегменте. При написании exe-программы приходится использовать полное описание команды *movsb*, согласно которому исходная строка начинается с логического адреса (*DS:SI*), а строка-копия – с логического адреса (*ES:DI*).

Рассмотрим более внимательно вопрос о заполнении сегментных регистров для *exe*-программы. Регистр сегмента кода *CS* первоначально заполняется загрузчиком и содержит номер первого параграфа (адрес-сегмент) того виртуального сегмента, который содержит оператор с меткой *..start*. В результате данный виртуальный сегмент кода «загружается» в логический сегмент кода. Дальнейшая замена «загружаемых» виртуальных сегментов кода осуществляется в результате выполнения самой программы, а именно – в результате выполнения команд межсегментных переходов – *jmp, int, call, iret, ret*. Еще меньшего участия со стороны программиста требует логический сегмент стека. Первоначально загрузчик помещает в регистр *SS* адрес-сегмент виртуального сегмента стека и, таким образом, «загружает» этот виртуальный сегмент в логический сегмент стека. Дальнейшая замена содержимого *SS* в процессе выполнения программы обычно не производится.

В отличие от логических сегментов кода и стека, манипуляции с логическими сегментами данных полностью выполняются командами программы. Иными словами, программист сам отвечает за заполнение регистров сегментов данных *DS* и *ES*. Первоначально загрузчик помещает в *DS* и *ES* адрес-сегмент *PSP* програмы. Этого достаточно для работы с содержимым самого *PSP*, но данные, обрабатываемые программой, на этом, естественно, не заканчиваются. Данные программы могут находиться, в частности, в любом виртуальном сегменте кода.

Например, если мы размещаем локальные данные процедуры сразу же за последним оператором этой процедуры *ret*, то, естественно, что эти данные будут находиться в том же виртуальном сегменте кода, что и процедура. Допустим, что в состав локальных данных процедуры входит сообщение, выводимое на экран с помощью системного вызова «*int 21h*» (функция 9). Если где-то перед этим системным вызовом программа не загрузит в регистр *DS* адрес-сегмент рассматриваемого виртуального сегмента кода, то требуемая строка символов не будет найдена. Вместо нее подпрограмма DOS выведет на экран нечто странное.

Если данное программы, обрабатываемое машинной командой или целым системным вызовом, находится в виртуальном сегменте кода или данных, то прежде, чем обрабатывать это данное, необходимо виртуальный сегмент данных (или кода) сделать «видимым» для ЦП, «загрузив» его в логический сегмент данных. (Напомним, что для этого требуется загрузить в *DS* номер начального параграфа требуемого виртуального сегмента.)

В качестве примера рассмотрим *exe*-программу, которая сначало выполняет вывод на экран своих параметров, а затем выводит «До свидание». Для этого переделаем *com*-программу из п. 17.1:

```
;       EXE-программа выводит на экран свои параметры
;       ----------------------------------------------
;
;   Виртуальный сегмент absolute
        absolute  80h           ; Следующий байт имеет адрес-смещение 80h
Lparam resb   1                 ; Длина хвоста команды
Param  resb   1                 ; Первый байт хвоста
;   Виртуальный сегмент стека
        segment   stack
        times     64 db  0
```

```
;     Виртуальный сегмент кода
        segment   code
..start: xor    cx, cx                          ; CX ← 0
        mov   cl, [Lparam]                      ; CL ← Длина хвоста
        cmp   cx, 0                             ; Хвост отсутствует ?
        jz    .Exit                             ; Если да
        mov   ah, 0eh                           ; 0Eh – функция вывода символа
        mov   bh, 0                             ; Нулевая страница экрана
        xor   si, si                            ; SI ← 0
.Next:  mov   al, [Param+si]                    ; Вывод следующего
        int   10h                               ;   символа
        inc   si
        loop  .Next                             ; Повторить для след. символа
.Exit:  mov   ax, cs                            ; Логический сегмент данных
        mov   ds, ax                            ;  на основе вирт. сегмента кода
        mov   ah, 9                             ; Вывод
        mov   dx, Msg                           ;    сообщения
        int   21h                               ;    на экран
        mov   ax,  4c00h                        ; Возврат в DOS с
        int   21h                               ;   кодом завершения 0
Msg     db  0dh, 0ah, 'До свидание', 0dh, 0ah, '$'   ; Сообщение
```

Первоначально данная программа выводит на экран свои параметры, используя области памяти *Lparam* и *Param*, принадлежащие виртуальному сегменту *absolute*. Так как перед выполнением программы загрузчик поместил в *DS* номер первого параграфа *PSP*, а все адреса-смещения в виртуальном сегменте *absolute* подсчитаны относительно начала программы (и *PSP*), то никакой замены содержимого *DS* не требуется. Замена содержимого *DS* должна предшествовать выводу на экран строки *Msg*. Так как данная строка определена в виртуальном сегменте *code*, то в *DS* копируется содержимое регистра *CS*. (Первоначально загрузчик уже поместил в *CS* номер первого параграфа сегмента *code*). Если бы строка *Msg* находилась в виртуальном сегменте *data* (см. п. 17.4), то оператор программы «*mov ax,cs*» пришлось бы заменить на оператор «*mov ax,data*».

В рассмотренном примере программа последовательно обрабатывает данные, принадлежащие разным виртуальным сегментам (*absolute* и *code*, или *absolute* и *data*). Трудности, связанные с заполнением регистра *DS*, увеличиваются, если программа должна одновременно работать с данными, принадлежащим разным виртуальным сегментам. В этом случае приходится многократно менять содержимое *DS*, выполняя, таким образом, смену виртуального сегмента, «загружаемого» в логический сегмент данных. Рассмотрим вопрос замены виртуальных сегментов более подробно.

В последнем примере программы замену виртуального сегмента, загружаемого в логический сегмент кода, выполняют два оператора:

```
.Exit:  mov   ax, cs     ; Логический сегмент данных
        mov   ds, ax     ;   на основе вирт. сегмента кода
```

Здесь использован тот факт, что перед выполнением программы загрузчик уже поместил в *CS* начальный адрес виртуального сегмента кода. Аналогичную «загрузку» виртуального сегмента кода в основной сегмент данных можно сделать, например, следующим образом:

```
.Exit: mov  ax, seg ..start     ; Логический сегмент данных
       mov  ds, ax              ;   на основе вирт. сегмента кода
```

Здесь использован вспомогательный псевдооператор ***seg***, требующий вычислить адрес-сегмент элемента программы с заданным именем. При этом имя «*..start*» можно заменить на имя (метку) любого элемента программы, определенного в том же виртуальном сегменте кода, что и метка «*..start*». Это объясняется тем, что адреса-сегменты у всех этих элементов программы одинаковые. Кроме того, в данном операторе можно задать имя самого виртуального сегмента. При этом псевдооператор *seg* не требуется:

```
.Exit: mov  ax, code            ; Логический сегмент данных
       mov  ds, ax              ;   на основе вирт. сегмента кода
```

Аналогично виртуальный сегмент кода или данных можно «загрузить» в дополнительный сегмент данных. Например, следующие два оператора «загружают» виртуальный сегмент кода в дополнительный сегмент данных:

```
       mov  dx, code
       mov  es, dx
```

Описанные выше способы «загрузки» виртуальных сегментов не пригодны для «загрузки» виртуальных сегментов типа *absolute*. Первоначально этот виртуальный сегмент «загружен» и в основной, и в дополнительный сегменты данных, так как перед выполнением программы загрузчик записывает в *DS* и в *ES* адрес-сегмент начала программы. Желательно, чтобы программа сохранила бы этот адрес-сегмент в каком-то слове данных, принадлежащем одному из виртуальных сегментов, за исключением виртуального сегмента *absolute*. Это делается на тот случай, если в процессе выполнения программы потребуется вернуться к обработке данных из виртуального сегмента *absolute*.

Например, пусть для запоминания адреса-сегмента начала программы выделено слово данных *Segin* в виртуальном сегменте данных *data*. Тогда начало виртуального сегмента кода может быть, например, следующим:

```
       segment  code
..start: mov ax, data           ; «Загрузка» виртуального сегмента data
       mov es, ax               ;   в дополнительный сегмент данных
       mov ax, ds               ; Запоминание в Segin начала
       mov [es:Segin], ax       ;   программы
. . . . . .
```

Если в ходе выполнения программы потребуется вернуться к обработке данных из виртуального сегмента *absolute*, или если требуется выполнить обработку содержимого памяти программы напрямую, не используя разбиение программы на виртуальные сегменты, то достаточно выполнить оператор:

```
       mov  ds, [es:Segin]      ;   DS ← начало программы
```

Количество замен содержимого *DS* можно уменьшить, если использовать возможность задавать адрес-сегмент ячейки ОП явно. Как сказано выше, для неявного задания адресов-сегментов ячеек ОП используется сегментный регистр *DS*. Если ячейка памяти (байт или слово) находится в виртуальном сегменте, который «загружен» не в основной, а в дополнительный сегмент данных или в сегмент кода, то для задания сегмента-адреса ячейки можно использовать имя сегментного регистра, после которого находится символ «:». Примеры:

```
       mov ax, [cs:Mem]         ; Слово Mem в логическом сегменте кода
```

```
mov bl, [es:Mem]        ; Байт Mem в дополнительном сегменте данных
mov dx, Mem             ; DX ← адрес-смещение ячейки Mem
```

Обратите внимание, что в последнем примере адрес-сегмент не задан явно. Причина этого в том, что при выполнении соответствующей машинной команды на ЦП адрес-сегмент ячейки ОП вообще не нужен. ЦП использует лишь смещение данной ячейки относительно начала того виртуального сегмента, в котором ячейка находится. Величину смещения записывает в поле машинной команды транслятор.

Указанная возможность явного задания в командах адресов-сегментов имеет ограниченное применение. Она не пригодна, например, при подготовке к выполнению системного вызова, который предназначен для обработки ячеек из основного сегмента данных.

Подводя итог, следует заметить, что редактор связей по-разному преобразует объектный модуль в *com*- и в *exe*-программу. При получении *com*-программы виртуальные сегменты программы не оказывают какого-либо влияния на адреса-смещения, проставляемые в поля машинных команд. Все эти смещения вычисляются транслятором относительно начала программы. Никаких действий по получению логических сегментов сама программа не выполняет. Все четыре логических сегмента для *com*-программы совпадают.

При получении *exe*-программы каждый адрес-смещение вычисляется относительно начала того виртуального сегмента, которому принадлежит адресуемый элемент программы. Логический адрес любой ячейки памяти есть пара: (нач. параграф вирт. сегмента; смещение относительно начала вирт. сегмента). Соответствие между логическим и виртуальным сегментами кода, а также между логическим и виртуальным сегментами стека не требуют наличия дополнительных команд программы. Наоборот, за соответствие между логическими сегментами данных (основным и дополнительным) и виртульными сегментами (кода и данных) отвечают команды программы. При этом в логический сегмент данных (основной или дополнительный) может быть загружен любой виртуальный сегмент данных или кода.

17.6. Применение нескольких объектных модулей

Получение и последующее связывание нескольких объектных модулей программы обусловлено несколькими причинами. Во-первых, напомним, что каждый объектный модуль получается в результате отдельного выполнения транслятора. Поэтому обрабатывать исходные фрагменты могут разные трансляторы и, следовательно, эти исходные фрагменты могут быть написаны на разных языках программирования.

Второе преимущество использования при получении программы нескольких объектных модулей – наиболее используемые из этих модулей могут быть получены один раз, а затем они могут быть включены в состав того или иного библиотечного файла. Это позволит использовать данные объектные модули многократно, в составе многих программ, получаемых многими программистами.

Третье преимущество многомодульной объектной структуры программы не используется в рассматриваемой простой ВС (DOS + i8086), но применяется очень широко в более сложных системах. Его сущность состоит в том, что фрагменты машинной программы, каждый из которых получается в результате преобразования своего объектного модуля, могут быть загружены в различные разделы (сегменты) ОП так, что последующая работа программ с этими разделами будет производиться по отличным друг от друга правилам. Например, для некоторых разделов ОП допустима только операция «чтение». В такие разделы целесообразно помещать неизменяемые данные программы, т. е.

ее константы. Для других разделов ОП система допускает только операцию «выполнение». Сюда целесообразно помещать команды («код») машинной программы. Некоторые разделы ОП могут содержать данные (в том числе и «код»), одновременно используемые (разделяемые) несколькими программами. Подобное свойство широко используется в мультипрограммных ВС для реализации *DLL* – динамически связываемых библиотек.

Многомодульная объектная структура оказывает влияние на тексты исходных фрагментов. Причиной этого является наличие в тексте рассматриваемого исходного фрагмента внешних имен (см. п. 17.2). При этом внешние выходные имена (метки) должны быть описаны с помощью псевдооператоров ***global***. А внешние входные имена – с помощью псевдооператоров ***extern***.

Например, переделаем «вежливую» *exe*-программу так, чтобы она была получена не из одного, а из двух исходных фрагментов. В первый исходный файл *test2a.asm* включим виртуальные сегменты кода и стека, а во второй *test2b.asm* – виртуальный сегмент данных. Текст первого исходного файла:

```
;       Программа выводит сообщение «Здравствуйте» на экран
;       ----------------------------------------------------
;
        segment  data              ; Сегмент данных в другом файле
        segment  stack             ; Сегмент
        times    64 db   0         ;  стека
        segment  code              ; Сегмент кода
..start:
        extern   Msg               ; Сообщение в другом модуле
        mov      ax, data          ; Сделаем сегмент данных
        mov      ds, ax            ;  адресуемым
        mov      dx, Msg           ; DX ← адрес сообщения
        mov      ah, 9             ; Вывод строки
        int      21h               ;  на экран
        mov      ax, 4c00h         ; Возврат в DOS с
        int      21h               ;  кодом завершения 0
```

Текст второго файла:

```
;       Сегмент данных программы, которая выводит
;       сообщение «Здравствуйте» на экран
; ------------------------------------------------------
;
cr      equ      0dh                        ; Код ASCII возврата каретки
lf      equ      0ah                        ; Код ASCII перевода строки
        segment  data                       ; Сегмент данных
        global   Msg                        ; Используется в другом файле
Msg     db   'Здравствуйте', cr, lf, '$'    ; Выводимое сообщение
```

Обратите внимание, что, несмотря на то, что виртуальный сегмент данных находится во втором файле, его заголовок «*segment data*» присутствует и в первом (главном) файле. Кроме заголовка этот виртуальный сегмент может содержать и любые данные. В любом случае редактор связей соединяет в один виртуальный сегмент одноименные виртуаль-

ные сегменты данных, кода или стека, объявленные в разных фрагментах исходной программы. Напомним, что фрагмент исходной программы состоит из одного или нескольких исходных файлов (исходных модулей), соединяемых в единое целое с помощью операторов *%include*.

Напомним, что имя виртуального сегмента может быть задано с помощью необязательного первого параметра оператора *segment*. Если этот параметр оператора *segment* опущен, то виртуальному сегменту присваивается имя, совпадающее с его типом. Например, в приведенном выше примере виртульные сегменты имеют имена: *code*, *data* и *stack*.

После того, как объектные модули программы получены, их необходимо связать в единый загрузочный модуль (*exe*-файл). Пример соответствующей команды DOS:

```
alink test2a.obj test2b.obj -oEXE  -o test2.exe
```

Все одноименные виртуальные сегменты, независимо от того, в каких исходных файлах и фрагментах они находятся, представлены в готовой машинной программе в виде единого виртуального сегмента. Если на начало этого сегмента указывает регистр *CS* (или *DS*, *ES*, *SS*), то данный единый виртуальный сегмент становится логическим сегментом кода (или сегментом данных, или дополнительным сегментом данных, или сегментом стека).

Наберите и выполните приведенную выше программу.

Лабораторная работа 15

Требуется переделать полученную на предыдущей лабораторной работе программу редактирования, внеся в нее следующие изменения:

- Имя редактируемого файла передается в командной строке DOS в качестве параметра команды запуска программы редактирования. Если это имя отсутствует, то производится редактирование ОП.
- Программа редактирования имеет тип *exe*. Для ее получения необходимо объединить два объектных модуля, в одном из которых находятся виртуальные сегменты кода и стека программы, а также сегмент *absolute*. Во втором объектном модуле содержится виртуальный сегмент данных, в котором собраны глобальные переменные программы.

☑ **Примечания:**

1. Рекомендуется «загружать» виртуальные сегменты *absolute* и *code* по мере надобности в основной сегмент данных. А основным местом пребывания виртуального сегмента данных рекомендуется сделать дополнительный сегмент данных.

2. В связи с тем, что виртуальный сегмент данных «загружен» в дополнительный сегмент данных, требуется скорректировать исполнительные операторы, выполняющие действия с глобальными переменными, во всех процедурах, за исключением *Write_sector*. Корректировка заключается в том, что адреса-сегменты обрабатываемых глобальных переменных должны задаваться в операторах явно.

3. Если процедура *Write_sector* выполняет копирование содержимого глобальной переменной *Sector* в оперативную память, то перед тем, как выполнять копирование с помощью команды *movsb*, требуется «загрузить» виртуальный сегмент данных в основной сегмент данных, а виртуальный сегмент *absolute* – в дополнительный сегмент данных. Где-то после указанного копирования следует вновь «загрузить» виртуальный сегмент данных в дополнительный сегмент данных.

4. Если процедура *Read_sektor* выполняет копирование сектора ОП в глобальную переменную *Sector*, то перед тем, как выполнить данное копирование с помощью команды *movsb*, требуется «загрузить» виртуальный сегмент *absolute* в основной сегмент данных.

5. Если в процедуре имеются операторы *mov*, выполняющие запись в сегментные регистры, то в начале процедуры операторы *push* должны запоминать в стеке, а в конце процедуры операторы *pop* должны восстанавливать прежнее содержимое этих регистров.

18. Управление памятью и запуск программ

18.1. Распределение памяти

Оперативная память – ресурс, требуемый для выполнения любой программы. В рассматриваемой ВС (DOS + i8086) объем адресуемого пространства ОП составляет 1 Мбайт. Только часть этого объема (640 К) соответствует реальной ОП (рис. 2.29). Остальная часть адресов используется для работы с другими видами памяти (ПЗУ и видеопамять). Благодаря общему адресному пространству работа с этими видами памяти выполняется командами ЦП аналогично ОП, за исключением того, что в ПЗУ нельзя выполнять запись.

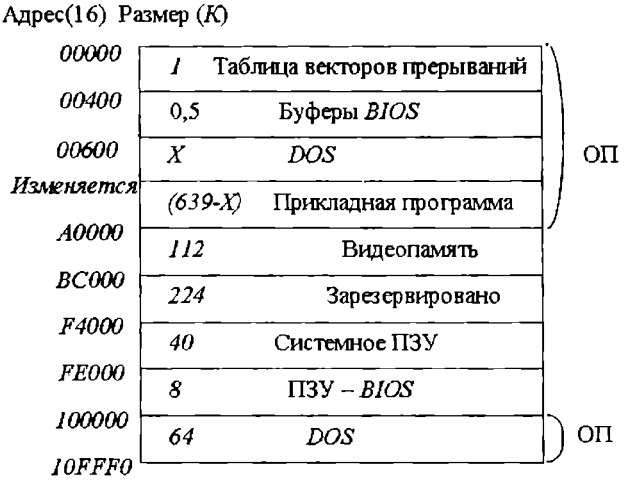

Рис. 2.29. Распределение адресного пространства ОП

Как видно из рис. 2.29, общий объем адресуемой памяти составляет не ровно 1 Мбайт, а немного больше (почти на 64 К). Дело в том, что если поместить в регистр сегмента предельно большое значение (*FFFFh*), то за счет содержимого регистра-смещения можно превысить число *FFFFFh*, которое является номером старшего байта в первом мегабайте памяти. Предельная величина адреса получается суммированием чисел *FFFF0h* и *FFFFh* и равна *10FFEFh*. Адреса за пределами 1 Мбайта соответствуют реальной ОП, которую DOS обычно использует для своих дополнительных нужд. Основная часть ОП, занимаемая ОС, находится в начале адресного пространства.

Что касается прикладной программы, то ей может быть распределена вся память, оставшаяся от 640 К после размещения DOS. Распределение памяти программе, сделанное до начала ее выполнения, называется ***статическим распределением памяти***, а во время ее выполнения – ***динамическим распределением***. Независимо от типа программы (*com* или *exe*), статически ОС распределяет ей всю память, свободную на момент создания программы.

Что касается динамического распределения памяти, то может возникнуть вопрос о том, зачем оно вообще нужно в однопрограммной системе, каковой является DOS. Первой причиной его применения является то, что даже при выполнении единственной прикладной программы ее потребности в памяти для хранения своих данных могут превышать объем памяти, которая может быть предоставлена программе. Поэтому программа может выполнять освобождение областей памяти, занимаемых одними данными, для того, чтобы позже запрашивать области памяти для других своих данных.

Вторая причина применения динамического распределения состоит в том, что DOS имеет зачатки мультипрограммных операционных систем, позволяя нескольким прикладным программам одновременно находиться в ОП. Это происходит, во-первых, тогда, когда одна прикладная программа запускается по запросу другой прикладной программы. А во-вторых, прикладная программа может выполнять какие-то системные функции, дублируя или дополняя подпрограммы ОС, и поэтому должна находиться в памяти и во время выполнения других прикладных программ. Подобные запуски прикладных программ будут рассмотрены в следующих разделах, а пока лишь заметим, что для их реализации требуется, чтобы одна прикладная программа добровольно отказывалась бы от своей неиспользуемой памяти с целью обеспечения будущей загрузки другой программы (программ).

Динамическое распределение памяти выполняется DOS по запросу самой прикладной программы. Для этого прикладная программа передает в ОС один из трех системных вызовов, рассматриваемых далее.

int 21h (функция ***48h***) – выдать программе область памяти требуемого размера. Для передачи размера области, измеряемого в параграфах, используется регистр *BX*. В случае успеха DOS возвращает в регистре *AX* адрес-сегмент выделенной области памяти. Иначе – устанавливает флаг переноса *CF=1*.

Пример. Для создания своего буфера программа запрашивает область памяти длиной *500h* параграфов:

```
        . . . . . .
Bufseg  dw   0              ; Адрес-сегмент новой области
        . . . . . .
        mov  ah, 48h        ; Номер функции
        mov  bx, 500h       ; Размер области (в параграфах)
        int  21h            ; Запрос памяти
        jc   Error          ; Переход при ошибке
        mov  [Bufseg], ax   ; Сохранить адрес-сегмент области
      . . .          .
```

int 21h (функция ***49h***) – освободить область памяти, выделенную ранее динамически программе с помощью системного вызова «*int 21h*» (функция *48h*). Адрес-сегмент области программа передает в регистре *ES*. В случае ошибки DOS устанавливает флаг переноса *CF=1*.

Пример. Программа освобождает область памяти, адрес-сегмент которой находится в слове *Bufseg*:

```
        . . . . .
Bufseg  dw   0              ; Адрес-сегмент области
        . . . . . .
        mov  ah, 49h        ; Номер функции
        mov  es, [Bufseg]   ; Адрес-сегмент области
        int  21h            ; Освобождение памяти
        jc   Error          ; Переход при ошибке
        . . . . . .
```

int 21h (функция ***4ah***) – изменить (увеличить или уменьшить) область памяти, принадлежащую программе. В регистре *ES* программа передает адрес-сегмент модифицируемой области, а в регистре *BX* – новый размер области памяти в параграфах. В случае ошибки DOS устанавливает флаг переноса *CF=1*.

Пример. Программа просит уменьшить размер области, выделенной ранее (в примере для функции *48h*) до *200h* параграфов (8 К):

```
        . . . . .
Bufseg  dw   0              ; Адрес-сегмент области
        . . . . . .
        mov  ah, 4ah        ; Номер функции
        mov  bx, 200h       ; Новый размер области
        mov  es, [Bufseg]   ; Адрес-сегмент области
        int  21h            ; Изменить размер области
        jc   Error          ; Переход при ошибке
        . . . . . .
```

Перейдем к рассмотрению принципов реализации динамического распределения памяти в DOS. Во-первых, вся ОП в системе разбита на области неодинаковой длины. Это разбиение со временем может изменяться, так как одни области могут разбиваться, а другие (соседние), наоборот, могут соединяться. Любая прикладная программа занимает минимум две области ОП. В одной из них находится сама программа (*PSP*, код, данные и стек), а во второй области находится блок окружения программы (размер этой области кратен 512 байт). Число областей ОП, занимаемых программой, увеличивается в результате системных вызовов «*int 21h*» (функция *48h*).

Во-вторых, каждая область памяти начинается с 16-байтового (один параграф) ***блока управления области***, который содержит:

- в байте 0 – символ ***M***, если область не последняя; ***Z*** – если область последняя в ОП;
- в байтах 1 и 2 – адрес-сегмент *PSP* программы, являющейся владельцем данной области ОП. Если область свободна, то *0000h*;
- в байтах 3 и 4 – длина области в параграфах;
- байты 5–15 – зарезервированы.

В-третьих, динамический запрос программы на получение дополнительной памяти DOS выполняет одним из трех способов:

- Последовательно просматривает блоки управления областей, двигаясь в сторону больших адресов, до тех пор, пока не встретится область памяти, не меньшая требуемой. Начальный адрес-сегмент этой области возвращается в программу, а ее излишек

оформляется в новую свободную область. При отсутствии требуемой области устанавливается флаг *CF*.

- Последовательно просматриваются все свободные области памяти с целью нахождения минимальной области, которая не меньше требуемой.
- Отличается от способа 1 тем, что просмотр областей производится от больших адресов к меньшим.

Обычно DOS ищет требуемую область, пользуясь первым способом. Но с помощью специального системного вызова программа может запросить у ОС другой алгоритм поиска.

Следует отметить, что при выполнении системных вызовов распределения памяти DOS «попутно» проверяет целостность всей цепочки блоков управления памятью. Поэтому нельзя портить содержимое этих блоков. Иначе ОС завершит работу системы, выдав соответствующее сообщение на экран.

В последующих разделах будут приведены примеры программ, содержащих системные вызовы для динамического распределения памяти.

18.2. Запуск прикладных программ

18.2.1. Описание системного вызова

Работа интерпретатора команд DOS основана на использовании системного вызова ***EXEC*** – «***int 21h***» (функция ***4bh***, подфункция ***00h***). Именно этот вызов обеспечивает загрузку заданного *com-* или *exe*-файла, его настройку (для *exe*-файла), выполнение полученной машинной программы, а затем возврат в ту точку программы ИК, откуда был сделан этот системный вызов.

Применив вызов *EXEC* в своей прикладной программе, мы сможем запустить (загрузить и выполнить) любую другую прикладную или системную обрабатывающую программу. В результате наша прикладная программа начнет выполнять некоторые функции ИК, а при некоторой доработке сможет даже заменить его. В этом нет ничего удивительного, так как основная часть ОС (ядро ОС) рассматривает ИК как обычную обрабатывающую программу (лингвистический процессор). Коль скоро вызов *EXEC* так важен, то внимательно рассмотрим его.

Во-первых, обратим внимание на то, что отношение между «родительской» и «дочерней» программами логически эквивалентно отношению между программой и подпрограммой, вызываемой с помощью команды *call* или команды *int*. Подобно подпрограмме, дочерняя программа является по отношению к родительской программе логической процедурой, так как на время выполнения дочерней программы выполнение родительской программы приостанавливается, а затем продолжается с той команды, которая расположена в программе сразу же после вызова *EXEC*.

Во-вторых, подобно обычным программным процедурам, дочерние программы могут быть вложенными. То есть дочерняя программа сама может содержать вызов *EXEC*, порождающий другую программу, и так далее.

В-третьих, прежде чем выдавать вызов *EXEC*, родительская программа должна позаботиться о том, чтобы имелась в наличии свободная память, достаточная для размещения дочерней программы. С этой целью родительская программа должна освободиться от излишков памяти, ранее выделенной ей, используя системные вызовы, рассмотренные выше.

В-четвертых, запуск программы с помощью вызова *EXEC* предполагает наличие между родительской и дочерней программами не только управляющего, но и информационного взаимодействия. При этом информация может быть передана как в прямом направлении – от родительской программы к дочерней, так и в обратном направлении.

Передача информации к дочерней программе выполняется с помощью ее *PSP*, который создается при выполнении вызова *EXEC*. Если информация небольшая, то она может быть размещена в «хвосте». (Напомним, что «хвост» – последнее поле *PSP*.) Большой объем информации (до 32 К) может быть размещен в блоке окружения программы. Кроме того, для передачи данных на вход дочерней программы может использоваться поле *PSP*, предназначенное для хранения устаревших типов блоков управления файлами.

Для того чтобы выполнение вызова EXEC привело к созданию требуемого PSP, в момент вызова регистры должны содержать следующие данные:

(DS:DX) – (адрес-сегмент: адрес-смещение) символьной строки, содержащей имя *com-* или *exe*-файла, подлежащего загрузке. Сразу же после имени файла должен находиться нулевой байт *(00h)*;

(ES:BX) – (адрес-сегмент: адрес-смещение) блока параметров.

Блок параметров содержит:

- 2 байта – адрес-сегмент блока окружения. Если эти байты содержат 0, то дочерняя программа получает копию блока окружения родительской программы;
- 4 байта – адрес-смещение и адрес-сегмент символьной строки, содержащей «хвост» команды. Эта строка заканчивается символом «возврат каретки» (*0Dh*), который не учитывается при подсчете длины «хвоста»;
- 8 байт – адрес-смещение и адрес-сегмент двух блоков управления файлами (на каждый блок по 4 байта).

В простейшем случае, когда ни «хвост», ни блоки управления файлами не используются для передачи информации на вход дочерней программы, а блок окружения такой же, как и у родительской программы, блок параметров может быть оформлен в программе как последовательность 14 байт, заполненных нулями.

В случае ошибочного завершения вызова *EXEC* в родительскую программу возвращается флаг *CF=1*, а в регистре *AX* возвращается код ошибки.

В случае успешного запуска и выполнения дочерней программы, она может передать в родительскую программу любой объем информации, используя для этого ту область ОП, адрес которой был получен ей ранее в составе другой информации, переданной от родительской программы.

18.2.2. Запуск программ из com-программы

Запуск дочерних программ зависит от типа родительской программы – *com* или *exe*. Построение родительской *com*-программы рассмотрим на следующем примере. Допустим, что программа исполняет роль простейшего интерпретатора команд DOS. Она запрашивает у пользователя имя файла, содержащего загрузочный модуль требуемой программы, а затем эту программу запускает. При этом могут запускаться только такие программы, которые не требуют от родительской программы каких-то входных данных. После запуска дочерней программы (успешного или неуспешного) на экран опять выводится приглашение ввести имя файла. Для прекращения работы программы достаточно вместо набора имени файла нажать клавишу <Enter>.

```
;          Com-программа запускает другие программы
;          ----------------------------------------
;
;
          org     100h              ; Следующий байт имеет адрес-смещение 100h
Start:    jmp     Begin
;  Данные программы
Buff      db      81                ; Максимальная длина имени файла
Lname     db      0                 ; Фактическая длина имени файла
Namefile  times   81  db  0         ; Здесь размещается имя файла
Msg1      db      'Ошибка распределения памяти', 0Dh, 0Ah, '$'
Msg2      db      'Введите имя com или exe-файла', 0Dh, 0Ah, '$'
Msg3      db      'Ошибка запуска программы',  0Dh, 0Ah, '$'
Blocpar   times   14  db 0          ; Блок параметров
;  Освобождение лишней памяти
Begin:    mov     sp, Lprog         ; SP ← новая база стека
          mov     ah, 4ah           ; Функция изменения области памяти
          mov     bx, (Lprog + 0Fh)/16  ; Новый размер области памяти
          int       21h             ; Изменение размера области
          jnc     .M1               ; Переход при отсутствии ошибки
          mov     ah, 09h           ; Функция вывода строки
          mov     dx, Msg1          ; DX ← адрес сообщения об ошибке
          int     21h               ; Вывод строки
          jmp     .Exit             ; Переход на завершение программы
;  Ввод имени файла
.M1:      mov     ah, 09h           ; Функция вывода строки
          mov     dx, Msg2          ; DX ←  адрес строки-приглашения
          int     21h               ; Вывод строки
          mov     ah, 0ah           ; Функция ввода строки
          mov     dx, Buff          ; DX ← адрес-смещение буфера ввода
          int     21h               ; Ввод строки
          mov     ah, 2             ; Функция вывода символа
          mov     dl, 0ah           ; DL ← символ перевода строки
          int     21h               ; Вывод символа
          cmp     [Lname], byte 0   ; Имя файла отсутствует ?
          jz      .Exit             ; Если да
;  Запуск новой программы
          xor     bx, bx            ; BX ← 0
          mov     bl, [Lname]       ; BL ← длина имени файла
          mov     [Namefile+bx], byte 0  ; Запись нуля после имени файла
          mov     dx, Namefile      ; DX ← смещение имени файла
          mov     bx, Blocpar       ; BX ← смещение блока параметров
          mov     ax, 4b00h         ; Функция и подфункция запуска
          int     21h               ; Запуск дочерней программы
          jnc     .M2               ; Переход при отсутствии ошибки
          mov     ah, 9             ; Функция вывода строки
          mov     dx, Msg3          ; Адрес строки-сообщения об ошибке
          int     21h               ; Вывод строки
```

```
.M2:    jmp    .M1                    ; Повторение для нового файла
;       Завершение программы
.Exit:  mov    ax, 4c00h              ; Возврат в DOS с
        int    21h                    ;  кодом завершения 0
;       Определение области стека
Stek    times  64  dw 0               ; Новая область стека
Lprog   equ    $ - Start +100h        ; Длина программы (включая PSP)
```

Приведенная выше программа выполняет обмен с экраном и клавиатурой, используя следующие системные вызовы DOS:

- «*int 21h*» (функция 9) – вывод строки. Данный вызов уже использовался ранее.
- «*int 21h*» (функция 2) – вывод символа. Перед применением вызова программа помещает в регистр *DL* код выводимого символа. В программе данный системный вызов используется для перевода строки, что исключает наложение на экране строк, принадлежащих родительской и дочерней программам.
- «*int 21h*» (функция *0Ah*) – ввод с клавиатуры символьной строки. В качестве последнего символа строки вводится символ *0Dh* (возврат каретки). (Этот символ помещается в водимую строку драйвером клавиатуры при нажатии клавиши <Enter>.) Перед применением вызова программа помещает в регистры *DS* и *DX* соответственно адрес-сегмент и адрес-смещение для младшего байта буфера, в который должна быть введена строка. Данный буфер имеет три поля:

1) младший байт содержит максимальное число вводимых символов. Запись этого числа выполняет сама программа;

2) второй байт содержит число фактически введенных символов. Запись в этот байт выполняет DOS;

3) остальные байты буфера содержат введенную строку, заканчивающуюся кодом *0Dh*.

Первый исполнительный оператор *jmp* «перепрыгивает» через данные программы на ее фрагмент, выполняющий возврат в систему «лишней» памяти. Вспомним, что для программы DOS выделяет всю ОП, оставшуюся после загрузки самой ОС. Так как наша программа использует лишь небольшую часть этой памяти, то она желает сообщить об этом системе. Простейший вариант этого – оставить за программой те 64 К, в которые первоначально загружается *com*-программа. Но наша программа делает больше: она создает свой новый стек, разместив его сразу за своим кодом, а от всей остальной памяти отказывается.

Само освобождение памяти выполняет ранее рассмотренный системный вызов «*int 21h*» (функция *4Ah*). При этом количество запрашиваемых параграфов памяти определяется в результате целочисленного деления длины программы (константа *Lprog*), увеличенной на 15 (*0Fh*), на длину одного параграфа – 16.

После того, как память освобождена, программа выводит на экран просьбу ввести имя загружаемого файла. Если искомый файл находится в текущем каталоге, то достаточно ввести простое имя файла и расширение имени (*com* или *exe*). Иначе – следует ввести имя-путь файла. Если файл находится на другом логическом диске, то имя-путь должно предваряться именем этого диска, например: *c:\distant\pr1.com* .

Введенное имя файла передается на вход системного вызова «*int 21h*» (функция *4Bh*, подфункция *0*). Этому предшествует небольшая корректировка имени: в последний байт (с кодом *0Dh*) записывается число 0.

Интересно отметить, что данная программа может запускать сама себя. В результате создается новый экземпляр программы, обладающий всеми ее свойствами. *Наберите* текст программы и проверьте практически это и другие свойства данного простейшего интерпретатора команд.

18.2.3. Запуск программ из exe-программы

Освобождение лишней памяти в *exe*-программе выполняется иначе по сравнению с *com*-программой. В то время как виртуальные сегменты *com*-программы «загружены» в единый логический сегмент, который одновременно является сегментом кода, сегментом данных, дополнительным сегментом данных и сегментом стека, виртуальные сегменты *exe*-программы «загружаются» в логические сегменты командами самой программы. При этом в отличие от *com*-программы в *exe*-программе разрешены операции с адресами-сегментами. Перечисленные особенности существенно упрощают определение длины *exe*-программы, измеренной в параграфах. Для этого достаточно записать в конце текста программы фиктивный (пустой) виртуальный сегмент данных с любым именем. Длина программы (в параграфах) есть разность адреса-сегмента фиктивного виртуального сегмента и адреса-сегмента начала всей программы. Для нахождения этой разности следует вспомнить, что перед запуском *exe*-программы загрузчик помещает адрес-сегмент начала программы в регистры *DS* и *ES*.

В качестве примера ниже приведен текст *exe*-программы, выполняющей то же самое, что и рассмотренная выше *com*-программа.

```
;       Exe-программа запускает другие программы
;       ----------------------------------------------------
;
;       Виртуальный сегмент данных
            segment     data
Buff        db    81                        ; Максимальная длина имени файла
Lname       db    0                         ; Фактическая длина имени файла
Namefile    times 81  db  0                 ; Здесь размещается имя файла
Msg1        db    'Ошибка распределения памяти', 0Dh, 0Ah, '$'
Msg2        db    'Введите имя com или exe-файла', 0Dh, 0Ah, '$'
Msg3        db    'Ошибка запуска программы',  0Dh, 0Ah, '$'
Blocpar     times   14   db   0          ; Блок параметров
;       Виртуальный сегмент стека
            segment     stack
            times 64    db   0
;       Виртуальный сегмент кода
            segment     code
..start:
;   «Загрузка» виртуального сегмента данных в логический сегмент данных
            mov   ax, data                ; AX ←нач. параграф сегмента data
            mov   ds, ax                  ; DS ← AX
;   Освобождение  лишней памяти
            mov   ax, es                  ; AX ← адрес-сегмент начала программы
            mov   bx, data1               ; BX←начальный параграф фикт. сегмента
            sub   bx, ax                  ; Новый размер области памяти
```

```
        mov   ah, 4ah                 ; Функция изменения области памяти
        int   21h                     ; Изменение размера области
        jnc   .M1                     ; Переход при отсутствии ошибки
        mov   ah, 09h                 ; Функция вывода строки
        mov   dx, Msg1                ; DX ← адрес сообщения об ошибке
        int   21h                     ; Вывод строки
        jmp   .Exit                   ; Переход на завершение программы
; «Загрузка» виртуального сегмента данных в дополн. сегмент данных
.M1:    mov   ax, data                ; AX ←нач. параграф сегмента data
        mov   es, ax                  ; ES ← AX
;   Ввод имени файла
.M2:    mov   ah, 09h                 ; Функция вывода строки
        mov   dx, Msg2                ; DX ←  адрес строки-приглашения
        int   21h                     ; Вывод строки
        mov   ah, 0ah                 ; Функция ввода строки
        mov   dx, Buff                ; DX ← адрес-смещение буфера ввода
        int   21h                     ; Ввод строки
        mov   ah, 2                   ; Функция вывода символа
        mov   dl, 0ah                 ; DL ← символ перевода строки
        int   21h                     ; Вывод символа
        cmp   [Lname], byte 0         ; Имя файла отсутствует?
        jz    .Exit                   ; Если да
;  Запуск новой программы
        xor   bx, bx                  ; BX ← 0
        mov   bl, [Lname]             ; BL ← длина имени файла
        mov   [Namefile+bx], byte 0   ; Запись нуля после имени файла
        mov   dx, Namefile            ; DX ← смещение имени файла
        mov   bx, Blocpar             ; BX ← смещение блока параметров
        mov   ax, 4b00h               ; Функция и подфункция запуска
        int   21h                     ; Запуск дочерней программы
        jnc   .M3                     ; Переход при отсутствии ошибки
        mov   ah, 9                   ; Функция вывода строки
        mov   dx, Msg3                ; Адрес строки-сообщения об ошибке
        int   21h                     ; Вывод строки
.M3:    jmp   .M2                     ; Повторение для нового файла
;   Завершение программы
.Exit:  mov   ax, 4c00h               ; Возврат в DOS с
        int   21h                     ;   кодом завершения 0
;   Фиктивный виртуальный сегмент данных с именем data1
        segment     data1     data
```

18.3. Многопрограммный редактор

18.3.1. Структура многопрограммного редактора

Переделаем разработанный ранее редактор так, чтобы он состоял не из одной, а из нескольких программ. В результате должна получиться программная система, структура которой приведена на рис. 2.30.

На рис. 2.30 программы изображены в виде параллелограммов. Подобное изображение программ используется для того, чтобы подчеркнуть возможную независимость (параллельность) выполнения программ. При этом выполнение каждой программы представляет собой отдельный программный процесс, изображаемый в виде параллелограмма. Несмотря на то, что подобная параллельность реализуется лишь в мультипрограммных ОС, к которым DOS не относится, мы будем изображать выполнение программ в виде процессов.

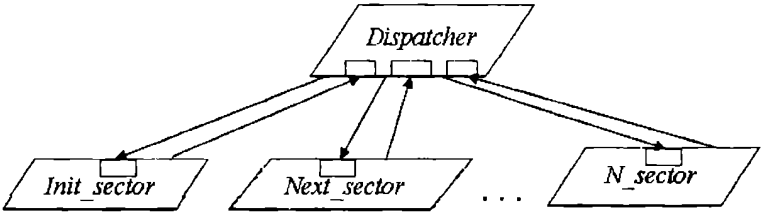

Рис. 2.30. Структура многопрограммного редактора

Небольшие прямоугольники внутри процессов на рис. 2.30 обозначают точки входа. ***Точка входа в процесс*** – место в алгоритме программы процесса, находясь в котором процесс «простаивает», ожидая инициирующего (запускающего) воздействия со стороны других процессов.

Как видно из рис. 2.30, главной программой многопрограммного редактора является *Dispatcher*. Но в отличие от однопрограммного варианта, она инициирует не свои внутренние процедуры, а отдельные программы, каждая из которых инициируется при нажатии пользователем своей функциональной клавиши. Перечислим эти программы: *Init_sec*, *Next_sec*, *Edit_sec*, *Prev_sec*, *N_sector*. Прежде чем инициировать соответствующую программу, диспетчер должен загрузить в ОП загрузочный модуль программы – файл с расширением имени *com* или *exe*. И загрузка и инициирование программы выполняются диспетчером с помощью рассмотренного ранее системного вызова *EXEC*.

Стрелки на рис. 2.30 обозначают управляющие (инициирующие) воздействия. Такое воздействие поступает на точку входа инициируемой программы. Одновременно инициирующая программа сама оказывается в одной из точек своих входов. Например, программа *Dispatcher* после запуска любой своей дочерней программы сразу же оказывается в той своей точке входа, которой соответствует оператор исходной программы, расположенный сразу же после вызова *EXEC*.

18.3.2. Главная программа

Применение вызова *EXEC* для запуска различных программ редактора отличается лишь именами *exe*- или *com*-файлов, содержащих загрузочные модули этих программ. Для того чтобы осуществлять поиск этих имен, в программе диспетчера удобно использовать измененную таблицу переходов. содержащую вместо имен процедур имена до-

черних файлов. При этом в соответствии с требованиями вызова *EXEC* каждое имя файла должно заканчиваться нулевым байтом.

Одновременно с запуском дочерней программы программа *Dispatcher* передает ей на вход данные, предназначенные для обработки в этой запускаемой программе. В качестве таких данных используется совокупность глобальных переменных, примером которых является переменная *Sector*. Несмотря на то, что эти глобальные переменные использовались нами и ранее для информационной взаимосвязи между модулями программы, механизм их передачи будет теперь другой. Дело в том, что в отличие от процедур, входящих в состав одной программы, самостоятельные программы не имеют общих областей (разделов) памяти.

Поэтому данные, передаваемые из родительской программы в дочернюю программу, включаются: 1) или в состав «наследуемых» данных – в состав *PSP* и (или) блока окружения. Эти данные копируются из *PSP* (или из блока окружения) родительской программы в *PSP* (или в блок окружения) дочерней программы; 2) родительская программа передает с помощью *PSP* адрес области ОП, которая содержит данные, передаваемые на вход дочерней программы. Второй из этих подходов и будет использоваться нами в многопрограммном редакторе. Причем область памяти, указатель на которую находится в *PSP* дочерней программы, будет использоваться не только для передачи ее входных, но и выходных данных.

Что касается типа загрузочных модулей программ, то удобнее передавать данные между *exe*-программами. В этом случае передаваемую область данных следует оформить в родительской *exe*-программе в виде виртуального сегмента данных, адрес-сегмент которого и передается в дочернюю программу. Для размещения такого указателя (адреса-сегмента) на общую область данных можно использовать любое свободное слово в *PSP* дочерней программы. Например, для этого пригодно слово *PSP*, предназначенное для устаревших блоков управления файлами и расположенное по относительному адресу *5ch*.

Для сравнения, при передаче данных между программами типа *com* указатель на общую область данных включает два слова – адрес-сегмент и адрес-смещение начала области данных. Это является следствием того, что при получении *com*-программы адрес-сегмент переменной вычисляется не для начала виртуального сегмента, в котором переменная определена, а для начала программы. Еще одним достоинством использования *exe*-программ является относительная простота операции освобождения лишней памяти.

Следует добавить, что для того, чтобы записать в *PSP* дочерней программы (например, по адресу *5ch*) указатель (адрес-сегмент) на свой виртуальный сегмент данных, родительская программа обращается к DOS, используя системный вызов *EXEC*. Напомним, что в одно из полей блока параметров, формируемого перед выдачей *EXEC*, родительская программа может поместить один или два указателя на строки байтов, помещаемые в *PSP* дочерней программы со смещением *5ch* относительно начала *PSP* (и всей программы). Каждый такой указатель, помещаемый в блок параметров, есть совокупность адреса-смещения (записывается первым) и адреса-сегмента. Длина одной строки байтов, помещаемой в *PSP*, не может превышать 12 байт. Этого вполне достаточно для передачи указателя (адреса-сегмента) на область глобальных переменных, так как длина такого указателя всего 2 байта.

Нетрудно заметить, что в нашем случае имеет место двойная косвенная адресация – указатель, помещаемый в блок параметров, есть адрес (смещение и сегмент) другого указателя, помещаемого сначала в новую глобальную переменную программы –

Seg_par, а затем – в *PSP* дочерней программы. Этот второй указатель представляет собой адрес (сегмент) виртуального сегмента данных главной программы, содержащего глобальные переменные.

Для получения исходного файла *Dispatch.asm* следует воспользоваться ранее полученным его текстом, внеся в него ряд изменений. Ниже приведен результат этих изменений.

```
;       Главная программа многопрограммного редактора
;       ---------------------------------------
;
cr          equ     0dh
lf          equ     0ah
            absolute  80h
Lparam      resb    1                   ; Длина хвоста
Param       resb    1                   ; Первичный байт хвоста
;   Виртуальный сегмент данных
            segment     data
; Глобальные переменные
Sector      times   256  db  0          ; Буфер для сектора памяти
Address     dw      0                   ; Нач. адрес сектора в области памяти
.   .   .   .   .   .   .   .
Segin       dw      0                   ; Адрес-сегмент начала программы
Seg_par     dw      0                   ; Адрес-сегмент передаваемых данных
; Блок параметров для запуска дочерней программы
Blocpar     times   6   db   0          ; Первые 6 байт блока параметров
Ukas        times   8   db   0          ; Указатель на передаваемую информацию
;   Виртуальный сегмент стека
      segment     stack
      times       64  db  0
;   Виртуальный сегмент кода
      segment     code
..start:
; «Загрузка» виртуального сегмента данных в дополнит. сегмент данных
      mov     ax, data                  ; AX ← нач. параграф сегмента data
      mov     es, ax
      mov     [es:Segin], ds            ; Запоминание адреса-сегмента программы
;  Подготовка данных, передаваемых в дочернюю программу
      mov     bx, Seg_par               ;BX ← смещение Seg_par в сегменте data
      mov     [es:Ukas],bx              ; Ukas ← смещение переменной Seg_par
      mov     bx,data                   ; BX ← нач. параграф сегмента data
      mov     [es:Ukas+2],bx            ; Ukas+2 ← нач. параграф сегмента data
      mov     [es:Seg_par],bx           ; Seg_par ← нач. параграф сегмента data
;    Освобождение лишней памяти
.   .   .   .   .   .   .   .
;    Определение наличия имени файла
.   .   .   .   .   .   .   .
;    Получение длины  имени файла
.   .   .   .   .   .   .   .
;    Открытие файла
```

```
;    Определение длины файла
;    Сообщение о длине файла
;    Прежний диспетчер
     mov    ax,4c00h              ; Возврат в DOS
     int    21h                   ;   с кодом возврата 0
;
;        Интерпретатор команд
;        --------------------
;  Входы:  AH – скан-код, соответствующий команде
;  Чтение: .Table – таблица переходов
;
Command:
     push   ax
     push   bx
     push   cx
     push   dx
     push   ds
; «Загрузка» виртуального сегмента кода в основной сегмент данных
     mov    bx, cs
     mov    ds, bx
; Поиск в таблице имени файла запускаемой программы
     mov    bx, .Table            ; BX ← адрес-смещение таблицы
.M1: cmp    byte[bx], 0ffh        ; Конец таблицы?
     je     .Exit                 ; Да, кода  нет в таблице
     cmp    ah,[bx]               ; Это вход в таблицу?
     je     .Dispatch             ; Да, выполнить команду
     add    bx,14                 ; Нет, переход к
     clc                          ;   следующему
     jmp    .M2                   ;   элементу таблицы
; Обработчик команды выполняет запуск соответствующей программы
.Dispatch:
     inc    bx                    ; В BX адрес-смещение  имени файла
     mov    dx, bx                ; DX ← BX
     mov    bx, Blocpar           ; BX ← адрес-смещение блока параметров
     mov    ax, 4b00h             ; AX ← номер функции и подфункции
     int    21h                   ; Запуск дочерней программы
     jnc    .Exit                 ; Если запуск успешный
     mov    dx, .Msg              ; Сообщение
     mov    ah, 9                 ;   об ошибке
     int    21h                   ;   запуска
.Exit: stc                        ; CF ← 1
.M2: jnc    .M1                   ; Повторение для нового элемента таблицы
     pop    ds
     pop    dx
     pop    cx
```

```
        pop   bx
        pop   ax
        ret                          ; Возврат из процедуры Command
.Msg    db    cr, lf, 'Ошибка запуска программы', cr, lf, '$'
;   Таблица переходов содержит скан-коды управляющих клавиш
;     и имена программ, выполняющих соответствующие команды
.Table  db    3bh                    ; <F1>
        db    'Init_sec.exe', 0
        db    3dh                    ; <F3>
        db    'Next_sec.exe', 0
        db    3eh                    ; <F4>
        db    'Edit_sec.exe', 0
        db    3fh                    ; <F5>
        db    'Prev_sec.exe', 0
        db    40h                    ; <F6>
        db    'N_sector.exe', 0
        db    0ffh                    ; Конец таблицы
        %include  'Video_io.asm'
        %include  'Cursor.asm'
        %include  'Kbd_io.asm'
; Фиктивный виртуальный сегмент
        segment   data1   data
```

В приведенной выше программе глобальные переменные программы собраны вместе в единый виртуальный сегмент данных *data*. Начальный адрес-сегмент этого блока записываются в глобальную переменную *Seg_par*. Указатель на эту переменную записывается сначала в поле *Ukas*, а затем копируется в *PSP* дочерней программы во время выполнения системного вызова *EXEC* (оператор «*int 21h*», функция *4b00h*).

