

АССЕМБЛЕР – ЭТО ПРОСТО

Учимся программировать
2-е издание

Основные команды процессоров Intel

16- и 32-разрядные регистры

Основы работы с сопроцессором

Управление XMS-памятью

Разработка и написание резидентных программ,
файловой оболочки, вируса и антивируса

Исследование работы отладчиков,
принципы отладки программ сторонних авторов



Олег Калашников

АССЕМБЛЕР ЭТО ПРОСТО

**УЧИМСЯ ПРОГРАММИРОВАТЬ
2-е издание**

Санкт-Петербург

«БХВ-Петербург»

2011

УДК 681.3.068+800.92Ассемблер

ББК 32.973.26-018.1

К17

Калашников О. А.

К17 Ассемблер — это просто. Учимся программировать. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 336 с.: ил. + CD-ROM

ISBN 978-5-9775-0591-8

Подробно и доходчиво объясняются все основные вопросы программирования на ассемблере. Рассмотрены команды процессоров Intel, 16- и 32-разрядные регистры, основы работы с сопроцессором, сегментация памяти в реальном масштабе времени, управление клавиатурой и последовательным портом, работа с дисками и многое другое. Описано, как разработать безобидный нерезидентный вирус и антивирус против этого вируса, как написать файловую оболочку (типа Norton Commander или FAR Manager) и как писать резидентные программы.

Каждая глава состоит из объяснения новой темы, описания алгоритмов программ, многочисленных примеров и ответов на часто задаваемые вопросы. Во второе издание внесены исправления и добавлены новые примеры. Компакт-диск содержит исходные коды всех примеров, приведенных в книге, с подробными описаниями.

Для программистов

УДК 681.3.068+800.92Ассемблер

ББК 32.973.26-018.1

Группа подготовки издания:

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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.01.11.

Формат 70×100¹/₁₆. Печать офсетная. Усл. печ. л. 27,09.

Тираж 2000 экз. Заказ №

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 978-5-9775-0591-8

© Калашников О. А., 2011

© Оформление, издательство "БХВ-Петербург", 2011

Оглавление

Предисловие.....	1
Несколько советов	2
Ответы на некоторые вопросы	3
ЧАСТЬ I. ЗНАКОМЬТЕСЬ: АССЕМБЛЕР.....	7
Глава 1. Первая программа	9
1.1. Шестнадцатеричная система счисления	9
1.2. Наша первая программа	12
1.3. Введение в прерывания	13
1.4. Резюме	16
Глава 2. Регистры процессора.....	17
2.1. Введение в регистры микропроцессоров 8086—80186.....	17
2.1.1. Регистры данных	17
2.1.2. Регистры-указатели.....	18
2.1.3. Сегментные регистры	18
2.2. Команды сложения и вычитания	19
2.2.1. Оператор <i>add</i>	19
2.2.2. Оператор <i>sub</i>	19
2.2.3. Оператор <i>inc</i>	20
2.2.4. Оператор <i>dec</i>	21
2.3. Программа для практики.....	21
Глава 3. Сегментация памяти в реальном режиме.....	23
3.1. Двоичная система счисления. Бит и байт	23
3.1.1. Как перевести двоичное число в десятичное.....	25
3.1.2. Как перевести десятичное число в двоичное.....	25
3.1.3. Как перевести шестнадцатеричное число в десятичное	26
3.2. Сегментация памяти в реальном режиме.....	26
3.2.1. Исследование программы в отладчике	28
3.3. Наше первое прерывание	32
3.3.1. Что такое ASCII?.....	32
3.4. Программа для практики.....	33
3.5. Подведем итоги.....	34
ЧАСТЬ II. УСЛОЖНЯЕМ ЗАДАЧИ.....	37
Глава 4. Создание циклов	39
4.1. Еще немного о сегментации памяти.....	39
4.1.2. Введение в адресацию	39
4.2. Создание циклов	42
4.2.1. Пример высокουровневой оптимизации	43
4.3. Условный и безусловный переходы	44
4.3.1. Пример низкоуровневой оптимизации.....	45

4.4. Программа для практики	45
4.4.1. Принцип работы программы.....	46
Глава 5. Подпрограммы	47
5.1. Исправляем ошибку	47
5.2. Подпрограммы	48
5.3. Программа для практики.....	51
5.4. Несколько слов об отладчике AFD.....	53
Глава 6. Работа со стеком	54
6.1. Стек	54
6.2. Программа для практики.....	61
6.2.1. Оператор <i>por</i>	61
6.2.2. Хитрая программа.....	62
Глава 7. Операторы сравнения.....	64
7.1. Разбор программы из главы 6	64
7.2. Оператор сравнения.....	66
7.3. Понятия условного и безусловного переходов.....	69
7.4. Расширенные коды ASCII	69
7.5. Программа для практики.....	71
Глава 8. Учимся работать с файлами.....	74
8.1. Программа из прошлой главы	74
8.2. Основы работы с файлами	76
8.3. Программа для практики.....	82
Глава 9. Работа с файлами.....	84
9.1. Программа из прошлой главы	84
9.2. Программа для практики.....	87
ЧАСТЬ III. ФАЙЛОВАЯ ОБОЛОЧКА, ВИРУС, РЕЗИДЕНТ	91
Глава 10. Введение в вирусологию. Обработчик прерываний	93
10.1. Программа из прошлой главы	93
10.2. Вирус.....	97
10.2.1. Структура и принцип работы вируса	98
Что должен делать вирус?	98
Какой объем памяти занимает вирус?	98
Что может вирус?	98
Какой вирус мы будем изучать?	98
Что будет делать вирус?	98
Как оформляется вирус?	98
10.3. Резидент	99
10.3.1. Подробней о прерываниях	99
10.4. Первый обработчик прерывания	101
10.4.1. Новые операторы и функции прерываний.....	104
10.5. Работа с флагами процессора	104
10.5.1. Как проверить работу программы?	106
Глава 11. Управление видеоадаптером	109
11.1. Оболочка.....	109
11.2. Управление видеокартой.....	112

Глава 12. Повторная загрузка резидента	115
12.1. Резидент	115
12.2. Проверка на повторную загрузку резидента.....	115
12.3. Команды работы со строками	118
12.4. Использование xor и sub для быстрого обнуления регистров.....	125
12.5. Задание для освоения информации из данной главы.....	126
Глава 13. Поиск и считывание файлов: вирус.....	127
13.1. Теория	127
13.2. Практика	128
13.3. Команда пересылки данных <i>movs</i>	132
13.4. Передача управления программе, расположенной в другом сегменте	134
13.5. Поиск файлов	135
Глава 14. Вывод окна в центре экрана	137
14.1. Модели памяти.....	137
14.1.1. Почему мы пишем только файлы типа СОМ?.....	137
14.1.2. Что такое модель памяти и какие модели бывают?	137
14.2. Оболочка SuperShell	139
14.2.1. Управление курсором	139
14.2.2. Операторы работы со стеком процессора 80286+	140
14.3. Процедура рисования рамки (окна).....	142
14.3.1. Прямое отображение в видеобуфер.....	142
14.3.2. Процедура <i>Draw_frame</i>	143
Что такое линейный адрес и зачем он нужен?	144
14.4. Практика	145
14.5. Новые операторы	145
Глава 15. Обработка аппаратных прерываний	148
15.1. Теория	148
15.1.1. Сохранение предыдущего вектора прерывания	150
15.1.2. Способы передачи управления на прежний адрес прерывания	151
Первый способ	151
Второй способ	151
15.2. Инструкции <i>ret</i> и <i>ref</i>	152
15.2.1. Оператор <i>ret</i>	152
15.2.2. Оператор <i>ref</i>	153
15.3. Механизм работы аппаратных прерываний. Оператор <i>iret</i>	155
15.4. Практика	157
15.5. Логические команды процессора	159
15.5.1. Оператор <i>or</i>	159
15.5.2. Оператор <i>and</i>	160
15.5.3. Оператор <i>xor</i>	161
15.6. Аппаратные прерывания нашего резидента	162
15.6.1. Аппаратное прерывание <i>05h</i>	162
15.6.2. Аппаратное прерывание <i>09h</i>	162
15.6.3. Аппаратное прерывание <i>ICh</i>	163
15.7. Резюме	164
Глава 16. Принципы работы отладчиков	165
16.1. Как работает отладчик.....	165
16.1.1. Прерывание <i>03h</i>	165