Обратим внимание, что все поля таблицы переходов, содержащие имена запускаемых файлов, имеют одинаковую длину 13 байт. Кроме того, следует заметить, что в конце программы не подсоединен файл *Disp_sec.asm*, но зато подсоединены файлы *Video_io.asm*, *Cursor.asm* и *Kbd_io.asm*. Эти файлы подсоединялись к программе и раньше, но через файл *Disp_sec.asm*, который теперь в главной программе не нужен.

18.3.3. Дочерние программы

В рассматриваемом варианте многопрограммного редактора каждая из взаимодействующих программ имеет дело со своим набором глобальных переменных. Поэтому сразу же после своего инициирования дочерняя программа копирует в свою собственную память набор глобальных переменных, принадлежащий программе *Dispatcher*. И, наоборот, перед своим завершением дочерняя программа выполняет обратное копирование глобальных переменных из своей области памяти в область памяти диспетчера. Такое копирование глобальных переменных приводит к повышенным затратам памяти (каждая программа имеет свой экземпляр глобальных переменных), но повышает надежность всей программной системы, так как сбой в работе одной из программ не приведет к нарушению правильности главной совокупности глобальных переменных.

Кроме того, использование в каждой программе своего набора глобальных переменных существенно упрощает процесс программирования. Дело в том, что, имея указатель (адрес-сегмент) области памяти, содержащей общие данные (в нашем случае такими

общими данными являются глобальные переменные программы), дочерняя программа может обрабатывать требуемую глобальную переменную, используя ее численное смещение относительно начала области. Для программиста это неудобно, так как мы привыкли использовать в исходной программе не численные смещения, а символьные имена переменных. При работе с областью памяти, принадлежащей другой программе, нельзя использовать символьные имена (метки), определенные в другой программе, так как процессы получения разных программ никак не связаны между собой. Это принципиально отличает получение программной системы из нескольких программ от получения одной, хотя бы и большой программы. Выходом является копирование в дочерней программе глобальных переменных родительской программы в свою собственную область и дальнейшая работа с этими переменными по их именам.

Для выполнения самой операции копирования удобно использовать рассмотренный нами ранее оператор *movsb* с префиксом *rep*. Напомним, что перед применением данного оператора следует сбросить флаг *DF* (оператором *cld*) и записать в регистр *CX* число переписываемых байтов. Что касается начальных адресов области-источника и области-приемника, то каждый из них задается двумя регистрами:

(*DS:SI*) – адрес области-источника;

(*ES:DI*) – адрес области-приемника.

Что касается исходных текстов дочерних программ, то они мало отличаются от текстов соответствующих процедур. Эти отличия находятся лишь в начале и в конце программ. Причем во всех дочерних программах отличия одинаковы:

```
;    Программа выполняет одну из команд пользователя
;        ---------------------------------------------
;
absolute    5ch                 ; Следующий байт имеет адрес-смещение 5ch
Seg_glob    resw  1             ; Адрес-сегмент области глобальных переменных
;   Виртуальный сегмент данных
            segment     data
; Глобальные переменные (экземпляр данной программы)
Sector    times 256 db   0          ; Буфер для сектора памяти
Address   dw  0                     ; Нач. адрес сектора в области памяти
.   .   .   .   .
Segin     dw   0                    ; Адрес-сегмент начала программы
;   Виртуальный сегмент стека
            segment stack
            times     64     db   0
;   Виртуальный сегмент кода
            segment code
..start:
; «Загрузка» виртуального сегмента данных в дополнит. сегмент данных
            mov     ax, data
            mov     es, ax
; Копирование глобальных переменных в сегмент данных
            mov     cx, 268             ; CX ← длина копируемой области
            cld                         ; DF ← 0
            push    ds                  ; Запоминание прежнего содержимого DS
            mov     ax, [Seg_glob]      ; «Загрузка» копируемой области
            mov     ds, ax              ;   в основной сегмент данных
```

```
        xor     si, si                  ; SI ← 0
        xor     di, di                  ; DI ← 0
rep     movsb                           ; Операция копирования
        pop     ds                      ; Восстановление содержимого DS
; Выполнение действий, обусловленных функциями программы
. . . . . .
; Обратное копирование глобальных переменных
        mov     cx, 268                 ; CX ← длина копируемой области
        cld                             ; DF ← 0
        mov     ds, [es:Segin]          ; DS ← адрес-сегмент начала программы
        mov     bx, [Seg_glob]          ; BX ← адрес глобальных переменных
        mov     ax, es                  ; «Загрузка» в основной сегмент данных
        mov     ds, ax                  ;   виртуального сегмента данных
        mov     es, bx                  ; «Загрузка» в дополнит. сегмент данных
        xor     si, si                  ; SI ← 0
        xor     di, di                  ; DI ← 0
rep     movsb                           ; Операция копирования
;   Завершение программы
        mov     ax, 4c00h
        int     21h
;   Подсоединение файлов операторами %include
    . . .
```

Исходные тексты дочерних программ *Init_sec.asm*, *Next_sec.asm*, *Edit_sec.asm*, *Prev_sec.asm* и *N_sector.asm* мало отличаются от ранее полученных текстов соответствующих процедур, которые были записаны в файлы *Disp_sec.asm, Edit_sec.asm*. Поэтому для получения текстов указанных программ целесообразно получить копии этих двух файлов, присвоив копиям требуемые имена (*Init_sec.asm* и т. д.), а затем внести в содержимое файлов-копий требуемые изменения.

Лабораторная работа 16

Требуется реализовать описанный выше многопрограммный редактор информации.

19. Прерывания и резидентные программы

19.1. Прерывания

19.1.1. Внешние аппаратные прерывания

Прерыванием называется навязанное принудительно центральному процессору прекращение выполнения текущей программы и переход его на выполнение подпрограммы, которая называется ***обработчиком прерываний***. В работе любой ВС (за исключением простейших) прерывания играют исключительно важную роль. Приводимые ниже примеры иллюстрируют это.

Допустим, что на ЦП выполняется прикладная программа, вошедшая в бесконечный цикл вследствие ошибки программирования. Тогда для прекращения выполнения этой программы достаточно нажать ту комбинацию клавиш, которая специально предназначена для этой цели в используемой ВС. Например, в DOS это одновременное нажатие клавиш <Ctrl> и <C>. Следствием этого является выдача интерфейсным устройством

клавиатуры аппаратного сигнала прерывания, который поступает в ЦП по ОШ. Процессор прерывает выполнение зациклившейся программы и начинает выполнять обработчик прерываний клавиатуры. Эта подпрограмма распознает код нажатой комбинации клавиш (<Ctrl>+<C>) и обеспечивает завершение программы. (Обработчик прерываний клавиатуры прекращает выполнение текущей программы не сам, а использует для этого программное прерывание «*int 23h*».)

В качестве второго примера рассмотрим обслуживание таймера. ***Таймер*** – аппаратное устройство, выдающее сигналы прерываний в ЦП через фиксированные промежутки времени. Эти сигналы называются ***тиками***. Их можно использовать в ВС, например, для определения времени суток. Для этого по приходу каждого тика ЦП прерывает выполнение текущей прикладной программы и начинает выполнять обработчик прерываний таймера, который, в свою очередь, увеличивает на единицу счетчик тиков и, возможно, пересчитывает текущее время, увеличивая значения счетчиков секунд, минут и часов.

Прерывания от клавиатуры и таймера являются примерами ***внешних аппаратных прерываний***. Сигналы этих прерываний выдаются в ЦП различными ПУ (а точнее – их ИУ), которым требуется внимание со стороны программ ЦП. Эти сигналы передаются в ЦП по шине управления, которая входит в состав ОШ. Характерной особенностью этих прерываний является то, что их первичной причиной являются процессы, асинхронные (независимые) по отношению к текущей работе ЦП. Примером такого асинхронного процесса является деятельность пользователя по нажатию клавиш, приводящая к возникновению прерываний от клавиатуры и от мыши.

Аппаратные процессы, протекающие в ПУ, также могут являться первичными источниками внешних аппаратных прерываний. Характерной особенностью соответствующих сигналов прерываний является то, что они должны обрабатываться в реальном времени. Термин ***реальное время*** (***РВ***) означает, что интервал времени между моментом появления сигнала прерывания и завершением его обработки не должен превышать предельно допустимой величины, которая зависит от типа сигнала прерывания. Например, для таймера этот интервал не должен превышать один тик. Задержка завершения обработки на большую величину приведет к потере тика и, следовательно, к нарушению правильности системного времени.

Внешние аппаратные прерывания разделяются на маскируемые и немаскируемые прерывания. ***Немаскируемые прерывания*** – наиболее важные прерывания, обработка которых не может быть отложена ни на какое время. Сюда относятся прерывания, возникающие при сбоях питания. Соответствующие сигналы прерываний выдаются в ЦП аппаратными схемами контроля питания. ***Маскируемые прерывания*** – прерывания, обработка которых может быть отложена на время, требуемое ЦП для выполнения какой-то другой, более важной операции. Запрет всех маскируемых прерываний выполняет машинная команда ***cli***, а разрешение – команда ***sti***. Первая из этих команд сбрасывает, а вторая устанавливает флажок разрешения прерываний *IF* в регистре *FLAGS*.

19.1.2. Исключения

Кроме внешних аппаратных прерываний существуют также ***внутренние аппаратные прерывания***, называемые обычно ***исключениями***. Источником аппаратного сигнала исключения является одна из аппаратных схем самого ЦП. Он выдается в том случае, если при выполнении на ЦП очередной машинной команды возникла ситуация, требующая помощи со стороны системных программ. Например, если при выполнении на

ЦП команды *div* (деление), получилось частное, величина которого превышает предельно допустимую величину для 16-битового регистра, то возникает исключение «деление на нуль». Стандартный обработчик данного исключения завершает выполнение программы, а также выводит соответствующее сообщение на экран.

В качестве второго примера рассмотрим исключение «трассировка». Данное исключение будет происходить при выполнении на ЦП каждой машинной команды, если установлен флаг *TF* в регистре *FLAGS*. В результате этого наша прикладная программа будет «спотыкаться» – по завершению каждой команды программы будет инициироваться обработчик исключения. Результат работы этого обработчика зависит от целей трассировки. Так как обычно трассировка выполняется с целью обнаружения ошибок в программе, то обработчик данного исключения выполняет выдачу на экран содержимого регистров ЦП. Подобный обработчик исключений входит в состав системной программы, называемой ***отладчиком***. Пример отладчика – Debug.

В завершение примера отметим, что в i8086 отсутствуют специальные команды для установки и сброса флага *TF*. Поэтому для выполнения требуемого действия отладчик должен: 1) с помощью команды *pushf* записать (скопировать) содержимое *FLAGS* в стек; 2) скорректировать слово в вершине стека, установив или сбросив бит 8; 3) с помощью команды *popf* выбрать слово из вершины стека и поместить его в регистр *FLAGS*.

19.1.3. Программные прерывания

Третий тип прерываний – ***программные прерывания*** (термин неудачный, но общепринятый). Причиной такого прерывания является сама программа (отсюда и название), а именно – попадание на ЦП машинной команды «*int n*», где *n* – номер прерывания. Подобно исключениям программные прерывания являются синхронными к текущей работе ЦП, так как обслуживают выполняемую на нем программу. Но в отличие от исключений, моменты возникновения которых автору прикладной программы заранее не известны, команды *int* помещаются программистом в текст программы сознательно с целью выполнения системных вызовов.

Более того, следует четко представлять себе, что в однопрограммной ВС не существует иного способа вызова из прикладной программы системных управляющих подпрограмм (ОС или BIOS) кроме применения команды программного прерывания *int*.

Примером системного вызова DOS является команда «*int 21h*». Особенностью этого вызова является то, что под номером *21h* «скрывается» программное прерывание, инициирующее ***диспетчер функций*** DOS. Эта подпрограмма обрабатывает огромное большинство системных вызовов, адресованных DOS из обрабатывающих программ. Для того чтобы диспетчер функций понял, какую внутреннюю процедуру DOS ему следует вызвать, в прикладной программе следует задать номер требуемой функции (в регистре *AH*), а если этого недостаточно, то и номер подфункции (в регистре *AL*).

Как и процедура, обработчик программного прерывания имеет входные и (или) выходные параметры. Для их передачи используются регистры данных и, возможно, стек программы. Использование стека менее предпочтительно, так как входные параметры будут в нем «пригружены» адресом возврата и содержимым регистра *FLAGS*. А выходные параметры должны быть «пригружены» этими же данными самим обработчиком, так как иначе адрес возврата не будет найден. Несмотря на то, что перечисленные трудности устраняются программой с помощью простых команд *push* и *pop*, на их выполнение требуются дополнительные затраты времени ЦП.

Команды программного прерывания *int* могут применяться не только для реализации системных вызовов, но и для реализации вызова прикладных подпрограмм. Реализация прикладной подпрограммы не в виде процедуры, а в виде обработчика прерываний це-

лесообразна только тогда, когда эта подпрограмма вызывается не из одной, а из нескольких прикладных программ. Только в этом случае удается сократить размер прикладных программ за счет того, что в их код не будет включен код подпрограммы. (При использовании процедуры, наоборот, ее код должен быть включен в код каждой программы, использующей ее.) Реализация подобного вызова подпрограмм требует, чтобы они были резидентны, т. е. постоянно находились бы в ОП. В противном случае выполнение каждой прикладной программы, использующей подпрограмму, должно было бы предваряться инициализацией соответствующего обработчика прерываний, что очень неудобно. Вопросы разработки резидентных программ будут рассмотрены в п. 19.2. Там же будет приведен пример обработчика программных прерываний.

Наряду с процедурами обработчики программных прерываний относятся к подпрограммам. Поэтому команда вызова такой подпрограммы *int* очень похожа на команду дальнего вызова процедуры *call*, а команда завершения обработчика прерываний *iret* – на команду возврата из дальней процедуры – *ret*. Работа данных команд будет рассмотрена далее совместно с другими этапами общего алгоритма обработки прерываний.

19.1.4. Алгоритм выполнения прерывания

Прежде чем приводить данный алгоритм, рассмотрим два понятия, играющих в нем важную роль. Во-первых, это ***номер прерывания***. Подобно другим однотипным объектам в ВС, все типы прерываний пронумерованы. Номер – ***численный идентификатор*** объекта, пользуясь которым система находит дескриптор объекта в своих таблицах. В данном пособии нам уже встречались и еще встретятся другие численные идентификаторы. Отметим их общее свойство: однотипные идентификаторы уникальны в пределах всей однопрограммной ВС. Например, в системе, построенной на основе i8086 и DOS номера упомянутых выше прерываний такие:

00h – Деление на нуль
01h – Трассировка
02h – Немаскируемое прерывание
. . . .
08h – Таймер
09h – Клавиатура
. . . .
10h–1Fh Прерывания BIOS
20h–3Fh Прерывания DOS
. . . .

Вторым понятием, необходимым для изложения алгоритма обработки прерываний, является ***таблица векторов прерываний***. При использовании в качестве ЦП i8086 эта таблица занимает первые 1024 байта ОП (рис. 2.31) и эти байты никогда нельзя использовать для других целей. Один элемент этой таблицы занимает два слова (4 байта) ОП и называется ***вектором прерываний***. Он соответствует своему типу прерывания и содержит адрес (сегмент и смещение) первой ячейки обработчика прерываний в ОП. При этом первое слово содержит смещение, а второе – сегмент. Общее число векторов прерываний *256 = 100h*. Следовательно, таково максимальное число типов прерываний, которые могут существовать в системе.

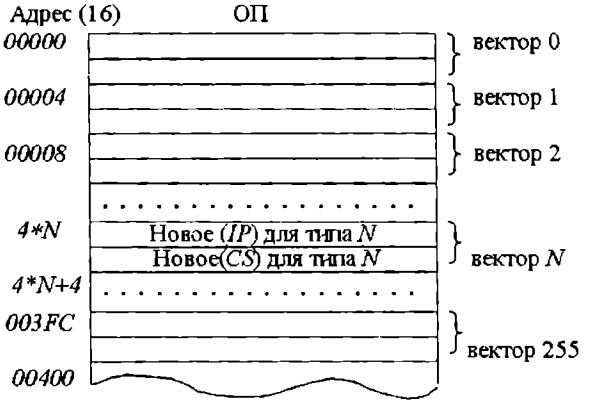

Рис. 2.31. Таблица векторов прерываний

Обработка прерывания производится совместными усилиями аппаратуры ЦП и программы обработчика прерываний. Общий алгоритм обработки любого типа прерываний:

1. Поступление сигнала прерывания в ЦП. При этом сигнал маскируемого внешнего аппаратного прерывания поступает на вход *INTR* процессора, сигнал немаскируемого внешнего аппаратного прерывания – на вход *NMI*, а сигнал исключения или программного прерывания поступает на внутреннюю линию процессора.

2. Собственно прерывание. Оно выполняется ЦП по окончанию выполнения текущей машинной команды и включает действия:

- Текущее содержимое *FLAGS*, *CS* и *IP* помещается в стек (рис. 2.32). Последние два слова представляют собой адрес возврата в прерванную программу.
- Считывание с ОШ или с внутренней шины процессора 8-битового числа – номера прерывания *N*.
- Копирование вектора прерываний с номером *N* в регистры *IP* и *CS*. В результате эти регистры содержат стартовый адрес обработчика прерываний.
- Сбрасывание в нуль флажков *IF* и *TF* в регистре *FLAGS*. В результате запрещаются все маскируемые внешние прерывания (кроме того, которое уже обрабатывается), а также запрещаются исключения «трассировка».

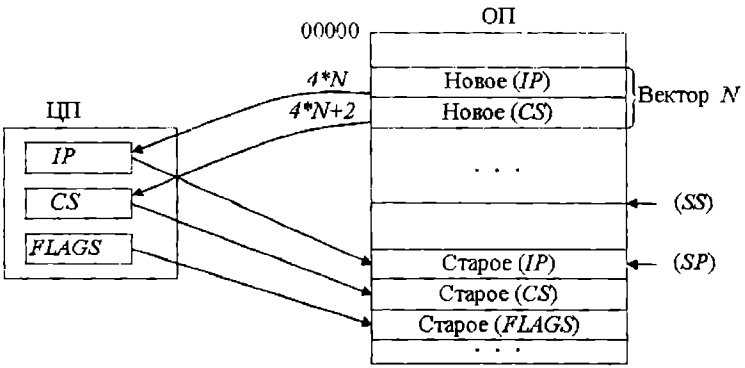

Рис. 2.32. Перемещение содержимого регистров при прерывании с номером *N*

3. Начальный этап программной обработки прерывания. Он выполняется обработчиком прерываний и включает действия:

- Разрешение маскируемых прерываний с помощью команды *sti*. В результате важные маскируемые прерывания, например от таймера, не будут более откладываться и вызовут прерывание текущего обработчика прерываний точно так, как прерывается обыкновенная программа.
- Сохранение в программном стеке содержимого тех регистров, с которыми будет работать программа обработчика.
- Запись в сегментный регистр данных *DS* значения, которое соответствует адресу-сегменту данных обработчика прерываний. Это выполняется потому, что в результате прерывания из всех сегментных регистров только *CS* «переключается» на обработчик прерываний, а остальные сегментные регистры по-прежнему адресуют данные прерванной программы.

4. Действия, определяемые типом прерывания. Эти действия обработчик прерываний выполняет сам или обращается за помощью к другим подпрограммам ОС, вызывая их как обычные процедуры командами *call* или используя для их вызова команды программного прерывания *int*.

5. Завершение обработки прерывания. Сюда относятся действия:

- Восстановление содержимого регистров, запомненного ранее в стеке.
- Возврат из прерывания выполняет команда *iret*, завершающая программу обработчика прерываний. (При попадании этой команды на ЦП его аппаратура выталкивает из стека в регистры *IP*, *CS* и *FLAGS* прежнее их содержимое. В результате следующей выполняемой на ЦП командой будет очередная команда прерванной программы.)

В дальнейшем мы не раз вернемся к данному алгоритму, используя его для написания программ обработчиков прерываний, а также для получения алгоритма выполнения маскируемых внешних прерываний (применение дополнительной аппаратуры потребует уточнения некоторых этапов рассмотренного алгоритма).

В заключение заметим, что среди трех типов прерываний (программные, исключения, внешние аппаратные) программные прерывания – наименее, а внешние аппаратные – наиболее приоритетные прерывания. Это означает, что обработчики программных прерываний могут прерываться сигналами исключений и сигналами внешних прерываний так, как прерываются обыкновенные прикладные программы. Обработчики исключений, в свою очередь, могут прерываться сигналами внешних аппаратных прерываний. Что касается обработчиков внешних прерываний, то они могут прерываться сигналами других, более приоритетных внешних прерываний. (Чтобы такие прерывания были действительно возможны, прерываемые обработчики прерываний должны предварительно выполнить команду *sti*.)

19.2. Резидентные программы

19.2.1. Особенности резидентных программ

Обычные прикладные программы присутствуют в ОП только во время своего выполнения. Когда это выполнение завершено (например, в результате системного вызова «*int 21h*», функция *4Ch*), вся занимавшаяся программой память освобождается и может быть распределена другой программе. Но если данная программа может потребоваться в дальнейшем для многократного обслуживания других прикладных программ и (или) для

особой обработки каких-то событий, возникающих в ВС, то желательно сделать данную программу резидентной. ***Резидентная программа*** – программа, постоянно находящаяся в ОП. Например, резидентна основная часть ОС, называемая ***ядром ОС***.

Для того чтобы сделать свою программу резидентной, достаточно воспользоваться системным вызовом «***int 27h***». Этот вызов завершает выполнение программы, оставив ее в памяти. Перед применением вызова программа должна поместить в регистр *DX* адрес-смещение для байта, расположенного непосредственно после резидентной программы.

Сразу же после своего создания резидентная программа занимает место в памяти, но ничего не делает (не выполняется на ЦП). Это продолжается до тех пор, пока программа не будет инициирована (запущена). Единственно возможным источником такого инициирования является поступление в ЦП сигнала прерывания. Поэтому резидентная программа обязательно является обработчиком прерываний.

Вспомним, что обязательным условием запускаемости обработчика прерываний является размещение его стартового адреса в векторе прерываний, номер которого совпадает с номером прерывания. Для такого размещения следует использовать системный вызов «***int 21h***» (функция ***25h***). Перед применением вызова программа должна поместить в регистр *AL* номер прерывания, а в регистры *DS* и *DX* соответственно адрес-сегмент и адрес-смещение для стартового адреса обработчика прерываний.

Выбор номера прерывания зависит от того, какую функцию будет выполнять обработчик прерываний. Если выполняемая им функция уже реализована в системе, то новый обработчик прерываний должен ориентироваться на уже используемый номер прерывания. Иначе – для новой функции необходимо использовать и новый номер прерывания. Вначале рассмотрим второй случай.

19.2.2. Использование свободных прерываний

Для выбора свободного номера прерывания можно использовать отладчик Debug. С помощью его команды «*d 0:0*» просмотрим начало области памяти, занимаемой таблицей векторов прерываний. (Для продолжения просмотра достаточно вводить команду *d*.) В этой области выберем любую последовательность из четырех нулевых байтов, номер младшего из которых делится нацело на четыре. Тогда частное от такого деления и есть искомый номер прерывания. Например, пусть выбранная последовательность четырех байтов начинается с номера *200h* = *512*. Тогда искомый номер прерывания *n* = *= 200h/4 = 80h = 128*.

Пример. Сделаем резидентной «вежливую» *com*-программу, рассмотренную ранее. Для ее инициирования будем использовать программное прерывание с номером *80h*. Текст программы на ассемблере:

```
;    Резидентная программа выводит сообщение «Здравствуйте» на экран
;    ---------------------------------------------------------------
;      Вызов программы:  команда  int 80h
;
cr          equ     0dh             ; Код ASCII возврата каретки
lf          equ     0ah             ; Код ASCII перевода строки
            absolute 2ch            ; Следующий байт имеет адрес-смещение 2ch
Blocokr     resw    1               ; Адрес-сегмент блока окружения
            segment  code           ; Виртуальный сегмент кода
            org     100h
```

```
Start:  jmp     Init            ; Переход на инициализацию
;     Обработчик прерываний
Msg     db    'Здравствуйте', cr, lf, '$'   ; Выводимое сообщение
Begin:  sti                     ; Разрешить маскируемые прерывания
        push    ax              ; Сохранение содержимого
        push    dx              ;   регистров
        push    ds              ;   в стеке
        mov     ax, cs          ; DS  будет адресовать данные
        mov     ds, ax          ;   резидентной программы
        mov     dx, Msg         ;  DX ← адрес строки-сообщения
        mov     ah, 9           ;  Вывод строки
        int     21h             ;   на экран
        pop     ds              ; Восстановление содержимого
        pop     dx              ;   регистров
        pop     ax              ;   из стека
        iret                    ; Возврат из прерывания
;   Инициализационная часть программы
Init:   mov     ax, 2580h       ; Запись стартового адреса обработчика
        mov     dx, Begin       ;   в вектор прерываний
        int     21h             ;   с номером 80h
        mov     ah, 49h         ; Освобождение области памяти,
        mov     es, [Blocokr]   ;   занимаемой
        int     21h             ;   блоком окружения
        mov     dx, Init        ;  Возврат в DOS, оставшись
        int     27h             ;   резидентным
```

Данная программа состоит из двух основных частей: 1) обработчик прерываний (резидентная часть); 2) инициализационная часть. Инициализационная часть нерезидентна по той причине, что она выполняется всего один раз, и нет смысла постоянно держать ее в памяти. При этом следует учитывать то, что так как резидентная программа находится в ОП во время выполнения других программ, то занимаемую ею память следует минимизировать.

Новым средством уменьшения затрат памяти, которое применено в программе, является освобождение области, выделенной для блока окружения программы. Напомним, что такая область статически назначается любой программе, и ее размер кратен 512 байт. Подобное освобождение возможно потому, что наш обработчик прерываний (как и любой другой) эту область не использует.

Обратим внимание, что первые команды обработчика прерываний запоминают в стеке, а последние команды извлекают из стека содержимое тех регистров, в которые выполняется запись в программе обработчика. Среди этих регистров есть и регистр сегмента данных *DS*, в который обработчик записывает новое значение, совпадающее с содержимым регистра *CS*. Это обусловлено тем, что при вызове обработчика (через прерывание) новое значение записывается лишь в единственный сегментный регистр *CS* (это значение берется из вектора прерываний). Остальные регистры сегментов (*DS*, *SS* и *ES*) сохраняют свое значение, полученное ранее в вызывающей программе. Поэтому для того, чтобы программа обработчика могла использовать свои данные, она помещает в *DS* соответствующий адрес-сегмент (для *com*-программ он совпадает с содержимым *CS*).

Интересно добавить, что, так как содержимое регистров *SS* и *SP* поступает в обработчик прерываний из вызывающей программы без изменений, то обработчик использует тот же самый стек, что и программа. Это свойство позволяет достаточно просто выполнять информационный обмен между программой и обработчиком, используя стек.

Выполните данную резидентную программу. Для ее тестирования необходимо после инициализации резидентной программы запустить вторую *com*-программу, выполняющую запуск резидентной программы с помощью команды *int*.

19.2.3. Перехват прерываний

Перейдем к рассмотрению более сложного случая, когда обработчик прерываний должен использовать уже занятый вектор прерываний. В этом случае требуется выполнить ***перехват прерываний*** – запись в вектор прерываний стартового адреса нового обработчика. Это нетрудно сделать, используя рассмотренный ранее системный вызов «*int 21h*», функция *25h*. Но дополнительная задача заключается в том, что обычно требуется сохранить прежнее содержимое вектора прерываний для того, чтобы прежний обработчик прерываний можно было бы впоследствии инициировать.

Для чтения прежнего содержимого вектора прерываний используется системный вызов «***int 21h***», функция ***35h***. Перед выполнением вызова программа должна поместить в регистр *AL* номер прерывания. В результате выполнения вызова регистры *ES* и *BX* содержат соответственно адрес-сегмент и адрес-смещение, находящиеся в векторе прерываний. Далее эти адреса могут быть переписаны для сохранения в любое место ОП.

Инициирование прежнего обработчика прерываний может потребоваться в двух случаях. Во-первых, в том случае, если новый обработчик прерываний не заменяет, а лишь расширяет функции, выполнявшиеся прежним обработчиком. При этом после того, как новый обработчик выполнится, он должен передать управление старому обработчику. Во-вторых, часто обработчик прерываний должен находиться в ОП не до конца работы системы. В этом случае перед тем, как освободить память, необходимо восстановить старый обработчик прерываний. Это нетрудно сделать с помощью системного вызова «*int 21h*», функция *25h*. Рассмотрим реализацию первого случая.

Пример. Следующая резидентная программа «дополняет» обработку прерывания с номером 0. Системный обработчик данного прерывания-исключения прекращает выполнение текущей программы, если ее команда выполнила деление на 0. Наш обработчик спрашивает пользователя о желании завершить программу. В случае нажатия клавиши *y* (или *Y*) выполнение программы прекращается. При нажатии любого другого символа выполнение программы продолжается.

```
;  Резидентная программа выполняет обработку исключения 0 (деление на 0)
;  ----------------------------------------------------------------------
;
;
cr          equ     0dh             ; Код ASCII возврата каретки
lf          equ     0ah             ; Код ASCII перевода строки
            absolute  2ch           ; Следующий байт имеет адрес-смещение 2ch
Blocokr     resw    1               ; Адрес-сегмент блока окружения
            segment   code          ; Виртуальный сегмент кода
            org   100h
Start:      jmp   Init              ;  Переход на инициализацию
;       Обработчик прерываний
```

```
Msg     db      'Завершить программу ?', cr, lf, '$'  ; Выводимое сообщение
Old     dd  0                       ; Прежнее содержимое вектора пр-й
Begin:  sti                         ; Разрешить маскируемые прерывания
        push ax                     ; Сохранение содержимого
        push dx                     ;   регистров
        push ds                     ;   в стеке
        pushf                       ; Сохранение регистра флагов
        mov  ax, cs                 ; DS будет адресовать данные
        mov  ds, ax                 ; резидентной программы
        mov  dx, Msg                ;  DX ← адрес строки-сообщения
        mov  ah, 9                  ;  Вывод строки
        int  21h                    ;   на экран
        mov  ah,1                   ; Ввод символа с
        int  21h                    ;   клавиатуры
        or   al, 20h                ; Перевод символа на нижний регистр
        cmp  al, 'y'                ; Введен символ «y» ?
        jnz  .Exit                  ; Если нет, на выход из прерывания
        popf                        ; Восстановление регистра флагов
        pop  ds                     ; Восстановление содержимого
        pop  dx                     ;   регистров
        pop  ax                     ;   из стека
        jmp  far  [cs:Old]          ; Вызов прежнего обработчика
.Exit:  mov  ah, 2                  ; Перевод
        mov  dl, lf                 ;   строки
        int  21h                    ;   экрана
        popf                        ; Восстановление регистра флагов
        pop  ds                     ; Восстановление содержимого
        pop  dx                     ;   регистров
        pop  ax                     ;   из стека
        iret                        ; Возврат из прерывания
;   Инициализационная часть программы
Init:   mov  ax, 3500h              ; Сохранение прежнего содержимого
        int  21h                    ; вектора 0 в регистрах ES и BX
        mov  [Old], bx              ; А затем
        mov  [Old+2], es            ;   в поле Old
        mov  ax, 2500h              ;Запись стартового адреса обработчика
        mov  dx, Begin              ;   в вектор прерываний
        int  21h                    ;   с номером 0
        mov  ah, 49h                ; Освобождение области памяти,
        mov  es, [Blocokr]          ;   занимаемой
        int  21h                    ;   блоком окружения
        mov  dx, Init               ;  Возврат в DOS, оставшись
        int  27h                    ;   резидентным
```

Как и предыдущая программа, данная программа состоит из инициализационной и резидентной частей. Инициализационная часть начинает свою работу с сохранения прежнего содержимого вектора прерываний 0 в двойном слове ОП с меткой *Old*. Затем

она помещает в вектор 0 стартовый адрес нашего обработчика прерываний и сокращает объем занимаемой им памяти. В завершение инициализационная часть выполняет возврат в DOS, оставив в памяти обработчик прерываний.

Обработчик прерываний (резидентная часть) запускается в результате прерывания-исключения с номером 0. В начале своего выполнения он разрешает маскируемые внешние прерывания, а также сохраняет в стеке используемые им регистры. Обратите внимание, что среди сохраняемых регистров находится и регистр флагов. Это делается для того, чтобы прежний обработчик прерываний получил такое содержимое этого регистра, которое установила ранее программа. (На тот случай, если некоторые флаги используются для передачи информации на вход обработчика.)

Далее резидентная часть выводит на экран вопрос о желании прекратить выполнение программы, выполнившей деление на 0. Ответ пользователя (один символ) переводится в строчный символ. Такой перевод выполняется установкой единственного бита 5, которым отличаются коды строчной и прописной одноименных букв. Если пользователь не желает прекращать выполнение программы, то выполняется возврат из прерывания.

Иначе – управление передается системному обработчику. Это делается с помощью команды *jmp*, выполняющей безусловный переход на стартовый адрес этого обработчика. Рассмотрим внимательно соответствующий программный оператор:

```
jmp   far   [cs:Old]    ; Вызов прежнего обработчика
```

Во-первых, данный оператор выполняет межсегментный переход. Об этом говорит наличие вспомогательного псевдооператора ***far*** (дальний). Кроме дальних переходов в программе возможны очень близкие переходы (не более чем на 128 байт) и близкие переходы – в пределах сегмента кода. Для задания таких переходов используются, соответственно, псевдооператоры ***short*** и ***near***. По умолчанию оператор *jmp* выполняет очень близкий переход и поэтому оператор *short* можно не записывать. Запись оператора *near* аналогична записи оператора *far* – между *jmp* и операндом.

Во-вторых, рассматриваемый оператор выполняет косвенный переход. Об этом говорит наличие имени *Old*, заключенного в квадратные скобки. Данное имя является не именем процедуры, а именем поля данных длиной два слова (четыре байта). В первом из этих слов находится адрес-смещение первого байта запускаемой процедуры, а во втором слове – адрес-сегмент этого байта. Таким образом, поле *Old* содержит стартовый логический адрес вызываемой процедуры.

В-третьих, используется явное задание адреса-сегмента ячейки *Old* – *«cs:Old»*. Это объясняется тем, что перед выполнением команды *jmp* команда *pop* восстановила прежнее содержимое регистра *DS*, и поэтому неявное задание адреса-сегмента ячейки памяти *Old*, принадлежащей обработчику прерываний, использоваться уже не может.

Закономерен вопрос: как системный обработчик возвратит управление (если пожелает) в прерванную программу? Ответ: обычным способом, т. е. с помощью команды *iret*. Это объясняется тем, что системный обработчик «наследует» от нашего обработчика стек прерванной программы, в который был помещен ранее адрес возврата (в момент прерывания).

Для проверки работоспособности рассмотренного обработчика прерываний необходимо: после того, как приведенная выше программа будет выполнена (сделав обработчик прерываний резидентным), запустить на выполнение простейшую прикладную программу, выполняющую деление на нуль в бесконечном цикле.

Если резидентная программа полностью заменяет систему в обработке каких-то типов прерываний, то по истечению какого-то времени у пользователя (или у прикладной программы) может возникнуть желание вернуться к системной обработке этих прерываний. Для этого следует выполнить следующие три действия:

1) закрыть файлы, открытые резидентной программой;

2) восстановить векторы прерываний так, чтобы они указывали на системные обработчики;

3) освободить память, занимаемую резидентной программой.

Перечисленные действия должна выполнить сама резидентная программа при получении ею команды о завершении. Например, если резидентная программа выполняет обработку прерываний от клавиатуры, то код одной из клавиш (комбинации клавиш) может использоваться в качестве такой команды.

Для восстановления векторов прерываний можно использовать рассмотренные ранее системные вызовы «*int 21h*» (функции *35h* и *25h*). Первый из этих вызовов позволяет запомнить, а второй – восстановить прежнее содержимое вектора прерывания.

Для освобождения памяти, занимаемой резидентной программой, можно воспользоваться системным вызовом «*int 21h*» (функция *49h*), который требует задания в регистре *ES* адреса-сегмента области резидентной программы. Так как адрес-сегмент резидентной программы находится во время ее выполнения в регистре *CS*, следующий фрагмент программы показывает, как можно осуществить такой вызов:

```
push  cs        ; Копирование CS
pop   es        ;  в ES
mov   ah, 49h   ; Освобождение
int   21h       ;  памяти
. . . . . .     ; Восстановление регистров из стека
iret            ; Возврат из прерывания
```

19.3. Резидентный редактор информации

Допустим, что мы хотим обеспечить запуск нашего редактора не из командной строки DOS или из утилиты типа DOS Navigator, а путем нажатия одной из клавиш, например, <F7>. Для этого редактор информации должен иметь резидентную часть, предназначенную для ожидания нажатия клавиши <F7>. Кроме этой резидентной части редактор имеет и основную (нерезидентную) часть, в качестве которой можно взять один из ранее разработанных *com*- или *exe*-файлов (*Redaktor.com* или *Redaktor.exe*). Для этой цели наиболее подходит вариант редактора *Redaktor.com*, полученный в лабораторной работе 14. Выбор именно этого варианта обусловлен тем, что он не предусматривает какой-либо обмен информацией между резидентной и основной частями редактора. При этом имя редактируемого файла (если оно есть) передается в программу редактора не как параметр (хвост) команды запуска редактора, а запрашивается динамически, т. е. самим редактором. Следует заметить, что каких-либо изменений в файл *Redaktor.com* вносить не требуется.

Кроме клавиши <F7> резидентная часть выполняет еще одну команду – <F8>. Нажатие этой клавиши приводит к уничтожению резидентной части редактора. При этом освобождается занимаемая этой частью память, а вся обработка прерываний клавиатуры возвращается системному обработчику. Следует заметить, что обработку обеих своих команд (<F7> и <F8>) резидентная часть редактора выполняет только в том случае, если

в данный момент не запущена основная часть редактора. Если основная часть редактора запущена, то резидентная часть игнорирует нажатия всех клавиш (в том числе <F7> и <F8>). Обработка этих клавиш оставляется другим программам.

В основе работы резидентной части редактора лежит перехват прерываний № 9 – прерываний от клавиатуры. Данная программа имеет две части – резидентную, называемую далее резидентом, и инициализирующую. Функции инициализирующей части не отличаются от функций соответствующей части для приведенного выше примера перехвата прерываний с номером 0. Что касается резидента, то его алгоритм несколько сложнее, чем алгоритм рассмотренной в предыдущем разделе программы для обработки прерывания 0. Этот алгоритм приведен на рис. 2.33.

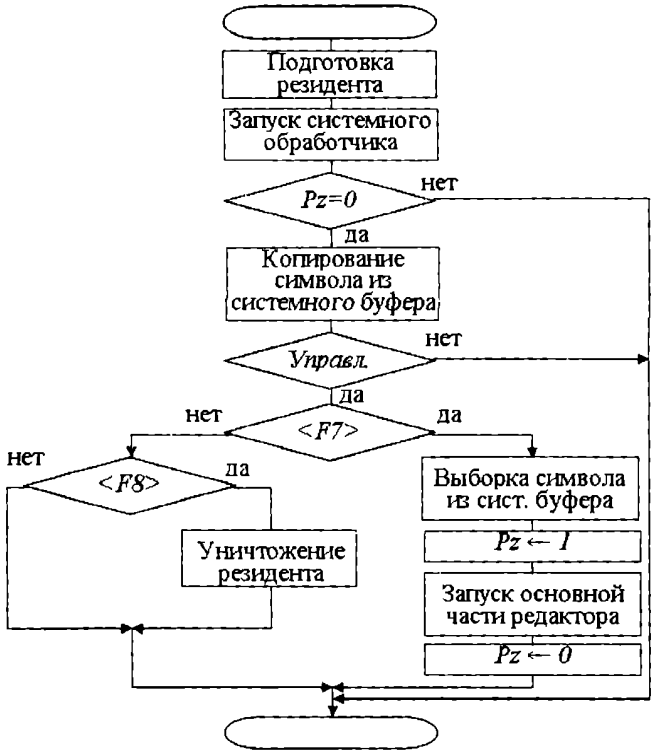

Рис. 2.33. Алгоритм резидентной части редактора:
Pz – признак запуска (0 – основная часть редактора не запущена, 1 – запущена)

На первом этапе алгоритма резидента производится его подготовка к последующей работе. Сюда относятся три действия: 1) разрешение маскируемых прерываний; 2) запоминание в стеке содержимого всех регистров, содержимое которых меняется при выполнении резидента; 3) «загрузка» виртуального сегмента кода резидента в оба логических сегмента данных – основной и дополнительный. Напомним, что последнее действие выполняется с целью обеспечить адресуемость данных, обрабатываемых резидентом.

На следующем этапе алгоритма осуществляется запуск системного обработчика прерываний клавиатуры. Кратко работа системного обработчика сводится к следующему: ***системный обработчик прерываний клавиатуры*** выполняет перенос скан-кода нажатой клавиши (или комбинации одновременно нажатых клавиш) из контроллера клавиа-

туры в системный буфер клавиатуры. ***Системный буфер*** (как и системный обработчик прерываний) находится в драйвере клавиатуры и представляет собой очередь, в которую помещаются скан-коды нажатых клавиш в порядке их нажатия. Так как наш резидент, являясь одновременно и обработчиком прерываний клавиатуры, никаких операций ни с системным буфером клавиатуры, ни тем более с контроллером клавиатуры, выполнять «не желает», то при каждом своем запуске (через прерывание) ему приходится инициировать системный обработчик, который и выполняет перенос кода нажатой клавиши из контроллера клавиатуры в системный буфер клавиатуры.

Операция запуска системного обработчика включает два действия: 1) сохранение в стеке нового адреса возврата; 2) переход на системный обработчик. Первое из этих действий является подготовкой к выполнению второго действия. Вспомним, что в момент прерывания ЦП помещает в стек прерванной программы (этот стек наследуется обработчиком прерываний) три слова – (*FLAGS*), (*CS*), (*IP*). После того, как программа обработчика выполнится, его последняя команда *iret* извлекает из вершины стека три последних слова и помещает их в соответствующие регистры. Таким образом, обеспечивается возврат в прерванную программу, а также восстановление прежнего содержимого регистра *FLAGS*. Поэтому прежде, чем выполнить переход на системный обработчик, наш резидент помещает в стек новый адрес возврата из трех слов: 1) текущее содержимое *FLAGS*, 2) текущее содержимое *CS*; 3) адрес-смещение того байта программы, который расположен в памяти сразу же за командой перехода на системный обработчик прерываний клавиатуры.

Например, допустим, что переход на системный обработчик выполняет оператор:

```
    jmp   far  [Old]
.M:
```

где *Old* – двойное слово ОП, в котором ранее инициализирующая часть сохранила стартовый адрес системного обработчика. Тогда для записи третьего слова в стек может использоваться оператор:

```
    push  .M
```

Интересно отметить, что для перехода на системный обработчик можно использовать не только оператор *jmp*, но и оператор *call*:

```
    call   far  [Old],
```

который выполняет дальний косвенный вызов процедуры. Выполнение данного оператора похоже на выполнение оператора *jmp* с тем лишь отличием, что он сам заносит в программный стек адрес возврата – сначала (*CS*), а затем (*IP*). Поэтому из трех операторов записи в стек в программе следует оставить лишь оператор *pushf*.

В результате записи в стек своего адреса возврата наша программа делает системный обработчик прерываний «ручным». Это означает, что после своего выполнения системный обработчик возвратит управление в наш резидент на метку *.M*. Напомним, что во время этого выполнения системный обработчик выполнил перенос кода нажатой клавиши из контроллера клавиатуры в системный буфер.

На следующем этапе алгоритма производится сравнение с нулем значения переменной *Pz* – признак запуска основной части редактора (файла *Redaktor.com*). Эта переменная (1 байт) имеет значения: 0 – файл *Redaktor.com* в данный момент не выполняется, 1 – выполняется. Первоначальное значение *Pz* = 0 записывает или транслятор (при обработке оператора *db*), или инициализирующая часть резидентной программы. Если $Pz \neq 0$,

то резидент не может выполнять свои команды <F7> и <F8>. Поэтому он завершает свое выполнение (с помощью *iret*), предварительно восстановив содержимое регистров.

Если *Pz* = 0, то на следующем этапе алгоритма резидента производится копирование символа из системного буфера. Напомним, что символы (точнее – скан-коды клавиш) в порядке их поступления помещаются в системный буфер клавиатуры системным обработчиком прерываний. Далее эти символы находятся в системном буфере до тех пор, пока не будут извлечены оттуда подпрограммами DOS или BIOS, выполняющими системные вызовы ввода с клавиатуры. Для того чтобы выполнить копирование символа из системного буфера, наша программа обращается за помощью к BIOS, используя системный вызов «***int 16h***», функция ***01h***. Особенностью данного системного вызова является то, что, выполняя чтение из системного буфера кода символа, он этот код из буфера не извлекает. Кроме этого, данный системный вызов отличается от ранее рассмотренного «*int 16h*», функция *00h*, тем, что он не дожидается появления в системном буфере кода символа, а при его отсутствии возвращает управление в вызывающую программу, установив флаг *ZF = 1*. Если символ в системном буфере есть, то в результате выполнения данного системного вызова флаг *ZF = 0*, а в регистрах: *AH* – скан-код символа, *AL* – ASCII-код символа.

На следующем этапе алгоритма проверяется, является ли символ, скопированный из системного буфера, управляющим. Напомним, что ASCII-код управляющего символа равен нулю. Если символ неуправляющий, то выполнение резидента заканчивается. Иначе – делается проверка того, не является ли введенный символ кодом клавиши <F7>. Если является, то, во-первых, выполняется выборка символа из системного буфера (с помощью системного вызова «*int 16h*», функция *00h*). Это делается потому, что в нашем случае символ <F7> предназначен для использования только в резиденте, и если его оставить в системном буфере, то он будет использоваться ошибочно в какой-то другой программе.

Во-вторых, после выборки символа из системного буфера осуществляется запуск основной части редактора, т. е. файла *Redaktor.com*. Напомним, что для этого следует использовать системный вызов «*int 21h*», функция *4bh*, подфункция *00h* (системный вызов *EXEC*). При этом заметим, что резидент не должен заниматься освобождением ОП, пригодной для размещения основной части редактора, так как ранее это было сделано в инициализирующей части резидентной программы. Единственное – после того, как основная часть редактора завершится (в результате обработки команды <F10>), резидент обнуляет содержимое переменной *Pz*.

Если символ, скопированный из системного буфера, не <F7>, то выполняется проверка на совпадение кода символа с кодом клавиши <F8>. Если ответ сравнения положительный, то резидент выполняет собственное уничтожение. Иначе – выполнение резидента завершается обычным образом. Уничтожение резидента включает: 1) освобождение памяти, занимаемой резидентом; 2) выборка символа из системного буфера. Реализация второго из этих двух действий описана выше, а для реализации первого из них используется системный вызов «*int 21h*», функция *49h*.

В заключении заметим, что в отличие от всех других ранее рассмотренных алгоритмов, алгоритм нашего резидента не является последовательным. При этом напомним, что принципиальной особенностью последовательного алгоритма является невозможность одновременного выполнения более чем одного этапа алгоритма. Для резидента это свойство не выполняется, так как одновременно могут выполняться этапы «запуск редактора» и «запуск системного обработчика» (при *Pz*=1). Причиной такой непоследова-

тельности является то, что одна и та же программа нашего резидентного редактора одновременно предназначена и для выполнения высокоуровневых функций по редактированию информации, и для выполнения низкоуровневых функций в качестве обработчика прерываний клавиатуры.

Лабораторная работа 17

Требуется реализовать описанный выше резидентный редактор.

☑ **Примечание**. Резидентный редактор должен правильно выполнять свои команды одновременно с выполнением в системе другой (нерезидентной) обрабатывающей программы. Эта обрабатывающая программа должна отвечать двум условиям: 1) выполнять ввод символов с клавиатуры; 2) освобождать лишнюю для себя память.

Подумайте, как можно использовать резидентный редактор для анализа выполнения обрабатывающей программы.

ЗАДАНИЯ ДЛЯ ПРОГРАММИРОВАНИЯ

Данные задания могут использоваться при самостоятельном изучении языка ассемблера. Кроме того, они могут использоваться преподавателем в качестве тем контрольных заданий для индивидуального программирования.

Первые 20 заданий предназначены для закрепления знаний по начальным главам части II пособия (гл. 9–11). Остальные 20 заданий позволяют закрепить знания по остальным разделам части II.

Задание 1. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>. Программа выводит на экран сумму этих чисел, представленную в десятичной и шестнадцатеричной системах счисления.

Задание 2. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число *N*. По следующему запросу он вводит с клавиатуры *N* целых трехзначных положительных десятичных чисел, разделенных пробелами. Программа выводит на экран сумму этих чисел, представленную в десятичной и двоичной системах счисления.