16.2. Способы обойти отладку программы.....	170
16.2.1. Таблица векторов прерываний.....	170
16.3. Практика	172
Глава 17. Заражение файлов вирусом.....	174
17.1. Определение текущего смещения выполняемого кода	174
17.2. Вирус	176
17.2.1. Первые байты "файла-жертвы".....	180
17.2.2. Передача управления "файлу-жертве"	181
Глава 18. Высокоуровневая оптимизация программ	183
18.1. Пример высокоуровневой оптимизации	183
18.2. Ошибка в главе 17	184
18.3. Оболочка Super Shell	185
18.3.1. Передача данных процедуре через стек	185
18.3.2. Передача параметров в стеке	192
18.3.3. Вычисление длины строки на стадии ассемблирования.....	192
18.3.4. Процедуры <i>Copy_scr / Restore_scr</i> (<i>display.asm</i>)	193
18.3.5. Оператор <i>scas</i>	194
18.3.6. Подсчет длины нефиксированной строки.....	196
18.3.7. Вывод строки на экран путем прямого отображения в видеобуфер	198
18.4. Резюме	199
Глава 19. Создание резидентного шпиона	200
19.1. Резидент	200
19.2. Что нужно вам вынести из этой главы?	204
Глава 20. Финальная версия вируса	205
20.1. Вирус	206
20.1.1. Альтернативы <i>ret, call</i> и <i>jmp</i>	206
20.1.2. Заражение файла	207
20.1.3. Общая схема работы вируса	210
20.2. Резюме	211
Глава 21. Работа с блоками основной памяти	213
21.1. Оболочка SuperShell	213
21.1.1. Теория	213
21.1.2. Практика	214
Новшество первое	214
Новшество второе	215
21.1.3. Оператор <i>test</i>	215
21.2. Работа с основной памятью DOS	219
21.2.1. Управление памятью	219
21.2.2. Считываем файлы в отведенную память	222
Глава 22. Часто задаваемые вопросы	223
Глава 23. Область PSP и DTA. Системные переменные (окружение DOS).....	225
23.1. Структура командной строки.....	226
23.2. Системные переменные (окружение MS-DOS).....	227
23.3. Основной резидент	231
23.3.1. Команды безусловного перехода.....	232

23.3.2. Команды управления флагами	233
23.3.3. Изменение параметров резидента "на лету"	235
23.4. Задание для закрепления сведений из данной главы	237
Глава 24. Резидентный антивирус	238
24.1. Регистры микропроцессоров 80386/80486. Хранение чисел в памяти	238
24.1.1. 16- и 32-разрядные отладчики	240
24.1.2. Директива <i>use16/use32</i>	241
24.1.3. Сопоставление ассемблера и языков высокого уровня	241
24.2. Резидентный антивирус. Практика.....	242
24.3. Резюме	247
Глава 25. Работа с сопроцессором	248
25.1. Ответы на некоторые вопросы	248
25.2. Введение в работу с сопроцессором	249
25.3. Первая программа с использованием сопроцессора.....	254
25.4. Вывод десятичного числа с помощью сопроцессора	255
25.5. Оболочка.....	256
25.5.1. Получение и вывод длинного имени файла.....	256
Глава 26. История развития ПК	258
26.1. Краткая история развития IBM-совместимых компьютеров	258
26.2. С чего все начиналось	259
26.3. Оболочка.....	260
26.3.1. Чтение файлов из каталога и размещение их в отведенной памяти	261
26.3.2. Размещение файлов в памяти нашей оболочки.....	262
Глава 27. Удаление резидента из памяти.....	264
27.1. Обзор последнего резидента	264
27.1.1. Перехват прерывания <i>21h</i>	264
27.1.2. Как удалять загруженный резидент из памяти?	267
27.1.3. Случай, когда резидент удалить невозможно	268
27.2. Практика	269
Глава 28. Алгоритм считывания имен файлов в память	271
28.1. Новый алгоритм считывания файлов в память	271
28.2. Процедура вывода имен файлов на экран.....	273
28.3. Новые переменные в оболочке	274
28.4. Обработка клавиш <PageUp> и <PageDown>.....	276
28.5. Обработка клавиш <Home> и <End>	276
Глава 29. Загрузка и запуск программ	278
29.1. Подготовка к запуску программы и ее загрузка.....	278
29.1.1. Выделяем память для загружаемой программы.....	279
Зачем необходимо урезать память перед загрузкой?.....	279
Зачем во второй строке мы сдвигаем на 4 бита вправо это смещение?	280
А для чего увеличиваем <i>bx</i> на единицу (3)?.....	280
29.1.2. Переносим стек в область PSP	280
29.1.3. Подготовка EPB	281
Еще несколько слов о системных переменных (сегменте окружения DOS)	282
Для чего нужно создавать свое окружение DOS?	283
Сегмент и смещение командной строки	283
Первый и второй адрес блоков FCB	284

29.1.4. Сохранение регистров	284
29.1.5. Запуск программы.....	285
29.2. "Восстановительные работы"	286
Глава 30. Работа с расширенной памятью	288
30.1. Расширенная (XMS) память. Общие принципы.....	288
30.2. Программа XMSmem.asm. Получение объема XMS-памяти	289
30.2.1. Подготовка к использованию расширенной памяти и вывод объема XMS-памяти.....	289
30.3. Программа XMSblock.asm. Чтение файла в расширенную память и вывод его на экран	291
30.3.1. Работа с расширенной памятью.....	293
30.3.2. Структура массива при работе с XMS-памятью	293
30.4. Программа XMScopy.asm. Копирование файла с использованием расширенной памяти	294
Глава 31. Обзор дополнительных возможностей оболочки	296
31.1. Оболочка Super Shell	296
31.1.1. Вызов внешних вспомогательных программ	297
31.1.2. Редактирование файла	298
31.2. Антивирусные возможности оболочки	299
31.2.1. Как защитить компьютер от заражения его резидентными вирусами	299
31.2.2. Как защитить компьютер от программ-разрушителей дисковой информации	300
Глава 32. Все о диске и файловой системе	302
32.1. Что находится на диске?	302
32.1.1. Таблица разделов жесткого диска	302
32.1.2. Загрузочный сектор	303
32.1.3. Таблица размещения файлов (FAT)	304
32.2. Удаление и восстановление файла	305
32.3. Ошибки файловой системы	306
32.3.1. Потерянные кластеры файловой системы FAT, FAT32	306
ПРИЛОЖЕНИЯ	307
Приложение 1. Ассемблирование программ (получение машинного кода из ассемблерного листинга)	309
П1.1. Загрузка MASM 6.10—6.13	309
П1.2. Ассемблирование	309
П1.3. Компоновка	310
П1.3.1. Ассемблирование и компоновка программ пакетами Microsoft (MASM)	311
Приложение 2. Типичные ошибки при ассемблировании программы	312
Приложение 3. Таблицы и коды символов	313
П3.1. Основные символы ASCII	313
П3.2. Расширенные коды ASCII	320
П3.3. Скан-коды клавиатуры	322
Приложение 4. Содержимое компакт-диска	324
Предметный указатель	325

Предисловие

Итак, вы решили начать изучение языка ассемблера. Возможно, вы уже пробовали его изучать, но так и не смогли освоить до конца, поскольку он показался вам очень трудным. Обилие новых, неизвестных читателю терминов и сложность языка, которым написаны многие книги, делают их трудными для начинающих программистов. В этой книге автор старался излагать материал так, чтобы он был понятен любому пользователю: и начинающему программисту, и человеку, который ни разу не сталкивался ни с каким языком программирования.

Основой для книги послужили материалы разработанной автором рассылки "Ассемблер? Это просто! Учимся программировать". Используя данную рассылку, более 18 000 подписчиков научились писать такие программы на ассемблере, которые казались им раньше чрезвычайно сложными и недоступными для понимания или написания. Большая часть подписчиков пыталась раньше изучать язык ассемблера, но так и не смогла пройти полный курс (прочитать ту или иную книгу до конца). Материал рассылки помог им понять ассемблер и научил писать довольно-таки сложные программы под операционными системами MS-DOS и Windows.

Во втором издании автор учел и исправил все недоработки и ошибки, допущенные в первом. Более того, автор попытался сделать обучение как можно более интересным для вас, перейдя с первой же главы к практической части. Это поможет вам изучить базовые основы ассемблера за короткое время.

Автор не претендует на то, что материал, изложенный в данной книге, поможет вам освоить ассемблер во всех его проявлениях и покажет все возможности языка. Ассемблер настолько многогранен, что просто невозможно подробно описать все его операторы, команды, алгоритмы, области применения в одной книге. Тем не менее, прочитав уже несколько глав, вы сможете научиться писать собственные программы, разбирать чужие, а также поймете, как в целом работает компьютер.

Уникальность этой книги заключается в следующем:

- в *части I* книги рассматриваются базовые операторы ассемблера, основы программирования в реальном режиме (консольные приложения), в *части II* — основы программирования на ассемблере под Windows; в *части III* — как создать файловую оболочку и написать/нейтрализовать резидентный вирус;
- каждая глава соответствует одному занятию, в конце главы приводится файл для практического изучения;

- материал изложен на простом языке, все новые термины подробно объясняются;
- исследуется работа отладчиков и способы обойти отладку программы;
- в процессе изучения ассемблера, начиная с главы 10, рассматриваются четыре программы:
 - безобидный нерезидентный вирус;
 - резидентный антивирус против написанного нами вируса;
 - файловая оболочка (типа Norton Commander®, FAR Manager® и т. п.) с поддержкой длинных имен файлов и использованием XMS-памяти;
 - несколько видов резидентных программ (программ, которые постоянно находятся в памяти).