Задание 3. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>. Программа выводит наибольшее число из введенных, представленное в десятичной и двоичной системах счисления.

Задание 4. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число *N*. По следующему запросу он вводит с клавиатуры *N* целых трехзначных положительных десятичных чисел, разделенных пробелами. Программа выводит наибольшее число из введенных, представленное в десятичной и шестнадцатеричной системах счисления.

Задание 5. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами.

Ввод последовательности заканчивается нажатием <Enter>. Программа выводит наименьшее число из введенных, представленное в десятичной и шестнадцатеричной системах счисления.

Задание 6. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число *N*. По следующему запросу он вводит с клавиатуры *N* целых трехзначных положительных десятичных чисел, разделенных пробелами. Программа выводит наименьшее число из введенных, представленное в десятичной и двоичной системах счисления.

Задание 7. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>. Программа выводит последовательность этих же чисел, но записанных в обратном порядке и в шестнадцатеричной системе счисления.

Задание 8. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число *N*. По следующему запросу он вводит с клавиатуры *N* целых трехзначных положительных десятичных чисел, разделенных пробелами. Программа выводит последовательность этих же чисел, но записанных в обратном порядке и в двоичной системе счисления.

Задание 9. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>. Программа выводит эти же числа на экран в порядке возрастания величины числа, причем в шестнадцатеричной системе счисления.

Задание 10. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число *N*. По следующему запросу он вводит с клавиатуры *N* целых трехзначных положительных десятичных чисел, разделенных пробелами. Программа выводит эти же числа на экран в порядке возрастания величины числа, причем в двоичной системе счисления.

Задание 11. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>. Программа выводит эти же числа на экран в порядке убывания величины числа, причем в двоичной системе счисления.

Задание 12. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число *N*. По следующему запросу он вводит с клавиатуры *N* целых трехзначных положительных десятичных чисел, разделенных пробелами. Программа выводит эти же числа на экран в порядке убывания величины числа, причем в шестнадцатеричной системе счисления.

Задание 13. По запросу программы пользователь вводит с клавиатуры сообщение на русском языке, заканчивающееся символом «.» или «!». Программа выводит на экран это же сообщение, записанное только заглавными буквами.

Задание 14. По запросу программы пользователь вводит с клавиатуры сообщение на русском языке, заканчивающееся символом «.» или «?». Программа выводит на экран это же сообщение, записанное только строчными (малыми) буквами.

Задание 15. По запросу программы пользователь вводит с клавиатуры сообщение на английском языке, заканчивающееся символом «.» или «?». Программа выводит на экран это же сообщение, записанное только заглавными буквами.

Задание 16. По запросу программы пользователь вводит с клавиатуры сообщение на английском языке, заканчивающееся символом «.» или «!». Программа выводит на экран это же сообщение, записанное только строчными (малыми) буквами.

Задание 17. По запросу программы пользователь вводит с клавиатуры два целых четырехзначных положительных десятичных числа, разделенных знаком операции «+» или «-». Программа выводит на экран результат операции в двух системах счисления – в десятичной и в двоичной (в дополнительном коде).

Задание 18. По запросу программы пользователь вводит с клавиатуры два целых четырехзначных положительных десятичных числа, разделенных знаком операции «*». Программа выводит на экран результат операции умножения.

Задание 19. По запросу программы пользователь вводит с клавиатуры два целых четырехзначных положительных десятичных числа, разделенных знаком операции «/». Программа выводит на экран результат операции деления (частное и остаток).

Задание 20. По запросу программы пользователь вводит с клавиатуры два целых трехзначных положительных десятичных числа. Программа выводит на экран сообщение о том, делится ли первое число на второе без остатка, а затем сообщение – делится ли без остатка второе число на первое.

Задание 21. Резидентная программа инициируется нажатием клавиши <F1> и выводит на экран ваши имя и фамилию, записанные английскими буквами. При нажатии клавиши <F2> программа уничтожается.

Задание 22. Резидентная программа инициируется нажатием клавиши <F2> и выводит на экран ваши имя и фамилию, записанные русскими буквами. При нажатии клавиши <F1> программа уничтожается.

Задание 23. Разработать простейший отладчик программ, который получает имя загрузочного модуля прикладной программы в качестве своего параметра, и выполняет эту программу покомандно, выдавая после завершения каждой ее команды на экран содержимое регистров *AX* и *BX* в шестнадцатеричной системе счисления. (Некоторый аналог команды *T* Debug.)

☑ **Примечание**. Работа программы основана на запуске трассируемой программы и обработке исключения «Трассировка».

Задание 24. Разработать простейший отладчик программ, который вводит с клавиатуры имя загрузочного модуля прикладной программы, и выполняет эту программу покомандно, выдавая после завершения каждой ее команды на экран содержимое регистров *CX* и *DX* в двоичной системе счисления. (Некоторый аналог команды *T* Debug.)

☑ **Примечание**. Работа программы основана на запуске трассируемой программы и обработке исключения «Трассировка».

Задание 25. Разработать простейший интерпретатор команд, выполняющий обработку командных файлов (*bat*-файлов). Имя командного файла вводится с клавиатуры. Имя запускаемой программы (строка *bat*-файла) не имеет параметров. После завершения запуска очередной программы на экран выводится сообщение об успешности запуска.

Задание 26. Разработать простейший интерпретатор команд, выполняющий обработку командных файлов (*bat*-файлов). Имя командного файла интерпретатор команд получает при своем запуске (в качестве параметра команды). Имя запускаемой программы (строка *bat*-файла) не имеет параметров. После завершения запуска очередной программы на экран выводится сообщение об успешности запуска.

Задание 27. Прикладная программа выполняет запуск другой (дочерней) прикладной программы, получив предварительно ее имя с клавиатуры. При своем запуске дочерняя программа получает на входе (в *PSP*) строку символов, которую она выводит на экран.

Задание 28. Прикладная программа выполняет запуск другой (дочерней) прикладной программы, получив ее имя в качестве своего параметра (хвоста команды). При своем запуске дочерняя программа получает на входе (в *PSP*) строку символов, которую она выводит на экран.

Задание 29. Прикладная программа выводит на экран содержимое своего блока окружения.

Задание 30. Прикладная программа выполняет уничтожение файла. Имя уничтожаемого файла вводится с клавиатуры.

Задание 31. Прикладная программа вводит с клавиатуры имя нового текстового файла, записывает в него содержимое своего блока окружения, а также «хвоста», а затем выводит содержимое этого файла на экран.

Задание 32. Прикладная программа вводит с клавиатуры имя существующего текстового файла, «дописывает» в него содержимое своего блока окружения, а также «хвоста», а затем выводит содержимое этого файла на экран.

Задание 33. Резидентная программа инициируется из прикладной программы, выполняя вывод на экран блока окружения и «хвоста» прикладной программы.

☑ **Примечание**. Для того чтобы обрабатывать данные вызывающей программы, содержимое регистров сегментов данных должно соответствовать прикладной, а не резидентной программе.

Задание 34. Прикладная программа вводит с клавиатуры имена двух существующих текстовых файлов, а затем инициирует резидентную программу (передав ей на вход через *PSP* имена файлов), которая «расширяет» первый файл, добавив в него содержимое второго файла.

Задание 35. Прикладная программа вводит с клавиатуры имя существующего текстового файла, а затем выводит его содержимое на экран.

Задание 36. Прикладная программа получает в качестве параметра команды имя существующего текстового файла, а затем выводит его содержимое на экран.

Задание 37. Прикладная программа вводит с клавиатуры имя текстового файла, содержащего имена других текстовых файлов. А затем выводит на экран содержимое этих текстовых файлов.

Задание 38. Прикладная программа получает в качестве параметра команды имя текстового файла, содержащего имена других текстовых файлов. А затем выводит на экран содержимое этих текстовых файлов.

Задание 39. Прикладная программа выполняет копирование файла. Имя исходного файла программа получает в качестве параметра команды, а имя файла-копии вводится с клавиатуры.

Задание 40. Прикладная программа выполняет копирование файла. Имя исходного файла вводится с клавиатуры, а имя файла-копии программа получает в качестве параметра команды.

Часть III. РАЗРАБОТКА АВТОНОМНЫХ ПРОГРАММ

Автономная программа выполняется на ВС, не имеющей операционную систему. Поэтому она не может обратиться за помощью к ОС, используя для этого соответствующие системные вызовы. Следовательно, автономная программа должна выполнять управление ресурсами системы сама, возможно, используя для этого системные вызовы BIOS. Разработка автономных программ особенно актуальна для небольших встроенных систем, устанавливаемых непосредственно на объекте управления. Кроме того, в качестве автономной программы можно рассматривать и саму ОС.

Причинами разработки автономных программ являются не только необходимость экономии аппаратных ресурсов, но и недостатки, присущие современным ОС. Однопрограммные ОС типа DOS достаточно просты и надежны, но они позволяют использовать лишь небольшую часть аппаратных ресурсов современной ЭВМ. В частности, DOS, выполняясь только в реальном режиме процессоров *Intel* и их клонов, позволяет использовать лишь первый мегабайт ОП. Для многих современных прикладных программ этого очень мало.

Что касается Windows , то эта ОС ориентирована на офисные цели и плохо адаптируется для решения задач оперативного управления техническими системами. При этом Windows забирает для собственных нужд изрядную часть ресурсов компьютера и ограничивает доступ к ПУ. Несмотря на то, что существует много ОС, специально предназначенных для решения задач управления, существенным доводом в пользу разработки автономных программ является то, что существующие ОС не предоставляют никаких гарантий своим пользователям при решении ответственных задач. Чтобы убедиться в этом, достаточно прочитать любое лицензионное соглашение и убедиться, что разработчик служебной программы ни за что не несет ответственности. За все неисправности и аварии отвечает лишь потребитель этой служебной программы, т. е. разработчик прикладных программ. Еще одно свойство, присущее современным ОС – огромный размер их программного кода, который совершенно нельзя протестировать.

В отличие от процессора i8086, современные процессоры имеют не один, а несколько режимов работы. Несмотря на то, что основным из этих режимов является защищенный режим, выполняющий поддержку мультипрограммирования, разработку автономных программ целесообразно начать с реального режима. ***Реальный режим*** – режим, в котором процессор оказывается сразу же после включения питания. Этот режим очень похож на работу процессора i8086.

Важнейшими вопросами, которые необходимо решить при разработке автономных программ, являются вопросы использования в программах периферийных устройств (ПУ). Несмотря на то, что автономные программы имеют возможность использовать для работы с ПУ системные вызовы BIOS, многие из этих программ работают с устройствами напрямую, т. е. на аппаратном уровне – через порты ввода-вывода. Подобному программному управлению ПУ в данной части пособия уделяется существенное внимание.

Несмотря на то, что автономная программа выполняется без помощи ОС, разработка такой программы, т. е. получение для нее исходных, объектных и загрузочных файлов, как и раньше, будет происходить в среде конкретной операционной системы. Соответствующая ВС (аппаратура + ОС) называется ***инструментальной системой***. Естественно, что для автономной программы инструментальная и выполняющая системы не совпада-

ют. Тем не менее такое совпадение имеет место для некоторых разрабатываемых далее программ, которые не только разрабатываются, но и исполняются в среде одной и той же вычислительной системы i8086+DOS. Позднее, когда у нас появятся загрузчики автономных программ, мы сможем «доделать» наши программы так, чтобы они действительно были автономными. Пока лишь заметим, что такая «доделка» производится достаточно просто.

20. Управление устройствами без прерываний

20.1. Управление устройствами на аппаратном уровне

Никакая прикладная или системная обрабатывающая программа не может выполняться без операций с периферийными устройствами (ПУ). Кроме стандартных устройств ввода-вывода, например, экрана и клавиатуры, и внешней памяти (дисководы), существует огромное количество нестандартных ПУ. Подобные устройства используются, например, для управления технологическими процессами. Сюда относятся различные датчики (устройства ввода), а также различные задвижки и вентили (устройства вывода).

Для работы со стандартными ПУ программа может использовать системные подпрограммы ОС и BIOS, применяя для этого соответствующие системные вызовы. При этом программе предоставляется возможность работать с ПУ не только как с устройствами, но и как с файлами. Подобные системные вызовы широко использовались нами в рассмотренных ранее примерах программ, заметно сокращая время программирования.

К сожалению, системных вызовов часто бывает недостаточно для управления ПУ по следующим причинам. Во-первых, время выполнения драйверов BIOS, и особенно DOS, слишком велико. Во-вторых, для многих типов ПУ, например, для нестандартных, соответствующие системные драйверы отсутствуют. Следствием этого является то, что многие прикладные программы выполняют управление ПУ на аппаратном уровне, т. е. фактически используют свои собственные драйверы. Далее мы рассмотрим вопросы построения таких драйверов.

Допустим, что ЭВМ имеет структуру с общей шиной (см. рис. 1.4). С помощью одного ИУ к ОШ подсоединяются однотипные ПУ. Простейшее ИУ состоит из одного или более регистров, в состав которых обязательно входят регистр состояния и управления и буферный регистр данных. ***Буферный регистр данных*** (***RD***) предназначен для временного хранения одного или двух байтов данных, передаваемых на ПУ при выводе или принимаемых с ПУ при вводе.

Регистр состояния и управления (***RS***) играет для ИУ и подключенных к нему ПУ ту же роль, что для ЦП играет регистр *FLAGS*, т. е. фактически является их блоком управления. Количество регистров состояния и управления, а также назначение их битов для разных ИУ существенно отличаются. На рис. 3.1 приведен пример типичного *RS*, который будет использоваться нами в дальнейшем. В этом *RS* биты b_0-b_3 – ***управляющие биты***. Они устанавливаются программами, которые исполняются на ЦП, и предназначены:

- если $b_0 = 1$, то разрешается работа ПУ по вводу (выводу) единицы информации;
- $b_1 – b_2$ используются для уточнения операции (функции), выполняемой устройством;
- если $b_3 = 1$, то в ЦП может быть выдан сигнал прерывания.

На рис. 3.1 биты b_4–b_7 – ***биты состояния*** **ПУ**. Они устанавливаются ПУ или его ИУ. При этом b_7 описывает состояние буферного регистра данных *RD*. Для устройства ввода $b_7 = 1$ означает, что *RD* заполнен данными, которые могут быть перенесены в ЦП. Для устройства вывода это означает, что выходной буфер данных сейчас пуст и ждет, когда ЦП загрузит в него новые данные.

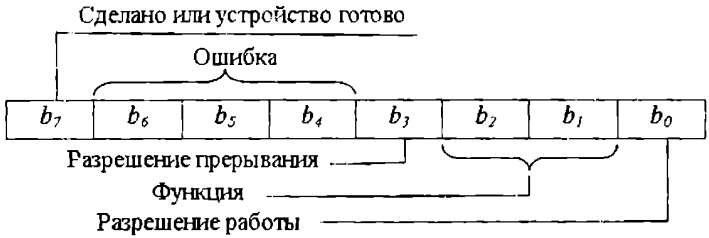

Рис. 3.1. Пример регистра состояния и управления *RS*

Биты b_4 – b_6 содержат информацию о том, произошла ли ошибка при выполнении последней операции ввода-вывода. Нулевое содержимое всех этих битов сообщает об отсутствии ошибки, иначе эти биты содержат код ошибки.

При использовании простейшего ИУ всю работу по управлению ПУ выполняет сам ЦП. Поэтому он отвлекается от выполнения самих программ и, таким образом, значительно снижается его фактическая производительность. Применение более сложных ИУ позволяет возложить на них значительную часть работы по управлению ПУ, которая теперь, в принципе, может выполняться параллельно работе ЦП. Наиболее сложным ИУ является ***контроллер*** – специализированный процессор ввода-вывода. (Как и всякое ИУ, контроллер имеет свои *RS* и *RD*.)

Регистры ИУ, непосредственно доступные из программ ЦП (являются «программно-видимыми»), называются ***портами***. Портами являются многие *RS* и *RD* (но не все), а также служебные регистры, используемые для программного доступа к тем *RS* и *RD*, к которым нет непосредственного доступа. Каждый порт в ЭВМ имеет свой уникальный номер, называемый ***адресом порта***. Для некоторых типов ЦП программные операции с портами (чтение и запись) выполняются точно так же, как и операции с ячейками ОП. При этом для адресации портов используется часть адресного пространства ОП, в которой каждый реальный адрес соответствует не ячейке ОП, а порту.

В i8086 используется другой подход, при котором существуют два адресных пространства – одно для ячеек ОП, а второе – для портов. При этом адрес порта – число в диапозоне от 0 до 65535, а сам порт представляет собой 8-битный регистр. Следствием наличия двух адресных пространств является использование для работы с портами специальных команд ЦП: ***in*** для ввода из порта и ***out*** – для вывода в порт.

Одной командой *in* можно или ввести в регистр *AL* один байт, или ввести в регистр *AX* 16-битовое слово. Во втором случае ввод осуществляется из двух портов, имеющих смежные адреса, причем порт с меньшим адресом содержит младший вводимый байт. Адрес порта в команде *in* можно задать двумя способами: 1) в виде константы; 2) в качестве содержимого регистра *DX*. Первый из этих способов позволяет использовать в команде *in* лишь младшие адреса портов (от 0 до 255). Второй способ позволяет задавать любые порты (от 0 до 65535). Примеры:

```
in   al, 50h      ; AL ← (50h)
in   ax, 50h      ; AX ← (51h : 50h)
```

```
in   al, dx      ; AL ← ((DX))
in   ax, dx      ; AX ← ((DX)+1 : (DX))
```

Одна команда *out* позволяет вывести (скопировать) байт из регистра *AL* в заданный порт или вывести слово из регистра *AX* в два порта со смежными адресами. Адрес порта задается аналогично команде *in* с той лишь разницей, что теперь порт является получателем и должен записываться в качестве первого операнда. Примеры:

```
out  50h, al     ; (50h) ← (AL)
out  50h, ax     ; (51h : 50h) ← (AX)
out  dx, al      ; ((DX)) ← (AL)
out  dx, ax      ; ((DX)+1 : (DX)) ← (AX)
```

Любой драйвер должен выполнять три основные функции:

1) подготовка ПУ и, возможно, самого драйвера к последующим операциям по обмену данными;

2) инициирование очередной операции ПУ по вводу (выводу) единицы информации;

3) обеспечение завершения очередной операции ввода-вывода.

Реализация перечисленных функций в значительной степени зависит от принципа ввода-вывода, принятого при разработке драйвера. Различают следующие принципы: синхронный; асинхронный с общей памятью; асинхронный с прерываниями; прямой доступ в память. Первые два из этих принципов предназначены для посимвольного обмена с ПУ без использования прерываний.

20.2. Синхронный ввод-вывод

При синхронном вводе-выводе драйвер ожидает завершения текущей операции ввода-вывода активно, занимая в процессе ожидания ЦП. В результате этого нет никакого физического параллелизма между ЦП, с одной стороны, и контроллером и устройством – с другой. Реализацию синхронного принципа рассмотрим на следующем гипотетическом примере. (Реальный пример потребовал бы рассмотрения второстепенных технических деталей, присущих конкретному реальному ПУ.)

Пусть требуется разработать драйвер для управления работой считывателя перфоленты. Структура *RS* (адрес порта – *90h*) приведена на рис. 3.1. Код вводимого символа помещается в *RD* (адрес порта *91h*). Результаты разработки драйвера рассматриваются в следующей последовательности: 1) логическая схема; 2) программа на ассемблере.

20.2.1. Логическая схема драйвера

Логическая схема драйвера приведена на рис. 3.2. Драйвер представляет собой совокупность двух интерфейсных (доступных для вызова извне драйвера) логических процедур. Процедура «Инициализация» без параметров. Она выполняет два действия: 1) делает процедуру «Ввести символ» резидентной; 2) запрещает прерывания от считывателя перфоленты.

Вспомним, что инициирование резидентной подпрограммы возможно только через прерывание. Так как подпрограмму «Ввести символ» вызывают прикладные программы, то они должны использовать для этого команду программного прерывания *int*. В качестве номера прерывания (этот номер совпадает с операндом команды *int* и с номером вектора прерываний) можно взять любой неиспользуемый в системе номер, например *EEh*.

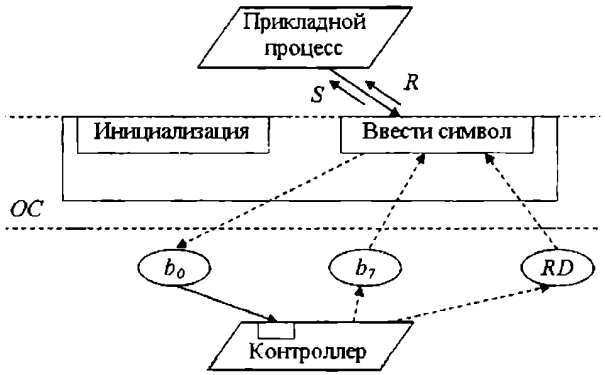

Рис. 3.2. Логическая схема синхронного драйвера:
S – код символа; R – признак результата (0 – успешно; 1 – ошибка);
b_0 – бит разрешения работы; b_7 – бит готовности

Запрет аппаратных прерываний от ПУ, управляемого синхронным драйвером, выполняется потому, что при синхронном вводе-выводе эти прерывания не нужны. Для выполнения такого запрета в нашем примере достаточно сбросить бит b_3 в регистре *RS* (см. рис. 3.1). (На практике для этого можно выдать команду маскирования в программируемый контроллер прерываний, который рассматривается в п. 21.1.)

Логическая процедура «Ввести символ» обслуживает запросы прикладных и системных программ по вводу символов с перфоленты. После своего вызова эта логическая процедура устанавливает в единицу бит b_0 в *RS* и тем самым инициирует устройство ввода, которое продвигает перфоленту на один шаг. В ожидании завершения этой операции опрашивается в цикле бит b_7 в *RS*. Затем, если нет ошибки, код символа переписывается из *RD* в *S*. После этого следует возврат управления в прикладную программу.

20.2.2. Программа синхронного драйвера

Программа синхронного драйвера ввода символов с перфоленты:

```
;          Синхронный драйвер ввода символов с перфоленты
;          ----------------------------------------------
; Содержит: 1) п/п-у инициализации (вызов из DOS)
;           2) п/п-у вывода символа (вызов командой «int 0eeh»)
;
absolute 2ch                ; Следующий байт имеет адрес-смещение 2ch
Blocokr  resw    1          ; Адрес-сегмент блока окружения
         segment code        ; Виртуальный сегмент кода
         org     100h
Start:   jmp     Init       ; Переход на инициализацию
;
;    Ввод символа – обработчик программного прерывания с номером EEh
;    ---------------------------------------------------------------
;    Выходы:  флаг CF – признак результата (0 – успешно; 1 – ошибка)
;             AL – код символа
Begin:  sti                 ; Разрешить маскируемые прерывания
        push ax             ; Сохранение содержимого
```

```
        push  ds                ;   регистров в стеке
        mov   ax, cs            ; DS будет адресовать данные
        mov   ds, ax            ;   резидентной программы
        in    al, 90h           ; Чтение RS в AL
        or    al, 00000001b     ; Запуск
        out   90h, al           ;   устройства
.Povt:  in  al, 90h             ; Чтение RS в AL
        test  al, 10000000b     ; Символ готов?
        jz    .Povt             ; Нет
        test  al, 01110000b     ; Ошибка есть?
        jnz   .Error            ; Да
        in    al, 91h           ; Прием символа из RD
        clc                     ; Сбрасывает флаг переноса CF
        jmp   .Exit
.Error: stc                     , Устанавливает флаг переноса CF
.Exit:  pop   ds                ; Восстановление содержимого
        pop   ax                ;   регистров из стека
        iret                    ; Возврат в программу из прерывания
;   Инициализационная часть программы
Init:   mov   ax, 25eeh         ; Запись стартового адреса обработчика
        mov   dx, Begin         ;   в вектор прерываний
        int   21h               ;   с номером EEh
        in    al, 90h           ; Чтение RS в AL
        and   al, 11110111h     ; Запрет прерываний
        out   90h, al           ;   от устройства
        mov   ah, 49h           ; Освобождение области памяти,
        mov   es, Blocokr       ;   занимаемой
        int   21h               ;   блоком окружения
        mov   dx, Init          ; Возврат в DOS, оставшись
        int   27h               ;   резидентным
```

Инициализационная часть этой программы выполняется в результате запуска программы из DOS. Так как эта часть нерезидентна, то она размещена в конце программы. В начале своего выполнения она помещает стартовый адрес резидентной части в выбранный вектор прерываний, а затем запрещает прерывания от устройства ввода перфоленты, сбросив бит 3 в *RS*. Далее производится подготовка резидентной части путем освобождения памяти, занимаемой блоком окружения программы. После этого инициализационная часть выполняет возврат в DOS, сопровождаемый просьбой сделать резидентной начало программы (до метки *Init*).

Резидентная часть программы инициируется в результате программного прерывания с номером *EEh*. Выполнение этой части начинается так же, как начинается любой обработчик прерываний:

1) разрешение маскируемых прерываний;

2) сохранение содержимого используемых регистров в стеке;

3) запись в *DS* адреса-сегмента данных резидентной части.

После этого производится запуск устройства ввода перфоленты (установкой бита 0 в *RS*). Далее в цикле опрашивается бит 7 в *RS* до тех пор, пока он не будет установлен в

1. В зависимости от наличия ошибки перед возвратом в прикладную программу сбрасывается или устанавливается флаг *CF*.

20.3. Асинхронный вывод с общей памятью

20.3.1. Видеоадаптер

Принцип асинхронного вывода с общей памятью заключается в том, что ЦП и ИУ имеют доступ к общей области памяти, используя ее в качестве буфера для информационного обмена между ними. При этом операции записи и чтения с общей памятью ЦП и ИУ производят асинхронно, т. е. независимо друг от друга. Данный принцип широко используется для управления экраном (дисплеем).

Вспомним (см. п. 18.1), что часть адресного пространства ОП (112 К, начиная с адреса *A0000h*) используется для адресации видеопамяти. Она является той самой областью памяти, с которой ЦП и ИУ работают асинхронно. ***Видеопамять*** входит в состав ИУ экрана, называемого ***видеоадаптером***. Конструктивно такой адаптер представляет собой печатную плату, вставляемую в разъем расширения компьютера. Среди сотен существующих видеоадаптеров нами будет рассматриваться только один – ***CGA*** (*Color Graphic Adapter*). Являясь одним из первых видеоадаптеров (создан в 1981 г.), *CGA* имитируется аппаратно многими другими, более современными видеоадаптерами.

Видеоадаптер (в том числе и *CGA*) имеет два принципиально разных режима работы – текстовый и графический. Для каждого из этих режимов *CGA* имеет несколько форматов. Примером формата для текстового режима является формат ***80*25*** – 25 строк по 80 символов в каждой строке.

Центральным модулем *CGA* является контроллер экрана ***CRT***. Он устанавливает и поддерживает режим работы экрана, выполняет основную работу по интерпретации кодов ASCII, а также управляет курсором. Кроме контроллера экрана, *CGA* имеет порты ввода-вывода, ПЗУ с матрицей знаков, видеопамять, а также 18 регистров управления (рис. 3.3).

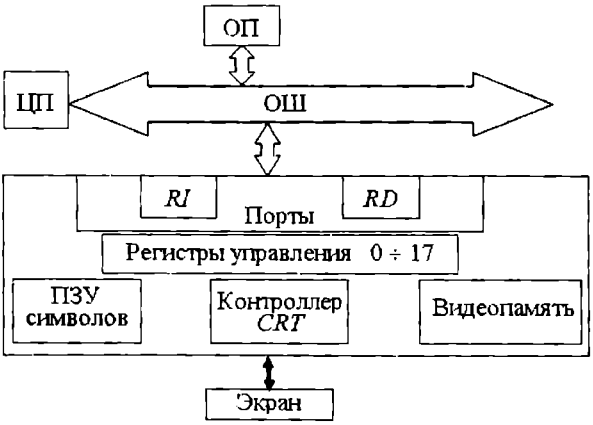

Рис. 3.3. Структура адаптера дисплея

В зависимости от выполняемых функций все 18 управляющих регистров контроллера можно разделить на две группы:

1) регистры, фиксирующие горизонтальные и вертикальные параметры экрана;

2) регистры визуализации.

К первой из перечисленных групп относятся десять регистров с номерами от 0 до 9. Они устанавливаются один раз при задании режима работы экрана. Это делает BIOS при включении ЭВМ в сеть. Ко второй группе относятся 8-битовые регистры визуализации, которые приведены в табл. 3.1. Некоторые из них связаны в пары, чтобы хранить 16-битовые величины.

Таблица 3.1. Регистры визуализации

Номер регистра	*Назначение регистра*	*Операции*
10	Начало курсора	Запись
11	Конец курсора	Запись
12	Начальный адрес сканирования (ст.)	Запись
13	Начальный адрес сканирования (мл.)	Запись
14	Адрес курсора (ст.)	Чтение/запись
15	Адрес курсора (мл.)	Чтение/запись
16	Световое перо (ст.)	Чтение
17	Световое перо (мл.)	Чтение

Все регистры визуализации можно разделить на три группы:

1) регистры управления курсором;

2) регистры светового пера;

3) регистры начального адреса сканирования. Этот адрес представляет собой порядковый номер в видеопамяти того символа, который будет выводиться в верхнем левом углу экрана. Например, это может быть номер 0.

Что касается портов, то их в *CGA* два (см. рис. 3.3):

1) порт с адресом *3D4h*, называемый также индексным регистром *RI*;

2) порт с адресом *3D5h*, называемый также регистром данных *RD*.

Регистр *RI* используется для задания требуемого управляющего регистра. А регистр *RD* используется для записи нового содержимого того управляющего регистра, на который указывает *RI*. Ниже будет рассмотрено применение этих портов для управления курсором.

20.3.2. Видеопамять

Видеопамять в *CGA* имеет размер 16 Кбайт и начинается по адресу *B8000h* (соответствующий логический адрес – *B800h:0000h*). Каждый символ занимает в видеопамяти два байта. В младшем из этих байтов содержится код ASCII символа (информация о том, что выводить), а в старшем байте – атрибуты символа (информация о том, как выводить). Структура байта атрибутов приведена на рис. 3.4.

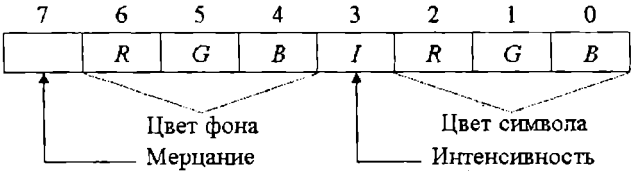

Рис. 3.4. Структура байта атрибутов

В табл. 3.2 приведены цвета, которые можно получить, задавая комбинации битов красного, зеленого и синего цветов, а также их интенсивности.

Таблица 3.2. Генерация цвета

R	*G*	*B*	*I*	*Цвета*
0	0	0	0	Черный
0	0	1	0	Синий
0	1	0	0	Зеленый
0	1	1	0	Голубой
1	0	0	0	Красный
1	0	1	0	Сиреневый
1	1	0	0	Коричневый
1	1	1	0	Белый
0	0	0	1	Серый
0	0	1	1	Ярко-синий
0	1	0	1	Ярко-зеленый
0	1	1	1	Ярко-голубой
1	0	0	1	Ярко-красный
1	0	1	1	Ярко-сиреневый
1	1	0	1	Желтый
1	1	1	1	Белый (повышен. интенсивности)

На экране одновременно могут находиться 2000 символов (80*25). Эти символы занимают 4000 байт (около 4 Кбайт) видеопамяти. Такая область памяти, занимаемая данными для вывода экрана, называется ***дисплейной страницей***. Нетрудно подсчитать, что в видеопамяти *CGA* объемом 16 К могут разместиться 4 дисплейные страницы. При этом данные экрана располагаются в памяти построчно (начиная с верхнего левого угла), а небольшие пустоты между страницами усекаются аппаратно (рис. 3.5).

Аппаратура адаптера периодически считывает содержимое видеопамяти (точнее – дисплейной страницы) и помещает его на экран. Электронный луч, управляемый системой отклонения, пробегает по экрану строка за строкой слева направо и сверху вниз (развертка). При этом контроллер включает и выключает интенсивность луча, повторяя «узор» битов в видеопамяти. За секунду электронный луч 50 раз пробегает по всему экрану (кадру). Между кадрами луч должен из правого нижнего угла вернуться в левый верхний угол. Это движение называется ***обратным ходом луча***.

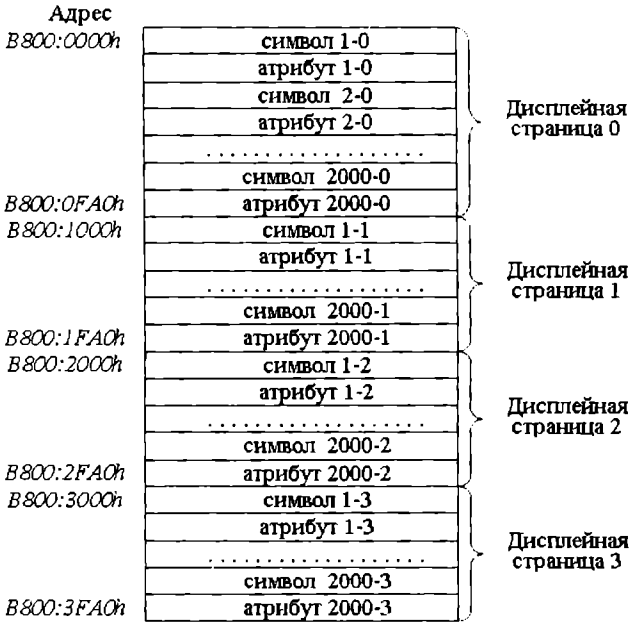

Рис. 3.5. Содержимое видеопамяти

Для того чтобы вывести символ с определенными атрибутами в заданную позицию экрана, необходимо выполнить последовательность действий:

1) рассчитать смещение *L* относительно начала видеопамяти;

2) записать по адресу *B800:L* код ASCII выводимого символа;

3) записать по адресу (*B800:L +1*) байт атрибутов выводимого символа.

Для расчета *L* можно воспользоваться формулой

$L = (80*Y + X)*2,$

где *X*, *Y* – текущие координаты (столбец и строка) курсора.

20.3.3. Управление курсором

Под термином ***управление курсором*** понимается изменение формы курсора и его координат на экране. При этом аппаратно курсор никак не связан с выводом символов на экран. Например, если даже очистить от символов весь экран, курсор все равно останется неподвижным.

Общепринято, что курсор должен указывать на то место экрана, куда будет выведен следующий символ. Если мы используем системные вызовы DOS или BIOS, то их подпрограммы сами устанавливают курсор в нужное положение. Но если мы программируем на уровне портов, то должны сами позаботиться об этом. Так как за отображение курсора на экране отвечает *CRT*, то для управления курсором необходимо выполнить действия по программированию этого контроллера.

Для того чтобы курсор был виден на экране, его координаты могут меняться в пределах 25 строк (0...24) и 80 столбцов (0...79), т. е. в пределах экрана. При этом положение курсора содержится в регистрах 14 и 15 (см. табл. 3.1) как число от 0 до 1999, что соответствует 2000 (25*80) позициям экрана. Если содержимое регистров 14 и 15 изменить, то положение курсора также изменится. Для этого достаточно выполнить действия:

1) поместить в *RI* запрос на использование регистра 15;
2) поместить в *RD* младший байт позиции курсора;
3) поместить в *RI* запрос на использование регистра 14;
4) поместить в *RD* старший байт позиции курсора.

После выполнения этих действий курсор будет немедленно перемещен контроллером на заданную позицию экрана.

Форма курсора может меняться – от тонкой линии до максимального размера, отводимого под символ. Это обеспечивается за счет того, что курсор строится из коротких горизонтальных отрезков, верхний из которых называется начальной строкой курсора, а нижний – конечной строкой. В *CGA* для каждого символа (а, следовательно, и для курсора) отводятся только 8 строк, пронумерованных, начиная сверху, от 0 до 7. Значения начальной и конечной строк содержатся соответственно в управляющих регистрах 10 и 11. Запись в эти регистры выполняется точно так же, как и изложенная выше запись в регистры 14 и 15. (Заметим, что интересный эффект получается при задании начальной строки больше конечной.)

20.3.4. Логическая схема

На рис. 3.6 приведена логическая схема простейшего драйвера экрана. Этот драйвер позволяет любой обрабатывающей программе вывести на экран символ с заданным цветом. Драйвер состоит из двух логических процедур, доступных извне драйвера: «Инициализация экрана», «Вывод символа». Кроме того, драйвер включает внутреннюю процедуру «Установка курсора» и три структуры данных. Первые две из них – переменные *X* и *Y*, содержащие текущие координаты курсора. Использование этих переменных позволяет повысить скорость пересчета новых координат курсора при обработке символов «Возврат каретки» и «Перевод строки». Еще одна структура данных – переменная *A*, содержащая байт атрибутов (см. рис. 3.4).

Логическая процедура «Инициализация экрана» выполняет первоначальную подготовку экрана (через его адаптер) и самого драйвера к последующей работе, а также осуществляет деинициализацию – восстановление прежнего содержимого видеопамяти. Данная процедура имеет три входных параметра – *u* (0 – выполнить инициализацию, 1 – выполнить деинициализацию), цвет фона *F* и цвет символа *M*. Параметр *F* содержит число ($0 \le F \le 15$), содержащееся в последовательности четырех битов 3–6 в байте атрибутов, а параметр *M* ($0 \le M \le 7$) – число, содержащееся в последовательности трех битов 0–2 этого же байта (см. рис. 3.4). Например, *M=2=010b* задает зеленый цвет символов. Заметим, что для передачи входных параметров *u*, *F* и *M* могут быть использованы любые три байтовых регистра общего назначения, например: *AL*, *DL* и *DH*.

В том случае, если драйвер экрана предназначен для обслуживания одной-двух прикладных программ, инициируемых достаточно редко, нецелесообразно постоянно держать драйвер в ОП, сделав драйвер резидентным. Гораздо лучше объединить его и прикладную программу в единый загрузочный модуль. В этом случае вести речь о двух программах (прикладной и драйвере) можно лишь с большой условностью, так как подобное разделение есть лишь на уровне исходных модулей (*asm*-файлов), соединяемых с помощью псевдооператора *%include*.

Рис. 3.6. Логическая схема драйвера экрана:
u – тип операции (0 – инициализация, 1 – деинициализация); *F* – цвет фона ($0 \le F \le 15$); *M* – цвет символа ($0 \le M \le 7$); *S* – ASCII-код символа; *A* – байт атрибутов; *X*, *Y* – текущие координаты курсора

С целью улучшить качество вывода на экран информации, выполняемого DOS после завершения нашей программы, во время инициализации экрана (*u = 0*) производится запоминание используемой (нулевой) видеостраницы. При выполнении деинициализации экрана (*u = 1*) запомненное содержимое нулевой видеостраницы восстанавливается.

Алгоритм логической процедуры «Инициализация экрана» включает шаги:

Ш а г 1. Если *u = 0* то переход к шагу 3.

Ш а г 2. Восстановление нулевой видеостраницы. Переход к шагу 10.

Ш а г 3. Сохранение нулевой видеостраницы.

Ш а г 4. Получение требуемого байта атрибутов и его запись в переменную *A*.

Ш а г 5. Очистка используемой (нулевой) видеостраницы путем заполнения ее символами пробела.

Ш а г 6. Задание нулевой видеостраницы в качестве отображаемой на экране. Для этого в регистры начального адреса сканирования (см. табл.3.1) следует поместить 0.

Ш а г 7. Задание максимальной толщины курсора.

Ш а г 8. Запись нулевых значений в переменные *X* и *Y*.

Ш а г 9. Установка курсора в начальную позицию экрана.

Ш а г 10. Завершение процедуры инициализации.

Логическая процедура «Вывод символа» имеет один входной параметр: *S* – ASCII-код символа. Алгоритм данной логической процедуры:

Ш а г 1. Если символ *S* есть «Перевод строки» (*0Ah*), то *Y←Y+1*. Переход к шагу 4.

Ш а г 2. Если символ *S* есть «Возврат каретки» (*0Dh*), то *X←0*. Переход к шагу 4.

Ш а г 3. Запись символа и его атрибутов в видеопамять. *X←X+1*. Если *X=80*, то *X←0*, *Y←Y+1*.

Ш а г 4. Если Y=25, то $Y \leftarrow 0$.

Ш а г 5. Установка курсора.

Внутренняя процедура драйвера «Установка курсора» не имеет параметров. Она выполняет установку курсора в ту позицию, которая соответствует текущим значениям переменных X и Y.

Лабораторная работа 18

Задание

Требуется реализовать программно нерезидентный драйвер экрана с видеоадаптером *CGA*, логическая схема которого приведена на рис. 3.6. Кроме этого, требуется разработать прикладную программу, выполняющую вывод на экран вашу фамилию и имя, и, возможно, другую информацию (по вашему усмотрению), используя для вывода на экран ваш драйвер. Цвет символов, выводимых на экран, а также цвет фона должны быть выбраны из табл. 3.3 в зависимости от номера вашего варианта.

Таблица 3.3. Цвета символов и фона

K	*Цвет символов*	*Цвет фона*
1	Черный	Зеленый
2	Синий	Коричневый
3	Зеленый	Красный
4	Голубой	Черный
5	Красный	Сиреневый
6	Сиреневый	Зеленый
7	Коричневый	Голубой
8	Белый	Черный
9	Серый	Ярко-красный
10	Ярко-синий	Ярко-зеленый
11	Ярко-зеленый	Ярко-белый
12	Ярко-голубой	Ярко-красный
13	Ярко-красный	Желтый
14	Ярко-сиреневый	Ярко-синий
15	Желтый	Ярко-голубой
16	Ярко-белый	Желтый
17	Серый	Ярко-красный
18	Ярко-синий	Серый
19	Ярко-зеленый	Ярко-синий
20	Ярко-голубой	желтый

Результат выполнения работы оформляется в виде двух файлов с расширением *asm* (исходные тексты прикладной программы и драйвера) и одного файла с расширением *com* (загрузочный модуль прикладной программы и драйвера).

Рекомендуемый план отладки

Рекомендуется выполнять отладку драйвера экрана не целиком, а постепенно наращивая его функции в соответствии со следующим планом.

Ш а г 1. Прикладная программа выполняет вызов процедуры инициализации, за которым следуют ожидание нажатия клавиши и возврат в DOS. Процедура инициализации выполняет только очистку экрана заданным цветом фона.

Ш а г 2. Отличается от шага 1 только тем, что процедура инициализации выполняет не только очистку экрана, но и первоначальную установку курсора.

Ш а г 3. Прикладная программа выполняет вызов процедуры инициализации, за которым следуют ожидание нажатия клавиши. Далее она в цикле вызывает процедуру вывода символа, передавая ей каждый раз на вход код следующего символа из символьной строки, заданной в прикладной программе. Данная строка завершается каким-то особым байтом, например *24h*. После завершения вывода прикладная программа ожидает нажатия клавиши и выполняет возврат в DOS.

Ш а г 4. Отличается от шага 3 тем, что выводимая на экран символьная строка содержит кроме других символов «возврат каретки» и «перевод строки».

☑ **Примечания:**

1. При выполнении отладки обычно требуется уметь остановить выполнение программы в заданной точке. В данной работе при выполнении прикладной программы подобная остановка используется дважды. Во-первых, после завершения процедуры инициализации, так как требуется время на перемещение курсора. Во-вторых, программа приостанавливается перед возвратом в DOS. В противном случае сразу же после возвращения управления из прикладной программы в DOS последняя заменит на экране выходные данные программы своими данными. Для выполнения остановки программы можно воспользоваться системным вызовом BIOS «ожидание ввода с клавиатуры» – «*int 16h*» (функция *0*).

2. Программа не должна содержать системных вызовов DOS и BIOS, выполняющих вывод на экран.

3. При отладке подпрограммы, выполняющей работу с курсором, надо помнить, что подпрограммы DOS и BIOS, выполняющие вывод на экран, будут игнорировать вашу установку курсора и вернут его в то положение, которое он занимал до начала выполнения вашей программы (соответствующее 2-байтовое значение хранится в области данных BIOS). Например, если вы пользуетесь отладчиком, то во время работы курсор перестает «слушаться» вашу программу и не двигается с места.

21. Управление устройствами с прерываниями

21.1. Контроллер прерываний

При асинхронном вводе-выводе с прерываниями драйвер ожидает завершения текущей операции ввода-вывода пассивно, не занимая для этого ЦП. Поэтому программа, сделавшая запрос на ввод-вывод, может выполнять в это же время (параллельно) какую-то другую свою работу на ЦП. При завершении операции ввода-вывода ИУ выдает в ЦП сигнал прерывания.

Прерывания от ИУ относятся к маскируемым внешним аппаратным прерываниям (см. п. 19.1.1). В большинстве ВС сигналы таких прерываний поступают не сразу в ЦП, а сначала в ***программируемый контроллер прерываний***. Данный контроллер упорядочивает поступление сигналов внешних прерываний в ЦП, освобождая его от необходи-

мости наводить порядок среди этих сигналов. Далее речь будет идти о программируемом контроллере прерываний *i8259A*.

Контроллер *i8259A* имеет 8 входов – *IRQ0*, *IRQ1*...*IRQ7*, позволяющих принимать запросы на прерывание от восьми ИУ:

IRQ0 – таймер;
IRQ1 – клавиатура;
IRQ2 – второй контроллер прерываний;
IRQ3 – последовательный порт *COM2*;
IRQ4 – последовательный порт *COM1*;
IRQ5 – параллельный порт *LPT2*;
IRQ6 – гибкий диск;
IRQ7 – параллельный порт *LPT1*.

Данные входы имеют приоритеты: чем меньше номер входа, тем приоритет выше. Поэтому самыми приоритетными являются сигналы прерываний от таймера. Приоритет сигнала используется контроллером прерываний для определения возможности направления этого сигнала в ЦП. При этом если в контроллер одновременно поступили несколько сигналов прерываний, то из них в ЦП будет отправлен тот сигнал, приоритет которого выше.

Кроме того, если ЦП уже занят обработкой какого-то прерывания, то новый сигнал будет отправлен контроллером прерываний только в том случае, если его приоритет выше, чем у того прерывания, обработчик которого выполняется на ЦП. (Вспомним, что для того, чтобы новое прерывание действительно прервало обработку старого прерывания, дополнительно требуется, чтобы в начале обработчика старого прерывания стояла команда *sti*.)

Номер прерывания, соответствующий конкретному ИУ, можно определить следующим образом: к базовому номеру, соответствующему контроллеру прерываний (8), следует прибавить номер входа в этот контроллер. Например, номер прерывания от клавиатуры: $N = 8 + 1 = 9$.

Как и любое ИУ, контроллер прерываний имеет порты, которые используются программами (драйверами) для управления им. Эти порты имеют адреса *20h* и *21h*. Порт *21h* используется, в частности, для маскирования (то есть для запрета) прерываний от устройств требуемого типа. Для этого требуется записать единицу в тот бит порта, который соответствует номеру входа для устройства. Например, следующие две команды выполняют маскирование прерываний от клавиатуры:

```
mov   al, 00000010b   ; Вход для клавиатуры - IRQ1
out   21h, al
```

В конце обработчика внешних прерываний следует разрешить обработку прерываний с более низкими приоритетами. Для этого достаточно выполнить две команды:

```
mov   al, 20h         ; Число 20h используется для двух
out   20h, al         ;   разных величин
```

Существуют другие команды управления контроллером прерываний, позволяющие размаскировать прерывания от любого ИУ (независимо от маскирования в ЦП), а также изменять приоритеты ИУ. Аналогичные команды используются и для управления вторым (ведомым) контроллером прерываний.

Ведомый контроллер подсоединяется к входу *IRQ2* ведущего контроллера. Поэтому ИУ, обслуживаемые ведомым контроллером, выдают сигналы прерываний с приоритетами, расположенными между приоритетами клавиатуры и последовательного порта *COM2*. Для управления ведомым контроллером используются порты *A0h* и *A1h*, аналогичные портам *20h* и *21h* для ведущего контроллера. Базовый номер прерываний, соответствующий ведомому контроллеру прерываний, равен *70h*.

Перечислим два типа устройств, обслуживаемых ведомым контроллером прерываний (с указанием его входов):

IRQ13 – математический сопроцессор;

IRQ14 – жесткий магнитный диск.

21.2. Алгоритм обработки прерываний

Использование при выдаче внешних маскируемых прерываний контроллера прерываний требует уточнения алгоритма обработки прерываний, рассмотренного в п. 19.1.4. Новый вариант этого алгоритма:

- Поступление сигнала прерывания в ЦП. Для этого должны быть выполнены действия:
  - ИУ посылает сигнал прерывания в контроллер прерываний *i8259A*;

    если данный тип прерываний незамаскирован, *i8259A* выдает сигнал прерывания на вход *INTR* процессора.
- Собственно прерывание. Оно выполняется ЦП по окончанию выполнения текущей машинной команды и включает действия:
  - ЦП возвращает в ИУ сигнал подтверждения;

    ИУ передает по ОШ в ЦП номер прерывания *N*;

    ЦП помещает текущее содержимое *FLAGS*, *CS* и *IP* в программный стек;

    в *IP* загружается слово ОП, имеющее реальный адрес *4N*, а в *CS* загружается слово, имеющее адрес – *4N + 2*;

    ЦП запрещает маскируемые прерывания путем сброса в нуль флажка *IF* в регистре *FLAGS*.
- Начальный этап программной обработки прерывания. Он выполняется обработчиком прерываний и включает действия:
  - разрешение маскируемых прерываний с помощью команды *sti*;

    сохранение в программном стеке содержимого тех регистров, с которыми будет работать программа обработчика;

    запись в сегментный регистр данных *DS* значения, которое соответствует адресу-сегменту данных обработчика прерываний.
- Действия обработчика прерываний, определяемые типом прерывания.
- Завершение обработки прерывания. Сюда относятся действия:
  - запись обработчиком прерываний кода *20h* в порт *20h*;

    восстановление содержимого регистров, запомненного ранее в стеке;

    выполнение обработчика прерываний завершается командой *iret*, которая извлекает из стека прежнее содержимое *IP*, *CS* и *FLAGS*, тем самым, возвращая управление в прерванную программу.

21.3. Пример драйвера

21.3.1. Логическая схема

В качестве примера опять рассмотрим драйвер ввода с перфоленты. Для связи с ИУ используются порты *90h* (*RS*) и *91h* (*RD*). Структура *RS* приведена на рис. 3.1. Допустим, что устройству ввода с перфоленты соответствует вектор прерываний с номером *60h*. Кроме того, предположим, что контроллер данного устройства соединен с входом *IRQ5* ведущего контроллера прерываний.

Логическая схема драйвера представляет собой активный пакет, состоящий из двух интерфейсных логических процедур, логического процесса и трех структур данных (рис. 3.7).

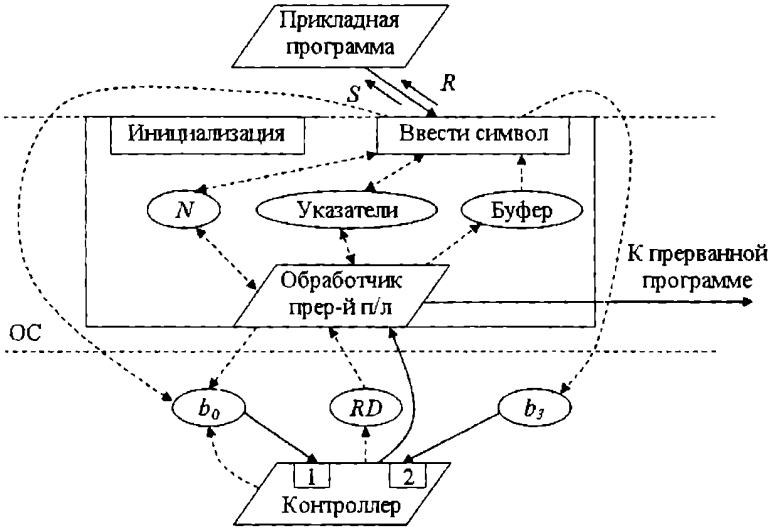

Рис. 3.7. Логическая схема асинхронного драйвера с прерываниями:
S – код символа; *R* – признак результата (0 – успешно, 1 – символа пока нет); *N* – число занятых позиций в буфере; b_0 – бит разрешения работы; b_3 – бит разрешения прерываний

Логическая процедура «Инициализация» без параметров. Она выполняет три действия:

1) делает остальные модули драйвера резидентными;

2) заполняет вектор прерываний с номером *EEh* (для последующего инициирования резидентной процедуры «Ввести символ»);

3) заполняет вектор прерываний с номером *60h* (для последующего инициирования обработчика прерываний от перфоленты).

Логическая процедура «Ввести символ» обслуживает запросы прикладных и системных программ по вводу символов с перфоленты. Обработчик прерываний перфоленты – логический процесс без параметров. Рассмотрим совместную работу перечисленных модулей по логической схеме.

После своего вызова логическая процедура «Ввести символ» проверяет значение *N*. Если *N = 0* (в буфере ничего нет), то устанавливаются в «1» биты b_0 и b_3 *RS*. После этого данная логическая процедура выполняет возврат в вызвавший ее модуль, т. е. в прикладную программу, с признаком результата *R = 1*.

Значение $b_0=1$ инициирует работу контроллера (вход 1) по вводу очередного символа. Контроллер помещает код символа в *RD* (сбрасывая при этом бит b_0 *RS*) и оказывается в состоянии «вход 2». Так как бит разрешения прерывания $b_3=1$, то контроллер инициируется по данному входу и выдает сигнал прерывания. Обработчик прерываний считывает код символа из *RD* в буфер и увеличивает значение *N* на единицу. Если при этом буфер оказывается полным ($N=N_{MAX}$), то управление сразу же возвращается прерванной программе. Иначе сначала установкой $b_0=1$ инициируется работа контроллера по вводу следующего символа.

Если при вызове логической процедуры «Ввести символ» $0<N<N_{MAX}$, то из буфера выбирается очередной символ и устанавливается $R=0$. Если $N=N_{MAX}$, то перед выполнением данных действий устанавливается $b_0=1$ (этот бит не был установлен обработчиком прерываний).

В основе алгоритмов логической процедуры «Ввести символ» и обработчика прерываний лежат операции с буфером. Обычно используют буфер в виде несвязанной циклической очереди. Напомним, что подобно стеку, очередь является линейным списком, исключение элементов из которого производится только из начала списка. Но в отличие от стека добавление нового элемента производится не в начало, а в конец списка. Поэтому для работы с очередью используются два указателя, один из которых *S* указывает на начало очереди, а второй *F* – на ее конец. В отличие от обычной несвязанной очереди, ***циклическая несвязанная очередь*** позволяет лучше использовать пространство памяти. Поясним ее организацию на примере.

Пусть имеется область памяти длиной десять ячеек, предназначенная для размещения буфера (рис. 3.8). Кроме того, есть две ячейки ОП, являющиеся указателями: *F* указывает на очередную пустую ячейку буфера, в которую можно занести новый элемент (символ), а *S* указывает на ячейку, содержащую первый из еще не взятых элементов. Цикличность буфера заключается в том, что при достижении максимального граничного адреса (на рис. 3.8 этот адрес равен 9) указатель корректируется так, что вместо 9 он будет содержать 0 (то есть будет указывать на начало области памяти).

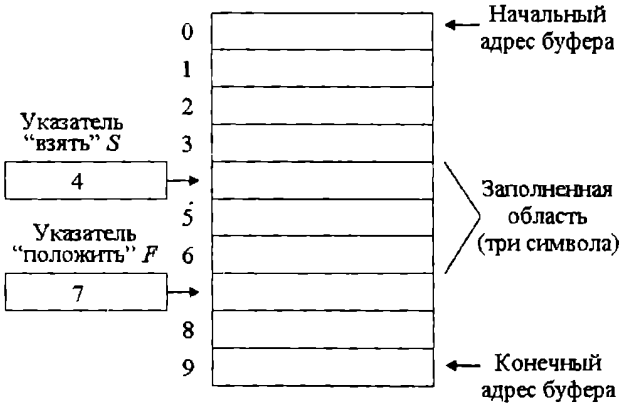

Рис. 3.8. Циклический буфер и указатели

Заметим, что если указатель «взять» движется в сторону больших адресов (на рис. 3.8 вниз) и становится равным указателю «положить», то очевидно, что буфер пуст, и счетчик элементов равен нулю. А если указатель «положить» «догонит» указатель «взять», то буфер полон и счетчик элементов равен максимально возможному числу (10).

21.3.2. Программа драйвера

Программа асинхронного посимвольного драйвера, использующего прерывания:

```
;          Асинхронный драйвер ввода символов с перфоленты
;          ------------------------------------------------
;       Содержит: 1) п/п-а инициализации (вызов из DOS)
;       2) п/п-а ввода символа (вызов командой «int 0eeh»)
;       3) обработчик прерываний перфоленты (инициируется аппаратно)
;
Nmax    equ       20          ; Емкость буфера = 20 символов
        absolute  2ch         ; Следующий байт имеет адрес-смещение 2ch
Blocokr resw      1           ; Адрес-сегмент блока окружения
        segment   code        ; Виртуальный сегмент кода
        org       100h
Start:  jmp       Init        ;  Переход на инициализацию
;       Определение данных
N       db        0           ; Счетчик символов в буфере
S       dw        0           ; Указатель «взять»
F       dw        0           ; Указатель «положить»
Buf     times  Nmax db 0      ;  Буфер
;
;     Ввод символа - обработчик программного прерывания с номером EEh
;     ---------------------------------------------------------------
;  Выходы:  флаг CF – признак результата (0–успешно; 1–символа пока нет)
;                AL – код символа
Begin:  sti                   ; Разрешить маскируемые прерывания
        push  ax              ; Сохранение
        push  bx              ;   содержимого
        push  ds              ;  регистров в стеке
        mov   ax, cs          ; DS будет адресовать данные
        mov   ds, ax          ;  резидентной программы
        cmp   [N], byte 0     ; Буфер пуст ?
        jnz   .Simv           ; Если не пуст, то переход
        in    al, 90h         ; Чтение RS в AL
        or    al, 00001001b   ; Запуск устройства
        out   90h, al         ;  и разрешение прерываний от п/л
        stc                   ; CF ← 1
        jmp   .Exit
.Simv:  cmp   [N], byte Nmax  ; Буфер полон?
        jnz   .M1             ; Если не полон, то переход
        in    al, 90h         ; Чтение RS в AL
        or    al, 00000001b   ; Запуск
        out   90h, al         ;  устройства
.M1:    mov   bx, [S]         ; Запись в AL
        mov   al, [Buf+bx]    ;   символа из буфера
        dec   [N]             ; Уменьшение счетчика символов на 1
        cmp   [S], word  Nmax –1  ; S указывает на конец буфера ?
```

```
        jz      .M2                 ; Если да, то переход
        inc     word [S]            ; Увеличение S на 1
        jmp     .M3
.M2:    mov     [S], word 0         ;  S ← 0
.M3:    clc                         ;  CF ←0
.Exit:  pop     ds                  ; Восстановление содержимого
        pop     bx                  ;   регистров
        pop     ax                  ;   из стека
        iret                        ; Возврат в программу из прерывания
;
; Обработчик аппаратных прерываний с номером 60h от устройства ввода
; перфоленты
;-------------------------------------------------------------------
Obrpr:  sti                         ; Разрешить маскируемые прерывания
        push    ax                  ; Сохранение
        push    bx                  ;   содержимого
        push    ds                  ;   регистров в стеке
        mov     ax, cs              ; DS будет адресовать данные
        mov     ds, ax              ;   резидентной программы
        in      al, 91h             ; Прием символа из RD в AL
        mov     bx, [F]             ; Запись символа из
        mov     [Buf+bx], al        ; AL в буфер
        inc     word   [N]          ; Увеличение счетчика символов на 1
        cmp     [F], word  Nmax –1  ; F указывает на конец буфера ?
        jz      .L1                 ; Если да, то переход
        inc     word[F]             ;  F ← F +1
        jmp     .L2
.L1:    mov     [F], word  0        ;  F ← 0
.L2:    cmp     [N], byte  Nmax     ; Буфер полон ?
        jz      .Exit               ; Если да, то переход
        in      al, 90h             ; Чтение RS в AL
        or      al, 00000001b       ; Запуск
        out     90h, al             ;   устройства
.Exit:  mov     al, 20h             ; Разрешение менее приоритетных
        out     20h, al             ;    прерываний
        pop     ds                  ; Восстановление содержимого
        pop     bx                  ;   регистров
        pop     ax                  ;   из стека
        iret                        ; Возврат в программу из прерывания
;
; Инициализационная часть программы
Init:   mov     ax, 25eeh           ; Запись стартового адреса обработчика
        mov     dx, Begin           ;   в вектор прерываний
        int     21h                 ;    с номером EEh
        mov     ax, 2560h           ; Запись стартового адреса обработчика
        mov     dx, Obrpr           ;   в вектор прерываний
        int     21h                 ;   с номером 60h
```

```
mov     ah, 49h         ; Освобождение области памяти,
mov     es, Blocokr     ;   занимаемой
int     21h             ;   блоком окружения
mov     dx, Init        ; Возврат в DOS, оставшись
int     27h             ;   резидентным
```

Приведенная программа состоит из трех модулей, предваряемых данными, совместно используемыми этими модулями. Первый по порядку модуль начинается с метки *Begin* и представляет собой программную реализацию логической процедуры «Ввести символ». Данный модуль выполнен как резидентный обработчик программных прерываний с номером *EEh*. (Напомним, что в качестве этого номера разработчик может взять любой неиспользуемый номер.)

Второй модуль начинается с метки *Obrpr* и представляет собой программную реализацию логического процесса «Обработчик прерываний перфоленты». Этот модуль выполнен как резидентный обработчик внешних аппаратных прерываний с номером *60h*.

Третий модуль начинается с метки *Init* и представляет собой программную реализацию логической процедуры «Инициализация». Этот модуль выполнен как фрагмент программы, инициируемый при запуске программы из DOS.

21.4. Программирование драйвера клавиатуры

21.4.1. Логическая схема ввода с клавиатуры

На рис. 3.9 приведена логическая схема, обеспечивающая посимвольный обмен с клавиатурой. В реализации этого обмена кроме драйвера клавиатуры участвуют контроллер клавиатуры, а также разработанный в гл. 20 драйвер экрана.

Для того чтобы выполнить ввод с клавиатуры, прикладная программа вызывает интерфейсную логическую процедуру драйвера клавиатуры «Ввод символа». Если в буфере драйвера имеется хотя бы один символ ($N>0$), то этот символ в качестве выходного параметра передается в прикладную программу. При отсутствии символа в прикладную программу возвращается соответствующее значение признака результата *R*.

Исходной причиной появления кода символа в буфере драйвера является нажатие соответствующей клавиши на клавиатуре. Взаимосвязь между этими двумя событиями обеспечивают контроллер клавиатуры и обработчик прерываний. Как показано на рис. 3.9, логическая структура контроллера клавиатуры представляет собой совокупность двух процессов – «Прием» и «Выдача». Процесс «Прием» инициируется при нажатии, а также при отпускании любой клавиши и помещает генерируемый клавишей скан-код во внутренний 4-позиционный буфер контроллера. В результате появления кода в буфере скан-кодов инициируется процесс «Выдача», находившийся в состоянии «Вход 1». Данный процесс выбирает код из буфера скан-кодов, помещает его в порт *60h* и выдает в ЦП сигнал прерывания.

Если прерывание разрешено, инициируется модуль «Обработчик прерываний клавиатуры», который считывает скан-код из порта *60h* и сообщает контроллеру о завершении этой операции установкой бита 7 в порте *61h*. Это приводит к инициированию процесса «Выдача», находившегося в состоянии «Вход 2». Данный процесс возвращается в состояние «Вход 1». Далее опять выбирается скан-код из буфера, и описанные выше действия повторяются.

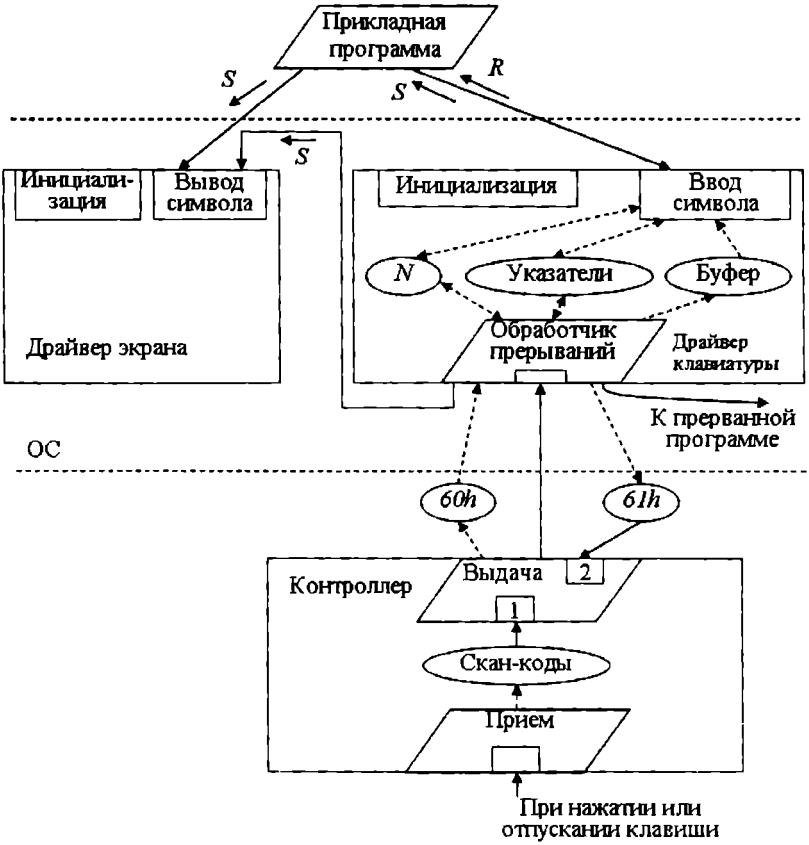

Рис. 3.9. Логическая схема ввода с клавиатуры:
S – ASCII-код символа; *N* – счетчик символов; *R* – признак результата (0 – успешно; 1 – символа пока нет)

После установки бита 7 в порте *61h* обработчик прерываний выполняет обработку принятого скан-кода. Обработка кода, вызванного нажатием алфавитно-цифровой клавиши, сводится к преобразованию скан-кода в код ASCII, размещению полученного кода в буфере драйвера и к выводу «эха» символа.

Необходимость вывода «эха» обусловлена тем, что клавиатура и экран никак не связаны между собой на уровне контроллеров. Для того чтобы видеть на экране вводимый символ, необходимо выполнить связывание на уровне драйверов. Как видно из рис. 3.9, для этого достаточно, чтобы при нажатии отображаемого символа обработчик прерываний клавиатуры вызвал логическую процедуру «Вывод символа».

Логическая процедура «Инициализация» имеет единственный входной параметр *u*: *u=0* – выполнить замену системного обработчика прерываний клавиатуры на свой обработчик; *u=1* – выполнить обратную замену своего обработчика прерываний клавиатуры на системный обработчик.

21.4.2. Скан-коды

Ранее в п. 14.1 шла речь о скан-кодах BIOS. Эта таблица скан-кодов далеко не единственная. Сейчас речь будет идти о скан-кодах клавиш (или комбинаций клавиш), которые контроллер клавиатуры передает в управляющую программу, т. е. в драйвер.

Скан-код – однобайтовое число, младшие 7 бит которого представляют идентификационный код клавиши. Старший бит кода (бит 7) указывает на то, была ли клавиша нажата (0) или освобождена (1). Например, 7-битовый скан-код клавиши <v> есть 47 (*0101111b*). Когда эту клавишу нажимают, в порт *60h* контроллер записывает код *00101111b*, а когда отпускают – *10101111b*.

Значения скан-кодов приведены в табл. 3.4. В этой таблице приведены скан-коды только тех клавиш, которым соответствуют отображаемые на экране символы. Что касается управляющих клавиш, то приведен код только для клавиши *CapsLock*, которая обрабатывается нашим драйвером.

Таблица 3.4. Скан-коды клавиш

Клавиша	*Скан-код*	*Клавиша*	*Скан-код*
1 !	0000010	Enter	0011100
2 @	0000011	a A	0011110
3 #	0000100	s S	0011111
4 $	0000101	d D	0100000
5 %	0000110	f F	0100001
6 ^	0000111	g G	0100010
7 &	0001000	h H	0100011
8 *	0001001	j J	0100100
9 (	0001010	k K	0100101
0)	0001011	l L	0100110
- _	0001100	; :	0100111
= +	0001101	' "	0101000
q Q	0010000	\ |	0101011
w W	0010001	z Z	0101100
e E	0010010	x X	0101101
r R	0010011	c C	0101110
t T	0010100	v V	0101111
y Y	0010101	b B	0110000
u U	0010110	n N	0110001
i I	0010111	m M	0110010
o O	0011000	, <	0110011
p P	0011001	. >	0110100
[{	0011010	/ ?	0110101
] }	0011011	Пробел	0111001
		Caps Lock	0111010

21.4.3. Алгоритмы программных модулей

Алгоритмы программных модулей драйвера клавиатуры имеют много общего с алгоритмами программных модулей драйвера считывателя перфоленты (см. п. 21.3). В част-

ности, буфер драйвера клавиатуры представляет собой несвязанную циклическую очередь, алгоритмы работы с которой очень похожи на соответствующие алгоритмы драйвера ввода с перфоленты. Но между этими драйверами есть и существенные отличия, вызванные, прежде всего техническими деталями управления конкретным устройством, т. е. клавиатурой.

Алгоритм программной реализации логической процедуры «Инициализация» состоит из следующих шагов:

Ш а г 1. Если *u=1*, переход к шагу 4 .

Ш а г 2. Сохранение в ОП прежнего содержимого вектора прерываний с номером *09h*.

Ш а г 3. Запись стартового адреса обработчика прерываний клавиатуры в вектор прерываний с номером *09h*. Переход к шагу 5.

Ш а г 4. Восстановление прежнего содержимого вектора прерываний с номером *09h*.

Ш а г 5. Возврат из процедуры.

Алгоритм обработчика прерываний клавиатуры включает шаги:

Ш а г 1. Разрешение маскируемых прерываний.

Ш а г 2. Сохранение в программном стеке содержимого регистров.

Ш а г 3. Запись в сегментный регистр *DS* значения, которое соответствует адресу-сегменту данных обработчика прерываний.

Ш а г 4. Чтение из порта *60h* значения скан-кода.

Ш а г 5. Установка бита 7 порта *61h*.

Ш а г 6. Сброс бита 7 порта *61h*.

Ш а г 7. Если была отпущена клавиша *CapsLock*, то инвертирование флага статуса клавиатуры. Переход к шагу 15.

Ш а г 8. Если было отпускание клавиши, то переход к шагу 15.

Ш а г 9. Если буфер полон, то переход к шагу 15.

Ш а г 10. Если скан-код неизвестен, то переход к шагу 15.

Ш а г 11. Перекодировка скан-кода в символ ASCII.

Ш а г 12. Запись символа ASCII в буфер драйвера.

Ш а г 13. Вывод «эха» полученного символа на экран.

Ш а г 14. Если полученный символ является символом «возврат каретки» (этот символ есть результат нажатия <Enter>), то запись в буфер и вывод на экран дополнительного символа «перевод строки».

Ш а г 15. Выдача команды в программируемый контроллер прерываний для разрешения менее приоритетных прерываний.

Ш а г 16. Восстановление содержимого регистров из стека.

Ш а г 17. Возврат из прерывания в прерванную программу.

Дополнение к шагам 5 и 6. Порт *61h* используется не только клавиатурой (она использует только бит 7), но и другими устройствами. Поэтому к моменту завершения шага 6 содержимое этого порта должно быть тем же, что было до начала шага 5.

Дополнение к шагам 7 и 10. Большинству клавиш соответствует не один, а два символа. Например, одна и та же клавиша соответствует символам «1» и «!». Каждой большой (прописной) букве соответствует малая (строчная) буква. Для того чтобы обработчик прерываний выполнял правильное преобразование скан-кода в код ASCII, необходимо ввести понятие «состояние клавиатуры». В одном состоянии в буфер драйвера записывается код ASCII большой буквы, а во втором – малой. Часто состояния клавиатуры называют «нижним регистром» и «верхним регистром».

Для отслеживания состояния клавиатуры обработчик прерываний должен иметь двоичную переменную «флаг статуса клавиатуры». Для управления этим флагом рекомендуется использовать управляющую клавишу *<CapsLock>*. Нажатие, а затем отпускание *<CapsLock>* обеспечивает долговременное изменение флага статуса, которое действует до следующего нажатия этой же клавиши.

При программировании шага 10 удобно свести значения скан-кодов и соответствующих кодов ASCII в единую таблицу. Первая строка этой таблицы может иметь вид

```
Tabl db   02h, '1', '!'
```

Строка таблицы, соответствующая пробелу:

```
     db   39h, ' ', ' '
```

В данной таблице отсутствует строка, соответствующая управляющему символу *CapsLock*. В качестве последнего байта таблицы рекомендуется взять нулевой байт.

Шаг 12 рекомендуется реализовать в виде программной процедуры.

Алгоритм подпрограммы «Ввод символа» состоит из таких шагов:

Шаг 1. Сохранение в программном стеке содержимого регистров.

Шаг 2. Если буфер пуст, то установка признака результата и переход к шагу 7.

Шаг 3. Переписка символа из буфера в регистр, используемый для передачи символа из подпрограммы в вызывающую программу.

Шаг 4. Увеличение указателя «взять» и уменьшение счетчика символов.

Шаг 5. Если указатель «взять» вышел за границу буфера, то установка его на начало буфера.

Шаг 6. Обнуление признака результата.

Шаг 7. Восстановление регистров из стека.

Шаг 8. Возврат из подпрограммы.

21.5. Прямой доступ в память

В рассмотренных выше драйверах для передачи каждого байта данных между ИУ и ОП приходится выполнять несколько машинных команд. Если в программе требуется выполнить ввод-вывод не одного, а целой последовательности байтов, то затраты времени ЦП на их передачу будут весьма существенны. Например, обмен данными с магнитным диском производится посекторно, т. е. блоками по 512 байт.

Применение специального ***контроллера прямого доступа в память*** **(ПДП)** позволяет существенно снизить затраты времени ЦП, так как программа драйвера будет выполнять операции инициирования и завершения ввода-вывода не каждого байта, а лишь всего блока. В процессе ввода-вывода с использованием ПДП обмен байтами данных между ИУ и ОП производится не через регистр RD и регистры ЦП, а через контроллер ПДП. При этом роль ЦП ограничивается выдачей разрешений на занятие ОШ (а точнее – шины данных).

Обычно контроллер ПДП рассчитан на одновременное обслуживание нескольких ИУ (каждое ИУ подключается к нему набором проводников). Часть контроллера ПДП, предназначенная для обслуживания одного ИУ, называется ***каналом*** **ПДП**. Например, совместно с ЦП типа i8086 наиболее часто используется 4-канальный контроллер ПДП i8237. Этот контроллер расположен на системной плате и имеет следующее назначение каналов:

0 – предназначен для «освежения» памяти, постоянно восстанавливает заряд ячеек ОП;

1 – свободен;

2 – обмен с контроллером гибких дисков;

3 – обмен с контроллером жестких дисков.

Контроллер i8237 имеет общие для всех каналов регистры управления, а также для каждого канала свои регистры канала. В отличие от обычных RS, ***регистры управления*** не имеют битов состояния. К таким регистрам относятся регистр режима и регистр масок. ***Регистр режима*** (порт *0Bh*) – 8-битовый регистр (рис. 3.10). Поясним смысл двух терминов: автоинициализация и тип передачи.

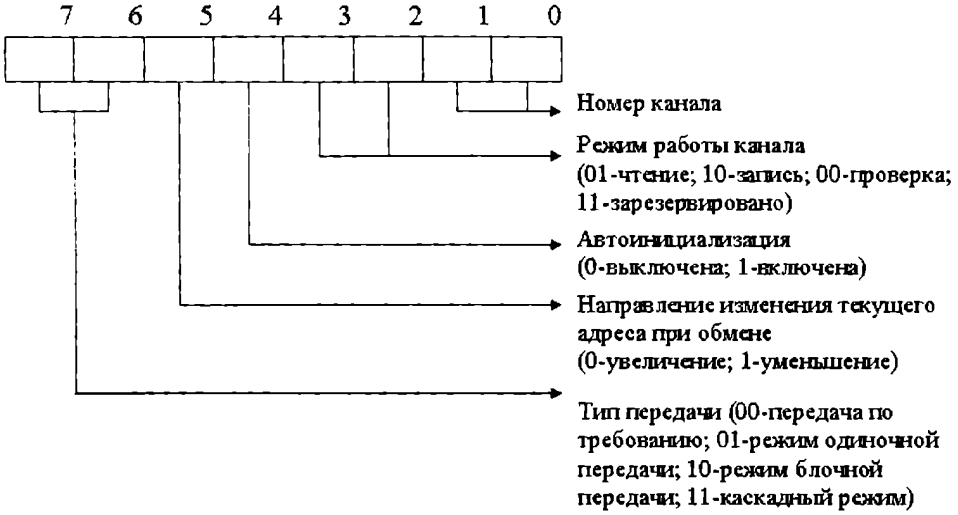

Рис. 3.10. Структура регистра режима

Автоинициализация. После завершения обычной передачи используемый канал ПДП маскируется и должен быть перепрограммирован для дальнейшей работы с ним. При автоинициализации маскировка канала после окончания передачи не происходит и контроллер ПДП полностью готов для повторения обмена по данному каналу.

Тип передачи. Контроллер ПДП поддерживает четыре различных режима передачи:

1) режим одиночной передачи – для передачи каждого байта в ОП (из ОП) контроллер ПДП запрашивает доступ к ОШ у ЦП;

2) режим блочной передачи – запрос на занятие ОШ делается один раз для всего блока;

3) режим передачи по требованию – контроллер ПДП ожидает подготовку байта контроллером устройства и делает запрос на занятие ОШ;

4) каскадный режим – используется в том случае, если ЭВМ имеет более одного контроллера ПДП.

Регистр масок – 4-битовый регистр. Значение одного бита маскирует (1) или демаскирует (0) соответствующий канал. Доступ к регистру масок осуществляется через 8-битовый порт *0Ah* (рис. 3.11).

Регистры канала предназначены для размещения численных параметров предстоящего обмена между ИУ и ОП. Это регистры:

- ***Регистр начального адреса*** – содержит 16 младших битов начального адреса буфера – области ОП, в которую будет считываться (при чтении) блок байтов с ПУ. При записи, наоборот, информация из буфера будет переписываться на ПУ.

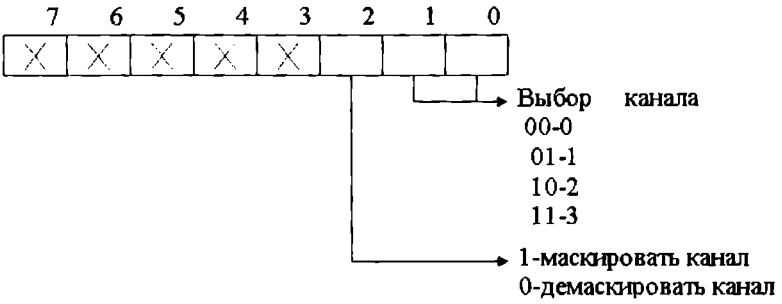

Рис. 3.11. Структура порта для доступа к регистру масок

Регистр страницы – содержит четыре старших бита 20-битового адреса буфера, с которым будет производиться обмен.

Регистр счетчика – содержит число байтов, которое предстоит передать между ИУ и ОП, уменьшенное на 1.

Номера портов, используемых для доступа к регистрам канала, зависят от номера канала. Например, запись в регистры канала 2 производится через следующие 8-битовые порты: регистр начального адреса – порт *4h*; регистр страницы – порт *81h*; регистр счетчика – порт *5h*. Так как регистры начального адреса и счетчика 16-битовые, то запись в них через 8-битовые порты имеет следующую особенность.

Специальный регистр, называемый «защелкой», направляет содержимое 8-битового порта в младший или старший байт 16-битового регистра канала. Перед записью младшего байта в порт, соответствующий регистру канала, необходимо записать какое-нибудь значение в порт *0Ch*, что приведет к сбросу «защелки». После записи младшего байта в регистр канала, установка «защелки» производится автоматически, и следующий байт из порта переписывается в качестве старшего байта регистра канала.

Управление контроллером ПДП рассмотрим на примере использования его для информационного обмена с контроллером гибких дисков. Соответствующая логическая схема приведена на рис. 3.12. (На этой схеме не показаны внутренние модули драйвера дисковода, а также опущены взаимосвязи между драйвером и контроллером дисковода.)

Драйвер дисковода предоставляет прикладным и системным программам возможность выполнять информационный обмен с установленным на дисковод магнитным диском. В результате одного обращения к драйверу производится чтение или запись одного сектора диска, т. е. блока из 512 байт, имеющего номер L ($0 \le L \le L_{MAX}$).

Получив от прикладной программы запрос на чтение или запись сектора, драйвер дисковода вызывает логическую процедуру «Инициализация канала 2», входящую в состав драйвера ПДП. Данная процедура имеет два входных параметра: «Начальный адрес прикладного буфера» и «Тип операции» (0 – чтение, 1 – запись). В процессе выполнения процедуры производится запись в регистры управления и в регистры канала 2. В частности, в регистры начального адреса и страницы помещается начальный адрес прикладного буфера, а в регистр счетчика – число передаваемых байтов, уменьшенное на единицу.

После завершения инициализации канал 2 контроллера ПДП переходит в состояние «Вход 2», ожидая прихода от контроллера дисковода команды «Начать передачу». Эта команда инициирует работу контроллера ПДП по побайтовой передаче данных между контроллером дисковода и ОП.

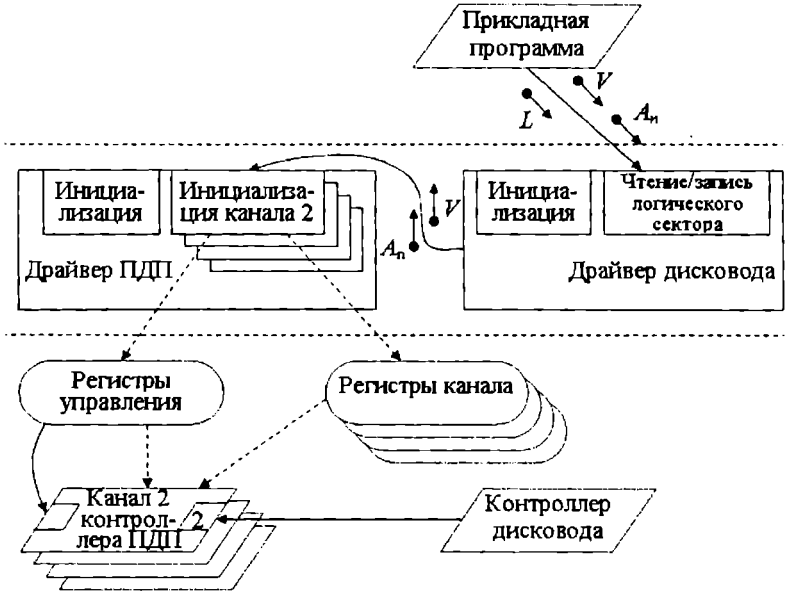

Рис. 3.12. Логическая схема применения ПДП для информационного обмена с дисководом:
V – тип операции (0 – чтение, 1 – запись); A_n – начальный адрес прикладного буфера;
L – номер логического сектора

Для передачи очередного байта в ОП (при чтении) или из ОП (при записи) контроллер ПДП выдает в ЦП запрос на занятие ОШ. Далее, при получении согласия он выводит содержимое регистров адреса и страницы на шину адреса, а в контроллер дисковода передает сигнал о том, чтобы тот вывел на шину данных очередной передаваемый байт (при чтении) или считал байт с шины данных (при записи). После этого контроллер ПДП увеличивает (или уменьшает) на единицу содержимое регистра адреса и уменьшает на единицу счетчик байтов. При достижении счетчиком нуля контроллер ПДП или выдает в ЦП сигнал прерывания, или сообщает о завершении передачи в контроллер дисковода.

Логическая процедура «Инициализация» выполняет две функции: 1) делает процедуры инициализации каналов резидентными; 2) заполняет векторы прерываний для доступа к этим резидентным процедурам. Программная реализация логической процедуры «Инициализация» аналогична реализации соответствующих процедур, рассмотренных ранее (см. пп. 20.2 и 21.3).

Логическая процедура «Инициализация канала 2» включает такие шаги:

Ш а г 1. В зависимости от типа операции (чтение или запись) сделать запись в регистр режима. При этом следует учесть следующие характеристики режима:

1) автоинициализация выключена;

2) текущий адрес при обмене увеличивается;

3) тип передачи одиночный.

Ш а г 2. Вычисление полного 20-битового адреса буфера в ОП и загрузка этого адреса в регистр начального адреса и в регистр страницы. Для вычисления полного адреса необходимо просуммировать адрес-сегмент, предварительно умноженный на 16, с адресом-смещением. Полученные четыре старших бита адреса записываются в регистр страницы, а остальные 16 бит – в регистр адреса.

Ш а г 3. Запись в регистр счетчика числа байтов, предназначенных для пересылки, уменьшенного на единицу, т. е. числа 511.

Ш а г 4. Запись в регистр маски контроллера ПДП числа, разрешающего работу канала 2.

Ш а г 5. Возврат из процедуры.

Ниже приведен фрагмент программы драйвера ПДП, выполняющий инициализацию канала 2. Так как данный фрагмент резидентен, то он вызывается через программное прерывание. В качестве номера прерывания выбрано число *EDh*.

```
; Подготовка канала 2  ПДП к чтению-записи сектора гибкого диска –
; обработчик программного прерывания с номером EDh
; ------------------------------------------------------------------------
;  Входы:   DX:BX – адрес буфера (сегмент:смещение)
;       AL – тип операции (0 – чтение, 1 – запись)
;
  Begin:    sti                 ; Разрешить маскируемые прерывания
        push  ax                ; Сохранение
        push  bx                ;   содержимого
        push  cx                ;   регистров
        push  bp                ;   в стеке
        cmp   al, 0             ; Операция – чтение ?
        jnz   .Sap              ; Нет
        mov   al, 01000110b     ; Задание режима для
        out   0bh, al           ;   чтения диска
        jmp   .Adr
.Sap:   mov   al, 01001010b     ; Задание режима для
        out   0bh, al           ;   записи на диск
.Adr:   mov   ax, dx            ; Адрес-сегмент  → AX
        mov   bp, 16
        mul   bp                ; (AX)*(BP) →DX:AX
        add   bx, ax            ; Младшие 16 бит адреса → BX
        mov   cx, 0             ;  0 → CX
        adc   dx, cx            ; Старшие биты адреса → DX
        mov   al, dl            ; Запись в
        out   81h, al           ;   регистр страницы
        out   0ch, al           ; Сброс «защелки»
        mov   al, bl            ; Запись младшего байта
        out   4h, al            ;   в регистр начального адреса
        mov   al, bh            ; Запись старшего байта
        out   4h, al            ;   в регистр начального адреса
        out   0ch, al           ; Сброс «защелки»
        mov   ax, 511           ; Запись младшего байта
        out   5h, al            ;   счетчика байтов
        mov   al, ah            ; Запись старшего байта
        out   5h, al            ;   счетчика байтов
        mov   al, 2             ; Размаскирование
        out   0ah, al           ;   канала 2
```

```
pop   bp                ; Восстановление
pop   cx                ;  содержимого
pop   bx                ;  регистров
pop   ax                ;  из стека
iret                    ; Возврат в программу из прерывания
```

Лабораторная работа 19

Задание

Требуется разработать программу нерезидентного драйвера клавиатуры, выполняющего запросы прикладных программ по вводу символов с клавиатуры. Драйвер должен обрабатывать управляющую клавишу <CapsLock>, выполняющую переключение регистров клавиатуры. Перед возвратом в DOS выполняется восстановление системной обработки прерываний от клавиатуры.

Для проверки работоспособности драйвера клавиатуры используется прикладная программа, которая начинает свою работу с вывода на экран приглашения ввести с клавиатуры строку символов, заканчивающуюся символом «$». Далее прикладная программа, используя драйвер клавиатуры, выполняет ввод символьной строки с клавиатуры в свой внутренний (прикладной) буфер. Получив код символа «$» (*24h*), прикладная программа выводит содержимое прикладного буфера на экран. Для вывода на экран используется драйвер экрана, полученный в результате лабораторной работы 18.

При правильной работе программы введенная с клавиатуры строка должна отобразиться на экране дважды: один раз при наборе на клавиатуре – «эхо» символов, а второй раз – как результат вывода на экран содержимого прикладного буфера.

Результат выполнения работы оформляется в виде двух файлов с расширением *asm* (исходные тексты прикладной программы и драйвера клавиатуры) и одного файла с расширением *com* (загрузочный модуль, включающий прикладную программу, а также драйверы экрана и клавиатуры).

Рекомендуемый план отладки

Отладку программы драйвера клавиатуры рекомендуется выполнить в два этапа.

Этап 1. На данном этапе прикладная программа сначала выполняет инициализацию драйверов экрана и клавиатуры, используя для этого соответствующие процедуры, а затем с помощью драйвера экрана выводит на экран приглашение к вводу. (Приглашение к вводу, заканчивающееся байтом *24h*, помещается в память программы с помощью псевдооператора *db*.) После этого драйвер клавиатуры, а точнее – обработчик прерываний, реагирует на нажатие клавиш, выводя соответствующие им «эхо» на экран (с помощью драйвера экрана).

Так как после инициализации драйвера клавиатуры, он и прикладная программа работают независимо друг от друга, то необходимо предусмотреть меры по завершению ввода с клавиатуры. Для этого можно поступить, например, следующим образом. Временно добавим в прикладную программу байтовую переменную *K* – счетчик введенных байтов, а также добавим следующие операторы:

```
      mov   [K], byte 20    ; K ← число вводимых символов (20)
.L:   cmp   [K], byte 0     ; Сравнение K с нулем
      jnz   .L              ; Если не нуль
```

Для обычной последовательной программы записанные три оператора означают бесконечный цикл. Но дело в том, что наша программа является не последовательной, а па-

раллельной. В то время, как прикладная программа выполняет данный цикл, независимо от нее (асинхронно) обработчик прерываний реагирует на нажатие (и отпускание) каждой клавиши. Поэтому для того, чтобы ввод символов с клавиатуры был ограничен по числу символов, необходимо, чтобы ввод очередного символа уменьшал содержимое счетчика *K*. Для этого достаточно, например, временно добавить в процедуру драйвера экрана «Вывод символа» оператор:

```
dec     byte [K]
```

Так как процедура «Вывод символа» выполняется при выводе «эха» каждого символа, то требуемая цель ограничения ввода будет достигнута. После завершения ввода прикладная программа выполняет деинициализацию экрана и клавиатуры.

Этап 2. Окончательная отладка драйвера клавиатуры. Для этого изменяется прикладная программа, использовавшаяся на шаге 1: после завершения инициализации экрана и клавиатуры, а также вывода приглашения прикладная программа выполняет ввод с клавиатуры (с помощью процедуры «Ввод символа») строки символов в свой прикладной буфер. Ввод в прикладной буфер производится до тех пор, пока прикладная программа не получит символ с кодом *24h*. Далее прикладная программа выводит содержимое своего прикладного буфера на экран, а затем восстанавливает системный обработчик прерываний клавиатуры и делается возврат в DOS.

☑ **Примечания:**

1. Разработанные программы не должны содержать системных вызовов DOS и BIOS, выполняющих информационный обмен с клавиатурой и экраном.
2. Перед выполнением деинициализации экрана и клавиатуры, в прикладной программе следует выполнить задержку, используя свой драйвер клавиатуры.

22. Простейший начальный загрузчик

22.1. Общая схема начальной загрузки

Начальная загрузка программ операционной системы в ОП с устройства внешней памяти необходима в начале сеанса работы с ВС, когда ее оперативная память еще пуста, а также при серьезных сбоях в системе, приводящих к порче загруженных ранее программ ОС. При загрузке автономных программ ОС вообще отсутствует, и поэтому такая загрузка также является начальной.

До настоящего времени наши управляющие программы (драйверы) сосуществовали в ОП совместно с программами инструментальной ОС – DOS. Необходимость такого сосуществования приводила к тому, что программы вынуждены были располагаться в области прикладных программ. Рано или поздно у вас могла появиться мысль создать автономную программу (или группу программ), размещаемую (размещаемых) в любой области ОП. Реализация подобной мысли возможна лишь при наличии выполнения операции начальной загрузки.

Обычно в ВС реализуется следующая схема начальной загрузки (рис. 3.13). При включении ЭВМ в сеть или при нажатии клавиши <Reset> инициируется модуль «Аппаратный загрузчик», который выполняет подготовку к выполнению программы POST BIOS, входящей в состав BIOS, а затем передает ей управление. POST BIOS, во-первых, выполняет комплексное тестирование ЭВМ (проверяет объем и целостность ОП, определяет тип ЦП, тестирует ПУ и т. д.).

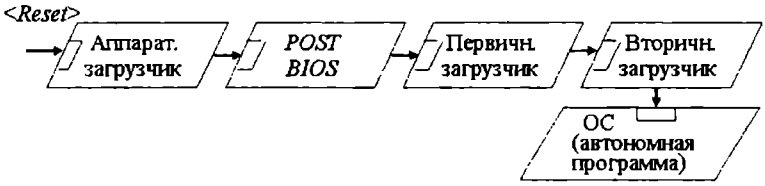

Рис. 3.13. Общая схема начальной загрузки

Во-вторых, с диска читается ***первичный начальный загрузчик*** и помещается в ОП по адресу *7C00h*. Для того чтобы выполнить данное считывание, POST BIOS обращается к первому по порядку диску и считывает с него логический сектор 0. (Порядок дисков может быть изменен во время настройки POST BIOS пользователем.) Если данный сектор не содержит первичный начальный загрузчик, то читается сектор 0 следующего по порядку диска. Если ни на одном из заданных дисков первичный загрузчик не найден, то из ПЗУ считывается Бейсик и ему передается управление. Если же первичный начальный загрузчик найден, то управление передается ему.

Так как размер сектора диска невелик (всего 512 байт), программный начальный загрузчик реализуется в виде двух модулей. Первичный начальный загрузчик, расположенный в секторе 0 диска, загружает информацию, расположенную в фиксированной области диска, а затем инициирует вторичный начальный загрузчик, входящий в состав этой информации. ***Вторичный загрузчик*** выполняет поиск в файловой структуре системы требуемых программ, а затем передает управление одной из программ, загруженных им или первичным загрузчиком. Прежде, чем передать это управление, вторичный загрузчик, возможно, переключает ЦП из реального режима в требуемый режим.

Например, при начальной загрузке операционной системы FreeDOS первичный загрузчик считывает в ОП фиксированную область, содержащую системный загрузочный модуль *Kernel.sys*. В состав этого файла входит вторичный начальный загрузчик, который после своего инициирования первичным начальным загрузчиком выполняет считывание в ОП недостающих модулей ОС, используя файловую структуру системного диска. Далее один из этих модулей – модуль настройки, считав файл *Fdconfig.sys*, в котором содержатся системные параметры, заданные пользователем (максимальное число одновременно открытых файлов, название и параметры интерпретатора команд и т. д.), выполняет настройку ОС. После этого в ОП загружается интерпретатор команд (по умолчанию это файл *Command.com*), которому передается управление.

Следует заметить, что современные первичные загрузчики обычно выполняют часть функций, отводимых выше вторичному загрузчику, а именно – поиск загружаемого файла по его имени в файловой структуре системы. В качестве загружаемого файла может выступать вторичный загрузчик, или небольшая автономная программа.

22.2. Программирование дисковода

Первичный начальный загрузчик должен считывать с диска загрузочный файл вторичного начального загрузчика. Для выполнения информационного обмена с диском можно использовать два подхода: 1) управление дисководом на уровне портов; 2) использование системных вызовов BIOS. Недостатками первого подхода являются не только трудоемкость программирования, но и сравнительно большие затраты дисковой памяти для размещения первичного начального загрузчика. Поэтому для того, чтобы

реализовать первичный загрузчик, выполняющий управление дисководом на уровне портов, требуется существенная оптимизация программы загрузчика относительно объема занимаемой памяти. Особенно это относится к созданию первичного загрузчика, предназначенного для загрузки с диска большой емкости.

Системные вызовы BIOS, основанные на команде «*int 13h*», являются универсальными, позволяя работать с любыми дисководами, с которых возможна начальная загрузка, в том числе, с дисководами для гибких дисков (дискет). Именно для такого дисковода мы будем разрабатывать свой первичный загрузчик.

Экономия памяти при использовании вызовов BIOS позволяет не только разместить без особого труда первичный загрузчик в одном секторе диска, но и возложить на него некоторые функции, выполняемые вторичным загрузчиком. А именно – поиск загружаемой программы в файловой структуре системы, пользуясь именем файла. При работе с дисководом на уровне портов такая функция в принципе нерелиазуема, так как приходится экономить каждый байт даже при реализации загрузки фиксированной области диска. Следствием такого расширения функций первичного загрузчика является то, что при загрузке небольших автономных программ, работающих исключительно в реальном режиме процессора, надобность во вторичном загрузчике отпадает. При этом всю загрузку автономной программы выполняет сам первичный загрузчик.

Системные вызовы BIOS для работы с дисками «*int 13h*» являются устаревшими, так как, во-первых, они не позволяют работать со всем объемом информации, находящейся на современном диске большой емкости. При программировании первичного загрузчика этот недостаток не является существенным, если загружаемая автономная программа (например, ОС) находится не очень далеко от начала диска.

Во-вторых, при использовании в своей программе указанных системных вызовов, приходится учитывать некоторые физические особенности того диска, с которым производится работа. Подобные физические особенности принято называть ***геометрией диска***. Достаточно типичная геометрия диска приведена на рис. 3.14.

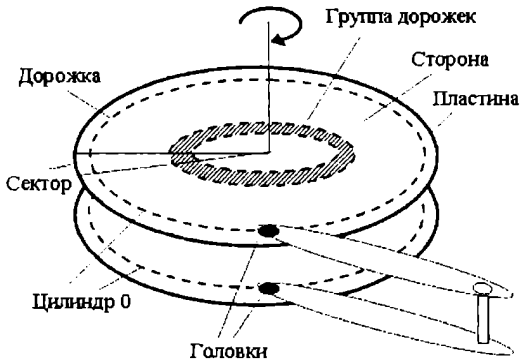

Рис. 3.14. Типичная геометрия магнитного диска

Магнитный диск состоит из одной, двух или большего числа металлических или стеклянных пластин, вращающихся вокруг общей оси. Одна или обе поверхности каждой пластины покрыты тонким однородным слоем магнитного материала. Подобная поверхность пластины условно разделена на тонкие кольца, называемые ***дорожками***. Все дорожки на каждой пластине пронумерованы, причем дорожка 0 находится около внешнего края пластины. Все дорожки диска, имеющие одинаковый номер, принадлежат од-

ному ***цилиндру***, номер которого определяется номерами входящих в него дорожек. Например, цилиндр 0 объединяет дорожки с номерами 0 (см. рис. 3.14).

Для чтения (записи) информации с пластины (на пластину) используется один или несколько элементов, называемых ***головками***. Количество головок зависит от используемого варианта конструкции дисковода. В первом из этих вариантов каждую поверхность пластины обслуживает единственная головка, которая может перемещаться по радиусу диска дискретными шагами. При этом каждый шаг соответствует одной дорожке пластины. После того, как головка «зависнет» над требуемой пластиной, она может выполнять информационный обмен с этой дорожкой. Так как все головки диска связаны жестко в единую «гребенку», то при установке одной из головок на требуемую дорожку все остальные головки «зависнут» над дорожками этого же цилиндра. Следствием этого является то, что переход между дорожками одного цилиндра не требует каких либо механических перемещений головок. Для такого перехода достаточно лишь выполнить электронное переключение головок. Другой вариант конструкции дисковода предполагает, что каждую дорожку поверхности пластины обслуживает своя отдельная головка. В этом случае никакого перемещения головок не производится, так как для выбора требуемой дорожки достаточно лишь электронно включить требуемую головку.

Что касается информационного обмена с дорожкой, то он производится следующим образом. Во-первых, заметим, что магнитная поверхность дорожки состоит из маленьких фрагментов, называемых ***магнитными доменами***. Во время движения дорожки мимо головки (вследствии вращения диска) магнитное поле головки может повлиять на полярность того домена, который находится в данный момент времени рядом с головкой. Благодаря этому магнитный домен может использоваться для записи одного бита информации. При этом одна полярность домена кодирует 0, а другая – 1. Во время чтения информации с диска, наоборот, полярность текущего домена влияет на магнитное поле головки. Это состояние магнитного поля считывается в электронную цепь и распознается ею как 0 или 1. Несмотря на то, что длины дорожек на пластине разные, все они содержат одинаковое количество доменов, кодирующих биты.

Поверхность пластины, как и любой круг, можно разделить на угловые секторы (см. рис. 3.14). Возьмем размер углового сектора таким, чтобы он «вырезал» из каждой дорожки на поверхности пластины дугу, содержащую 512 бит. Эта дуга дорожки называется ***сектором***. Один угловой сектор «вырезает» секторы одинаковой длины (измерение в битах) не только на одной поверхности одной пластины, но и на всех поверхностях всех пластин. Разбиение дорожки диска на секторы производится во время ***низкоуровневого форматирования диска***. Во время этого форматирования начало каждого сектора помечается специальной последовательностью битов. Кроме того, для каждого сектора записывается его номер.

С точки зрения драйвера дисковода сектор является минимальной единицей информационного обмена между диском и ОП. Для выполнения такого обмена каждый сектор задается своим адресом:

(номер цилиндра; номер поверхности; номер сектора).

В программе, при использовании системных вызовов BIOS «*int 13h*», также приходится пользоваться такими же «физическими» адресами секторов. Драйверы дисководов, входящие в состав ОС, а так же драйверы современных BIOS, позволяют использовать в программе линейные адреса секторов. Каждый такой адрес представляет собой номер сектора на диске, начиная с нуля. Например, драйверы дисководов, входящие в

состав DOS, предоставляют развитый программный интерфейс для работы с дисками, минуя файловую систему. Сюда относятся не только линейная адресация секторов диска, но и огромный объем адресуемого дискового пространства. Единственная мелочь – в момент начальной загрузки никакой DOS в ОП еще нет. Поэтому переходим к рассмотрению системных вызовов «*int 13h*».

В случае ошибки любой вызов «*int 13h*» устанавливает в единицу флаг *CF*, а в регистре *AH* возвращает код ошибки.

Рассмотрим 3 наиболее интересных функции прерывания *13h*.

1. ***int 13h*** (функция ***00h***) – сброс состояния дисковой системы. Данный системный вызов требуется выполнить перед началом работы с дисковой подсистемой, а также в случае любой ошибки работы с дисковой подсистемой.

Вход: *AH* – 00;

DL – номер дисковода. При установленном бите 7 в регистре *DL* происходит сброс всех жёстких и гибких дисков.

Выход: *CF* – признак результата (0 – успешно, 1 – ошибка);

AH – состояние дисковой подсистемы. При успешном завершении системного вызова (*CF*=0) *AH* содержит 0.

2. ***int 13h*** (функция ***02h***) – чтение сектора (секторов) диска. Данный системный вызов считывает в заданную область ОП содержимое одного или нескольких секторов диска.

Вход: *AH* – 02;

AL – число считываемых секторов (*AL* > 0);

DL – номер дисковода;

CH – номер цилиндра (*CH* ≥0);

DH – номер поверхности (*DH* ≥0);

CL – номер начального сектора на дорожке (*CL*≥1);

ES:BX – указатель на буфер, в который производится считывание.

Выход: *CF* – признак результата (0 – успешно, 1 – ошибка);

AH – состояние дисковой подсистемы. При успешном завершении системного вызова (*CF*=0) AH содержит 0;

AL – число считанных секторов.

3. ***int 13h*** (функция ***03h***) – запись сектора (секторов) диска. Данный системный вызов записывает на диск из заданной области ОП содержимое одного или нескольких секторов.

Вход: *AH* – 03;

AL – число записываемых секторов (*AL* > 0);

DL – номер дисковода;

CH – номер цилиндра;

DH – номер головки (номер поверхности);

CL – номер начального сектора;

ES:BX – указатель на буфер, из которого производится запись на диск.

Выход: *CF* – признак результата (0 – успешно, 1 – ошибка);

AH – состояние дисковой подсистемы. При успешном завершении системного вызова (*CF=0*) *AH* содержит 0;

AL – число записанных секторов.

22.3. Укрупненная структура первичного загрузчика

Первичный начальный загрузчик занимает первый сектор логического диска длиной 512 байт и имеет следующую укрупненную структуру:

```
        org     7c00h                   ; Смещение относительно начала сегмента ОП
        jmp     near  Begin             ; Переход на начало кода
;   Описательный блок
B_1   times   8     db  0               ; Название ОС
B_2   dw      0                         ; Размер сектора (байт)
        .  .  .  .  .  .  .  .  .
;   Другие данные программы
        .  .  .  .  .  .  .  .  .
Begin:                                  ; Начало кода
        .  .  .  .  .  .  .  .  .       ; Команды загрузчика
        .  .  .  .  .  .  .  .  .
; Заполнение оставшегося места в секторе нулями с целью точного
;   размещения загрузочной сигнатуры
        times  510-($-$$)     db  0
; Загрузочный сектор заканчивается загрузочной сигнатурой
boot_signature:  dw  0aa55h
```

По относительному адресу *7c00h* находится 3-байтовая команда перехода *jmp*, которая «перескакивает» через описательный блок загрузочного сектора на исполнительные команды (код) загрузчика. ***Описательный блок загрузочного сектора*** имеет длину 59 байт и содержит информацию об используемой файловой системе и о размещении информации на диске. Содержание описательного блока для файловых систем *FAT12* и *FAT16* приведено в табл. 3.5.

Таблица 3.5. Структура описательного блока

Назначение	*Размерность*
1. Название ОС (текстовая строка)	8 байт
2. Размер сектора в байтах (всегда 512)	слово
3. Количество секторов в кластере	байт
4. Число резервных секторов	слово
5. Количество копий таблицы *FAT* (всегда 2)	байт
6. Количество записей в корневом каталоге	слово
7. Общее количество секторов	слово
8. Тип носителя (*F0h* для гибкого диска)	байт
9. Число секторов, занимаемых одной *FAT*-таблицей	слово
10. Число секторов на дорожке	слово
11. Число головок (поверхностей) на один цилиндр	слово
12. Количество скрытых секторов (для дискеты всегда 0)	двойное слово
13. Всего секторов	двойное слово
14. Номер дисковода	байт
15. Зарезервировано	байт
16. Если *29h*, то следующие три поля имеют смысл	байт
17. Номер логического диска	двойное слово
18. Метка тома (текстовая строка)	11 байт
19. Идентификатор ФС (текстовая строка)	8 байт

Первоначальная запись в описательный блок производится транслятором в результате обычной инициализации данных. После того, как первичный загрузчик будет переписан программой POST BIOS с диска в ОП, прежде чем передать ему управление, POST BIOS заменяет содержимое большинства полей в описательном блоке, за исключением текстовых строк. При этом особый интерес представляет запись в поле 14 (номер дисковода). Некоторые BIOS записывают в это поле сам номер дисковода. Другие BIOS помещают в это поле число *FFh*, а номер дисковода передают на вход первичного загрузчика в регистре *DL*.

Последние два байта загрузочного сектора содержат загрузочную сигнатуру (слово) – *AA55h* (в младшем байте – *55h*, в старшем байте – *AAh*). Если POST BIOS обнаружит это слово в конце первого сектора логического диска, то он будет считать этот диск загрузочным.

Рассмотрим записанный перед загрузочной сигнатурой псевдооператор

```
times  510-($-$$)    db  0
```

Его наличие обеспечивает точное размещение загрузочной сигнатуры в двух последних байтах загрузочного сектора. Данный оператор «вычисляет» расстояние между кодом загрузчика и предпоследним байтом загрузочного сектора и заполняет это пространство памяти нулями. В самом деле, *$* – смещение текущей команды относительно начала сегмента ОП. *$$* – тоже смещение, но начала кода программы, т. е. смещение, заданное в операторе *org*. Таким образом, (*$-$$*) есть размер программы (в байтах) до текущей позиции.

22.4. Программа начального загрузчика

Далее рассматривается простейший начальный загрузчик, выполняющий повторную загрузку самого себя, но с другого адреса ОП – *700h*. Выбор данного начального адреса обусловлен тем, что согласно п. 18.1 первые *400h* байтов ОП используются для размещения таблицы векторов прерываний. Последующие *200h* байтов памяти используются драйверами BIOS для размещения своих буферов. Загрузив свою копию в ОП, первичный начальный загрузчик передает управление тому фрагменту программы-копии, который выводит на экран короткое сообщение «123».

Распределение начального участка ОП, используемое первой версией нашего начального загрузчика, приведено на рис. 3.15.

Из этого рисунка видно, что для стека, обслуживающего первичный загрузчик, отведено место, начиная с адреса *07C00*. Это нижняя граница области в 512 байт, занимаемой самим первичным загрузчиком. При этом стек, как всегда, растет в сторону меньших адресов. Этот же самый стек может обслуживать и автономную программу. Но ничто не мешает автономной программе, после того, как начальный загрузчик передаст ей управление самой разместить свой стек в памяти. Более того, автономная программа может использовать область памяти, занимаемую первоначально начальным загрузчиком, а также может занять любые области памяти с большими адресами. Напомним, что автономная программа – это или программа, выполняемая без помощи ОС, или сама ОС.

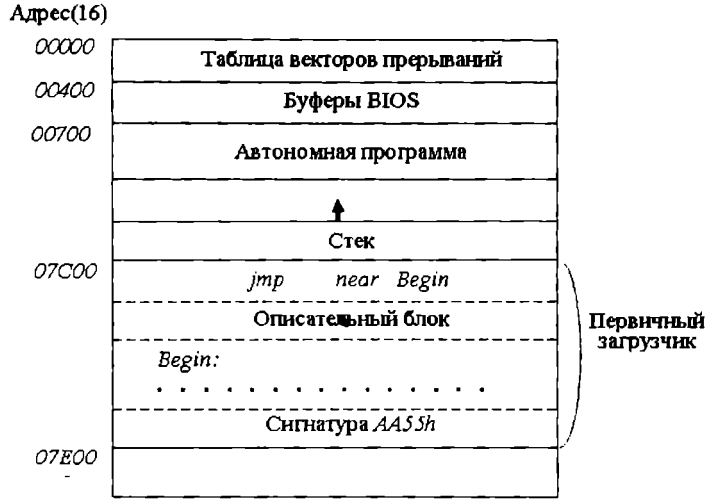

Рис. 3.15. Распределение начального участка ОП

Несмотря на небольшие размеры программы простейшего начального загрузчика, ее разработка требует знания некоторых дополнительных деталей. Поэтому данная программа приводится далее полностью (за исключением процедуры считывания сектора диска, подлежащей самостоятельному написанию).

```
;   Первичный начальный загрузчик
;   ------------------------------
; Загружается по адресу 7c00h, а затем загружает себя повторно
;   по адресу 700h
;
        org   7c00h             ; Смещение относительно начала сегмента ОП
        jmp   near   Begin      ; Переход на начало кода
;   Описательный блок
B_1     db    '1STLDR '         ; Длина поля 8 байт
B_2     dw    200h
B_3     db    1
B_4     dw    1
B_5     db    2
B_6     dw    0e0h
B_7     dw    0b40h
B_8     db    0f0h
B_9     dw    9
B_10    dw    12h               ; Число секторов на дорожке
B_11    dw    2                 ; Число поверхностей на цилиндр
B_12    dd    0
B_13    dd    0
B_14    db    0                 ; Номер дисковода
B_15    db    0
```

```
B_16    db      29h
B_17    dd      12344321h
B_18    db      'FIRSTLOADER'     ; Длина поля 11 байт
B_19    db      'FAT12   '        ; Длина поля 8 байт
Begin:
; «Загрузка» виртуальных сегментов
        mov     ax, cs
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, 7bfeh         ; Определение стека
;  Проверка номера дисковода
        cmp     [B_14], byte 0ffh
        jne     M1
        mov     [B_14], dl
M1:
;  Размещение копии загрузчика по адресу 700h
        xor     ax, ax            ; 0 – логический номер сектора диска
        mov     bx, 700h          ; 700h – адрес в ОП
        call    Sec_read          ; Чтение сектора диска
        jnc     M2                ; Если нет ошибки
        mov     si, Msg1          ; SI ← адрес сообщения
        call    Outstring         ; Вывод сообщения об ошибке
        jmp     $                 ; Бесконечный цикл
M2:
; Отключение двигателя дисковода
        mov     dx, 3F2h          ; DX ← номер порта
        xor     al, al            ; AL ← 0
        out     dx, al            ; Вывод команды в порт
; Переход на загруженную программу
        mov     bx, M3            ; BX ← адрес перехода  в оригинале загрузчика
        sub     bx, 7c00h         ; BX ← адрес относительно начала загрузчика
        add     bx, 700h          ; BX ← адрес перехода
        jmp     near  bx          ; Переход на копию загрузчика
M3:
;   Выполняемый фрагмент копии загрузчика
        mov     ah, 0eh           ; AH ← функция вывода символа
        xor     bh, bh            ; Нулевая видеостраница
        mov     al, '1'           ; AL ← код символа
        int     10h               ; Вывод символа
        mov     al, '2'
        int     10h
        mov     al,'3'
        int     10h
        jmp     $                 ; Бесконечный цикл
;
;    Чтение сектора диска
```

```
;   ----------------------
; Входы: AX – номер сектора диска (AX >=0)
;    ES:BX – начальный адрес (сегмент:смещение) буфера
; Выход: CF – признак результата (0 – успешно, 1 – ошибка)
;
Sec_read:
        . . . . . . . .          . . .
        ret                              ; Возврат из Sec_read
;
;    Вывод на экран строки, заканчивающейся нулем
;    ---------------------------------------------
; Вход: SI – адрес-смещение первого байта строки
;
Outstring:
        push  ax
        push  bx
        push  si
.Next:  lodsb                            ; AL ← si-й байт, SI ← SI+1
        or    al, al                     ; (AL)=0 ?
        jz    .Exit                      ; Если да
        mov   ah, 0eh                    ; Функция вывода символа
        xor   bh, bh                     ; Нулевая видеостраница
        int   10h                        ; Вывод символа
        jmp   .Next                      ; За следующим символом
.Exit:  pop   si
        pop   bx
        pop   ax
        ret                              ; Возврат из Outstring
Msg1    db    'Error loading data',0     ; Сообщение об ошибке загрузки
; Заполнение оставшегося места в секторе нулями с целью точного
;    размещения загрузочной сигнатуры
        times  510-($-$$)    db  0
; Загрузочный сектор заканчивается загрузочной сигнатурой
boot_signature:  dw  0aa55h
```

В приведенной программе используются две процедуры. Первая из них имеет имя *Sec_read* и выполняет чтение заданного сектора диска в буфер ОП. Данная процедура имеет три входных параметра: *AX* содержит логический номер считываемого сектора диска ($AX \geq 0$), *ES:BX* – начальный адрес (сегмент:смещение) буфера. Единственный выходной параметр процедуры: *CF* – признак результата (0 – успешно, 1 – ошибка). В основе данной процедуры лежит ранее рассмотренный системный вызов BIOS «*int 13h*» (функция *2h*), выполняющий чтение сектора диска.

Вторая процедура имеет имя *Outstring* и выполняет вывод на экран строки, заканчивающейся нулем. Единственный входной параметр процедуры: *SI* – адрес-смещение первого байта строки. В основе данной процедуры лежит системный вызов BIOS «***int 10h***» (функция ***0eh***), выполняющий вывод символа на экран. Входные параметры данного вызова: *AL* – код ASCII выводимого символа; *BH* – номер видеостраницы (мы используем только нулевую видеостраницу). Кроме того, данная процедура содержит новый

исполнительный оператор – ***lodsb*** (чтение байта из строки). Данный оператор копирует байт ОП по адресу *DS:SI* в регистр *AL* и увеличивает содержимое регистра *SI* на 1. Соответствующая машинная команда однобайтовая, что немаловажно при написании таких «экономных» программ как начальный загрузчик.

Записанный в начале программы загрузчика псевдооператор «*org 7c00h*» влияет на настройку некоторых адресов-смещений. Например, таким настраиваемым адресом является адрес, заменяющий метку *Msg1* в операторе «*mov si,Msg1*». Транслятор записывает величину этого адреса в предположении, что программа начального загрузчика будет размещена в сегменте ОП со смещением *7c00h*. С учетом этого подобный оператор вывода на экран нельзя использовать в копии начального загрузчика, так как эта копия загружается по другому адресу – *700h*. Поэтому для вывода на экран в исполняемом фрагменте копии начального загрузчика используется не процедура *Outstring*, требующая задания в качестве параметра начального адреса выводимой строки, а выполняется посимвольный вывод на экран, непосредственно используя вызов «*int 10h*» (функция *0eh*).

Еще одна особенность приведенной выше программы – выключение двигателя дисковода непосредственно перед тем, как передать управление на загруженную копию начального загрузчика. Это выключение производится путем выдачи команды в порт с адресом *3F2h*. Если в дальнейшем загруженная автономная программа будет выполнять информационный обмен с дисководом, то включение двигателя дисковода выполнит используемый программой системный вызов BIOS «*int 13h*». В нашем случае загруженная в качестве автономной программы копия начального загрузчика никаких дисковых операций не делает, а лишь выводит короткое сообщение на экран.

В приведенной программе транслятор выполняет инициализацию полей описательного блока конкретными данными. На самом деле, такая инициализация требуется лишь для текстовых полей. Все остальные поля можно заполнить нулями, так как BIOS все равно заполнит их требуемыми данными. Это нетрудно проверить, выполнив в приведенной выше программе нулевую инициализацию полей описательного блока, а также заменив запуск копии начального загрузчика на операторы вывода на экран содержимого требуемых полей описательного блока. В качестве таких операторов удобно использовать операторы вызова ранее разработанных процедур вывода на экран содержимого байта и содержимого слова. Так как исходные тексты этих процедур находятся в файле *Video_io.asm*, то этот файл следует временно подсоединить к исходному файлу начального загрузчика, используя псевдооператор *%include*. Следует добавить, что данный оператор следует поместить в программу где-то до оператора «*times 510-($-$$) db 0*». Иначе длина начального загрузчика превысит 512 байт на величину оттранслированной величины подсоединенного файла.

Вообще может показаться странным, что загружаемая копия начального загрузчика не только не делает ничего полезного, но и то, что она делает, уже содержится в программе начального загрузчика. На самом деле, решена главная промежуточная задача – начальный загрузчик считывает в нужное место ОП нужный сектор диска. В следующем разделе мы добьемся того, чтобы считываемая информация действительно была полезной.

22.5. Подготовка начального загрузчика к выполнению

На первом этапе подготовки начального загрузчика к выполнению необходимо преобразовать исходный *asm*-файл в бинарный файл. Термин ***бинарный файл*** означает, что данный файл содержит полностью готовую машинную программу, не требующую какой-либо настройки. После того, как бинарный файл будет загружен в ОП, соответствующая программа начнет выполняться (если на нее будет передано управление). При-

мером бинарного файла является хорошо знакомый нам *com*-файл. Данный файл подготовлен транслятором в предположении, что загрузка *com*-файла производится в сегмент ОП со смещением *100h* относительно начала сегмента. Напомним, что начальное смещение для нашего бинарного файла – *7c00h*.

Например, пусть исходный файл начального загрузчика имеет имя *Boot_1.asm*, тогда для получения соответствующего бинарного файла может быть использована команда DOS:

```
nasm  Boot_1.asm .
```

Обратим внимание, что в данном примере результирующий бинарный файл не имеет никакого расширения имени, в том числе, и расширения *com*.

После того, как первичный загрузчик получен в виде бинарного файла, этот файл должен быть записан в качестве самого первого сектора загрузочного диска. С целью уменьшить зависимость от наличия реальной аппаратуры, а также для устранения нежелательных последствий для имеющейся в системе ОС, подготовка начального загрузчика к выполнению, а также само это выполнение (загрузка автономных программ), будут производиться на виртуальной машине (ВМ). Из существующих программ, позволяющих создавать и поддерживать ВМ. в настоящем пособии используется программа *Virtual PC*. Краткие сведения об этой программе, необходимые для нашей дальнейшей работы, приведены в приложении 2. Прежде, чем перейти к использованию ВМ, подчеркнем, что работать с ВМ также интересно, как и с реальной машиной, но во много раз проще и безопаснее. После того, как первичный загрузчик и автономная программа отлажены с помощью ВМ, они могут быть без всякого труда перенесены на реальную машину.

Допустим, что ранее создана ВМ, имеющая в качестве операционной системы какую-то DOS (см. приложение 2). В этом случае возможный перечень действий по подготовке первичного загрузчика к выполнению на ВМ:

- создание виртуальной дискеты;
- «вставка» виртуальной дискеты в виртуальный дисковод *a:* ;
- форматирование виртуальной дискеты;
- запись на виртуальную дискету начального загрузчика;
- изменение приоритета начальной загрузки;
- перезапуск ВМ.

Ниже приводится подробное описание перечисленных действий.

1. Создание виртуальной дискеты.

1.1. В консольном окне *Virtual PC* (левый верхний угол экрана) выбрать требуемую ВМ (оснащенную DOS), но не запускать ее;

1.2. В главном меню консоли выбрать «*File*», а затем:

«*Virtual Disk Wizard*» → «*Create a new virtual disk*» → «*A virtual floppy disk*».

1.3. В появившемся окне следует набрать имя файла, который будет играть роль виртуальной дискеты. Расширение имени файла – *vfd*. Пример имени: *1.vfd*.

2. «Вставка» виртуальной дискеты в виртуальный дисковод.

2.1. Запустить требуемую ВМ.

2.2. Установить курсор мыши на изображении дискеты в левом нижнем углу запущенной ВМ и нажать правую клавишу мыши.

2.3. Выбрать «*Capture Floppy Disk Image…*».

2.4. В появившемся окне выбрать имя файла с расширением *vfd*, выполняющего роль дискеты и «нажать» «открыть».

3. Форматирование виртуальной дискеты. На запущенной ранее ВМ следует выполнить команду DOS – «*format a:*». На все последующие вопросы программы форматирования достаточно нажимать <Enter>.

4. Запись на виртуальную дискету начального загрузчика.

4.1. Поместить в какой-нибудь каталог (папку) физической машины два файла: 1) бинарный файл первичного загрузчика; 2) специальную утилиту DOS, предназначенную для записи требуемого бинарного файла на первый сектор диска. Например, для этого пригодна утилита *bootable.exe*.

4.2. Запустив ВМ с DOS, требуется сделать «видимой» каталог (папку) из предыдущего пункта (используя изображение папки в левом нижнем углу ВМ). В результате этого последующая работа с требуемым реальным каталогом будет выполняться с помощью выбранного виртуального логического диска, например, диска *z:*.

4.3. Выполнить на ВМ команду DOS для запуска утилиты, выполняющей запись бинарного файла первичного загрузчика на первый сектор гибкого диска. Пример такой команды:

```
z:\bootable.exe имя_бинарного_файла a: .
```

5. Изменение приоритета начальной загрузки. Цель данного шага – скорректировать порядок загрузочных дисков, используемый виртуальным POST BIOS, так, чтобы загрузочный гибкий диск (дискета) был первым.

5.1. Сразу после начала запуска ВМ нажать клавишу <Delete>. Результатом этого действия является приостановка запуска ВМ и вывод на экран окна BIOS.

5.2. С помощью клавиш <→> и <←> выбрать в горизонтальном меню пункт «*Boot*».

5.3. С помощью клавиш <↑> и <↓> выбрать в вертикальном меню «*Floppy Drive*» и нажимая клавишу <+> перевести строку «*Floppy Drive*» на первое (верхнее) место.

5.4. Закрепить результат изменения приоритета, нажав <F10>.

Заметим, что в процессе отладки первичного загрузчика приходится многократно менять порядок загрузочных дисков, используемый POST BIOS. В то время как наш загрузчик находится на дискете (строка «*Floppy Drive*»), инструментальная DOS находится на винчестере (строка «*Hard Drive*»).

6. Перезапуск ВМ производится обычным образом – с помощью «клавиши» «*Start*», или такой перезапуск делает сам POST BIOS (после нажатия <F10>).

В результате выполнения перечисленных действий первичная загрузка ВМ будет производиться с виртуальной загрузочной дискеты. Находящийся на ней первичный загрузчик сначала загрузит, а затем передаст управление или вторичному начальному загрузчику, или непосредственно автономной программе. Напомним, что пока в качестве загружаемой программы выступает копия самого первичного загрузчика.

Лабораторная работа 20

Требуется дописать и отладить простейший начальный загрузчик, выполняющий загрузку с виртуальной дискеты своей копии. Загруженная копия загрузчика должна вывести на экран вашу фамилию, записанную латинскими буквами.

☑ **Примечание**. При отладке программы часто возникает вопрос о том, какие ее участки выполняются при конкретном прогоне, а какие – не выполняются. Для ответа на данный вопрос в текст программы временно помещаются операторы, выполняющие вывод на экран. В качестве такого оператора, например, удобно использовать макрооператор *Slovo*, выводящий на экран слово, заданное в качестве параметра макрооператора. Соответствующее макроопределение:

```
%macro Slovo  1
;    Вывод слова на экран
;    --------------------
; Вход: %1 – слово, выводимое на экран (заключено в кавычки)
;
        push    si
        jmp     .L
.Msg    db      %1, 0
.L:     mov     si, .Msg
        call    Outstring
        pop     si
    %endmacro   Slovo
```

Пример записи макрооператора: *Slovo 'Proba_1'*. Данная запись может присутствовать в качестве любой строки исходной программы. Использование в приведенном выше теле макрооператора локальных меток приводит к ограничению: между двумя обычными (нелокальными) метками исходной программы не может быть записано более одного подобного макрооператора. Применение макрооператора *Slovo* будет еще более полезным на последующих лабораторных работах.

23. Начальная загрузка с использованием имени файла

На последней лабораторной работе мы создали первую версию первичного начального загрузчика, выполняющего загрузку с дискеты собственной копии. Выбор загружаемой программы обусловлен тем, что кроме первичного загрузчика, никакие неслужебные данные диска не привязаны к определенному участку дискового пространства. Поэтому реальные первичные загрузчики всегда выполняют загрузку вторичного загрузчика или загрузку автономной программы, используя не номер сектора диска, а используя имя соответствующего бинарного файла. Это же самое мы поручим второй версии нашего начального загрузчика, учтя при этом, что существенная часть начального сектора диска в 512 байт осталась у нас ранее свободной. Причем для упрощения задачи будем считать, что имя загружаемого бинарного файла находится в корневом каталоге того же диска, на котором находится начальный загрузчик.

23.1. Общая структура информации на диске

Как и ранее, далее речь будет идти о гибком диске (дискете) емкостью 1,44 Мбайт. Такой диск имеет две поверхности, 80 дорожек (цилиндров), 18 секторов по 512 байт на дорожке. Нетрудно проверить, что

512 × 2 × 80 × 18 = 512 байт × 2880 секторов = 1474560 байт = 1,44 Мбайт.

Кроме того, будем считать, что в результате форматирования дискета подготовлена для размещения на ней файловой системы *FAT12*. Выбор данной ФС обусловлен тем, что она используется в инструментальной операционной системе DOS для размещения информации на дискетах.

На рис. 3.16 приведено распределение пространства дискеты емкостью 1,44 Мбайт при использовании файловой системы *FAT12*.

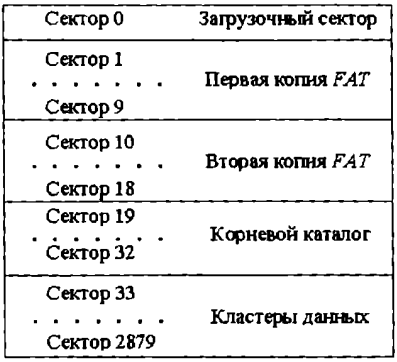

Рис. 3.16. Размещение информации на диске

Из этого рисунка видно, что каждая из двух копий таблицы *FAT* занимает 9 секторов. Корневой каталог требует для своего размещмния 14 секторов. Для размещения остальных каталогов и файлов данных остается 2846 секторов дискового пространства.

23.2. Структура каталога

Любой каталог, в том числе и корневой, состоит из 32-байтовых записей, каждая из которых соответствует одному дочернему файлу или каталогу. Первые две записи обычного (некорневого) каталога содержат служебную информацию. У корневого каталога служебную информацию содержит только самая первая запись. Обычная (неслужебная) запись каталога имеет следующую структуру.

Байты 0–7 содержат собственно имя файла данных или имя подкаталога. Имя всегда записывается большими буквами. Если это имя имеет длину менее 8 символов, то оно дополняется справа пробелами (код пробела – *20h*). Если элемент (запись) каталога не соответствует ни файлу данных, ни подкаталогу, то байт 0 содержит:

E5h – означает, что данный элемент каталога не используется, так как соответствует удаленному файлу. Теперь этот элемент может быть использован для создания нового файла или подкаталога;

00h – данный элемент свободен и является началом свободной области. После него в каталоге нет ни одного занятого элемента.

Байты 8–10 содержат расширение имени файла данных (записывается большими буквами). Если длина расширения менее трех символов, то оно дополняется справа пробелами.

Байт 11 описывает атрибуты файла (рис. 3.17). При этом под атрибутом понимается «специальное» свойство файла, кодируемое одним битом. Примером атрибута является «спрятанный» файл – файл, имя которого не выводится на экран при выполнении команды ОС вывода каталога (для DOS – команда *dir*). Так как разным атрибутам файла соответствуют и разные биты байта 11, то один и тот же файл может иметь несколько «специальных» свойств одновременно. Обычный файл данных не имеет «специальных» свойств и содержит *00h* в байте 11.

Байт 12 зарезервирован для Windows. В остальных случаях должен содержать *00h*.

Байт 13 уточняет время создания файла (содержит десятки миллисекунд). Содержимое поля: 0÷199.

Байты 14–15 содержит время создания файла.

Байты 16–17 содержат дату создания файла.

Байты 18–19 содержат дату последнего обращения к файлу для чтения или записи.

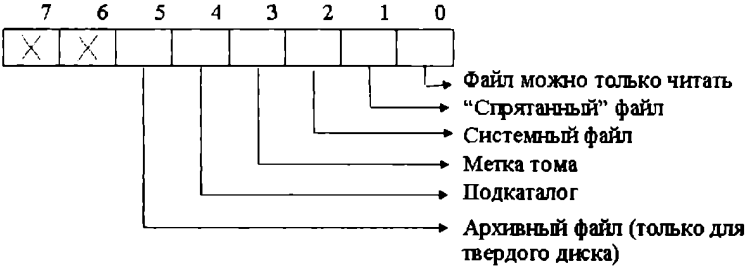

Рис. 3.17. Структура байта атрибутов элемента каталога

Байты 20–21 содержат старшее слово номера первого кластера файла (для *FAT12* всегда *0000h*).

Байты 22–23 содержат время последнего изменения файла.

Байты 24–25 содержат дату создания или последнего изменения файла.

Байты 26–27 содержат младшее слово номера начального кластера файла.

Байты 28–31 содержат размер файла в байтах. Для каталога это поле содержит *00000000h*.

Формат представления времени, используемый в байтах 14–15 и 22–23 приведен на рис. 3.18. Допустимое содержимое поля «Час» – 0÷23. Допустимое содержимое поля «Минута» – 0÷59. Поле «Секунда/2» содержит двухсекундный отсчет. Допустимое значение этого поля – 0÷29.

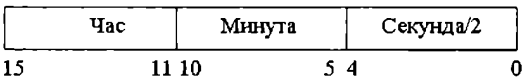

Рис. 3.18. Формат представления времени

Формат представления даты, используемый в байтах 16–17, 18–19, 24–25 приведен на рис. 3.19. При этом отсчет дат ведется с начала эпохи MS-DOS – 01.01.1980 г. Поле «Год» содержит номер года минус 1980. Допустимое содержимое этого поля – 0÷127. Допустимое содержимое поля «Месяц» – 1÷12. Допустимое содержимое поля «День» – 1÷31.

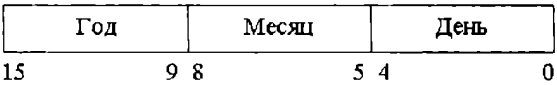

Рис. 3.19. Формат представления даты

Первые две записи в некорневом каталоге служебные. Запись 0 содержит указатель (байты 20–21, 26–27) на сам каталог, а поле «имя файла» (байты 0–7) этой записи содержит единственную точку: «.». Запись 1 содержит указатель на родительский каталог. Если родительский каталог является корневым, то поле указателя содержит ноль. Поле «имя файла» данной записи содержит две точки: «..». Единственная служебная запись корневого каталога почти вся пустая (содержит нули).

23.3. Структура таблицы FAT

Напомним, что применение таблицы *FAT* позволяет размещать файл в разрывном пространстве дисковой памяти. Каждый элемент этой таблицы соответствует одному

кластеру распределяемой части дискового пространства и содержит указатель (номер) на следующий элемент *FAT*-таблицы, соответствующий данному файлу. Так как номера элементов таблицы *FAT* и номера кластеров диска совпадают, то указатель на следующий элемент *FAT*-таблицы одновременно является указателем на следующий кластер данного файла. Поэтому, двигаясь по односвязанному списку, образованному из элементов *FAT*-таблицы, можно без труда найти все кластеры файла. Номер начального кластера файла одновременно является и номером первого элемента односвязанного списка. Напомним, что этот номер размещен в двух полях записи каталога (байты 26–27, 20–21), содержащих номер первого кластера файла. Для файловой системы *FAT12* используется только одно поле – байты 26–27. Другое поле (байты 20–21) содержит нули (*0000h*).

В файловой системе *FAT12* длина одного элемента *FAT*-таблицы – 12 бит (1,5 байта). Отсюда и название файловой системы. Последний элемент односвязанного списка, образованного из элементов *FAT*-таблицы, содержит пустой указатель – число *FFFh*. Если элемент *FAT*-таблицы соответствует свободному кластеру диска, то этот элемент содержит 0. Если элемент соответствует дефектному кластеру, то он содержит *FF7h*.

Нумерация распределяемых кластеров диска начинается с числа 2. Поэтому первые два элемента *FAT*-таблицы служебные. Первый элемент (элемент 0) всегда содержит сигнатуру – число *FF8h*. Второй элемент (элемент 1) содержит число *FFFh*. С учетом того, что первый распределяемый кластер диска имеет номер 2, а отводимое под файлы данных и каталоги дисковое пространство начинается с сектора номер 33 (см. рис. 3.16), следует, что кластер номер 2 начинается с сектора 33. Так как для рассматриваемой дискеты размер кластера равен одному сектору, то, следовательно, каждый распределяемый кластер есть сектор диска, для получения номера которого следует прибавить к номеру кластера число 31.

Например, пусть диск содержит только один файл с именем *test.asm*, занимающий кластеры 2, 3 и 4. Единственный непустой элемент корневого каталога содержит имя «*TEST*», расширение имени – «*ASM*», атрибуты обычного файла – *00h*, время и дату создания файла, номер начального кластера файла – *002h*, размер файла в байтах. Содержимое таблицы *FAT* приведено на рис. 3.20.

Номер элемента *FAT*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**...**
Содержимое элемента	*FF8*	*FFF*	003	004	*FFF*	000	000	...

Рис. 3.20. Пример таблицы *FAT*

В данном примере файл расположен в непрерывной области диска, занимая его секторы с 33-го по 35-й включительно, но благодаря применению *FAT* он мог бы успешно располагаться и в любой другой области, в том числе и разрывной.

23.4. Требования к загружаемой программе

Напомним, что полученная нами ранее версия первичного начального загрузчика выполняла загрузку своей собственной копии, а затем передавала управление на заключительный фрагмент этой копии, выполняющий вывод сообщения на экран. Новая версия будет загружать в ОП, а затем инициировать любой бинарный файл, отвечающий требованиям:

- Программа имеет точку входа со смещением 100*h* (256) относительно начала занимаемой области памяти. Этому требованию отвечает бинарный файл для *com-*

программы. Наличие данного требования обусловлено лишь желанием того, чтобы начальный загрузчик «знал» место, куда передавать управление после загрузки.

- Программа не должна использовать информацию из своего *PSP*. Поэтому, например, для начальной загрузки не подходит разработанная ранее *com*-программа, выполняющая вывод на экран своих параметров (см. п. 17.1). Наличие данного требования обусловлено тем, что подготовкой *PSP* всегда занимается тот загрузчик, который размещает программу в ОП. Наш начальный загрузчик (как и любой другой первичный начальный загрузчик) этим не занимается.
- Программа не должна содержать каких-либо вызовов ОС. Наличие данного требования обусловлено тем, что загружаемая программа является автономной, и никакой ОС в памяти нет.
- Размер бинарного файла не должен превышать величины 29696 байт. Данный размер получен следующим образом. Во-первых, как и в первой версии начального загрузчика, программа будет загружаться в ОП по адресу *700h*. Во-вторых, начиная с адреса *7C00h*, в памяти находится сам начальный загрузчик. В-третьих, с учетом того, что первая команда программы имеет смещение *100h* относительно начала области памяти, предельный размер программы составляет: *7C00h–700h–100h* = *7400h* = 29696 байт. Если разделить этот размер на длину сектора диска (512 байт), то получим предельный размер загружаемого бинарного файла, измеренный в секторах – 58.

Заметим, что, так как первые 256 байт области памяти, выделенной для загружаемой программы, не содержат какой-либо полезной информации, то эту область можно использовать, например, для размещения стека программы. Трудно даже вообразить, что такой стек когда-нибудь заполнится полностью.

Наряду с требованиями, предъявляемыми к самой загружаемой программе, с целью упростить алгоритм загрузчика ограничим также представление этой программы в используемой файловой структуре:

1) Загружаемый файл находится на том же диске, что и начальный загрузчик.

2) Имя загружаемого файла находится в корневом каталоге диска.

3) Для идентификации загружаемого файла используется лишь простое имя файла (расширение имени не используется). Поэтому если в корневом каталоге диска находятся несколько файлов с заданным собственно именем файла, но с разными расширениями имени, то из них будет загружен тот файл, которому соответствует запись корневого каталога с меньшим номером.

23.5. Алгоритм модифицированного начального загрузчика

В основе данного алгоритма лежат действия с корневым каталогом, а также действия с таблицей *FAT*. Прежде, чем выполнять анализ этих информационных структур, начальный загрузчик должен считать их с диска в какие-то свободные области ОП. Например, используемое распределение ОП может соответствовать тому, что приведено на рис. 3.21.

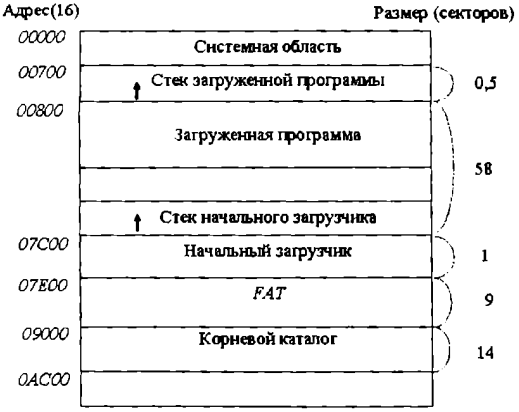

Рис. 3.21. Распределение ОП при начальной загрузке

Алгоритм второй версии начального загрузчика приведен на рис. 3.22. На этом рисунке блок завершения фиктивный, так как переход на этот блок не может быть сделан ни при каких условиях. Наличие блока завершения обусловлено лишь желанием улучшить «внешний вид» алгоритма.

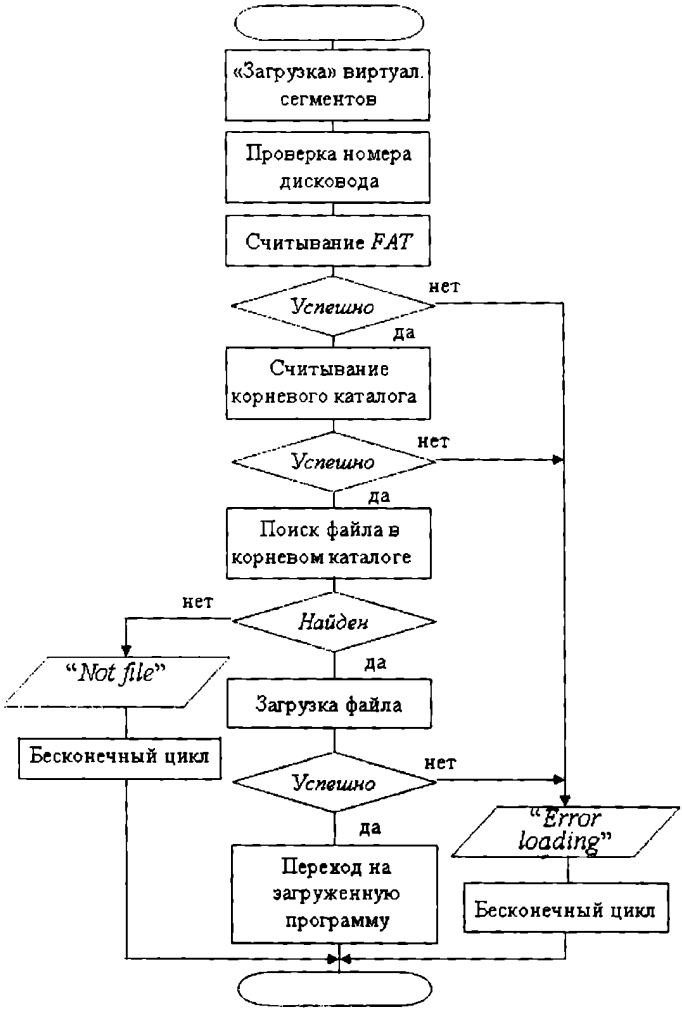

Рис. 3.22. Алгоритм второй версии начального загрузчика

Два этапа алгоритма начального загрузчика («Поиск файла в корневом каталоге» и «Загрузка файла») достаточно сложные, и поэтому их целесообразно реализовать в виде отдельных процедур. Кроме того, реализация данных этапов в виде процедур позволяет упростить программу вторичного загрузчика, который мы будем разрабатывать позднее.

Что касается этапов «Считывание *FAT*» и «Считывание корневого каталога», то их рекомендуется реализовать путем вызова одной и той же процедуры *Nsec_read*, которая считывает непрерывную область диска требуемой длины. Входные параметры процедуры *Nsec_read*: *CX* – число секторов (≥1) в области диска; *AX* – номер первого считываемого сектора диска; *ES:BX* – начальный адрес (сегмент:смещение) приемного буфера. Единственный выходной параметр данной процедуры: *CF* – признак результата (0 – загрузка произошла успешно, 1 – с ошибкой). В основе работы данной процедуры лежит вызов ранее разработанной процедуры *Sec_read*, выполняющей считывание с диска всего одного сектора.

После каждого из этапов: «Считывание *FAT*», «Считывание корневого каталога», «Загрузка файла», выполняющих считывание информации с диска в ОП, программа загрузчика выполняет проверку значения флага *CF*, и в случае ошибки (*CF*=1) она выводит сообщение об ошибке (используя вызов процедуры *Outstring*), а затем переходит в бесконечный цикл.

Последний этап алгоритма начального загрузчика (см. рис. 3.22) – «Переход на загруженную программу». Прежде, чем выполнить с помощью команды *jmp* сам переход, необходимо записать в сегментные регистры *DS*, *ES*, *SS* значения, соответствующие типу загружаемого бинарного файла. Так как мы загружаем *com*-программу, то в перечисленные регистры необходимо записать число *0700h* (см. рис. 3.21). Кроме того, в указатель стека *SP* необходимо поместить число *0800h*.

Что касается оператора безусловного перехода *jmp*, то применявшийся ранее для этой цели оператор «*jmp near*» теперь не пригоден, так как, выполняя внутрисегментный («близкий») переход, он не выполняет замену содержимого регистра сегмента кода *CS*. Тем не менее, такая замена наобходима и, подобно другим сегментным регистрам, *CS* должен указывать на начало области памяти, занимаемой программой, т. е. должен содержать число *0700h*. Напомним, что для замены содержимого *CS* команда *mov* непригодна. Это можно сделать только с помощью команды перехода, например, с помощью команды безусловного перехода, в которую транслируется оператор ***jmp far***. Данный оператор выполняет межсегментный («дальний») безусловный переход. В нашем случае для перехода на загруженную программу пригоден, например, оператор ***межсегментного косвенного перехода***. Такой оператор задается путем записи в качестве операнда оператора *jmp* не метки перехода, а имени поля данных длиной два слова, которое содержит логический адрес перехода. При этом младшее слово содержит целевое (*IP*) – адрес-смещение, а старшее слово содержит (*CS*) – адрес-сегмент. Имя поля, содержащего логический адрес, заключается в квадратные скобки и ему предшествует вспомогательный псевдооператор ***far*** (дальний). Пример:

```
. . . . . .
Old   dd    07000100h    ; Логический адрес перехода
. . . . . .
jmp   far   [cs:Old]     ; Межсегментный косвенный переход
```

Обратите внимание, что при задании в операторе *jmp* ячейки ОП с именем *Old* используется явное задание регистра сегмента *CS*. Причиной этого является то, что ис-

пользуемый при неявной адресации ячеек ОП регистр сегмента *DS* получил ранее значение, соответствующее не начальному загрузчику, а загруженной автономной программе. В момент передачи управления на загруженную программу, *CS* является единственным сегментным регистром, который можно использовать для адресации данных самого загрузчика.

23.6. Алгоритмы процедур

Процедура «Поиск файла в корневом каталоге» имеет один входной параметр: *BX* – адрес-смещение имени файла. Выходные параметры: *CF* – признак результата (0 – файл найден, 1 – не найден); *SI* – адрес-смещение записи каталога, соответствующей файлу.

Алгоритм процедуры «Поиск файла в корневом каталоге» приведен на рис. 3.23.

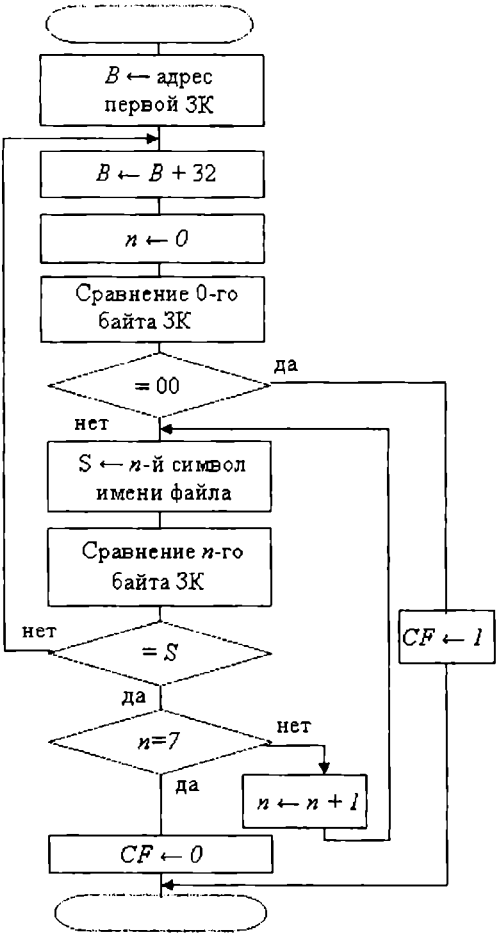

Рис. 3.23. Алгоритм процедуры «Поиск файла в корневом каталоге»:
B – адрес записи каталога (ЗК); *n* – номер байта ЗК ($n \geq 0$); *S* – символ имени файла

В основе данного алгоритма лежит сравнение содержимого полей «имя файла», принадлежащих записям каталога, с требуемым именем файла. Требуемое имя файла зада-

ется статически (с помощью псевдооператора *db*) и занимает ровно 8 байт в области памяти самого загрузчика. Все буквы имени заглавные. Если имя короче 8 символов, то пустые байты справа должны быть заполнены пробелами. Для хранения переменных алгоритма в программе удобно использовать регистры: для *B* – *SI*, для *n* – *CX*, для *S* – *DL*.

Процедура «Загрузка файла» имеет три входных параметра: *SI* – начальный адрес записи каталога, соответствующей файлу, *ES:BX* – начальный адрес (сегмент:смещение) загрузки в ОП. Единственным выходным параметром процедуры является признак завершения *CF* (0 – загрузка файла прошла успешно, 1 – с ошибкой).

В основе алгоритма процедуры «Загрузка файла» лежит поиск кластеров файла путем рассмотрения последовательности *FAT*-элементов, образующих односвязанный список, соответствующий данному файлу (рис. 3.24).

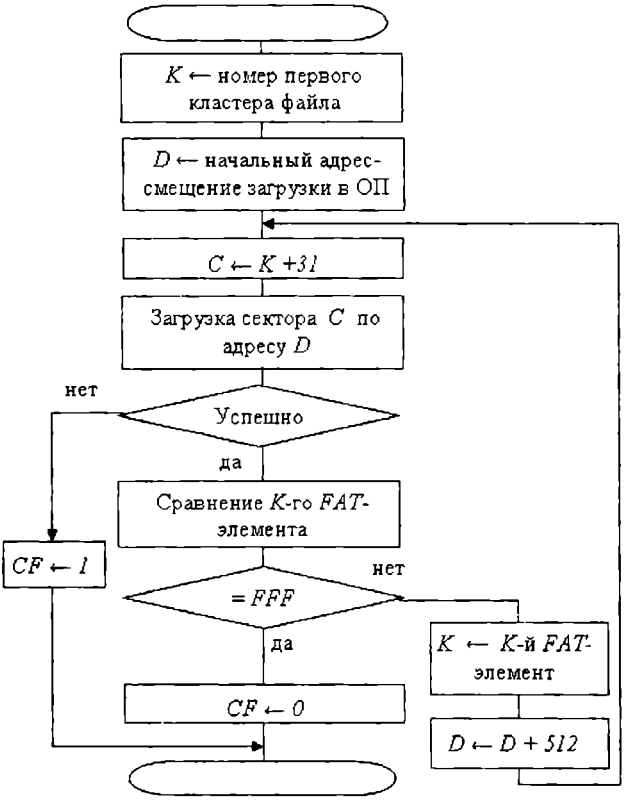

Рис. 3.24. Алгоритм процедуры «Загрузка файла»:
C – порядковый номер сектора диска; *K* – номер кластера файла;
D – начальный адрес-смещение сектора в ОП

Номер первого кластера файла определяется из анализа соответствующего поля записи корневого каталога, найденной при выполнении этапа «Поиск файла в корневом каталоге», и расположенной в ОП, начиная с адреса *B*. Далее рассчитывается тот адрес ОП, с которого должна осуществляться загрузка очередного сектора файла, и выполняется сама загрузка. Номер загружаемого сектора находится путем суммирования номера кластера и числа 31. Далее содержимое текущего *FAT*-элемента сравнивается с числом

FFFh. Если эти значения совпали, то загрузка файла завершена. Иначе – выполняется загрузка следующего сектора файла.

Особо следует остановиться на этапе «сравнение *K*-го *FAT*-элемента». Напомним, что в файловой системе *FAT12* длина одного *FAT*-элемента составляет 12 бит (1,5 байта), в то время как ЦП ориентирован на обработку данных, кратных восьми битам, т. е. на обработку байтов, слов, двойных слов. Следующий алгоритм позволяет отыскать 12-битовый *FAT*-элемент по его порядковому номеру.

Ш а г 1. Умножить номер *K* требуемого *FAT*-элемента на 1,5 и округлить результат вниз.

Ш а г 2. Прочитать 2 байта с полученным на шаге 1 смещением.

Ш а г 3. Если номер требуемого элемента четный, то следует взять младшие 12 бит, иначе – старшие 12 бит.

Данный алгоритм нетрудно реализовать на ассемблере. При этом умножение номера кластера на 1,5 производится копированием числа, сдвигом копии на один бит вправо для деления пополам и сложением копии с оригиналом. При этом одновременно выполняется округление вниз. Для программирования перечисленных действий требуется неоднократное применение известных нам операторов *shr* и *and*. Например, для определения четности элемента достаточно применить оператор *shr*. В результате младший бит элемента будет выдвинут во флаг *CF*, что позволит далее применить оператор условного перехода *jc* или *jnc*.

Лабораторная работа 21

Требуется написать и отладить программу первичного начального загрузчика, выполняющего загрузку и запуск программы, отвечающей требованиям п. 23.4 и имеющей длину более 512 байт. В качестве загружаемой программы используется шестнадцатеричный редактор.

Так как в тексте программы должны отсутствовать какие-либо системные вызовы DOS, то из всех версий редактора, полученных на лабораторных работах 13–17, для автономной загрузки пригодна только самая первая версия. Все остальные версии редактора используют системные вызовы DOS, не заменимые вызовами BIOS. Что касается первой версии редактора (лабораторная работа 13), то почти все используемые в ней вызовы DOS (*int 21h*) за исключением самого последнего вызова, можно заменить на вызовы BIOS (*int 16h* и *int 10h*). Для вывода на экран строк следует использовать недавно полученную процедуру *Outstring*. Естественно, что в коррекции нуждаются и сами строки: 1) текст строки должен быть записан латинскими буквами, так как BIOS не работает с русским алфавитом; 2) последний байт строки должен содержать не '*$*', а 0. Что касается последнего системного вызова, выполняющего возврат в DOS, то его следует заменить на бесконечный цикл, так как возвращать управление из автономной программы пока просто некому.

Рекомендуемый план отладки:

Этап 1. Загружается и выполняется короткая (менее 512 байт) программа, выполняющая посимвольный вывод на экран, используя вызовы BIOS. На данном этапе не используется таблица *FAT*, а также не осуществляется поиск файла в корневом каталоге. Использование корневого каталога ограничивается анализом записи 2, из которой определяется номер первого (и единственного) сектора загружаемого файла. Естественно,

что на данном этапе виртуальная дискета не должна содержать других файлов, кроме загружаемого бинарного файла.

Этап 2. Загружается и выполняется та же программа, что и на этапе 1. Отличие: виртуальная дискета должна содержать несколько файлов. На данном этапе по-прежнему не используется таблица *FAT*, но уже выполняется поиск файла в корневом каталоге.

Этап 3. Загружается и выполняется тестовая программа, имеющая длину не менее трех секторов по 512 байт (такую программу нетрудно получить из программы, использовавшейся на этапах 1 и 2, добавив, операторы резервирования памяти *db*, *resb* и т. д.). Естественно, что на данном этапе используется таблица *FAT*.

Этап 4. Загружается и выполняется программа шестнадцатеричного редактора.

24. Вторичный начальный загрузчик-русификатор

24.1. Функции вторичного загрузчика

В предыдущих разделах нами рассматривалась разработка одноступенчатого начального загрузчика. Применение такой схемы начальной загрузки оправдано в том случае, если функции начального загрузчика ограничены поиском загружаемого бинарного файла в используемой файловой структуре, а затем размещение этого файла где-то в начале ОП. Если от начального загрузчика требуется выполнение дополнительных функций, то из-за ограничения на размер первичного начального загрузчика (не более 512 байт) приходится использовать двухступенчатую схему начальной загрузки (см. рис. 3.13). В этом случае первичный начальный загрузчик считывает с диска в ОП, а затем инициирует не автономную программу, а вторичный начальный загрузчик, который затем выполняет загрузку и запуск автономной программы.

Одной из функций, выполнение которых можно поручить вторичному начальному загрузчику, является загрузка русского шрифта для текстового режима видеоадаптера. Наличие данной функции обусловлено, во-первых, тем, что BIOS (как это следует из предыдущего раздела) данную функцию не обеспечивает. Во-вторых, игнорирование данной функции, в общем случае существенно ухудшает пользовательский интерфейс автономной программы, лишая пользователя возможности использовать свой родной русский язык. Обычно данная функция выполняется ОС путем запуска утилиты, называемой ***русификатором***. Ввиду отсутствия ОС, русификатор можно включить в состав или вторичного загрузчика, или в состав автономной программы. Первый из этих вариантов выглядит гораздо более предпочтительным, так как оставляет выполнение служебной функции за системной программой, каковой в нашем случае является вторичный начальный загрузчик.

Выполняя поиск и загрузку бинарного файла автономной программы, вторичный начальный загрузчик во многом повторяет те действия, которые до этого выполнялись первичным начальным загрузчиком с целью подготовки к выполнению самого вторичного загрузчика. Если допустить, что и начальные загрузчики, и автономные программы находятся на одном и том же диске (дискете), то оказывается, что оба начальных загрузчика используют одни и те же структуры данных – описательный блок диска, корневой каталог, *FAT*-таблицу. Повторное использование этих структур данных особого труда не представляет, так как в ОП они расположены по фиксированным адресам. Что касается одинаковых фрагментов программ обоих начальных загрузчиков, то простейший подход

заключается в копировании таких фрагментов самим программистом при написании им исходного текста вторичного начального загрузчика. Мы займемся реализацией более изящного варианта, согласно которому программа вторичного начального загрузчика не содержит копий фрагментов программы первичного начального загрузчика, а непосредственно использует эти фрагменты. Забегая вперед, заметим, что данные фрагменты должны быть оформлены как процедуры, отвечающие определенным требованиям.

Еще одна особенность нашего вторичного загрузчика состоит в том, что он будет выполнять загрузку любой *com*-программы, отвечающей следующим условиям (соответствуют условиям из п. 23.4 за исключением требования к длине файла):

- *com*-программа не использует информацию из своего *PSP*;
- программа не должна содержать каких-либо вызовов ОС, за исключением трех системных вызовов, перечисленных ниже;
- бинарный файл программы находится в корневом каталоге той же дискеты, на которую записан начальный загрузчик.

Имя *com*-программы, подлежащей выполнению, вводится пользователем с клавиатуры.

Еще одна функция вторичного загрузчика – выполнение нескольких системных вызовов, аналогичных по форме соответствующим системным вызовам DOS. Перечень этих системных вызовов:

int 21h (функция *02h*) – вывод символа на экран. Перед применением вызова программа задает в регистре *AH* номер функции – 2, а в регистр *DL* заносится код ASCII выводимого символа.

int 21h (функция *09h*) – вывод строки символов на экран. Перед применением вызова программа задает в регистре *AH* номер функции – 9, а в регистры *DS:DX* заносится логический адрес (сегмент:смещение) первого байта выводимой строки. Последним символом выводимой строки является «*$*». На экран этот символ не выводится.

int 21h (функция *01h*) – ввод ASCII-символа с клавиатуры. Перед применением вызова программа задает в регистре *AH* номер функции – 1. В результате выполнения системного вызова ASCII-символ, соответствующий нажатой клавише, помещается в регистр *AL*. На экран выводится «эхо» символа.

Вторичный загрузчик, выполняющий перечисленные выше функции, может рассматриваться как миниатюрная однопрограммная ОС, позволяющая запускать *com*-программы, использующие не только латинский, но и русский шрифты. Во время выполнения программы ей предоставляется минимальный сервис, сходный по форме (но не по объему) с тем сервисом, который предоставляется прикладным программам со стороны DOS.

24.2. Алгоритм вторичного начального загрузчика

Алгоритм вторичного начального загрузчика приведен на рис. 3.25. Данный вторичный загрузчик выполняет загрузку программы в какую-то свободную область, расположенную недалеко от начала ОП и, следовательно, недалеко от самого вторичного загрузчика. В соответствии с используемым распределением ОП (см. рис. 3.21) эта область может, например, начинаться с реального адреса *0AC00h*. Обратите внимание, что все сообщения вторичного загрузчика делаются на русском языке.

На этапе «инициализация вектора *21h*» производится запись в вектор прерываний с номером *21h* стартового адреса своего обработчика прерываний, который используется

нами для имитации выполнения трех системных вызовов DOS. Если раньше инициализация векторов прерываний выполнялась с помощью DOS, то теперь такая инициализация должна выполняться без такой помощи. Что касается самих подпрограмм, выполняющих обработку системных вызовов «*int 21h*», то так как они являются обработчиками прерываний, они никак не связаны по управлению с другими подпрограммами вторичного загрузчика. Как следствие, эти подпрограммы не представлены на рис. 3.25, так как имеют свои отдельные алгоритмы. Реализация этапа «инициализация векторов прерываний» рассматривается в п. 24.5, а сама обработка системных вызовов – в п. 24.6.

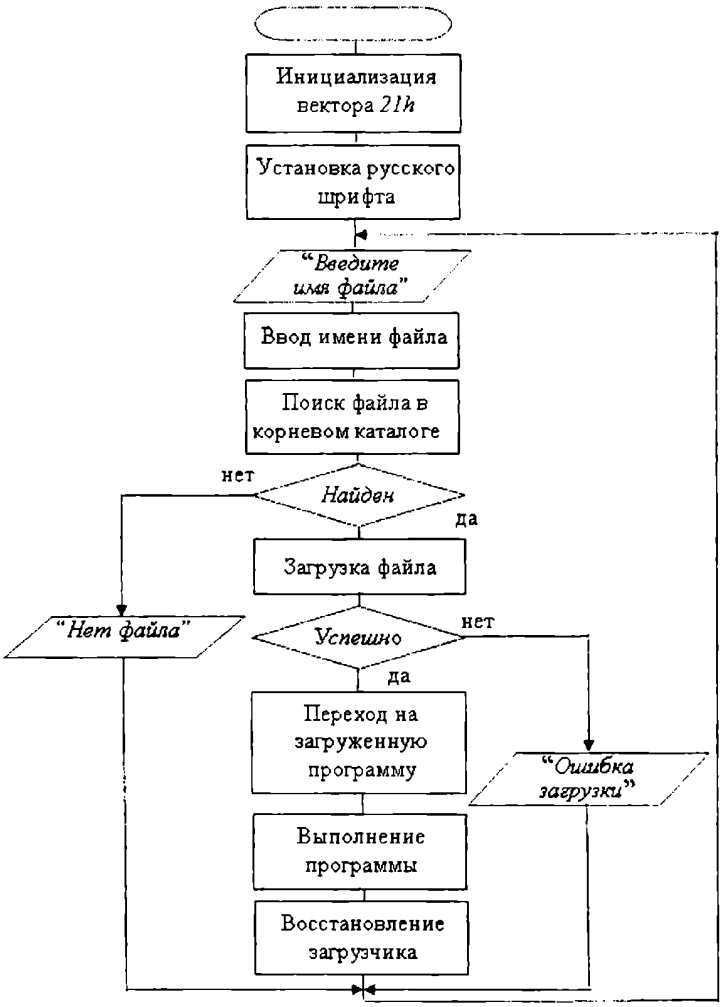

Рис. 3.25. Алгоритм вторичного начального загрузчика

На этапе установки русского шрифта производится коррекция памяти знакогенератора, ответственная за интерпретацию кодов ASCII символов, выводимых на экран. Реализация данного этапа рассматривается в п. 24.3.

Для реализации каждого из трех этапов, выполняющих вывод сообщений на экран, используется процедура *Outstring*, входящая в состав программы первичного загрузчика.

Аналогично, для программирования этапов «поиск файла в корневом каталоге» и «загрузка файла» также используются соответствующие процедуры, входящие в состав первичного загрузчика. С целью уменьшить размер программы вторичного загрузчика, в ней используются косвенные вызовы процедур, входящих в состав первичного загрузчика. Реализация косвенного вызова процедур рассматривается в п. 24.4.

В основе этапа «ввод имени файла» лежит использование системного вызова BIOS – «*int 16h*» (функция 0), выполняющего ввод символа с клавиатуры (без «эха»). При этом код ASCII символа помещается в регистр *AL*. На данном этапе введенные символы по одному записываются в поле для имени файла. Длина этого поля – 8 байтов. При достижении конца поля, или при нажатии клавиши <Enter> выполнение этапа завершается. Момент нажатия <Enter> определяется по получению кода ASCII символа – *0dh*. После создания своего обработчика прерываний с номером *21h*, в программе вторичного загрузчика следует заменить системный вызов «*int 16h*» (функция 0) на вызов «*int 21h*» (функция 1), позволяющий видеть набираемые символы на экране.

На этапе «переход на загруженную программу» выполняются: 1) загрузка в сегментные регистры *ds*, *es* и *ss* номера начального параграфа той области памяти, в которую загружается *com*-программа, т. е. числа *AC0h*; 2) загрузка в указатель команд *IP* числа *100h* (как для любой *com*-программы); 3) собственно переход на загруженную программу. Данный переход реализуется как межсегментный косвенный вызов процедуры. При таком подходе к инициированию автономной программы сама эта программа нуждается в небольшой корректировке – в качестве последнего исполнительного оператора программы, выполняющего возврат управления во вторичный загрузчик, должен быть записан оператор ***retf***. Подробности межсегментного косвенного вызова процедур изложены в п. 24.4.

Следующий этап алгоритма вторичного загрузчика, который выполняется сразу же после возврата из автономной программы, этап «восстановление загрузчика». На этом этапе выполняются действия, требуемые для подготовки вторичного загрузчика к запуску следующего *com*-файла. Это действия: 1) загрузка в сегментные регистры *ds*, *es* и *ss* числа 0; 2) загрузка в указатель стека *SP* числа *800h*; 3) заполнение восьмибайтового поля имени файла пробелами, т. е. символами с кодом *20h*..

24.3. Установка русского шрифта

Внутренняя память видеоадаптера включает не только видеопамять, содержащую ASCII-коды символов, предназначенных для вывода на экран. Она содержит также память знакогенератора, содержащую информацию о том, как выполнить изображение на экране каждого символа. При этом символ «рисуется» на экране в виде нескольких строк. Каждая строка кодируется одним или несколькими байтами, биты которых соответствуют точкам (пикселам) экрана. Если бит установлен в единицу, то соответствующая точка экрана должна светиться.

В устаревшем видеоадаптере *VGA*, который подобно *CGA* имитируется аппаратно многими другими, более современными видеоадаптерами, каждый символ «рисуется» в виде 16 строк, каждая из которых кодируется одним байтом. Например, следующее поле данных кодирует русскую букву «*А*»:

```
db  00h, 00h, 10h, 38h, 6ch, 0c6h, 0c6h, 0feh          ; Рисунок буквы
db  0c6h, 0c6h, 0c6h, 0c6h, 00h, 00h, 00h, 00h         ; А
```

Это же самое поле данных может быть задано следующим образом:

```
db  00000000b, 00000000b, 00010000b, 00111000b    ; Рисунок буквы
db  01101100b, 11000110b, 11000110b, 11111110b    ; A
db  11000110b, 11000110b, 11000110b, 11000110b    ;     –"–
db  00000000b, 00000000b, 00000000b, 00000000b    ;     –"–
```

На рис. 3.26 показана буква «*A*», построенная по точкам, заданным полем данных, приведенным выше. В более современных видеоадаптерах каждый символ рисуется большим числом строк, а каждая строка кодируется не одним, а несколькими байтами. Но BIOS, с помощью которого мы будем устанавливать русский шрифт, ориентирован на использование *VGA*.

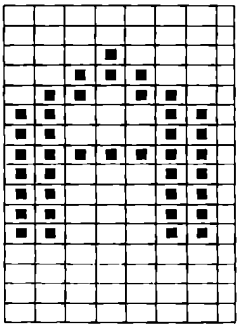

Рис. 3.26. Рисунок буквы «*A*», построенный по точкам

Так как код ASCII символа имеет длину 1 байт, то один шрифт может включать 256 различных символов. Сюда относятся символы, отображаемые на экране и символы, управляющие выводом на экран. Память знакогенератора *VGA* может содержать 8 различных шрифтов, из которых шрифт 0 является основным.

Для замены шрифта или его части можно использовать системный вызов BIOS «***int 10h***», функция ***11h***, подфункция ***0***. Входные параметры вызова: *BH* – число байтов в описании символа; *BL* – номер шрифта знакогенератора; *CX* – число устанавливаемых символов; *DX* – код ASCII первого устанавливаемого символа в шрифте; *ES:BP* – указатель на таблицу, содержащую описания устанавливаемых символов.

Из анализа табл. 1.4, содержащей коды ASCII русских букв, можно сделать вывод о том, что эти коды образуют не один, а два непрерывных фрагмента шрифта. Первый фрагмент включает коды *80h ÷ AFh* (символы «*А*» ÷ «*п*»), а второй фрагмент – коды *E0h ÷ EFh* (символы «*р*» ÷ «*я*»). Поэтому для установки русского шрифта требуется иметь две таблицы определения символов и требуется дважды выполнить системный вызов «*int 10h*», функция *11h*, подфункция *0*. Обе таблицы определения символов приводятся ниже.

```
Table1  db    00h,00h,10h,38h,6ch,0c6h,0c6h,0feh          ; А
        db    0c6h,0c6h,0c6h,0c6h,00h,00h,00h,00h         ; ----
        db    00h,00h,0feh,62h,60h,7ch,66h,66h            ;  Б
        db    66h,66h,66h,0fch,00h,00h,00h,00h            ; -----
        db    00h, 00h, 0fch, 66h,66h,66h,7eh,66h         ;  В
        db    66h,66h,66h,0fch,00h,00h,00h,00h            ; -----
        db    00h,00h,0feh,62h,62h,60h,60h,60h            ;  Г
        db    60h,60h,60h,0f0h,00h,00h,00h,00             ; -----
        db    00h,00h,3eh,66h,66h,66h,66h,66h             ;  Д
        db    66h,66h,66h,0ffh,0c3h,00h,00h,00h           ; -----
        db    00h,00h,0feh,62h,60h,64h,7ch,64h            ;  Е
        db    60h,60h,62h,0feh,00h,00h,00h,00h            ; ----
```

```
db    00h,00h,99h,0dbh,5ah,7eh,3ch,3ch        ; Ж
db    7eh,5ah,0dbh,99h,00h,00h,00h,00h        ; ------
db    00h,00h,3ch,66h,46h,06h,1ch,06h         ; З
db    06h,46h,66h,3ch,00h,00h,00h,00h         ; ------
db    00h,00h,0c6h,0c6h,0ceh,0deh,0feh,0F6h   ; И
db    0F6h,0c6h,0c6h,0c6h,00h,00h,00h,00h     ; ------
db    00h,18h,0d6h,0F6h,0ceh,0deh,0feh,0F6h   ; Й
db    0e6h,0c6h,0c6h,0c6h,00h,00h,00h,00h     ; ------
db    00h,00h,0e6h,66h,6ch,6ch,70h,70h        ; К
db    6ch,6ch,66h,0e6h,00h,00h,00h,00h        ; ------
db    00h,00h,1eh,3eh,66h,66h,66h,66h         ; Л
db    66h,66h,66h,0c6h,00h,00h,00h,00h        ; ------
db    00h,00h,0c6h,0eeh,0feh,0feh,0d6h,0c6h   ; М
db    0c6h,0c6h,0c6h,0c6h,00h,00h,00h,00h     ; ------
db    00h,00h,0c6h,0c6h,0c6h,0c6h,0feh,0c6h   ; Н
db    0c6h,0c6h,0c6h,0c6h,00h,00h,00h,00h     ; ------
db    00h,00h,38h,6ch,0c6h,0c6h,0c6h,0c6h     ; О
db    0c6h,0c6h,6ch,38h,00h,00h,00h,00h       ; ------
db    00h,00h,0feh,0c6h,0c6h,0c6h,0c6h,0c6h   ; П
db    0c6h,0c6h,0c6h,0c6h,00h,00h,00h,00h     ; ------
db    00h,00h,0fch,66h,66h,66h,7ch,60h        ; Р
db    60h,60h,60h,0f0h,00h,00h,00h,00h        ; ------
db    00h,00h,3ch,66h,0c6h,0c0h,0c0h,0c0h     ; С
db    0coh,0c6h,66h,3ch,00h,00h,00h,00h       ; ------
db    00h,00h,7eh,5ah,18h,18h,18h,18h         ; Т
db    18h,18h,18h,3ch,00h,00h,00h,00h         ; ------
db    00h,00h,66h,66h,66h,66h,66h,66h         ; У
db    3eh,06h,66h,3ch,00h,00h,00h,00h         ; ------
db    00h,00h,18h,7eh,0dbh,0dbh,0dbh,0dbh     ; Ф
db    0dbh,7eh,18h,18h,00h,00h,00h,00h        ; ------
db    00h,00h,0c6h,0c6h,0c6h,7ch,38h,7ch      ; Х
db    0c6h,0c6h,0c6h,0c6h,00h,00h,00h,00h     ; ------
db    00h,00h,0cch,0cch,0cch,0cch,0cch,0cch   ; Ц
db    0cch,0cch,0cch,0feh,06h,00h,00h,00h     ; ------
db    00h,00h,0c6h,0c6h,0c6h,0c6h,0c6h,0c6h   ; Ч
db    7eh,06h,06h,06h,00h,00h,00h,00h         ; ------
db    00h,00h,0d6h,0d6h,0d6h,0d6h,0d6h,0d6h   ; Ш
db    0d6h,0d6h,0d6h,0feh,00h,00h,00h,00h     ; ------
db    00h,00h,0d6h,0d6h,0d6h,0d6h,0d6h,0d6h   ; Щ
db    0d6h,0d6h,0d6h,0ffh,03h,00h,00h,00h     ; ------
db    00h,00h,0e0h,0e0h,60h,60h,60h,7ch       ; Ъ
db    66h,66h,66h,7ch,00h,00h,00h,00h         ; ------
db    00h,00h,0c6h,0c6h,0c6h,0c6h,0c6h,0F6h   ; Ы
db    0deh,0deh,0deh,0F6h,00h,00h,00h,00h     ; ------
db    00h,00h,00h,0c0h,0c0h,0c0h,0c0h,0fch    ; Ь
db    0c6h,0c6h,0c6h,0fch,00h,00h,00h,00h     ; ------
db    00h,00h,78h,0cch,0c6h,06h,06h,1eh       ; Э
```

```
        db    06h,0c6h,0cch,78h,00h,00h,00h,00h      ; -----
        db    00h,00h,0ceh,0dbh,0dbh,0dbh,0fbh,0dbh  ; Ю
        db    0dbh,0dbh,0dbh,0ceh,00h,00h,00h,00h    ; -----
        db    00h,00h,3fh,66h,66h,66h,3eh,36h        ; Я
        db    66h,66h,66h,0e7h,00h,00h,00h,00h       ; -----
        db    00h,00h,00h,00h,00h,78h,0ch,7ch        ; а
        db    0cch,0cch,0cch,76h,00h,00h,00h,00h     ; -----
        db    00h,00h,00h,00h,04h,7ch,0c0h,0F8h      ; б
        db    0cch,0cch,0cch,78h,00h,00h,00h,00h     ; -----
        db    00h,00h,00h,00h,00h,0F8h,0cch,0F8h     ; в
        db    0cch,0cch,0cch,0F8h,00h,00h,00h,00h    ; -----
        db    00h,00h,00h,00h,00h,0fch,0c0h,0c0h     ; г
        db    0c0h,0c0h,0c0h,0c0h,00h,00h,00h,00h    ; -----
        db    00h,00h,00h,00h,00h,7ch,6ch,6ch        ; д
        db    6ch,6ch,6ch,0feh,0c6h,00h,00h,00h      ; -----
        db    00h,00h,00h,00h,00h,78h,0cch,0fch      ; е
        db    0c0h,0c0h,0cch,78h,00h,00h,00h,00h     ; -----
        db    00h,00h,00h,00h,00h,92h,0d6h,7ch       ; ж
        db    38h,7ch,0d6h,92h,00h,00h,00h,00h       ; -----
        db    00h,00h,00h,00h,00h,7ch,8ch,38h        ; з
        db    0ch,0ch,8ch,78h,00h,00h,00h,00h        ; -----
        db    00h,00h,00h,00h,00h,0cch,0cch,0dch     ; и
        db    0fch,0ech,0cch,0cch,00h,00h,00h,00h    ; -----
        db    00h,00h,08h,18h,10h,0cch,0cch,0dch     ; й
        db    0fch,0ech,0cch,0cch,00h,00h,00h,00h    ; -----
        db    00h,00h,00h,00h,00h,0cch,0d8h,0f0h     ; к
        db    0d8h,0c8h,0cch,0cch,00h,00h,00h,00h    ; -----
        db    00h,00h,00h,00h,00h,1ch,3ch,2ch        ; л
        db    6ch,4ch,4ch,0cch,00h,00h,00h,00h       ; -----
        db    00h,00h,00h,00h,00h,0c6h,0eeh,0d6h     ; м
        db    0d6h,0c6h,0c6h,0c6h,00h,00h,00h,00h    ; -----
        db    00h,00h,00h,00h,00h,0cch,0cch,0cch     ; н
        db    0fch,0cch,0cch,0cch,00h,00h,00h,00h    ; -----
        db    00h,00h,00h,00h,00h,78h,0cch,0cch      ; о
        db    0cch,0cch,0cch,78h,00h,00h,00h,00h     ; -----
        db    00h,00h,00h,00h,00h,0fch,0cch,0cch     ; п
        db    0cch,0cch,0cch,0cch,00h,00h,00h,00h    ; -----

Table2  db    00h,00h,00h,00h,00h,0F8h,0cch,0cch     ; р
        db    0cch,0cch,0cch,0F8h,0c0h,0c0h,00h,00h  ; -----
        db    00h,00h,00h,00h,00h,78h,0cch,0c0h      ; с
        db    0c0h,0c0h,0cch,78h,00h,00h,00h,00h     ; -----
        db    00h,00h,00h,00h,00h,0fch,30h,30h       ; т
        db    30h,30h,30h,30h,00h,00h,00h,00h        ; -----
        db    00h,00h,00h,00h,00h,66h,66h,66h        ; у
        db    66h,66h,3eh,06h,66h,3ch,00h,00h        ; -----
        db    00h,00h,00h,00h,10h,7ch,0d6h,0d6h      ; ф
```

```
db    0d6h,0d6h,7ch,10h,10h,00h,00h,00h      ; -----
db    00h,00h,00h,00h,00h,0c6h,6ch,38h       ; х
db    10h,38h,6ch,0c6h,00h,00h,00h,00h       ; -----
db    00h,00h,00h,00h,00h,0cch,0cch,0cch     ; ц
db    0cch,0cch,0cch,0feh,06h,00h,00h,00h    ; -----
db    00h,00h,00h,00h,00h,0cch,0cch,0cch     ; ч
db    7ch,0ch,0ch,0ch,00h,00h,00h,00h        ; -----
db    00h,00h,00h,00h,00h,0d6h,0d6h,0d6h     ; ш
db    0d6h,0d6h,0d6h,0feh,00h,00h,00h,00h    ; -----
db    00h,00h,00h,00h,00h,0d6h,0d6h,0d6h     ; щ
db    0d6h,0d6h,0d6h,0ffh,03h,00h,00h,00h    ; -----
db    00h,00h,00h,00h,00h,0e0h,0e0h,7ch      ; ъ
db    66h,66h,66h,7ch,00h,00h,00h,00h        ; -----
db    00h,00h,00h,00h,00h,0c6h,0c6h,0F6h     ; ы
db    0deh,0deh,0deh,0F6h,00h,00h,00h,00h    ; -----
db    00h,00h,00h,00h,00h,0c0h,0c0h,0F8h     ; ь
db    0cch,0cch,0cch,0F8h,00h,00h,00h,00h    ; -----
db    00h,00h,00h,00h,00h,78h,0cch,0ch       ; э
db    3ch,0ch,0cch,78h,00h,00h,00h,00h       ; -----
db    00h,00h,00h,00h,00h,0dch,0d6h,0F6h     ; ю
db    0F6h,0F6h,0d6h,0dch,00h,00h,00h,00h    ; -----
db    00h,00h,00h,00h,00h,7ch,0cch,0cch      ; я
db    0cch,7ch,6ch,0cch,00h,00h,00h,00h      ; -----
```

24.4. Близкий косвенный вызов процедуры

Выполняя загрузку автономной программы, вторичный начальный загрузчик повторяет многие действия, выполняемые первичным начальным загрузчиком. Сюда относятся, в частности, поиск искомого файла в корневом каталоге диска, а затем загрузка найденного файла в ОП. С учетом того, что корневой каталог и таблица *FAT* загружены первичным загрузчиком в ОП по фиксированным адресам, отпадает потребность в повторной загрузке этих структур данных вторичным загрузчиком. Другое дело – фрагменты программ, совпадающие в обоих начальных загрузчиках. Простейший подход заключается в повторении совпадающих фрагментов в программе вторичного начального загрузчика. Но мы будем использовать другой, более изящный, хотя и более сложный подход, суть которого заключается в совместном использовании обоими загрузчиками одних и тех же областей памяти, содержащих общие программные фрагменты.

Каждый общий фрагмент программы оформляется в виде процедуры, называемой далее ***разделяемой процедурой***. Именно этим объясняется наличие такого большого колличества процедур (5) в программе первичного начального загрузчика. Машинный код разделяемой процедуры должен быть: 1) полностью независим от места расположения этой процедуры в ОП; 2) в тексте процедуры не должно быть никаких изменяемых данных. Например, исходный текст процедуры не может содержать оператор «*mov ax, Msg1*», так как полученная в результате трансляции этого оператора машинная команда будет содержать адрес-смещение байта памяти с меткой *Msg1*. В зависимости от расположения байта *Msg1* относительно начала своего виртуального сегмента, содержимое

соответствующего поля указанной машинной команды будет разным. Каждая из процедур нашего первичного начального загрузчика отвечает указанным требованиям, так как использует для хранения изменяемых данных исключительно регистры.

Как следует из предыдущего раздела, передаче управления из первичного начального загрузчика в загруженную им программу предшествует подготовка сегментных регистров *DS, ES, SS*, а также указателя стека *SP*. Теперь, прежде, чем выполнять подобную подготовку, первичный загрузчик должен поместить адреса-смещения разделяемых процедур в какие-то, специально для этого предназначенные, ячейки памяти, занимаемой вторичным загрузчиком. Иными словами, первичный загрузчик должен выполнить настройку программы вторичного загрузчика.

Так как содержимое всех сегментных регистров первичного и вторичного начальных загрузчиков совпадает (*0000h*), то во вторичный загрузчик должны быть переданы те же начальные адреса-смещения процедур, которые используются внутри первичного загрузчика. В таких случаях, когда для передачи управления процедуре достаточно знать лишь адрес-смещение ее первого оператора, используется ***близкий вызов*** процедуры. Если бы адреса-сегменты первичного и вторичного загрузчиков были бы разными, то пришлось бы использовать ***дальний вызов*** процедуры.

Как и в любой программе, данные, используемые вторичным загрузчиком, располагаются в ячейках ОП, которые ни при каких условиях не должны рассматриваться ЦП в качестве ячеек, содержащих машинные команды. Это относится и к ячейкам памяти, содержащим адреса-смещения разделяемых процедур. Например, начальный адрес-смещение процедуры *Outstring* может располагаться в программе вторичного загрузчика следующим образом:

```
;  Вторичный начальный загрузчик-русификатор
;  ------------------------------------------
                org     800h
                jmp     M1
U_Outstring     dw      0                    ; Для адреса процедуры Outstring
   . . . . . . . . . . .
M1:
;  Инициализация векторов прерываний
   . . . . . . . . . . .
```

Для того чтобы в любом месте программы вторичного начального загрузчика запустить процедуру *Outstring*, достаточно поместить в это место программы следующий оператор ***близкого косвенного вызова процедуры***:

```
call    [U_Outstring] .
```

Наличие квадратных скобок сообщает транслятору о том, что заключенный в них адрес есть адрес той ячейки ОП, которая содержит начальный адрес-смещение вызываемой процедуры. Естественно, что где-то перед этим оператором в программе вторичного загрузчика должен находиться оператор *mov*, записывающий в регистр *si* начальный адрес выводимой на экран строки байтов.

Для того чтобы программа вторичного начального загрузчика могла использовать операторы косвенного вызова процедуры *Outstring*, программа первичного загрузчика

должна поместить в ячейку с меткой *U_Outstring* начальный адрес-смещение этой процедуры. Метка *U_Outstring* определена только в программе вторичного загрузчика. Поэтому для адресации этой же ячейки в программе первичного загрузчика приходится использовать не метку, а численный адрес-смещение искомой ячейки. С учетом того, что программа вторичного загрузчика загружается по адресу *800h*, и то, что искомой ячейке предшествует лишь 3-байтовая машинная команда *jmp*, адрес-смещение этой ячейки равен *803h*. Запись по этому адресу начального адреса процедуры *Outstring* выполняют операторы:

```
mov  bx, Outstring    ; Запись адреса-смещения процедуры
mov  [803h], bx       ; Outstring
```

Аналогично процедуре *Outstring*, первичный загрузчик подготавливает стартовые адреса и других процедур, вызываемых в программе вторичного загрузчика.

24.5. Дальний косвенный вызов процедуры

Допустим, что две программы используют разные сегменты памяти размером 64 K. При этом под сегментом памяти, используемым программой, понимается тот сегмент, на который «указывает» ее регистр сегмента кода *CS*. Неважно, пересекаются или не пересекаются сегменты памяти двух программ. Единственное, что имеет значение, это разное содержимое их *CS*. В этом случае для того, чтобы одна программа могла вызывать процедуру, принадлежащую другой программе, она должна использовать оператор дальнего косвенного вызова процедуры.

При разработке нашего вторичного загрузчика вся запускаемая *com*-программа рассматривается как удаленная процедура, расположенная в другом сегменте памяти. Поэтому инициирование *com*-программы реализуется как межсегментный косвенный вызов процедуры с помощью следующего оператора:

```
call  far  [cs:Old],
```

где *Old* – метка двухсловного поля, содержащего логический адрес перехода (в старшем слове – адрес-сегмент, в младшем слове – адрес-смещение):

```
Old   dd    0AC00100h
```

Псевдооператор ***far*** означает, что выполняется дальний (межсегментный) вызов процедуры, расположенной в другом сегменте памяти. Использование сегментного регистра *CS* для адресации поля *Old* обусловлено тем, что к моменту выполнения команды *call* все остальные сегментные регистры уже указывают не на начало памяти, а на *com*-программу.

Так как инициирование *com*-программы из вторичного загрузчика производится с помощью оператора вызова процедуры *call*, то и возврат управления из *com*-программы во вторичный загрузчик выполняет оператор возврата из процедуры ***retf***. Последняя буква *f* означает, что выполняемый возврат управления является дальним (межсегментным).

В отличие от обычного (близкого) оператора *call* дальний оператор *call* помещает в стек не одно, а два слова, образующих адрес возврата – сначала адрес-сегмент, а затем – адрес-смещение. Оператор *retf* извлекает эти слова из стека в обратном порядке и помещает их в регистры *IP* и *CS*.

Операторы дальнего вызова процедуры *call far* и возврата из нее *retf* понадобятся нам не только для инициирования *com*-программ, но и для реализации системных вызовов.

24.6. Инициализация векторов прерываний

Так как мы поручили нашему вторичному загрузчику выполнение некоторых системных вызовов DOS, то обработчики этих системных вызовов «территориально» входят в программу вторичного загрузчика. Любая программа, в том числе, и сам вторичный загрузчик, может инициировать системный вызов, используя команду программного прерывания «*int 21h*». Для того чтобы при попадании этой команды на ЦП был запущен наш обработчик прерываний, необходимо на этапе «инициализация векторов прерываний» поместить стартовый адрес этого обработчика в вектор прерываний с номером *21h*.

Ранее (см. гл. 19 и 21) для записи в нужный вектор прерываний мы использовали системный вызов DOS: «*int 21h*», функция 25. Теперь, в виду отсутствия DOS, мы будем выполнять запись в вектор прерываний напрямую, используя обычные команды *mov*. При выполнении такой записи следует учитывать размещение в ОП вектора прерываний с номером *21h*. Данное размещение показано на рис. 3.27.

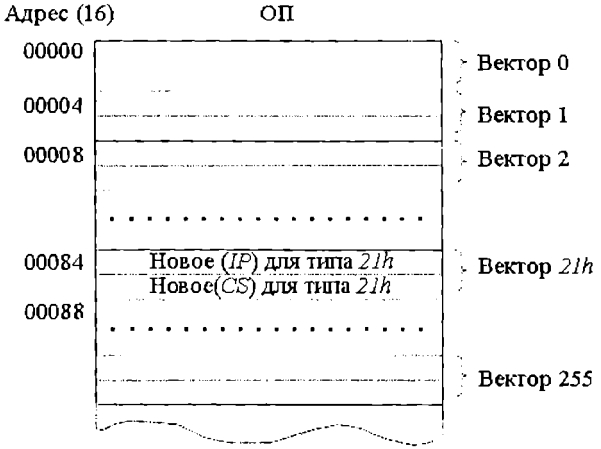

Рис. 3.27. Размещение вектора прерываний *21h* в ОП

Допустим, что первый оператор обработчика прерываний *21h* имеет метку *Obr_pr*, тогда инициализацию вектора *21h* можно выполнить с помощью следующего программного фрагмента:

```
; Запись стартового адреса обработчика в вектор прерываний с номером 21h
    xor     ax, ax              ; AX ← 0
    mov     es, ax              ; ES ← 0
    mov     dx, Obr_pr          ; DX ← адрес-смещение обработчика
    mov     [es:84h], dx        ; Запись адреса-смещения в вектор
    mov     dx, cs              ; DX ← адрес-сегмент обработчика
    mov     [es:86h], dx        ; Запись адреса-сегмента в вектор
```

Аналогично просто выполняется и замена системного вызова DOS – «*int 21h*», функция 35. При этом каждая такая замена увеличивает длину программы всего на один оператор.

24.7. Выполнение системных вызовов

Напомним, что наш вторичный загрузчик должен выполнять три системных вызова, аналогичных по форме соответствующим вызовам DOS: 1) «*int 21h*» (функция 1) – ввод кода ASCII символа с клавиатуры; 2) «*int 21h*» (функция 2) – вывод символа на экран; 3) «*int 21h*» (функция 9) – вывод строки символов на экран. Алгоритм обработки прерывания *21h* приведен на рис. 3.28.

Кодирование этапов алгоритма основано на использовании двух системных вызовов BIOS: 1) «*int 10h*» (функция *0eh*) – вывод символа на экран; 2) «*int 16h*» (функция 0) – ввод символа с клавиатуры. Первый из этих вызовов используется для реализации этапов «вывод эха символа», «вывод символа на экран» и «вывод строки символов». Второй вызов лежит в основе программирования этапа «ввод символа с клавиатуры».

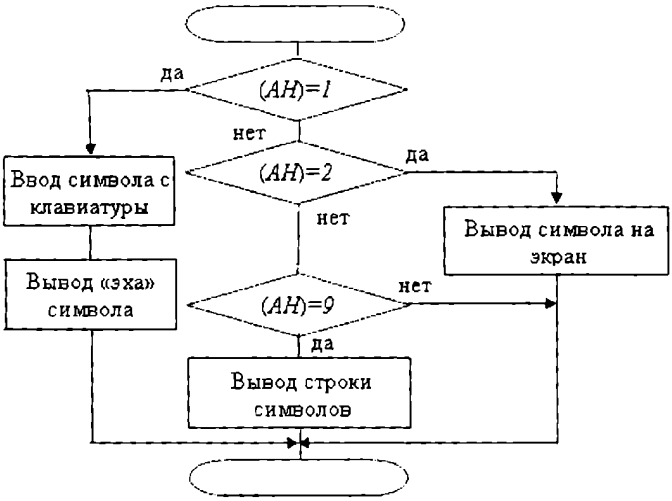

Рис. 3.28. Алгоритм обработки прерываний *21h*

Программная реализация этапа «вывод символа на экран»:

```
.M1:
; Вывод символа на экран (DL содержит ASCII-код символа)
    push    ax
    push    bx
    mov     ah, 0eh             ; Функция вывода символа
    mov     al, dl              ; AL ← код ASCII символа
    xor     bh, bh              ; Задание нулевой видеостраницы
    pushf                       ; Запись в стек регистра флагов
    call    far [cs:40h]        ; Вместо «int 10h»
    pop     bx
    pop     ax
    iret                        ; Возврат из обработчика 21h
```

Следует обратить внимание, что вместо команды «*int 10h*» мы использовали команду дальнего вызова процедуры «*call far [cs:40h]*». Выбор адреса-смещения *40h* обусловлен, во-первых, тем, что по реальному адресу *40h* начинается вектор прерываний с номером *10h* (*10h* x *4* = *40h*). Во-вторых, так как для вторичного загрузчика (*cs*)=*0*, то логический адрес [*cs:40h*] и реальный адрес *40h* указывают на одну и ту же ячейку памяти – первый байт вектора прерываний *10h*. С учетом того, что в младшем слове вектора записан стартовый адрес-смещение, а в старшем слове – адрес-сегмент подпрограммы-обработчика (см. рис. 3.27), то мы имеем именно тот порядок записи адресов перехода, который мы использовали в предыдущем подразделе для дальнего косвенного вызова процедур.

Единственное, чем подпрограмма-обработчик прерываний *10h* (как и обработчик любого другого прерывания) отличается от обычной дальней процедуры, заключается в том, что в конце подпрограммы обработчика записан не оператор *retf*, а оператор *iret*. Отличие в выполнении операторов *iret* и *retf* заключается в том, что *iret* выбирает из стека не два слова (адрес-сегмент и адрес-смещение возврата), а три. Последним выбираемым из стека слова является содержимое регистра *FLAGS* (см. рис. 2.32). Поэтому в программу добавлен оператор *pushf*, выполняющий запись содержимого *FLAGS* в стек.

Что касается этапа «вывод строки символов», то его программная реализация очень похожа на ранее записанную процедуру *Outstring* (см. п. 22.4). Главное отличие состоит в том, что конец строки определяется не по наличию нулевого байта, а по наличию байта с кодом *24h* («*$*»).

Лабораторная работа 22

Требуется написать и отладить программу вторичного начального загрузчика-русификатора, выполняющего загрузку и инициирование *com*-файлов, в том числе, и программы шестнадцатеричного редактора. Загружаемые *com*-программы могут использовать латинский и русский шрифты, а также могут содержать не только системные вызовы BIOS, но и вызовы, аналогичные DOS – «*int 21h*», функции 1, 2 и 9.

☑ **Примечание**. Учитывая большой размер таблиц, содержащих «рисунки» символов, разрешается ограничиться использованием фрагментов этих таблиц.

Рекомендуемый план отладки.

Этап 1. Первичный начальный загрузчик загружает автономную программу, выполняющую установку русского шрифта и последующий вывод на экран (с помощью косвенного вызова процедуры *Outstring*) короткого сообщения на русском языке.

Этап 2. Расширить программу русификатора, полученную на предыдущем этапе, до программы вторичного загрузчика, выполняющего загрузку *com*-программы. При этом имя *com*-программы задается во вторичном загрузчике статически (с помощью псевдооператора *db*). Сама *com*-программа выполняет вывод на экран нескольких русских букв, используя системные вызовы BIOS.

Этап 3. Вторичный загрузчик выполняет загрузку и инициирование *com*-программ, имена которых вводятся с клавиатуры. Среди загружаемых программ – шестнадцатеричный редактор. И вторичный загрузчик, и *com*-программы используют только системные вызовы BIOS.

Этап 4. Выполняется отладка окончательного варианта вторичного загрузчика-русификатора. В программе вторичного загрузчика все системные вызовы BIOS, а также

вызовы процедуры *Outstring*, заменены системными вызовами «*int 21h*» (функции 1, 2, 9). Эти же системные вызовы используются и в запускаемых *com*-программах.

25. Редактирование всей оперативной памяти

25.1. Режимы работы процессора

Мы могли бы и дальше развивать свою миниатюрную однопрограммную ОС, двигаясь по двум направлениям: 1) совершенствование пользовательского интерфейса; 2) совершенствование программного интерфейса. По обоим направлениям нам видны далекие «маяки». По первому направлению таким «маяком» является интерпретатор команд DOS, а по второму – ядро DOS. Но вместо того, чтобы стремиться к достижению величия DOS, мы займемся реализацией такой функции, которая в DOS отсутствует. Речь идет о работе с ОП, расположенной за пределами первого мегабайта.

Напомним, что операционная система DOS работает только с одно мегабайтовым адресным пространством ОП. Этим же свойством обладают и огромное большинство подпрограмм BIOS. Что касается прикладных программ, то для многих из них одномегабайтная память явно недостаточна. В качестве примера можно назвать текстовые и графические редакторы, геоинформационные системы, системы проектирования и т. д. Наш шестнадцатеричный редактор пока также работает лишь в пределах первого мегабайта ОП, хотя существует потребность в просмотре и коррекции других областей памяти.

Имея огромный объем ОП, современная ЭВМ аппаратно имитирует работу с одномегабайтной памятью «древнего» процессора i8086. Эта имитация выполняется в двух из трех основных режимов процессора i80386 и последующих процессоров фирмы *Intel*. Это ***реальный режим*** и ***режим V86*** – режим виртуального процессора i8086. Декларируется, что в реальном режиме процессор оказывается сразу же после включения питания (как будет показано ниже, это не совсем так). Выполняемая в реальном режиме программа имеет дело с самым первым мегабайтом ОП. Что касается режима *V86*, то предоставляемый программе мегабайт памяти является виртуальным, располагаясь, возможно, не только в ОП, но и на магнитном диске. Другим отличием режима *V86* от реального режима является то, что в этом режиме запрещены машинные команды, выполняющие перевод процессора в защищенный режим. С учетом этих отличий, а также того, что переход в режим *V86* возможен лишь из защищенного режима, режим *V86* далее не рассматривается.

Основным режимом работы современного процессора является ***защищенный режим***. В этом режиме для программного обеспечения доступен весь объем оперативной памяти. Основной функцией данного режима, как это следует из его названия, является защита параллельно выполняемых программ друг от друга. Поэтому защищенный режим процессора используется для организации работы мультипрограммных систем (см. п. 9.1). Огромная трудность разработки автономных программ, работающих в защищенном режиме процессора, заключается в том, что из-за неприменимости в этом режиме подпрограмм BIOS, приходится самому разрабатывать подпрограммы, выполняющие управление периферийными устройствами на уровне портов. Неприятная особенность этой задачи состоит в том, что управление большинством ПУ имеет существенные отличия при работе ЦП в защищенном режиме. Причем эти отличия достаточно громоздки и, самое главное, должным образом не документированы. По этой причине приходится от-

казываться от идеи разработки автономной мультипрограммной системы и использовать одну из готовых мультипрограммных ОС.

Применение готовой мультипрограммной ОС часто неприемлимо для приложений, выполняющих ответственные функции, по следующим причинам. Во-первых, читая любое лизензионное соглашение, нетрудно убедиться, что разработчик системного программного обеспечения не отвечает, фактически, ни за что. За все ошибки в работе программной системы несет ответственность разработчик приложения. Во-вторых, невозможно надежно протестировать огромный объем кода, который имеет современная мультипрограммная ОС. Например, для любой Windows надежно протестировано лишь небольшое ядро системы.

Ситуация существенно упрощается, если наша автономная программная система не должна быть мультипрограммной. В этом случае нам не нужно главное достоинство защищенного режима – защита параллельных программ, а достаточно лишь его свойство обеспечить работу со всем объемом ОП. В этом случае вместо «чистого» защищенного режима достаточно использовать «гибрид» между реальным и защищенным режимами, называемый далее ***режимом линейной адресации***.

Вообще говоря, количество «гибридных» режимов современного процессора чрезвычайно велико. Причиной этого является большое количество управляющих регистров, а также большое число битов в этих регистрах. К сожалению, из-за отсутствия документации на аппаратуру процессора, программист вынужден использовать лишь «чистые» режимы – реальный и защищенный. Причем, как отмечено выше, для использования защищенного режима приходится пользоваться услугами мультипрограммной ОС.

Приятным исключением из этого является режим линейной адресации, впервые предложенный в 1989 г. Томасом Роденом. Суть этого режима заключается в том, что, по крайней мере, один из сегментов данных программы имеет очень большую длину (до 4 Гбайт), позволяющую обрабатывать данные, расположенные в любом месте ОП. В то же время, использование в программе обычных для реального режима сегментов кода рамером 64 Кбайта позволяет использовать в ней любые системные вызовы BIOS. Приятной неожиданностью, которую авторы пособия выявили путем тестирования ряда современных процессоров, оказалось то, что сразу же после включения питания эти процессоры работают не в реальном режиме, а в режиме линейной адресации. Причем речь идет о всех сегментах данных длиной 4 Гбайта.

Поэтому для того, чтобы обрабатывать в своей автономной программе любые ячейки ОП, в ней не требуется выполнять каких-либо действий по переключению процессора в режим линейной адресации, а лишь достаточно использовать некоторые особенности аппаратуры процессора i80386, отсутствовавшие в процессоре i8086. Сюда относятся: 1) 32-битовые регистры данных; 2) 32-битовые регистры-указатели; 3) 32-битовый регистр флагов. Кроме того, для уяснения сути режима линейной адресации полезно рассмотреть аппаратные дескрипторы сегментов, а также некоторые управляющие регистры и таблицы.

25.2. Увеличение разрядности регистров

До сих пор мы использовали лишь аппаратные возможности процессора i8086. Если мы хотим, чтобы наши автономные программы не ограничивались использованием лишь первого мегабайта ОП, то должны использовать аппаратные возможности, предоставляемые более современными процессорами Для решения наших задач вполне доста-

точны аппаратные возможности процессора i80386, которые аппаратно поддерживаются более современными процессорами. Первая интересующая нас особенность процессора i80386 – увеличение длины рабочих регистров и регистров-указателей с 16 до 32 бит.

На рис. 3.29 приведены рабочие регистры, регистры-указатели, а также регистр флагов процессора i80386. Каждый из этих 32-битовых регистров содержит в качестве младшей половины соответствующий 16-битовый регистр процессора i8086.

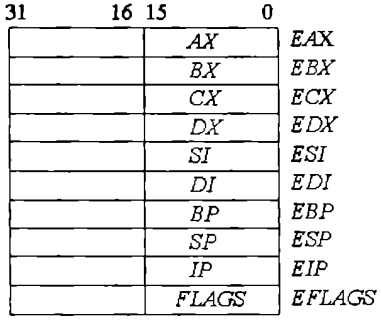

Рис. 3.29. 32-битовые регистры процессора i80386

В обычной программе на ассемблере (автономной или выполняемой в среде DOS) можно использовать 32-битовые рабочие регистры *EAX*, *EBX*, *ECX*, *EDX* без каких либо ограничений. При этом, например, можно поместить в стек или выбрать из него содержимое 32-битового регистра с помощью обычного оператора *push* или *pop*. При этом первым помещается в стек старшее слово регистра, а первым выбирается из стека – младшее слово. Пример:

```
mov  eax, 0000ffffh
push eax
pop  bx        ; BX ← FFFFh
pop  dx        ; DX ← 0000h
```

До сих пор наши программы включали лишь 16-битовые виртуальные сегменты кода. На самом деле, кроме таких сегментов программа может иметь и 32-битовые сегменты кода. Каждый такой сегмент начинается с псевдооператора «***BITS 32***». Если в исходном тексте программы 32-битовый сегмент непосредственно предшествует 16-битовому сегменту, то в начале 16-битового сегмента следует поместить псевдооператор «***BITS 16***». Если самый первый сегмент кода не имеет ни одного из этих двух псевдооператоров, то по умолчанию он считается 16-битным. Именно так обстояло дело во всех наших предыдущих программах.

Применение псевдооператоров, задающих разрядность сегментов кода, зависит от использования псевдооператоров, задающих тип процессора. Каждый такой псевдооператор сообщает о том, что следующий далее исходный текст программы должен содержать операторы, соответствующие указанному процессору. Примеры таких псевдооператоров:

«***CPU 8086***» – процессор i8086;
«***CPU 386***» – процессор i80386;
«***CPU 586***» – процессор Pentium;
«***CPU 686***» – процессор Pentium 2;
«***CPU P3***» – процессор Pentium 3;
«***CPU P4***» – процессор Pentium 4.

Если записан псевдооператор «CPU 8086», то псевдооператор «BITS 32» записан быть не может. Для процессоров, начиная с i80386, отсутствие оператора задания разрядности эквивалентно оператору «BITS 16». Оператор «BITS 32» должен задаваться явно.

Тип разрядности сегмента влияет не только на форму машинных команд, но и на состав этих команд. В 16-битовых виртуальных сегментах кода отсутствуют специальные машинные команды для работы с 32-битовыми регистрами. В этих сегментах для работы с 32-битовыми регистрами используются те же машинные команды, что и для работы с 16-битовыми регистрами, но снабженные специальными приставками (префиксами), уточняющими длину операндов. Например, машинная команда, соответствующая оператору «*mov eax,ebx*» отличается от команды, соответствующей оператору «*mov ax, bx*» наличием байта-префикса с кодом *66h*. В 32-битовом сегменте кода «обычными» являются машинные команды, операнды которых имеют длину не 16, а 32 бита. Команды для 16-битовых операндов имеют соответствующие префиксы. Например, машинная команда, соответствующая оператору «*mov ax,bx*» отличается от команды, соответствующей оператору «*mov eax,ebx*» наличием байта-префикса с кодом *66h*. Так как команды с префиксами выполняются медленнее соответствующих команд без префикса, то 16-битовые сегменты быстрее обрабатывают однословные, а 32-битовые сегменты – двусловные данные. Еще большие различия имеют 16- и 32-битовые сегменты при использовании регистров-указателей.

Нетрудно догадаться, что 32-битовые регистры-указатели нужны для адресации данных, принадлежащим «длинным» сегментам данных, так как для адресации данных, принадлежащих обычным 64 Кбайтным сегментам достаточно использовать 16-битовые регистры *BX, SI, DI, SP*. Оказывается, что команды 16-битового сегмента кода могут адресовать данные, принадлежащие «длинным» сегментам данных, используя 32-битовые регистры *EBX, ESI, EDI, EBP*. При этом обычная 16-битовая команда снабжается префиксом, уточняющим длины регистров. Например, если поместить оператор «*mov eax, [ebx]*» в 32-битовый сегмент, то ему будет соответствовать двух байтная машинная команда. А если поместить этот же оператор в 16-битовый сегмент, то – четырех байтная команда, из которых 2 байта – префиксы, уточняющие длины регистров. Следует добавить, что в 32-битовом сегменте кода для задания адресов-смещений разрешается использовать не четыре, а восемь 32-битовых регистра: *EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP*.

Еще одно отличие работы 16-битового и 32-битового сегментов кода с «длинными» сегментами данных состоит в том, что в 16-битовом сегменте кода нельзя использовать цепочечные (строковые) команды типа *movsb*. Поэтому для обмена данными с «длинным» сегментом приходится выполнять в цикле обычную команду *mov*.

25.3. Аппаратные дескрипторы сегментов

Напомним, что в процессоре i8086 имеются четыре 16-битовых сегментных регистра: *CS, DS, ES, SS*. В процессорах, начиная с i80386 имеются еще два 16-битовых сегментных регистра, которые можно использовать для адресации данных – *GS* и *FS*. В отличие от процессора i8086, каждый сегментный регистр более современного процессора имеет не только программно видимую 16-битовую часть, но и 64-битовую теневую часть. Видимая 16-битовая часть заполняется с помощью обычных команд типа *mov*. Программы реального режима имеют дело только с этими видимыми частями. 64-битовая теневая часть сегментного регистра содержит дескриптор (описатель) сегмента (рис. 3.30).

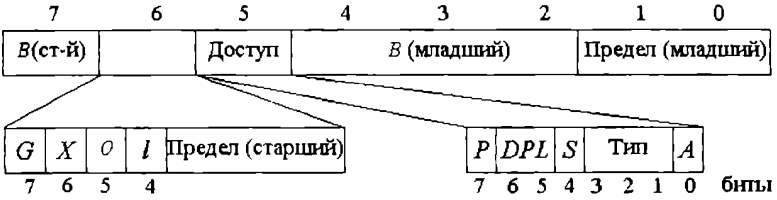

Рис. 3.30. Структура дескриптора сегмента

Дескриптор сегмента имеет длину 8 байт (64 бита), из которых ***базовый линейный адрес сегмента*** (*B*) занимает 4 байта, или 32 бита. При работе процессора в реальном режиме базовый линейный адрес сегмента совпадает с содержимым 16-битовой видимой части регистра сегмента, умноженным на 16. Поэтому, выполняя запись, например, с помощью обычной команды *mov*, в регистр сегмента, мы одновременно меняем содержимое поля *B* теневой части регистра сегмента. Именно содержимое этого поля суммируется со смещением при определении процессором реального адреса искомой ячейки.

Поле ***предел*** содержит размер сегмента в байтах, уменьшенный на 1. В реальном режиме размер сегмента ОП составляет 64 К. Число 64 К–1=65535 находится во всех теневых сегментных регистрах при работе ЦП в реальном режиме.

Бит *G* – ***бит гранулярности***. Если *G*=1, то поле предела задает предельный размер сегмента в страницах (по 4096 байтов), иначе – в байтах. Длина поля предела всего 20 бит и поэтому для того, чтобы использовать весь объем адресного пространства ОП, требуется записать 1 в бит *G*. В реальном режиме процессора размер сегмента всего 64 К и поэтому бит *G* сброшен.

Бит *X* задает размерность машинных команд, выполняемых ЦП. Если *X*=*1*, то выполняются 32-битовые команды, иначе – 16-битовые. Данный бит используется лишь при выполнении на ЦП цепочечных (строковых) команд. В реальном режиме или в режиме линейной адресации бит *X* всегда сброшен.

Бит *l* – свободный бит, используемый системным программистом по своему усмотрению.

Поле (байт) ***доступ*** ограничивает операции, которые можно выполнять над сегментом. При этом бит *S* – признак системного сегмента. Если этот бит сброшен в 0, то сегмент системный. Для сегментов прикладной программы бит *S* должен содержать 1. Если *S* содержит 1, то назначение сегмента описывается полем ***тип***.

Бит *P* – признак присутствия сегмента в ОП: 0 – сегмент «сброшен» на диск; 1 – сегмент находится в ОП. Так как свопинг (перекачка между диском и ОП) нами не используется, то бит *P* содержит 1.

2-х битовое поле *DPL* содержит ***уровень привилегий сегмента***. Наивысший уровень привилегий задается числом 0. Так как в наших программах распределение сегментов по приоритетам не используется, то поле *DPL* должно содержать всегда 0.

Бит *A* отражает текущую работу с сегментом. Если процессор выполняет операцию чтения или записи с данными сегмента, то он устанавливает бит *A* в единицу. Иначе бит *A* содержит 0. Это же значение записывается и при инициализации дескриптора сегмента.

На рис. 3.31 приведен байт доступа для сегмента кода, содержащего машинные команды программы. Здесь бит *C* – ***бит подчинения***. Он влияет на возможность инициирования данного сегмента из других сегментов кода. Если приоритеты сегментов совпадают, как это имеет место у нас, то величина бита *C* не имеет значения. Везде далее *C* =0.

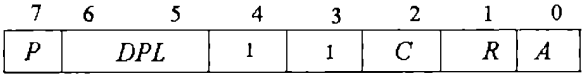

Рис. 3.31. Байт доступа для сегмента кода

R – бит разрешения чтения сегмента (1 – чтение разрешено, 0 – чтение запрещено). Что касается записи, то в сегмент кода она запрещена всегда.

На рис. 3.32 приведен байт доступа для сегмента данных. Здесь бит *D* задает направление расширения сегмента. Обычный сегмент данных расширяется в направлении старших адресов (*D=0*). Если же в сегменте расположен стек, расширение происходит в обратном направлении (*D=1*).

W – бит разрешения записи в сегмент (1 – запись допустима, 0 – нет). Если программа попытается записать в сегмент при *W=0*, то ее выполнение будет прервано. Для наших сегментов данных бит *W* содержит 1.

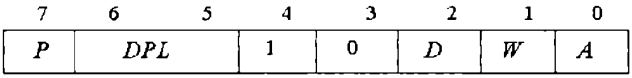

Рис. 3.32. Байт доступа для сегмента данных

25.4. Переход в режим линейной адресации

Несмотря на то, что после включения питания ЦП уже находится в режиме линейной адресации, переход в этот режим требуется в том случае, если система работала до этого в реальном режиме (например, под управлением DOS). Поэтому далее рассматриваются особенности такого перехода.

Когда процессор работает в реальном режиме, теневая часть каждого сегментного регистра программно «невидима», т. е. с помощью команды программы мы не можем прочитать содержимое этой части или записать в нее новое содержимое. Исключением являются 20 младших бита поля *B*, которые копируют программно видимую часть сегментного регистра. Тем не менее, теневая часть сегментного регистра «видима» аппаратно, заставляя аппаратуру ЦП относиться к соответствующему сегменту программы как к стандартному сегменту размером 64 Кбайта. Для того чтобы изменить содержимое любого поля теневой части регистра сегмента, необходимо перейти в защищенный режим работы процессора.

Допустим, что мы хотим сделать «длинным» один из виртуальных сегментов данных. Для этого достаточно скорректировать содержимое двух полей в соответствующем теневом регистре сегмента: 1) записать 1 в бит гранулярности *G*; 2) записать большое число в поле предела. Если мы запишем предельно большое содержимое поле предела (*0FFFFFh*), то получим предельный размер сегмента данных – 4 Гбайт. Сегмент данных такого размера «накрывает» всю ОП и поэтому из двух составляющих логического адреса реального режима (адрес-сегмент и адрес-смещение) для указания требуемой ячейки в программе достаточно задать лишь 32-битовый адрес-смещение. Отсюда вытекает и название режима работы процессора – режим линейной адресации. Что касается адреса-сегмента, то в качестве него (и содержимого поля *B*) часто помещают 0. В результате этого адрес-смещение ячейки ОП совпадает с ее реальным адресом. Напомним, что для размещения 32-битового адреса-смещения в 16-битовой программе можно использовать четыре, а в 32-битовой программе – восемь 32-битовых регистров.

Что касается выбора виртуального сегмента данных, который более подходит на роль «длинного» сегмента, то лучше подходит дополнительный сегмент данных, адресуемый или регистром *FS*, или регистром *GS*. Причиной этого является то, что ни DOS, ни BIOS эти регистры не используют. Поэтому, эти системные программы не изменят содержимое поля *B* дескриптора сегмента путем записи (команда *mov*) в видимую часть регистра сегмента *FS* или *GS*. Аналогично, наши программы не должны выполнять никаких действий с этими регистрами после того, как один из них или они оба будут настроены в защищенном режиме на «длинные» сегменты данных. Единственно допустимый способ использования *FS* и (или) *GS* заключается в явном задании сегмента при адресации данных. Пример такого использования:

```
mov  [fs: esi], eax
```

Если правильно использовать *FS* (*GS*) в программе, то после того, как эта программа завершится, этот же сегментный регистр можно продолжать использовать для адресации данных «длинного» сегмента в другой программе, не выполняя «перенастройку» сегментного регистра. Это может продолжаться до выключения питания. При этом неважно, выполняются ли автономные программы, или средой выполнения является DOS.

Из всех управляющих структур данных процессора i80386 сейчас нас будут интересовать регистры и таблицы, используемые для перевода этого процессора в режим линейной адресации. Укрупненный алгоритм такого перехода:

Ш а г 1. Подготовка к переходу процессора в защищенный режим.

Ш а г 2. Перевод процессора в защищенный режим.

Ш а г 3. Изменение теневой части регистра *FS* или *GS*.

Ш а г 4. Перевод процессора в режим линейной адресации.

В защищенном режиме содержимое «видимой» 16-битовой части сегментного регистра совершенно меняется. Данная перемена настолько серьезна, что меняется само название регистра – сегментный регистр становится ***селектором сегмента***. Структура селектора сегмента приведена на рис. 3.33.

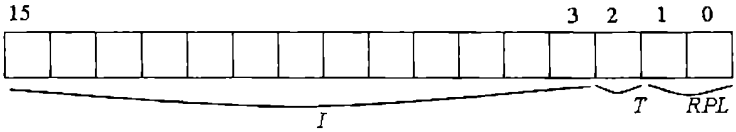

Рис. 3.33. Структура селектора сегмента: *I* – индекс дескриптора сегмента в таблице *GDT* или *LDT*; *T* – тип таблицы дескрипторов (*0* – *GDT*, *1* – *LDT*); *RPL* – уровень запрашиваемых привилегий

Селектор сегмента управляет заполнением 64-битовой теневой части сегментного регистра при выполнении команды записи в регистр *mov* или *pop*. В отличие от реального режима каждая из этих команд выполняет запись не только в младшие 20 бит поля *B*, но и во все остальные биты теневой части сегментного регистра. При этом бит *T* селектора сегмента указывает – из какой таблицы дескрипторов должен копироваться дескриптор сегмента в теневую часть регистра сегмента. При *T*=0 дескриптор сегмента копируется из таблицы *GDT*. При *T*=1 – из таблицы *LDT*.

GDT – глобальная таблица дескрипторов. В качестве нее может использоваться любая область ОП, длина которой кратна 64 байтам. Обратим внимание, что размер поля индексов *I* в селекторе сегмента составляет 13 бит. Поэтому максимальная длина (в байтах) *GDT* или *LDT* равна 2^{13} = 8192. Так как длина одного дескриптора равна 8 байтам, то максимальное число дескрипторов в таблице *GDT* или *LDT* будет: 8192/8=1024=1 K.

Такое максимальное число сегментов памяти может иметь конкретная прикладная программа, или ядро ОС.

Задание таблицы *GDT* производится в реальном режиме процессора, на этапе подготовки перехода процессора в защищенный режим. Первым этапом задания *GDT* является резервирование подходящей области ОП, а также инициализация (заполнение) этой области. Данный этап выполняется транслятором при обработке псевдооператоров исходной программы. Пример таких псевдооператоров:

```
Gdt:
; Первый дескриптор любой GDT или LDT пустой – содержит одни нули
    db  0, 0, 0, 0, 0, 0, 0, 0
; Дескриптор сегмента кода (выполнение, чтение, 32-битовые команды)
    db  0ffh            ; Байт 0: предел сегмента 4 Гбайта
    db  0ffh            ; Байт 1:       –"–
    db  00h             ; Байт 2:  база сегмента – 0
    db  00h             ; Байт 3:       –"–
    db  00h             ; Байт 4:       –"–
    db  10011010b       ; Байт 5: в ОП; DPL=0; несистемный; код;
                        ;   чтение разрешено
    db  11001111b       ; Байт 6: предел в страницах; 32-битовый код;
                        ; предел – наибольший
    db  00h             ; Байт 7: старший байт базы
; Дескриптор сегмента данных (рост вверх, запись разрешена)
    db  0ffh            ; Байт 0: предел сегмента 4 Гбайта
    db  0ffh            ; Байт 1:       –"–
    db  00h             ; Байт 2:  база сегмента – 0
    db  00h             ; Байт 3:       –"–
    db  00h             ; Байт 4:       –"–
    db  10010010b       ; Байт 5: в ОП; DPL=0; несистемный; данные;
                        ;   увеличение – вверх; запись разрешена
    db  11001111b       ; Байт 6: предел в страницах; 32-битовый код;
                        ;   предел – наибольший
    db  00h             ; Байт 7: старший байт базы
```

При рассмотрении данного примера следует учесть, что в программе сначала располагается младший байт дескриптора, а на рис. 3.30, наоборот, младший байт расположен справа. В качестве базового линейного адреса для обоих сегментов (кода и данных) выбран нулевой адрес. Что касается предельной длины сегментов, то она тоже одинакова (4 Гбайта). Следовательно, оба дескриптора описывают один и тот же сегмент памяти. В первом случае этот сегмент рассматривается как сегмент кода, а во втором – как сегмент данных. В результате программа имеет по отношению к своей собственной памяти неограниченные права доступа.

Для того чтобы найти *GDT* в памяти, ЦП имеет специальный 6-байтовый регистр ***GDTR***. Этот регистр содержит 4-байтовый реальный адрес первого байта *GDT* и 2-байтовый размер этой таблицы. Для заполнения регистра *GDTR* в программе используется специальная машинная команда ***lgdt***. Данная команда (и соответствующий оператор ассемблера) имеет единственный операнд – указатель (сегмент:смещение) на

6-байтовое поле, содержащее требуемое содержимое *GDTR*. В операторе указатель заключается в квадратные скобки. Пример применения оператора *lgdt*:

```
        mov   ax, cs              ; AX ← CS
        mov   cl, 4               ; CL ← величина сдвига
        shl   ax, cl              ; Умножение (AX) на 16
        add   ax, Gdt             ; AX ← реальный адрес Gdt
        mov   (Gd_reg+2), ax      ; Запись реального адреса Gdt
        lgdt  [Gd_reg]            ; Загрузка регистра GDTR
. . . . . .
Gd_reg: dw    24                  ; Длина таблицы Gdt
        dd    0                   ; Для реального адреса начала Gdt
```

В мультипрограммной системе дескрипторы сегментов каждой параллельной программы содержатся в своей ***LDT*** – в ***локальной таблице дескрипторов***. Дескриптор каждой *LDT* содержится в *GDT*. Специальный регистр ***LDTR*** содержит индекс (смещение относительно начала *GDT*) для дескриптора *LDT* выполняемой в данный момент программы. Если наша система однопрограммная, то дескрипторы и системных, и прикладных сегментов следует хранить в единой таблице *GDT*, а таблицы *LDT* не используются.

Что касается поля ***RPL*** селектора сегмента (см. рис. 3.33), то оно используется для проверки правомочности текущего сегмента кода по выполнению команды записи (*mov* или *pop*) содержимого селектора сегмента. Если все наши сегменты кода (системные и прикладные) имеют одинаковый приоритет – 0, то поле *RPL* каждого селектора сегмента также содержит одинаковое число, например, число 0.

Перевод процессора в защищенный режим выполняется путем установки в 1 бита 0 в 32-битовом управляющем регистре *CR0*. Этот бит управляет переключением режима процессора: *1*-защищенный режим; *0*-реальный режим. Следующая группа машинных команд выполняет переключение ЦП из реального режима в защищенный режим:

```
mov   ax, cr0      ; AX ← CR0
or    ax,1         ; Установка бита 0  AX в 1
mov   cr0, ax      ; CR0 ← AX
```

После того, как изложенные выше действия (подготовка *GDT* и перевод ЦП в защищенный режим) выполнены, изменение теневой части регистра *FS* (или *GS*) производится очень просто. Для этого достаточно с помощью команды *mov* записать в регистр *FS* (или *GS*) содержимое 16-битового поля ОП, содержащего индексную часть (поле *I*) требуемого селектора сегмента. Что касается двух других полей селектора (*T* и *RPL*), то их запишет в селектор сам ЦП, получив требуемые данные из дескриптора выполняемого сегмента кода. Например, следующие операторы загружают в теневую часть регистра *FS* дескриптор сегмента данных из таблицы *Gdt*:

```
FS_selektor  equ    0008h          ; Индексная часть селектора
. . . . . .
             mov    ax, FS_selektor
             mov    fs, ax
```

После того, как теневая часть регистра *FS* (или *GS*) изменена, необходимо вернуться в реальный режим, сбросив бит 0 регистра *CR0*. Теперь любые программы будут находиться в режиме линейной адресации (по отношению к *FS* или *GS*) до тех пор, пока эти

сегментные регистры используются лишь для адресации данных. Первая же операция записи в такой регистр вернет процессор в обычный реальный режим.

В завершение напомним, что если наша программа является автономной, т. е. запускается сразу же после включения питания, то никакая перенастройка сегментов данных не требуется, так как нужное содержимое теневой части каждого сегментного регистра записывается чисто аппаратно.

25.5. Совершенствование редактора

До сих пор наш шестнадцатеричный редактор мог выполнять редактирование 256-байтовых секторов, входящих в состав 64-Кбайтового сегмента памяти, в котором располагается сам редактор. Теперь мы переделаем редактор так, чтобы можно было редактировать любой сегмент ОП. Это нетрудно сделать с учетом того, что виртуальные сегменты данных программы могут иметь размер, не уступающий размеру всей ОП. Для работы с такими сегментами некоторые процедуры редактора должны быть скорректированы. В частности, некоторые 16-битовые регистры, используемые для хранения внутрисегментных смещений, должны быть заменены на 32-битовые регистры. Ниже перечислены изменения, которые требуется внести в исходные файлы редактора.

1. Изменения в файле *Redaktor.asm*:

- Требуется увеличить длину переменной *Address* с одного до двух слов. Теперь эта переменная содержит не адрес-смещение сектора относительно начала сегмента памяти, а начальный реальный адрес сектора в ОП.

  Увеличить длину переменной *N* с одного байта до двух слов. Теперь *N* содержит номер сектора во всей ОП.

2. Изменения в файле *Disp_sec.asm*:

- Корректировка процедур *Init_sector*, *Prev_sector*, *Next_sector*, *N_sector* с учетом новой длины переменных *Address* и *N*. При этом в процедуре *N_sector* вызов процедуры *Read_byte_hex* (ввод двухзначного шестнадцатеричного числа) следует заменить на вызов процедуры *Read_N* (ввод номера сектора). Текст *Read_N* будет приведен ниже.

  Корректировка процедуры *Disp_line* с учетом увеличения длины переменной *Address*. При этом вызов процедуры *Write_word_hex* (вывод 4-значного шестнадцатеричного числа) следует заменить на вызов процедуры *Write_dword_hex* (вывод 8-значного шестнадцатеричного числа). Кроме того, вместо регистра *DX* следует использовать регистр *EDX*. Процедуру *Write_dword_hex* нетрудно получить, используя вызовы процедуры *Write_byte_hex*.

  Корректировка процедур *Read_sector* и *Write_sector*. При этом, во-первых, регистры *si* и *di* заменяются на регистры *esi* и *edi*. Во-вторых, цепочечный оператор «*rep movsb*» заменяется на цикл, реализованный с помощью оператора *loop*. Причина такой замены заключается в том, что в 16-битовой программе при работе с «длинными» сегментами данных цепочечные (строковые) операторы использоваться не могут.

3. В файл *Video_io.asm* добавить текст процедуры *Write_dword_hex*.

4. В файле *Cursor.asm* выполнить коррекцию процедуры *Set_cursor*: заменить оператор «*add dl, 7*» на оператор «*add dl, 11*». Данная замена обусловлена тем, что адрес байта ОП занимает теперь на экране не 4, а 8 позиций.

5. Изменения в файле *Kbd_io.asm* сводятся к добавлению процедуры *Read_N*, а также к корректировке процедуры *Read_digit_hex*. Текст процедуры *Read_N*:

```
;   Ввод с клавиатуры 1–6 разрядного шестнадцатеричного числа
;   ----------------------------------------------------------
;   Выходы: EDX – содержит введенное число
;
Read_N: push  bx
        push  cx
        xor   edx, edx          ; EDX ← 0
        mov   cx, 6             ; Счетчик цифр числа
.B:     call  Read_digit_hex    ; Ввод цифры
        cmp   bl, 10h           ; Нажата <Enter>?
        jz    .V                ; Если да
        shl   edx, 4            ; Сдвиг EDX на одну цифру влево
        add   dl, bl            ; Добавление цифры в EDX
        loop  .B                ; За новой цифрой
.V:     pop   cx
        pop   bx
        ret
```

Данная процедура *Read_N* позволяет вводить с клавиатуры любое шестнадцатеричное число с количеством цифр от одного до шести. Это весьма удобно, так как при наборе короткого номера сектора пользователю не требуется набирать все 6 цифр, а достаточно нажать <Enter>.

В приведенном тексте *Read_N* выполняется вызов измененной процедуры *Read_digit_hex*. Такая процедура возвращает в своем выходном регистре *BL* или величину шестнадцатеричной цифры, или число *10h*. Это число возвращается в том случае, если пользователь нажал <Enter>. В самой процедуре *Read_digit_hex* нажатие клавиши <Enter> определяется по коду *0dh* (возврат каретки).

Лабораторная работа 23

Требуется получить программу автономного шестнадцатеричного редактора, выполняющего редактирование всего адресного пространства ОП. Данный редактор должен правильно выполняться на ВМ «Virtual PC».

При наличии у вас реального компьютера с дисководом гибких дисков, осуществить загрузку и выполнение указанного выше шестнадцатеричного редактора, используя свою загрузочную дискету.

Приложение 1. Основные операторы ассемблера

П1.1. Типы операторов

Любой алгоритмический язык программирования, в том числе и ассемблер, имеет операторы следующих типов.

Исполнительные операторы. Данные операторы преобразуются транслятором в машинные команды. Один исполнительный оператор ассемблера преобразуется в одну машинную команду. А один исполнительный оператор языка высокого уровня транслируется в несколько машинных команд. Все исполнительные операторы языка программирования делятся на операторы обработки данных и операторы передачи управления. Операторы обработки данных влияют на содержимое ячеек памяти (ячейки ОП, регистры, флаги), а операторы передачи управления изменяют ход выполнения программы.

Псевдооператоры определения данных. В отличие от исполнительного оператора псевдооператор ни в какие машинные команды не транслируется, а представляет собой указание транслятору со стороны программиста. Псевдооператор определения данных требует от транслятора выделить область памяти заданной длины. Кроме того, он может попросить транслятор поместить в выделенную область какие-то первоначальные данные. Впоследствии на этапе выполнения сама программа может менять содержимое этой области.

Другие псевдооператоры. Они информируют транслятор о структуре программы, помогая транслятору и редактору связей правильно преобразовать исполнительные операторы в машинные команды.

Макрооператоры. Каждый такой оператор заменяется транслятором на несколько обычных операторов языка программирования (в том числе, возможно, и псевдооператоров).

Комментарии. Это любые сообщения в исходной программе, предваряемые специальным символом. В рассматриваемом языке ассемблера это символ «;». Комментарии игнорируются транслятором и никак не влияют на текст машинной программы.

П1.2. Операторы обработки данных

Операторы обработки данных делятся:

1) на арифметические операторы;
2) логические операторы;
3) операторы передачи данных;
4) операторы манипуляций флажками;
5) операторы сдвигов;
6) цепочечные (строковые) операторы.

Далее рассматриваются эти типы операторов, а также адресация данных и псевдооператоры определения данных.

П1.2.1. Арифметические операторы

Арифметические машинные команды (и соответствующие операторы) делятся на ***двоичные*** и ***двоично-кодированные десятичные***. Второй из этих классов используется редко, так как применяемый в нем способ кодирования данных неэффективен по затратам памяти.

В свою очередь, двоичные арифметические операторы разделяются на ***знаковые*** и ***беззнаковые***. Первые из них выполняют операции как над положительными, так и над отрицательными двоичными числами, в то время как беззнаковые операторы имеют дело только с положительными числами. Рассмотрим основные типы операторов ассемблера, выполняющие операции над двоичными числами.

Сложение

1. ***add*** (***сложить***) суммирует два операнда (слова или байты). Результат записывается на место первого операнда. Примеры:

```
add     ax, [Mem]   ; (AX) + (Mem) →AX  , Mem - слово в ОП
add     [Mem], ax   ; (Mem) + (AX) →Mem
add     al, 40      ; (AL) + 40 → AL
add     [Mem], 0fh  ; (Mem) + 0fh → Mem
```

Запрещается суммировать содержимое двух ячеек ОП, а также записывать в качестве первого операнда непосредственное значение.

2. ***adc*** (***сложить с переносом***) суммирует два операнда (слова или байты), а также флаг переноса *CF*. Результат помещается на место первого операнда.

Совместное применение операторов *add* и *adc* позволяет выполнить суммирование двух чисел даже тогда, когда результат не вмещается в 16 битов. Например, следующие два оператора складывают 32-битовое число, находящееся в регистрах *CX* и *DX*, с 32-битовым числом, находящимся в регистрах *AX* и *BX*. Результат записывается в регистры *AX* и *BX*:

```
add     ax, cx      ; Суммируются младшие 16 бит
adc     bx, dx      ; Суммируются старшие 16 бит, а также
                    ;   перенос от предыдущего суммирования
```

3. ***inc*** (***инкремент***) увеличивает операнд на 1. Пример:

```
inc     ax          ; (AX) + 1 → AX
```

Операторы вычитания

1. ***sub*** (***вычесть***) выполняет вычитание второго операнда из первого операнда (операнды – байты или слова). Результат помещается в качестве первого операнда. Примеры:

```
sub     ax, cx      ; (AX) – (CX) →AX
sub     ax, [Mem]   ; (AX) – (Mem) →AX
sub     [Mem], ax   ; (Mem) – (AX) →Mem
sub     al, 10      ; (AL) – 10 → AL
```

Запрещается брать в качестве обоих операндов ячейки ОП, а также задавать в качестве первого операнда непосредственное значение;

2. ***dec*** (***декремент***) уменьшает операнд на 1. Пример:

```
dec     ax          ; (AX) - 1 → AX
```

3. ***neg*** (***изменить знак***) вычитает из нулевого значения значение операнда. Результат записывается на место операнда. Пример:

```
neg     ax          ; - (AX) → AX
```

4. ***cmp*** (***сравнить два операнда***) выполняет вычитание 1-го и 2-го операндов, но в отличие от оператора *sub* результат вычитания никуда не записывается, а лишь используется для установки флажков. Примеры:

```
cmp     ax, bx      ; (AX) – (BX)
cmp     [Mem], ah   ; (Mem) – (AH)
cmp     al, 10      ; (AL) – 10
```

Операторы умножения

1. ***mul*** (***умножить***) выполняет умножение двух беззнаковых чисел (слов или байтов). Единственный операнд содержит один из сомножителей и представляет собой регистр общего назначения или ячейку памяти размером в байт или слово. В качестве второго сомножителя используется содержимое регистра *AL* (в операциях над байтами) или регистра *AX* (в операциях над словами).

16-битовое произведение байтов помещается в регистры *AH* (старший байт) и *AL* (младший байт). 32-битовое произведение слов помещается в регистры *DX* (старшее слово) и *AX* (младшее слово). Примеры:

```
mul   bx                ; Умножить BX на AX без знака
mul   word [Mem_wor]    ; Умножить содержимое ячейки на AX
                        ;   без знака
mul   dl                ; Умножить DL на AL без знака
```

2. ***imul*** (***умножить целые числа***) выполняет умножение двух знаковых чисел (слов или байтов). Правило размещения сомножителей и результата аналогично оператору *mul*.

Операторы деления

1. ***div*** (***разделить***) выполняет деление чисел без знака. Единственный операнд представляет собой регистр общего назначения или ячейку памяти (байт или слово) и содержит делитель. Делимое должно иметь двойной размер; оно извлекается из регистров *AH* и *AL* (при делении на байт) или из регистров *DX* и *AX* (при делении на слово).

Результат возвращается следующим образом. Если делитель представляет собой байт, то частное возвращается в регистре *AL*, а остаток в регистре *AH*. Если делитель представляет собой слово, то частное возвращается в регистре *AX*, а остаток в регистре *DX*. Примеры:

```
div   bx                ; Разделить DX:AX на BX , без знака
div   byte [Mem_byte]   ; Разделить AH:AL на байт ОП
```

2. ***idiv*** (**разделить целые числа**) выполняет деление чисел со знаком. Правило записи делимого, делителя и результата аналогично *div*.

П1.2.2. Логические операторы

Они используются для установки и сброса битов в слове или в байте. В качестве операндов могут выступать два регистра, регистр с ячейкой ОП или непосредственное значение с регистром или ячейкой ОП.

1. ***and*** (***логическое И***) сравнивает два операнда (слова или байты) побитно. Результат записывается на место первого операнда. Если оба из сравниваемых битов равны 1, то результат равен 1, во всех остальных случаях результат равен 0. Например, пусть (*AL*) = = 10101111, а (*BH*) = 11100000, тогда «*and al, bh*» запишет в *AL* 10100000. Примеры:

```
and   ax, bx                ; Операция AND над двумя регистрами
and   ax, [Mem_wor]         ; Операция AND над регистром и словом ОП
and   [Mem_byte], al        ; Операция AND над байтом ОП и регистром
and   bl, 11110000b         ; Обнуление младших 4 битов регистра BL
and   [Mem_byte], 00001111b ; Обнуление старших 4 битов байта ОП
```

2. ***or*** (***логическое ИЛИ***) сравнивает два операнда (слова или байты) побитно. Если хотя бы один из сравниваемых битов равен 1, то результат равен 1, если оба сравниваемых

мых бита равны 0, то результат равен 0. Например, пусть (*AL*) =10101111, а (*BH*) = = 11100000, тогда «*or al, bh*» запишет в *AL* 11101111. Примеры:

```
or   bl, 11110000b          ; Установка в 1 старших 4 и сохранение
                            ;    значений 4 младших битов регистра BL
or   [Mem_byte], 00001111b  ; Сохранение значений 4 старших и установка
                            ;    в 1 младших 4 бит байта ОП
```

3. ***xor* (*исключающее ИЛИ*)** сравнивает два операнда (слова или байты) побитно. Если один из сравниваемых битов равен 0, а другой 1, то результат есть 1, если оба сравниваемых бита одинаковы (оба – 0 или оба – 1), то результат есть 0.

4. ***test* (*проверить*)** выполняет логическое побитовое умножение своих двух операндов (байтов или слов), но в отличие от *and* результат никуда не записывается, а лишь используется для установки флажков. Оба операнда остаются без изменения. Например, оператор:

```
test   bl, 11110000b
```

установит флаг нуля в 1 только тогда, когда ни один из четырех старших битов в *BL* не равен 1.

5. ***not* (*НЕ*)** инвертирует все биты в операнде: вместо 0 пишет 1, а вместо 1– 0.

П1.2.3. Операторы передачи данных

Они осуществляют обмен данными между регистрами и ячейками памяти.

1. ***mov* (*переслать*)** пересылает содержимое второго операнда (слово или байт) в качестве содержимого первого операнда. Можно пересылать байт или слово между регистром и ячейкой памяти или между двумя регистрами. А также можно помещать непосредственное значение в регистр или в ячейку памяти. При пересылке непосредственного значения в ячейку памяти необходимо указать длину пересылаемого данного с помощью вспомогательного псевдооператора *byte* (байт) или *word* (слово). Примеры:

```
mov    ax, [Table]       ; Пересылка из ячейки ОП в регистр
mov    [Table], ax       ;    и наоборот
mov    bl, al            ; Пересылка между регистрами
mov    cl, -30           ; Загрузка константы в регистр
mov    [Mem], byte 25h   ;    или в ячейку ОП
```

Следует отметить, что с помощью данного оператора нельзя выполнять пересылку данных из одной ячейки ОП в другую ячейку ОП. Такая пересылка может быть осуществлена двумя операторами *mov* с использованием регистра общего назначения.

С помощью оператора *mov* можно загружать в регистр общего назначения (но не в регистр сегмента) номер начального параграфа сегмента, задав в качестве второго операнда имя сегмента. Например:

```
mov    ax, Data_seg  ; Data_seg – имя сегмента данных
```

Для загрузки в регистр сегмента (кроме *CS*) номера начального параграфа сегмента требуются два оператора *mov*. Например:

```
mov    ax, Data_seg  ; Загрузка в AX
mov    ds, ax        ; Пересылка в регистр DS
```

Что касается регистра сегмента кода *CS*, то он не может быть использован в качестве первого операнда в операторе *mov*. Кроме того, нельзя пересылать содержимое одного

регистра сегмента в другой. Такая пересылка может быть осуществлена двумя операторами *mov* с использованием регистра общего назначения.

2. ***xchg*** (***обменять***) производит обмен содержимого двух регистров или регистра и ячейки ОП. Следующие два фрагмента программ делают одно и то же – меняют местами содержимое двух байтов в ОП – *Opr1* и *Opr2*:

```
а) mov   al, [Opr1]     ;  (Opr1) → AL
   mov   bl, [Opr2]     ;  (Opr2) → BL
   mov   [Opr2], al     ;  (AL) → Opr2
   mov   [Opr1], bl     ;  (BL) → Opr1
б) mov   al, [Opr1]     ;  (Opr1) → AL
   xchg  al, [Opr2]     ;  (Opr2) ↔AL
   mov   [Opr1], al     ;  (AL) → Opr1
```

В варианте (б) потребовалось на один оператор и на один регистр меньше, чем в (а).

3. ***push*** (***поместить слово в стек***) помещает содержимое регистра или ячейки ОП размером в 16-битовое слово на вершину стека. Примеры:

```
push   ax          ; (AX)→ стек
push   [Mem]       ; (Mem) →стек
```

4. ***pop*** (***извлечь слово из стека***) выбирает слово из вершины стека и помещает его в ячейку памяти или в регистр. Пример:

```
pop    ax          ; Слово из вершины стека → AX
```

5. ***lea*** (***загрузить адрес***) пересылает адрес-смещение ячейки памяти в любой 16-битовый регистр данных, регистр-указатель или индексный регистр. Оператор имеет два операнда. В качестве первого из них записывается регистр, а в качестве второго – метка ячейки ОП. Пример:

```
lea    bx, [Mem]   ; Адрес-смещение ячейки Mem → BX
```

Следует заметить, что данный оператор можно заменить оператором:

```
mov    bx, Mem     ; Адрес-смещение Mem → BX
```

6. ***lodsb*** (***чтение байта из строки***) копирует байт ОП по адресу *DS:SI* в регистр *AL* и увеличивает содержимое регистра *SI* на 1. Соответствующая машинная команда однобайтовая.

7. ***stosb*** (***запись байта в строку***) копирует байт из регистра *AL* в ОП по адресу *ES:DI* и увеличивает содержимое регистра *DI* на 1. Команда однобайтовая.

П1.2.4. Операции над регистром флагов

Большинство операторов не только выполняют действия над своими операндами, но и выполняют действия над флажками – битами регистра флагов *FLAGS* (рис. П.1).

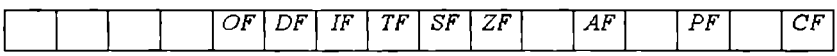

Рис. П.1. Регистр флагов микропроцессора i8086

Существуют 7 операторов, которые предназначены только для манипуляций флажками *FLAGS*. Они не имеют операндов и позволяют изменять флаги *CF*, *DF* и *IF*. Это:

stc – устанавливает флаг переноса *CF*;

clc – сбрасывает *CF*;

cmc – инвертирует *CF*;
std – устанавливает флаг направления *DF*;
cld – сбрасывает *DF*;
sti – устанавливает флаг разрешения прерываний *IF*;
cli – сбрасывает *IF*.

Следующие операторы выполняют пересылку содержимого регистра *FLAGS*:
lahf – пересылает младший байт *FLAGS* в регистр *AH*;
sahf – пересылает содержимое регистра *AH* в *FLAGS*;
pushf – записывает содержимое *FLAGS* в стек;
popf – выбирает слово из вершины стека и помещает его в регистр *FLAGS*.
Нетрудно заметить, что эти операторы позволяют изменять содержимое *FLAGS*.

П1.2.5. Операторы сдвига

Такие операторы перемещают все биты первого операнда (байта или слова) влево или вправо на число, заданное вторым операндом. В процессоре i8086 вторым операндом может быть только 1 или регистр *CL*, но, начиная с процессора *i80186*, в качестве второго операнда может использоваться как регистр *CL*, так и число, не меньшее 1.

Для всех восьми операторов сдвига флаг переноса *CF* является как бы расширением сдвигаемого операнда: в *CF* загружается значение бита, выдвинутого за пределы операнда.

Операторы сдвига разделяются на логические и арифметические. ***Логический оператор сдвига*** рассматривает знаковый бит операнда как обычный бит, а ***арифметический оператор сдвига*** обрабатывает бит знака особо.

1. ***shl – логический сдвиг влево***. Этот оператор сдвигает число без знака (рис. П.2). При каждом сдвиге в освободившийся нулевой бит заносится 0. Например, пусть (*AL*) = *10110100b*, *CF* = *0*, тогда:

```
shl al, 1      ; 01101000 → AL,  1 → CF
```

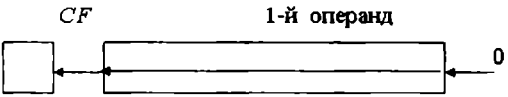

Рис. П.2. Логический сдвиг влево *shl*

Одним из применений оператора *shl* является умножение беззнаковых чисел на степень числа 2. Например, пусть (*CL*) = 2, тогда:

```
shl  ax, cl    ; Умножение числа без знака в AX на 4
```

По сравнению с обычным умножением время выполнения в 6–8 раз меньше.

2. ***sal – арифметический сдвиг влево***. Этот оператор сдвигает число со знаком. Действие аналогично *shl*. При этом содержимое знакового бита не сохраняется, но оно переписывается в флажок *OF*. Например, пусть (*AL*) = *10110100b*, *CF* = *0*, *OF* = *0*, тогда:

```
sal al, 1      ; 01101000 → AL,  1 → CF,  1 → OF
```

3. ***shr – логический сдвиг вправо***. Этот оператор сдвигает число без знака. При каждом сдвиге операнда в освободившийся старший бит (бит 7 для байта и бит 15 для слова) заносится 0 (рис. П.3). Например, пусть (*AL*) = *10110100b*, *CF* = *1*, тогда:

```
shr al, 1      ; 01011010 → AL,  0 → CF
```

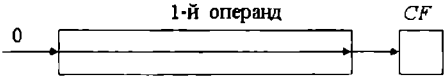

Рис. П.3. Логический сдвиг вправо *shr*

Одним из применений оператора *shr* является деление беззнаковых чисел на степень числа 2. Например, пусть (*CL*) = 2, тогда:

```
shr ax, cl    ; Деление беззнакового числа в AX на 4
```

4. ***sar – арифметический сдвиг вправо***. Данный оператор сдвигает число со знаком. При сдвиге в старшие освобождающиеся биты дублируется знак операнда (рис. П.4). Например, пусть (*AL*) = *10110100b*, *CF* = *1*, тогда:

```
sar al, 1     ; 11011010 → AL,  0 → CF
```

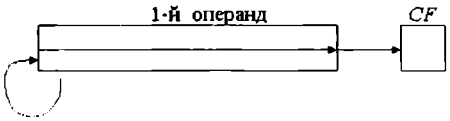

Рис. П.4. Арифметический сдвиг вправо *sar*

Одним из применений оператора *sar* является деление чисел со знаком на степень числа 2. Например, пусть *(CL) = 3*, тогда:

```
sar ax, cl    ; Деление числа со знаком в AX на 8
```

5. ***rol – циклический сдвиг влево***. При выполнении данного оператора (как и любого другого циклического оператора) вышедший за пределы операнда бит входит в него с противоположного конца (рис. П.5). Например, пусть *(AL) = 10110100b*, *CF = 0*, тогда:

```
rol al,1      ; 01101001 → AL,    1 → CF
```

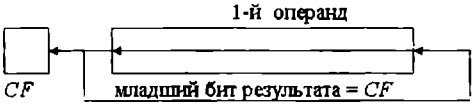

Рис. П.5. Циклический сдвиг влево *rol*

6. ***ror – циклический сдвиг вправо*** (рис. П.6). Например, пусть *(AL) = 10110100b*, *CF = 1*, тогда:

```
ror al, 1     ; 01011010 → AL,  0 → CF
```

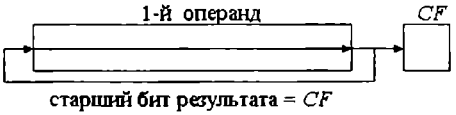

Рис. П.6. Циклический сдвиг вправо *ror*

7. ***rcl – циклический сдвиг влево через перенос*** (рис. П.7). Например, пусть *(AL) = = 10110100b*, *CF = 1*, тогда:

```
rcl al, 1     ; 01101001 → AL,    1 → CF
```

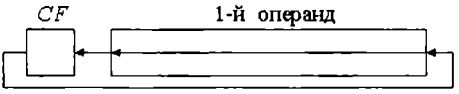

Рис. П.7. Циклический сдвиг влево через перенос *rcl*

8. ***rcr – циклический сдвиг вправо через перенос*** (рис. П.8). Например, пусть *(AL) = 10110100b*, *CF = 1*, тогда:

```
rcr  al,1      ; 11011010 → AL,  0 → CF
```

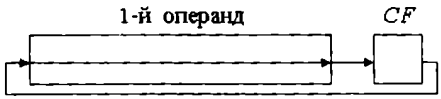

Рис. П.8. Циклический сдвиг вправо через перенос *rcr*

П1.2.6. Цепочечные (строковые) операторы

Такие операторы предназначены для того, чтобы обрабатывать одной командой последовательности из нескольких байт или слов. В результате программа становится короче.

Существуют пять групп цепочечных операторов. Из них далее рассматриваются лишь операторы пересылки строк. Это следующие операторы.

1. ***movsb – переслать цепочку байт***. Оператор не имеет операндов. Ему может предшествовать префикс ***rep***.

При отсутствии префикса оператор копирует всего один байт, расположенный в сегменте данных, в байт, принадлежащий дополнительному сегменту. Как задать байт в сегменте длиной 64 Кбайт? Для этого используются индексные регистры *SI* и *DI*. Регистр *SI* содержит номер (индекс) байта в основном сегменте данных, а *DI* – номер байта в дополнительном сегменте данных. В результате выполнения оператора происходит не только копирование байта, но и изменение на единицу содержимого регистров *SI* и *DI*. Направление изменения (увеличение или уменьшение) определяется флагом направления *DF* в регистре *FLAGS*. Если *DF = 0*, то адреса увеличиваются, а если 1, то уменьшаются.

Добавление к оператору *movsb* (как и к любому другому строковому оператору пересылки) префикса повторения команды *rep* многократно усиливает мощность строковой команды. При этом оператор *movsb* выполняется столько раз, каково содержимое регистра *CX*. При каждом выполнении содержимое *CX* уменьшается на единицу. Как только это содержимое станет равно нулю, то выполняется следующий за *movsb* оператор. Применение оператора *movsb* с префиксом *rep* требует выполнения следующих пяти шагов:

1) обнулить флаг *DF* (оператором *cld*) или установить его (оператором *std*) в зависимости от того, будет ли пересылка осуществляться от младших адресов к старшим или наоборот;

2) загрузить смещение адреса строки-источника в регистр *SI*;

3) загрузить смещение адреса строки-приемника в регистр *DI*;

4) загрузить число пересылаемых байтов в регистр *CX*;

5) выполнить оператор *movsb* с префиксом *rep*.

Пример 1. Следующий фрагмент программы копирует 100 байт из строки *Source*, находящейся в сегменте данных, в строку *Dest*, находящуюся в дополнительном сегменте.

```
      cld                ; 0 → DF
      lea   si, [Source] ; Адрес-смещение Source → SI
      lea   di, [es:Dest]; Адрес-смещение Dest → DI
      mov   cx, 100      ; Число пересылаемых байтов
rep   movsb              ; Копирование цепочки байтов
```

Пример 2. Следующая процедура выполняет копирование цепочки 256 байтов, находящейся в сегменте данных, в цепочку этой же длины, но расположенную в дополнительном сегменте.

```
;                Копирование 256 байт
;                --------------------
;   Входы :  SI – содержит смещение адреса цепочки-источника
;            DI – содержит смещение адреса цепочки-приемника
;
        push    cx          ; (CX) → стек
        pushf               ; (FLAGS) →стек
        cld                 ; 0 → DF
        mov     cx, 256     ; Счетчик байтов
rep     movsb               ; Копирование цепочки байтов
        popf                ;  Восстановление  FLAGS
        pop     cx          ; Восстановление  CX
        ret                 ; Возврат из процедуры
```

Так как флаг направления может использоваться и в программе, вызывающей нашу процедуру, то надо временно сохранить его в стеке. Для этого используются операторы *pushf* и *popf*, выполняющие запись в стек и извлечение оттуда содержимого регистра *FLAGS*.

2. ***movsw*** – пересылка цепочки слов. Данный оператор копирует цепочку слов из сегмента данных в цепочку, расположенную в дополнительном сегменте. Детали применения данного оператора аналогичны оператору *movsb*.

П1.3. Адресация данных

Многие исполнительные операторы ассемблера (арифметические, логические, передачи данных, строковые операторы) обрабатывают какие-то данные, являющиеся операндами этих операторов. Эти данные могут находиться в следующих местах: 1) в регистрах; 2) в ОП – в поле машинной команды, их обрабатывающей; 3) в ОП – вне машинных команд.

1. ***Регистровая адресация***. Обрабатываемое данное находится в регистре. В следующем операторе оба операнда используют такую адресацию:

```
mov   ax, bx
```

2. ***Непосредственная адресация***. Пример оператора, второй операнд которого содержит непосредственные данные:

```
mov   ax, 2
```

Непосредственно адресуемый операнд помещается в поле машинной команды, занимая в зависимости от типа команды 1 или 2 байта. Например, для приведенного выше оператора ассемблера соответствующая машинная команда имеет вид: *b80200*, где *b8h* – КОП, а два байта данных (*02h* и *00h*) следуют за ним (младший байт слова расположен первым в памяти). Данные, задаваемые в операторе непосредственно, можно только читать, но менять их нельзя.

Все остальные типы адресации предназначены для работы с данными, находящимися в ОП вне сегмента кодов. Основным местом, предназначенным для хранения таких данных является сегмент данных (рис. П.9). Местоположение любого байта или слова дан-

ных однозначно задается смещением *L* относительно начала сегмента данных. С учетом того, что регистр сегмента данных *DS* содержит начальный параграф сегмента данных, реальный адрес соответствующей ячейки в ОП: *R = (DS) x 16 + L*.

Рис. П.9. Расположение сегмента данных в ОП

Различные способы (режимы) адресации отличаются друг от друга тем, как они задают смещение *L*. Вот некоторые из них.

3. ***Прямая адресация***. 16-битовое смещение *L* является частью исполнительного оператора. Пример:

```
mov   [Max], al ,
```

где *Max* – метка ячейки в сегменте данных. В поле машинной команды находится 16-битовое слово, содержащее *L* для байта, помеченного в исходной программе как *Max*.

4. ***Регистровая косвенная адресация***. Искомое смещение *L* находится или в базовом регистре *BX* или в индексном регистре (*DI* или *SI*). Регистр задается в операторе ассемблера, заключенным в квадратные скобки. Например, следующий оператор

```
mov   [bx], dx
```

пересылает содержимое регистра *DX* в ячейку (слово) ОП, адрес которой находится в регистре *BX*.

5. ***Относительная регистровая адресация***. Это «гибрид» прямой и регистровой косвенной адресации для получения *L*. К заданному в операторе смещению прибавляется содержимое базового или индексного регистра. Смещение можно задать двумя способами: а) меткой; б) непосредственно, т. е. числом. Примеры:

```
а) mov   [Max+bx],dx,
```

содержимое регистра *DX* записывается в слово ОП, смещение *L* для которого получается путем суммирования смещения для метки *MAX* с содержимым регистра *BX*;

```
б) mov   [10+bx], dx,
```

содержимое *DX* записывается в ячейку ОП, имеющую *L=10+(BX)* .

6. ***Базовая индексная адресация***. Смещение *L* равно сумме содержимых базового регистра *BX* и индексного (*SI* или *DI*), заданных в операнде. Пример:

```
mov   al, [bx+si],
```

содержимое байта ОП, имеющего *L=(BX)+(SI)*, переписывается в регистр *AL*.

7. ***Относительная базовая индексная адресация***. Смещение *L* равно сумме трех слагаемых: а) заданного в операторе смещения; б) содержимого базового регистра *BX*; в) содержимого индексного регистра *SI* или *DI*. Примеры:

```
а)  mov    al, [Max+bx+di] ,
        L = смещение для Max + (BX) + (DI) .
б)  mov    al, [10+bx+di],
        L = 10+(BX)+(DI) .
```

П1.4. Определение данных

Начальное состояние сегмента данных к моменту начала выполнения программы задается программистом с помощью псевдооператоров определения данных. С помощью них присваиваются метки различным элементам сегмента данных и, возможно, загружаются в них первоначальные значения.

П1.4.1. Метки

Метка – символьное имя, которое может быть присвоено элементу данных, исполнительному оператору или модулю программы. Максимальная длина метки – 31 символ. Это могут быть такие символы:

1) буквы – от *A* до *Z* и от *a* до *z*;

2) цифры – от 0 до 9;

3) специальные символы – знак вопроса (?); знак @; точка (.); подчеркивание (_); доллар ($).

Примеры меток: *COUNT*, *Page25*, *$E10*. Весьма желательно, чтобы метки поясняли смысл программы, а не выбирались отвлеченно.

Первым символом в метке должна быть буква или специальный символ. Точка может использоваться в метке только в качестве первого символа. Если метка начинается с точки, то это ***локальная метка***. Область определения такой метки лежит между двумя соседними нелокальными (глобальными) метками. Это значит, что если локальная метка определена оператором программы, расположенном на участке программы между двумя глобальными метками, то и операторы, использующие эту метку, должны быть расположены на этом же участке программы.

Ассемблер *NASM* делает разницу между заглавными и строчными буквами. Поэтому метки *COUNT* и *Count* разные.

П1.4.2. Определение байтов

1. ***db*** – определение байта. Данный псевдооператор просит транслятор выделить один или несколько байтов ОП и сообщает ему, что первому из этих байтов присваивается указанная метка. Кроме того, от транслятора требуется записать в эти байты первоначальное содержимое. С помощью одного псевдооператора *db* можно определить один байт или ***массив*** байтов. Основные варианты:

а) нулевая инициализация:

```
F1db    db 0                ; 1 байт
F2db    db 0, 0, 0          ; Массив из 3 байт
F3db    times 3 db 0        ;    –"–
```

б) размещение в последовательности байтов символьной строки (в один байт записывается код ASCII одного символа):

```
F4db    db 'H', 'E', 'L', 'L', 'O'  ; В массиве из  5 байт
                                    ;   HELLO
F5db    db  'HELLO'                 ;          –"–
F6db    db  "HELLO"                 ;          –"–
```

в) размещение в байте десятичной константы:

```
F7db    db 32
F8db    times 10  db  0         ; Массив из 10 байт с нулями
```

г) шестнадцатеричная константа:

```
F9db    db 20h
```

д) двоичная константа:

```
F10db   db 01011001b
```

е) смешанные данные:

```
F11db   db      0, 'A', 'B', 'C', 0
F12db   db      'TABL1', 10h, 20h, 30h
```

2. ***resb*** – резервирование последовательности байтов без инициализации (заполнения). Первому байту присваивается заданное символьное имя. Пример:

```
F1resb  resb  10        ; Массив из 10 байт
```

П1.4.3. Определение слов

1. ***dw*** – определение слова. Основные варианты:

а) нулевая инициализация:

```
F1dw    dw  0               ; Одно слово
F2dw    times 10  dw  0     ; Массив из 10 слов
```

б) шестнадцатеричная константа:

```
F3dw dw 0fff0h
```

в) двоичная константа:

```
F4dw    dw  0101010101010101b
```

г) адресная константа:

```
F5dw    dw  F10DB       ; В слове смещение L для байта с меткой
                        ; F10DB
```

2. ***resw*** – резервирование последовательности слов без инициализации (заполнения). Первому байту присваивается заданное символьное имя. Пример:

```
F1resw  resw  10        ; Массив из десяти слов
```

3. ***dd*** – определение двойного слова. Резервируется область ОП длиной в два слова (4 байта). Максимальное число без знака *0FFFFFFFFh*.

4. ***dq*** – определение последовательности четырех слов (8 байт).

П1.4.4. Определение констант

1. ***equ*** – память не резервируется, а лишь задается инициализирующее значение. Например, пусть в сегменте данных имеется псевдооператор:

```
Numer   equ  10,
```

тогда в каком бы исполнительном операторе или псевдооператоре ни использовалось слово *Numer*, транслятор-ассемблер подставит вместо него 10. Например, он преобразует

```
Fielda  times   Numer  db  0
```

в оператор:

```
Fielda  times  10  db  0
```

Другой пример:

```
Countr  equ   05
        mov   cx,  Countr
```

транслятор сделает замену в операторе *mov*:

```
        mov   cx,  05.
```

В следующем примере переопределяется имя регистра *CX*:

```
Countr   equ   cx
```

2. Символ равенства «=» – применение данного псевдооператора схоже с оператором *equ*. Отличие: выражение справа может быть только числовым. Примеры:

```
Times = 10
Countr = 5
```

П1.4.5. Структуры

В отличие от простых данных, определяемых с помощью псевдооператоров *db, dw, dd, dq, resb, resw*, ***структура*** представляет собой сложный тип данных, так как ее элементы (поля) имеют разную длину. Программист определяет структуру с помощью псевдооператоров ***struc*** и ***endstruc***:

```
struc    <имя>
. . . . . .          ; Описание полей структуры
endstruc
```

Здесь описание полей представляет собой последовательность псевдооператоров определения данных (*db, dw, dd, dq, resb* и *resw*). Их операнды определяют размер полей. Пример описания структуры:

```
struc   WINDOW
        .Height     resw  1   ; Высота окна
        .Width      resw  1   ; Ширина окна
        .Color      resb  1   ; Цвета символов и фона
        .Text       resd  1   ; Дальний указатель (сегмент и смещение)
                              ;     на буфер, содержащий выводимый текст
endstruc
```

Где расположить описание структуры? Здесь можно использовать два варианта. В первом из них описание структуры располагается в начале того же файла, где эта структура будет использоваться. Недостаток такого варианта очевиден, так как в каждом файле, где предполагается использовать структуру, ее придется описывать заново. Во втором варианте описание структуры выносится в отдельный файл с расширением *inc* (текстовый файл, полученный с помощью любого текстового редактора). А в начале каждого ассемблерного файла, где предполагается использовать структуру, записывается псевдооператор *%include* (*включить*) с указанием имени включаемого файла. Например:

```
%include   Window.inc
```

Вне зависимости от того, каким из двух способов определена структура, она может использоваться в программе следующим образом. Во-первых, имя структуры фактически является типом данных, пользуясь которым можно требовать от транслятора создание любого числа экземпляров данной структуры в памяти. Для создания экземпляра структуры следует использовать пару псевдооператоров: ***istruc*** (***включить структуру***) и ***iend*** (***конец включения структуры***). Например, для создания трех экземпляров структуры *WINDOW* достаточно записать:

```
Wind1   istruc   WINDOW   iend
Wind2   istruc   WINDOW   iend
Wind3   istruc   WINDOW   iend
```

Для инициализации экземпляра структуры, т. е. для задания первоначального значения какого-то ее поля, используется вспомогательный псевдооператор ***at***. При этом поля экземпляра структуры должны указываться в той же последовательности, что и в описании структуры, но некоторые поля могут быть пропущены. Например:

```
Wind   istruc  WINDOW
       at      WINDOW.Height,  db 10
       at      WINDOW.Width,   db 20
       iend
```

Здесь высота и ширина окна инициализируются значениями 10 и 20, а цвет и выводимый в окно текст первоначально не задаются. Обратите внимание на запятую после имени поля. Ее запись обязательна.

Если экземпляр структуры определен в программе, с его полями можно работать так же, как с обычными данными, используя прямую адресацию. Напомним, что при использовании такой адресации имя (метка) элемента данных заключается в квадратные скобки. При работе с полем структуры в квадратные скобки записывается сумма имени экземпляра структуры и составного имени, образованного из имени структуры и имени ее поля, разделенных точкой. Например, следующие операторы выполняют запись в поле *Text* экземпляра *Wind1* адреса сообщения, находящегося в поле памяти с меткой *Buff*:

```
mov     [Wind1+WINDOW.Text], Buff          ; Запись адреса-смещения
mov     [Wind1+WINDOW.Text+2], seg Buff    ; Запись адреса-сегмента
```

П1.5. Операторы передачи управления

Существует 6 основных типов операторов передачи управления:

1) операторы условных переходов;
2) операторы безусловных переходов;
3) операторы циклов;
4) операторы процедур;
5) операторы программных прерываний;
6) операторы останова и холостого хода.

П1.5.1. Операторы условных переходов

Формат оператора условного перехода:

КОП α,

где α – метка, или метка + (–) выражение, вычисление которого дает константу. Операнд α указывает на тот оператор в программе, на который делается переход в случае выполнения предусмотренных в операторе перехода условий. Такими условиями являются значения одного, двух или трех флажков.

Машинная команда, соответствующая оператору условного перехода, имеет длину 2 байта. В первом байте находится КОП, а во втором – «расстояние» между содержимым указателя команды *IP* и искомым адресом. Это «расстояние» есть число со знаком, так как переходы могут делаться как вперед по программе, так и назад.

В одном байте можно разместить число со знаком (в дополнительном коде) от –128 до +127. Это приводит к тому, что оператор условного перехода может использоваться лишь для небольших переходов. Для выполнения больших переходов, в том числе и в

другие сегменты памяти, оператор данного типа дополняется операторами безусловного перехода.

Пример. Подсчитаем содержимое второго байта в операторе перехода в следующем фрагменте:

```
0050 Again:  inc   cx
0052         add   ax, [bx]
0054         jz    Again
0056 Next:   mov   [Result], cx
```

В момент выполнения *jz (IP)=0056h*. Следовательно, второй байт *jz* должен содержать –6. В дополнительном коде это *FAh*.

Некоторые операторы условного перехода:

jz (или ***je***) – перейти, если нуль или равно. Условием перехода является *ZF=1*;

jnz (или ***jne***) – перейти, если не нуль или не равно. Условие перехода: *ZF=0*.

Следующие операторы перехода записывают в программу только после операторов, выполняющих действие над беззнаковыми данными. Эти операторы перехода не учитывают ни флаг знака *SF*, ни флаг переполнения *OF*. Для них важен флаг переноса *CF*:

ja (или ***jnbe***) – перейти, если больше. Условие: *(CF = 0) & (ZF = 0)*;

jae (или ***jnb***) – перейти, если больше или равно. Условие: *CF = 0*;

jb (или ***jnae***) – перейти, если меньше. Условие: *CF = 1*;

jbe (или ***jna***) – перейти, если не больше. Условие: *(CF = 1) V (ZF = 1)* .

Следующие операторы перехода записывают в программу только после операторов, выполняющих действия над знаковыми данными. Для них важны флаг знака *SF* и флаг переполнения *OF*:

jg (или ***jnle***) – перейти, если больше. Условие: *(SF = OF)&(ZF = 0)*;

jge (или ***jnl***) – перейти, если больше или равно. Условие: *SF = OF*;

jl (или ***jnge***) – перейти, если меньше. Условие: *SF ≠ OF*;

jle (или ***jng***) – перейти, если меньше или равно. Условие: *(SF≠ OF) V (ZF=1)*.

Каждому условному оператору перехода соответствует противоположный по смыслу оператор. Несмотря на логическую эквивалентность двух фрагментов программ, продолжительность их выполнения скорее всего будет различной. Дело в том, что время выполнения условного оператора в случае перехода в четыре раза больше времени выполнения этого же оператора, если переход не делается. Поэтому из двух альтернативных фрагментов желательно использовать тот, в котором вероятность перехода по условному оператору меньше.

П1.5.2. Операторы безусловных переходов

Такой оператор заставляет ЦП извлечь новую команду не из следующей ячейки ОП, а из какой-то другой, известной еще до начала выполнения программы. Существуют пять машинных команд безусловных переходов. Все они имеют одну и ту же ассемблерную мнемонику *jmp* и один операнд. Это:

1) внутрисегментный прямой короткий переход;
2) внутрисегментный прямой переход;
3) внутрисегментный косвенный переход;
4) межсегментный прямой переход;
5) межсегментный косвенный переход.

Во ***внутрисегментном переходе*** оператор перехода находится в том же сегменте памяти, что и оператор, на который делается переход. А в ***межсегментном переходе*** – в разных сегментах. В операторах ***прямого перехода*** операндом является метка того оператора, на который делается переход. При внутрисегментном переходе данная метка имеет тип ***near*** (близкий). При межсегментном переходе метка перехода имеет тип ***far*** (дальний).

Внутрисегментный прямой короткий переход используется для переходов от +127 до –128 байт. Команда перехода имеет длину 2 байта. В первом байте КОП, а во втором число со знаком, представляющее собой «расстояние» до искомого оператора. Это число идентично соответствующему числу в операторе условного перехода.

Для указания транслятору, что переход короткий, используется слово ***short***. Например:

```
    jmp  short  .A90
              . . . . . .
.A90:
```

В ассемблере *NASM* по умолчанию прыжок короткий. Поэтому, если не задано другое значение, слово *short* обычно опускается.

Внутрисегментный прямой переход используется для переходов любой длины внутри сегмента памяти. При этом машинная команда безусловного перехода занимает 3 байта – один для КОП и 2 – для «расстояния» перехода. Для указания транслятору, что переход не короткий, используется слово ***near***. Например:

```
    jmp  near  .A50
              . . . . . .
.A50:
```

Внутрисегментный косвенный переход задается путем записи в качестве операнда оператора *jmp* не метки, а адреса данных. По этому адресу (в регистре или в слове ОП) записано требуемое содержимое *IP*. Допустим, имеется оператор:

```
    jmp   near  bx
```

Если в момент исполнения соответствующей машинной команды *(BX) = 1ABh*, то ЦП запишет в *IP* это число *1ABh* и извлечет следующую машинную команду из ОП по физическому адресу *(CS)x16 + 1ABh*.

Допустим, что слово ОП с именем *test* содержит адрес-смещение той команды программы, на которую требуется сделать переход:

```
    jmp  near  [test].
```

Межсегментный прямой переход задается путем записи в качестве операнда оператора *jmp* метки перехода типа *far* (дальняя). Пример:

```
    jmp   far  test
```

Межсегментный косвенный переход задается путем записи в качестве операнда оператора *jmp* не метки перехода, а имени поля данных длиной два слова, которое содержит логический адрес перехода. При этом первое слово содержит целевое (*IP*) – адрес-смещение, а второе слово содержит (*CS*) – адрес-сегмент. Имя поля, содержащего логический адрес, заключается в квадратные скобки и ему предшествует вспомогательный псевдооператор *far* (дальний). Пример:

```
. . . . . .
Old   dd    0              ; Для логического адреса перехода
. . . . . .
      jmp   far  [Old]     ; Межсегментный косвенный переход
```

П1.5.3. Операторы циклов

Цикл – многократное повторение группы операторов, называемых ***телом цикла***, до тех пор, пока выполняется некоторое условие (рис. П.10).

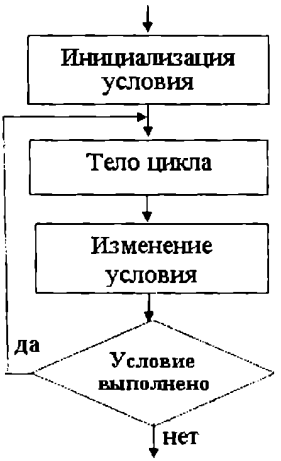

Рис. П.10. Блок-схема цикла

Типичное условие цикла: повторить тело цикла заданное число раз. Число повторений обычно заносится в регистр *CX*. Реализация такого цикла с помощью условного оператора *jnz*:

```
        mov   cx, N
Begin:
        . . . . . .      } Тело цикла
        dec   cx
        jnz   Begin
```

Это же самое можно сделать с помощью специального оператора цикла ***loop***:

```
        mov   cx, N
Begin:
        . . . . . .      } Тело цикла
        loop  Begin
```

Оператор *loop* уменьшает содержимое регистра *CX* на 1 и выполняет переход, если *(CX) ≠ 0*. Следующий фрагмент программы выполняет сложение *M* слов, начинающихся с адреса *Array*. Результат записывается в слово с именем *Total*:

```
        mov   cx, M
        mov   ax, 0
        mov   si, ax
.L1:    add   ax, [Array+si]
        add   si, 2                ; Инкремент индекса на 2
        loop  .L1
        mov   [Total], ax
```

Существуют еще два оператора циклов – ***loopz*** (или ***loope***) и ***loopnz*** (или ***loopne***). Условием повторения для *loopz* является *(CX)≠0 & ZF=1*, т. е. кроме ненулевого содержимого счетчика *CX* требуется, чтобы был установлен флаг нуля. Условием повторения для *loopnz* является *(CX)≠0 & ZF=0*. Подобный оператор обычно используется для поиска в массиве заданного элемента.

Пример. Пусть метка ASCII присвоена первой ячейке массива из *L* символов. Требуется найти в этом массиве пробел (код ASCII пробела – *20h*). Если пробела нет, требуется перейти на оператор с меткой *Not*. Соответствующий фрагмент программы:

```
        mov     cx, L
        mov     si, -1              ; Инициализировать индекс
        mov     al, 20h             ; Код пробела → AL
.Next:  inc     si                  ; Инкремент индекса
        cmp     al, [ASCII+si]      ; Проверка на пробел
        loopne  .Next               ; Если не пробел, то цикл
        jnz     Not
```

Машинная команда, соответствующая оператору цикла, имеет длину 2 байта. В первом байте находится КОП, а во втором – «расстояние» до команды, помеченной меткой перехода. Подобно команде условного перехода это «расстояние» может находиться в пределах от –128 до +127 байт.

П1.5.4. Операторы процедур

В гл. 7 рассматривались машинные команды, выполняющие вызов процедуры и возврат из нее. В языке ассемблера этим командам соответствуют операторы ***call*** и ***ret***. Основное отличие оператора *call* от соответствующей машинной команды заключается в том, что в качестве операнда записывается не адрес первой команды процедуры, а его заменитель – имя (метка) процедуры. Пример:

```
call Read_hex ,
```

где *Read_hex* – имя процедуры.

Что касается задания имени процедуры, то оно выполняется также просто, как и задание обыкновенной метки перехода, например:

```
Read_hex:
. . . . . .             ; Операторы процедуры
    ret                 ; Возврат из процедуры
```

При выполнении оператора *call*, прежде, чем осуществить переход на первый оператор процедуры, в стек помещается 16-битовое слово – адрес-смещение того исполнительного оператора, который находится в вызывающей программе сразу же за оператором *call*. Оператор *ret* извлекает это слово из стека и делает переход на соответствующий адрес-смещение.

Если процедура может вызываться и из других сегментов, то ей присваивается тип *far* (дальняя). Вспомогательный псевдооператор *far* записывается между оператором *call* и именем процедуры. Пример:

```
call  far  Read_hex.
```

Это межсегментный (дальний) прямой вызов процедуры. Для вызова процедуры из другого сегмента памяти можно использовать также косвенную адресацию. Пример:

```
. . . . . .             . . . . . .
Old  dd   0             ; Для логического адреса процедуры
. . . . . .             . . . . . .
    call  far  [Old]    ; Межсегментный вызов процедуры
```

Обратите внимание, что при межсегментном косвенном вызове процедуры не используется имя вызываемой процедуры. Поэтому так можно вызывать процедуры, принадлежащие чужим машинным программам. Имена процедур нужны лишь транслятору и редактору связей при получении конкретной машинной программы, а при межпрограммном взаимодействии эти имена использоваться не могут. Естественно, что для

межсегментного косвенного вызова процедуры необходимо знать ее стартовый логический адрес (сегмент:смещение).

Межсегментный возврат из процедуры выполняет оператор ***retf***.

Дальний оператор *call* помещает в стек два слова, образующих адрес возврата – сначала адрес-сегмент, а затем – адрес-смещение. Оператор *retf* извлекает эти слова из стека в обратном порядке и помещает их в регистры *IP* и *CS*.

П1.5.5. Другие операторы передачи управления

Это следующие операторы.

1. ***int***, ***iret*** – операторы программного прерывания и возврата из него. Данные операторы не имеют смысловых отличий от соответствующих машинных команд. Единственное, на что требуется обратить внимание, это тип системы счисления операнда в операторе *int*. Так как «родной» для ассемблера является десятичная система, то любое шестнадцатеричное число должно записываться по правилам, принятым в ассемблере. Это:

а) если старшей «цифрой» числа является буква, то перед ней записывается 0;

б) сразу за числом записывается буква *h*. Например: int 21h .

2. ***hlt*** – оператор останавливает ЦП до первого аппаратного прерывания. Такие операторы часто встречаются в диагностических программах, а также для снижения нагрузки на ЦП в циклах – обработчиках событий мыши и клавиатуры.

3. Оператор холостого хода ***nop*** не действует ни на флаги, ни на регистры, ни на ячейки ОП. Единственное – он увеличивает содержимое *IP* на 1. Данный оператор имеет много применений для упрощения отладки программы. Приводим три из них:

а) кодом оператора (*90h*) можно «забить» объектный код в том случае, если вам надо удалить машинную команду, не транслируя программу заново;

б) можно сделать оператор *nop* последним в тестируемой программе и тем самым получить удобное место для остановки трассировки;

в) при смене режимов работы процессора содержимое указателя команды *IP* часто увеличивается на один байт. Для устранения ошибок целесообразно разместить оператор *nop* сразу после оператора, выполняющего смену режима работы ЦП.

П1.6. Псевдооператоры для задания структуры программы

Эти псевдооператоры позволяют программисту влиять на структуру машинной программы и на размещение программы в памяти.

1. ***%include*** – псевдооператор, выполняющий включение в исходную программу требуемого текстового файла. Включаемый файл может содержать любые программные операторы, в том числе, макроопределения и описания структур. Операндом оператора *%include* является имя включаемого файла, заключенное в кавычки. Пример:

```
%include   'Video_io.asm'   ; Включение файла Video_io.asm
```

Встретив оператор *%include*, транслятор заменяет его на текст «дочернего» исходного файла, указанного в этом операторе.

2. ***absolute*** – псевдооператор, задающий виртуальный сегмент данных, а также расположение этого сегмента данных относительно начала сегмента памяти программы. Термин ***виртуальный сегмент*** означает, что данный сегмент создается в исходной программе явно, с использованием специально предназначенных для этого псевдооперато-

ров *absolute* или *segment*. В отличие от него ***логический сегмент*** существует не в исходной, а в машинной программе. Пример:

```
A1  absolute   80h     ; Следующий байт имеет адрес-смещение 80h
```

Виртуальный сегмент, начинающийся с оператора *absolute*, закончится тогда, когда в программе начнется следущий виртуальный сегмент. Его начало задается или другим оператором *absolute*, или оператором *segment*. Имя сегмента (в примере – *A1*) не является обязательным.

3. ***segment*** – псевдооператор задания виртуального сегмента кода, данных или стека. Тип сегмента ***code***, ***data*** или ***stack*** задается в качестве параметра оператора *segment*. Если данный параметр является единственным, то сегменту присваивается имя, совпадающее с типом сегмента. Если виртуальному сегменту требуется присвоить какое-то другое имя, то это имя записывается в качестве дополнительного параметра оператора *segment*. Причем этот параметр записывается первым. Например, следующий оператор присваивает сегменту кода имя «*Program*»:

```
segment   Program  code
```

Пример оператора, который присваивает виртуальному сегменту данных имя «*data*»:

```
segment   data
```

Виртуальный сегмент, начинающийся с оператора *segment*, закончится тогда, когда в программе встретится или другой оператор *segment*, или оператор *absolute*, или закончится текст программы.

Все одноименные виртуальные сегменты, независимо от того, в каких исходных файлах и фрагментах они находятся, представлены в готовой машинной программе в виде единого логического сегмента.

4. Псевдооператор ***org*** сообщает транслятору о том, что следующий оператор программы должен быть помещен в выделенный программе сегмент ОП с заданным смещением (параметр оператора *org*) относительно начала сегмента.

5. ***global***, ***extern*** – псевдооператоры, используемые для подготовки связывания раздельно транслировавшихся объектных модулей в единый загрузочный модуль. При этом оператор *global* используется для описания внешних выходных меток, а оператор *extern* – внешних входных меток. Внешние выходные метки определены в данном фрагменте исходной программы, а используются в других фрагментах исходной программы. Фрагмент исходной программы – один или несколько исходных файлов программы, соединенных с помощью операторов *%include*. Внешние входные метки, наоборот, используются в данном фрагменте исходной программы, а определены в других фрагментах. Примеры:

```
global  Label
extern  Array
```

Следует отметить, что в отличие от псевдооператоров *short, near, far* данные операторы никак не зависят от «расстояния» между операторами программы, а представляют собой лишь указания для транслятора о подготовке вспомогательной информации для редактора связей.

П1.7. Вспомогательные псевдооператоры

Вспомогательные псевдооператоры – дополняют основные операторы (исполнительные и псевдооператоры) с целью уточнения их операндов. Примером такого опера-

тора является рассмотренный ранее псевдооператор *short*. Рассмотрим некоторые другие вспомогательные операторы.

1. ***seg*** – вычисляет значение адреса-сегмента – номера параграфа, образующего совместно со смещением логический адрес программного объекта, заданного в ассемблерной программе своей меткой. Например, операторы

```
mov  ax, seg Table
mov  bx,     Table
```

загрузят номер параграфа и смещение адреса переменной *Table* в регистры *AX* и *BX* соответственно.

Оператор *seg* нельзя использовать в *com*-программах.

2. ***$*** – вычисляет адрес-смещение той точки программы, где находится данный оператор *$*. Например, при трансляции операторов

```
Mess  db   'Hello'
Messl equ  $-Mess
```

транслятор определит число символов в строке *Mess* и присвоит его константе *Messl*.

3. ***byte***, ***word***, ***dword*** – псевдооператоры, уточняющие длины операндов. Если оператор имеет два операнда, то достаточно уточнить длину лишь одного из них, так как длина другого операнда такая же.

Например, с помощью этого оператора можно получить доступ к отдельным байтам в таблице слов. Пусть таблица определена следующим образом:

```
Table  times  100  dw 0
```

Тогда оператор

```
First_byte  equ  byte   Table
```

присвоит имя *First_byte* первому байту таблицы *Table*. Можно присвоить имя любому другому байту этой таблицы, например, оператор:

```
Fifth_byte   equ   First_byte+4
```

присвоит имя *Fifth_byte* пятому байту таблицы *Table*.

Если один из операндов – регистр, то применение псевдооператора, уточняющего длину операндов, излишне, так как по имени регистра транслятор «знает» его длину.

4. ***short, near, far*** – псевдооператоры, задающие тип метки перехода.

Метка типа *short* (короткая) присваивается тому исполнительному оператору, переходы на который возможны только из этого же сегмента ОП и не превышают расстояние в 128 байт. Задание данного типа метки производится по умолчанию.

Метка типа *near* (ближняя) присваивается тому исполнительному оператору, переходы на который возможны только из этого же сегмента ОП. Псевдооператор *near* записывается между оператором *jmp* и меткой перехода. Пример:

```
jmp near Label.
```

Метка типа *far* (дальняя) присваивается тому исполнительному оператору, переходы на который возможны из любых сегментов ОП. Нетрудно заменить тип метки *short* (или *near*) на тип *far* и наоборот. Например, если программа содержит оператор

```
Start:  mov  cx, 100
```

то метка *Start* имеет тип *short* или *near*, что позволяет ссылаться на нее командам перехода, находящимся в том же сегменте памяти. Для того чтобы на это же место программы могли делать переходы команды *jmp*, расположенные в других сегментах ОП, необходимо задать в программе альтернативную метку типа *far*:

```
Far_start  equ  far  Start
```

Ранее были записаны прмеры использования оператора *far* для реализации межсегментных безусловных переходов и для межсегментных вызовов процедур.

П1.8. Макрооператоры

Макрооператор – оператор, заменяющий в программе на ассемблере последовательность обычных операторов (исполнительных операторов и псевдооператоров). То есть вместо того, чтобы записывать в программе несколько операторов, мы помещаем в нее единственный макрооператор. Структура макрооператора:

```
<имя макрооператора>  [список фактических параметров]
```

Пример макрооператора:

```
Out_str   Buff
```

Здесь макрооператор *Out_str* предназначен для вывода на экран строки символов, расположенной в области памяти с именем *Buff*. В этом примере всего один фактический параметр (*Buff*). Если таких параметров несколько, то они отделяются запятыми.

В начале своей работы транслятор-ассемблер выполняет для каждого макрооператора ***макрорасширение*** – заменяет макрооператор на соответствующую совокупность операторов ассемблера. После завершения макрорасширений текст программы не содержит макрооператоров и представляет собой обычную ассемблерную программу.

Для того чтобы транслятор мог выполнить макрорасширение, имя макрооператора должно быть определено в ***макроопределении***. Структура макроопределения:

```
%macro   <имя макрооператора> [количество формальных параметров]
        . . . . . .                    ; Тело макроопределения
%endmacro
```

Пример макроопределения:

```
%macro  Out_str  1
;     Вывод строки на экран
;     ---------------------
;  Вход:  %1 – адрес-смещение выводимой строки
        push   ax
        push  dx
        mov   ah, O9h
        mov   dx, %1
        int    21h
        pop    dx
        pop    ax
%endmacro
```

При выполнении макрорасширения для *Out_str* транслятор заменит этот макрооператор на тело макроопределения, причем он заменит формальный параметр %1в теле макроопределения на фактический параметр *Buff*. (В примере такая замена выполняется в единственном операторе *mov*.)

Где расположить макроопределение? Здесь можно использовать два варианта. В первом из них макроопределение располагается в начале того же файла, где оно будет использоваться. Недостаток: в каждом файле, где предполагается использовать аналогичный макрооператор, его придется описывать заново. Во втором варианте макроопреде-

ление выносится в отдельный файл (текстовый ASCII-файл, полученный с помощью любого текстового редактора). А в начале каждого ассемблерного файла, где предполагается использовать макроопределение, записывается псевдооператор *%include* с указанием имени включаемого файла. Например:

```
%include    "Outstr.inc"
```

Обычно в один включаемый файл помещают несколько макроопределений и, возможно, описания структур (которые фактически являются разновидностью макроопределений). При выполнении транслятором псевдооператора *%include* весь включаемый файл подсоединяется к тексту программы, содержащей *%include*, но на текст машинной программы наличие этого файла не влияет.

Полезно выполнить сравнение макроопределений с подпрограммами. Оба эти типа модулей позволяют программисту инициировать выполнение многих операторов с помощью единственного оператора (*call* или *int* для подпрограммы и макрооператор для макроопределения). Но механизм их реализации совершенно различен. В то время как тело макроопределения будет записано транслятором всюду, где он встретит соответствующий макрооператор, тело подпрограммы находится в памяти в единственном экземпляре, а операторы *call* (или *int*) выполняют передачу управления этому экземпляру. Отсюда можно сделать более обоснованным выбор между этими типами модулей.

Так как каждое выполнение подпрограммы связано с «накладными расходами» в виде двух машинных команд (*call* и *ret*, или *int* и *iret*), то вклад этих команд в продолжительность выполнения тем больше, чем меньше подпрограмма. С другой стороны, чем длиннее тело макроопределения, тем больше затраты памяти на его «тиражирование». Следовательно, короткие повторяющиеся последовательности операторов лучше оформлять в виде макроопределений, а длинные – в виде подпрограмм.

Приложение 2. Виртуальные машины «Virtual PC»

Программа «*Virtual PC*» фирмы Microsoft, запускаемая под управлением одной из современных Windows, выполняет имитацию работы аппаратуры ЭВМ требуемой конфигурации. Данная имитация выполняется настолько искусно, что появляется возможность запустить на «голой» аппаратуре любую другую ОС или какую-то автономную программу, например, загрузчик или драйвер.

Для установки «Virtual PC» достаточно запустить на выполнение программу Setup. Дальнейшая установка производится почти автоматически.

После установки «Virtual PC» на ваш компьютер можно создать любое число экземпляров виртуальной машины (ВМ). Это число ограничено лишь объемом свободной дисковой памяти, так как каждая ВМ хранится на магнитном диске в виде поддерева файловой структуры. Корневой каталог (корневая папка) этого поддерева имеет имя, совпадающее с именем виртуальной машины. Само создание экземпляра ВМ производится следующим образом.

Во-первых, следует запустить на выполнение программу

c:\Program Files\Microsoft Virtual PC\Virtual PC

Это можно сделать и с помощью кнопок:

[Пуск] → [Программы] → [Microsoft Virtual PC]

В результате в верхнем левом углу экрана появится окно, называемое «консолью управления». После нажатия кнопки этого окна «*New...*» откроется окно мастера конфигурации виртуального компьютера. Нажав кнопку «*Next*» получим меню из трех пунктов:

1) *Create a virtual machine* (создать виртуальную машину);

2) *Use default settings to create a virtual machine* (использовать настройки виртуальной машины по умолчанию);

3) *Add an existing virtual machine* (добавить существующую виртуальную машину).

Следует выбрать первый пункт, а затем, нажав «*Next*», набрать имя новой ВМ. Напомним, что это имя будет одновременно простым именем каталога (папки). После завершения создания машины оно будет включено в список имен виртуальных машин консоли управления.

Следующая вкладка позволяет нам выбрать тип операционной системы. Этот тип оказывает влияние лишь на параметры ВМ. Сама ОС, естественно, установлена пока быть не может. Если предполагается использовать ОС, отличную от Windows, то лучше выбрать пункт меню «*Other*» (другое).

На следующей вкладке мастер спрашивает пользователя о том, хочет ли он вручную назначить объём оперативной памяти ВМ или оставить без изменений предлагаемое значение. Выберем пункт «*Adjusting the RAM*» (указать объём ОП). Появится полоска с бегунком, для установки которого на требуемое значение следует использовать клавиши ↓ и ↑, или курсор мыши. При этом объем запрашиваемой памяти не может превышать объема реальной ОП на вашей ЭВМ. При попытке превысить этот объем на экран будет выведено соответствующее сообщение. Для установки FreeDOS, а также для запуска наших автономных программ вполне достаточен объем ОП в 32 Мбайта.

Следующий пункт мастера – создание жёсткого диска. Выберите пункт «*A new virtal hard disk*» (создать новый диск) и нажмите «*Next*». В появившемся окне следует ввести имя файла (с расширением *vhd*), в котором будет хранится информация жесткого диска,

а также задать размер этого диска. При этом имя файла логично поместить в ранее созданный каталог, имя которого совпадает с именем ВМ. Что касается размера диска, то фактическое его значение определяется правилами работы с файлами, а введенный размер представляет собой предельную величину этого значения. Напомним, что размер файла возрастает динамически – по мере записи в него новой информации. Причем размер файла не может превысить размер свободного пространства того диска, на котором файл находится. Этот предельный размер мастер выводит для нас на экран.

Работа по созданию новой ВМ завершена. При этом в окне консоли управления появился прямоугольник с именем новой ВМ. Пользуясь кнопками в окне консоли мы можем запустить ВМ на выполнение или продолжить ее настройку, задав ее расширенную конфигурацию. ВМ настраивается после нажатия кнопки «*Settings*». Далее можно указать конфигурацию аудио-видео подсистемы, жёстких дисков и настроить систему ввода-вывода. Если нам такая точная настройка не требуется, мы запускаем ВМ нажатием кнопки «*Start*».

На экране появляется окно ВМ. Теперь с виртуальной машиной можно работать, так же как и с реальной. При этом следует учесть, что заменены некоторые комбинации клавиш. Например, вместо <Ctrl>+<Alt>+<Delete> следует нажать <Правый Alt>+<Delete>. При этом заметим, что <Правый Alt> является управляющей клавишей физической машины (реальная аппаратура + Windows), а <Левый Alt> – управляющая клавиша виртуальной машины.

Запустив виртуальную машину, нам станет скучно, если на этой машине отсутствует какая-либо автономная программа, например, ОС. Запуск автономной программы производится автоматически, т. е. без участия пользователя. Для этого после «включения» виртуальной машины инициируется подпрограмма BIOS, выполняющая проверку наличия у виртуальной машины требуемой аппаратуры, а затем загружающая и запускающая первичный начальный загрузчик.

Первичный начальный загрузчик – программный модуль, расположенный на таком секторе диска (512 байт), который зарегистрирован в BIOS в качестве загрузочного сектора. Зная расположение первичного начального загрузчика, подпрограмма BIOS выполняет копирование его с диска в ОП, а затем передает ему управление. Далее первичный начальный загрузчик загружает в ОП из ВП автономную программу и передает ей управление, или загружает и инициирует промежуточный модуль – вторичный начальный загрузчик.

Если на виртуальной машине отсутствует подготовленная к загрузке автономная программа (например, ОС), то нам остается работать лишь с подпрограммами BIOS. Как известно, эти подпрограммы «зашиты» в ПЗУ и как бы являются продолжением аппаратуры. Виртуальная машина также снабжена своими (виртуальными) подпрограммами BIOS, которые находятся, естественно, не в ПЗУ, а где-то в ОП. Для того чтобы войти в BIOS виртуальной машины, следует во время ее запуска нажать <Delete>.

В нижнем левом углу окна виртуальной машины располагаются индикаторы ее устройств – жестких дисков, приводов CD (DVD)-ROM, дисководов гибких дисков, сетевого адаптера. Кликнув правой клавишей мыши на изображении компакт-диска, мы можем связать виртуальный привод с физическим приводом, или «установить» на этот виртуальный привод файл-образ компакт-диска в формате *ISO*. Например, прежде чем выполнить установку на виртуальной машине какой-либо ОС (например, FreeDOS), следует снять образ в формате *ISO* установочного компакт-диска с ОС, а затем связать его с вир-

туальным приводом. Некоторые ОС, в том числе и FreeDOS, распространяются сразу в виде *ISO*-образа компакт-диска, предполагающим их последующую запись на носитель.

Другой способ – в качестве жесткого диска (винчестера) требуется задать файл, содержащий виртуальный загрузочный жесткий диск с требуемой ОС. Расширение имени файла – *vhd*. Полный тип файла – «*Virtual Machine Hard Drive Image*». Собственно имя файла любое, например, *FreeDOS Hard Disk*. Данный файл создается заранее и размещается в любом каталоге физической машины. Далее, во время создания ВМ требуется выбрать строку «*An existing virtual hard disk*», «нажать» клавишу «*Browse...*», а затем выбрать в появившемся окне имя-путь для *vhd*–файла. Теперь при запуске ВМ ее виртуальный BIOS будет загружать в ОП и инициировать тот первичный загрузчик, который находится на самом первом «секторе» виртуального загрузочного диска.

Во время загрузки FreeDOS BIOS выводит на экран меню, в котором следует выбрать строку «*3 – Load FreeDOS including...*». Для того чтобы в начале следующего сеанса работы ВМ не повторять загрузку ОС, а сразу начать работу с загруженной системой, в конце данного сеанса, при закрытии окна ВМ следует выбрать строку «*Save state*».

Наличие на ВМ своей ОС или другой автономной программы обычно еще не достаточно для того, чтобы работа на ВМ была полезной. Для этого дополнительно требуется организация информационного обмена между ВМ и физической машиной. Организация такого обмена предполагает, что файлы, полученные на физической машине, могут затем использоваться при работе на ВМ и наоборот. Подобный информационный обмен реализуется в «Virtual PC», в отличие от некоторых других подобных систем, достаточно просто.

Для этого достаточно кликнуть правой клавишей мыши на значок папки (каталога) в левом нижнем углу окна ВМ. Затем следует «нажать» «*Share Folder...*». В результате появится окно, в левом нижнем углу которого можно выбрать имя логического диска, например, *z:*. В центральной части окна следует выбрать требуемый каталог (папку) и нажать <OK>. В результате содержимое выбранного каталога физической машины будет «видно» в ВМ как содержимое выбранного логического диска. Естественно, что при возврате в физическую машину выбранная ранее папка может содержать любые результаты (файлы), полученные в ходе работы ВМ.

Изложенная выше связь между логическим диском ВМ и каталогом (папкой) физической машины существует только во время данного сеанса работы ВМ. При закрытии окна ВМ эта связь ликвидируется. Если требуется освободить логический диск ВМ, связанный с каталогом физической машины, не дожидаясь завершения текущего сеанса, то достаточно кликнуть правой клавишей мыши на значок папки в левом нижнем углу окна ВМ, а затем «нажать» «*Remove <имя логического диска>*».

ЛИТЕРАТУРА

Основная

1. Нортон П., Соухэ Д. Язык ассемблера для IBM PC. М.: Компьютер, 1992.

2. Абель П. Язык ассемблера для IBM PC и программирования. М.: Высшая школа, 1992.

3. Кулаков В. Программирование на аппаратном уровне. Специальный справочник. СПб.: Питер, 2003.

Дополнительная

1. Зубков С.В. Ассемблер для DOS, Windows и Unix. М.: ДМК, 2005.

2. Данкан Р. Профессиональная работа в MS-DOS. М.: Мир, 1993.

3. Юров В. И. Assembler: Учебник для вузов. СПб.: Питер, 2005.

4. Магда Ю. Ассемблер для процессоров Intel Pentium. СПб.: Питер, 2006.

5. Пирогов В. Ю. Assembler. Учебный курс. М.: Нолидж, 2001.

6. Сайт разработчиков альтернативного Ассемблера [Электронный ресурс]. URL: www.nasm.us/pub/nasm/releasebuilds/2.09rc1/win32/.

Вышли в свет и имеются в продаже:

Архитектура персонального компьютера: организация, устройство, работа. Учебное пособие для вузов / В. В. Старков. – М.: Горячая линия – Телеком, 2009. – 536 с.: ил., ISBN 978-5-9912-0064-6.

Систематизированы сведения в области архитектуры и внутренней организации компьютерных систем. Подробно рассмотрены базовые понятия и основные принципы структурной (логической) организации современного персонального компьютера (ПК). Рассмотрены принципы работы и устройство основных компонентов ПК: процессора, системной платы, памяти, дисковой системы, видеосистемы. Показаны особенности организации каждой из систем и соединения их в единое целое в составе компьютера, построенного на базе «открытой архитектуры». Указаны факторы, способствовавшие успеху «открытой» архитектуры ПК. Книга содержит большое количество справочной информации, полезной в повседневной практике. Каждая глава снабжена вопросами для самопроверки.

Для студентов вузов, обучающихся по направлению 230100 – «Информатика и вычислительная техника», будет полезна специалистам в области сборки, модернизации и технического обслуживания компьютеров, сотрудникам предприятий по продаже компьютеров и комплектующих, читателям, самостоятельно осуществляющим сбору или модернизацию собственного компьютера.

Логическое программирование на языке Visual Prolog. Учебное пособие для вузов / Н. И. Цуканова, Т. А. Дмитриева. – М.: Горячая линия – Телеком, 2008. – 144 с.: ил., ISBN 978-5-9912-0033-2.

Изложены основы логического программирования на примере языка Visual Prolog. Рассмотрены описание предметной области и структура программы, алгоритм работы интерпретатора, ввод – вывод, приемы и средства организации интерактивных программ, вопросы недетерминированного программирования и управления выполнением программы, различные структуры данных и предикаты работы с ними. Книга содержит многочисленные примеры, иллюстрирующие теоретические положения их решения, а также контрольные вопросы и практические задания. Многие примеры ориентированы на создание реляционной базы данных и написание различных запросов к ней. Пособие может быть полезно при изучении курса «Функциональное и логическое программирование».

Для студентов высших учебных заведений, программистов, специалистов в области искусственного интеллекта и баз данных.

Основы современного программирования. Разработка гетерогенных систем в Интернет-ориентированной среде: Учебное пособие для вузов / С. В. Зыков. – М.: Горячая линия–Телеком, 2006. – 443 с., ил., ISBN 5-93517-228-3.

Систематизированы теоретические и практические сведения в области современного программирования. Рассмотрены основы функционального подхода к программированию на основе метаязыка SML.NET и подходы к объектно-ориентированному и компонентному программированию на языках SML и C#. Особое внимание уделено применению языковых конструкций для создания корректных, эффективных и надежных программ. Книга написана на основании курса лекций, читаемого автором на факультете информационной безопасности МИФИ, и построена на основе творческого синтеза избранных формальных теорий (лямбда-исчисление, комбинаторная логика, теория категорий и др.) и интернет-ориентированной технологической платформы Microsoft .NET, обеспечивающей практически прозрачную интеграцию кода на языках программирования различных типов. Каждая глава учебного пособия сопровождается контрольными вопросами.

Для студентов, аспирантов, специализирующихся в области компьютерных наук и информационных технологий, может быть полезной для опытных программистов.

Сервер приложений «Zope». Учебное пособие для вузов / С. Э. Грегер. – М.: Горячая линия – Телеком, 2009. – 256 с.: ил., ISBN 978-5-9912-0112-4.

Рассмотрен язык программирования Python и вопросы применения приложения ZOPE (Z Object Publishing Environment) для разработки Web-приложений. Дано описание синтаксиса языка Python, построения структур данных, разбираются вопросы объектно-ориентированного программирования. Рассмотрены принципы работы сервера приложений Zope, его объектная модель и интерфейс управления. Обсуждаются вопросы синтаксиса, приведены примеры использования языков разработки шаблонов DTML и TAL, взаимодействие с реляционными базами данных, создание поисковых служб. Рассмотрены вопросы настройки системы безопасности сервера. Пособие содержит большое количество примеров и контрольные вопросы по изучаемым темам, что может служить основой для построения лабораторного практикума.

Для студентов вузов, обучающихся по направлению 230100 – «Информатика и вычислительная техника», специальности 230105 – «Программное обеспечение вычислительной техники и автоматизированных систем».

В. В. Одиноков, В. П. Коцубинский

Программирование на ассемблере

Изучение самого «древнего» из языков программирования – ассемблера абсолютно необходимо для всех, кто желает в совершенстве овладеть искусством программирования. Почему же ассемблер столь важен?

Во-первых, ни один язык программирования кроме него не дает такого ясного представления о строении аппаратуры ЭВМ.

Во-вторых, язык ассемблера незаменим для разработки программ для встроенных устройств – микропроцессорных систем и микроконтроллеров, используемых в LCD телевизорах, автомагнитолах, сотовых телефонах, медицинской аппаратуре и других подобных разработках. Эти устройства часто не имеют операционных систем и поэтому управляющие программы должны быть автономными.

Материал изложен от написания и отладки простейших программ в среде отладчика debug до практической разработки автономных программ, не использующих не только операционную систему, но и BIOS. Приведенные сведения иллюстрируются примером достаточно сложной программы – шестнадцатеричного редактора, который позволяет редактировать как содержимое оперативной памяти, так и файлы на диске. Теоретический материал в совокупности с предлагаемыми индивидуальными заданиями оформлен в виде лабораторных работ, что значительно облегчает индивидуальное восприятие предмета читателем и повышает ценность книги при использовании в учебном процессе.

Для студентов технических вузов, будет полезна для широкого круга специалистов, желающих самостоятельно систематизировать свои познания в области программирования и устройства микропроцессорных систем.

Сайт издательства:

www.techbook.ru

ISBN 978-5-9912-0162-9