В ассемблере, как и в любом другом языке программирования, очень важна практика и опыт. На компакт-диске, прилагаемом к книге, приводятся готовые ассемблерные файлы в текстовом формате с подробными описаниями для практического изучения курса, а также необходимое ПО. На прилагаемом диске автор постарался собрать все, что нужно для полноценного изучения материала.

Несколько советов

- Обязательно скачайте файлы-приложения для практического изучения курса, а также необходимое ПО с сайта <http://www.Kalashnikoff.ru> (если таких нет). Без практики и вспомогательных программ данная книга вряд ли поможет вам научиться программировать на ассемблере.
- Чаще пользуйтесь отладчиком.
- Изменяйте код программ (файлов-приложений), больше экспериментируйте.
- Пробуйте написать собственную программу на основе изученного материала.
- Так как вначале будет довольно сложно ориентироваться в обилии инструкций, директив, прерываний, процедур ассемблера, то пробуйте вставлять в ваши собственные программы выдержки, процедуры, алгоритмы из файлов-приложений. Помните, что опыт приходит со временем!
- Внимательно следите за ходом мысли автора, за его логикой. Это особенно актуально при чтении *частей II и III*.
- *Не спешите!* Внимательно и досяконально изучайте каждую главу, выполняйте все, что автор просит сделать с прилагаемыми программами (запускать их под отладчиком, изменять код, думать над тем, что делает та или иная процедура и пр.).
- Все вопросы, которые у вас, несомненно, возникнут в процессе изучения ассемблера, вы можете задавать в любое время экспертам на портале профессионалов <http://RFpro.ru>. Этот сайт был специально разработан автором книги с целью оказания посетителям помощи по разным направлениям, в том числе и по ассемблеру.

Ответы на некоторые вопросы

1. Почему важно изучить работу процессора в реальном режиме (в MS-DOS) и только после этого переходить к программированию в защищенном режиме (в Windows)?

- Во-первых, для ассемблера не существует различий между операционными системами. Ассемблер — это язык самого процессора. Но он может использовать готовые подпрограммы операционной системы, на которой запущена программа.
- Во-вторых, функции WinAPI, о которых пойдет речь в данной книге, — это прототип прерываний MS-DOS. Но понять принцип работы WinAPI в разы проще на примерах прерываний, которые до сих пор поддерживаются Windows. Как только вы поймете данный принцип, вы без труда перейдете на использование WinAPI.
- В-третьих, учиться основам языка, создавая компактные COM-файлы, которые использовались в MS-DOS и до сих пор поддерживаются Windows, гораздо проще.

2. Под управлением каких операционных систем будут работать файлы-приложения?

Компания Microsoft придерживается политики поддержки работоспособности программ, написанных в более ранних версиях собственных операционных систем. Если программа разработана, например, для MS-DOS 3.30, то она будет выполняться и в более поздних версиях этой системы, если, конечно, в самой программе не установлены ограничения, или она не привязана в работе именно к той версии, для которой написана.

Появление на рынке ОС Windows 95/98/2000/XP/Vista продолжило эту традицию, оставив возможность загружать и выполнять программы, написанные под операционную систему MS-DOS, и даже запуск DOS-приложений, требующих загруженной "чистой" дисковой системы, к которым относятся, как правило, сложные графические игры, работающие с расширителями, например, DOS4GW.

Тем не менее, большинство программ прекрасно запускается напрямую из Продовника (Windows Explorer), не требуя перезагрузки системы. Например, Norton Commander, Far Manager, а также необходимые для изучения настоящего курса средства разработки и отладки, перечисленные далее.

В современных операционных системах компании Microsoft (Windows) есть возможность работать в эмуляторе MS-DOS: **Пуск | Выполнить | cmd | <Enter>**. В открывшемся окне и можно запускать все файлы из данной книги, не обращая внимания на то, что наши COM-программы используют прерывания MS-DOS.

Все примеры протестированы на работоспособность под управлением следующих операционных систем компаний Microsoft на IBM-совместимых компьютерах:

- Windows 2000 Pro и Server;
- Windows XP Home Edition и Pro;
- Windows Vista/Server 2008/7.

ВНИМАНИЕ!

На ассемблере мы будем напрямую взаимодействовать с аппаратурой, что не очень приветствуется Windows. Поэтому некоторые программы из учебного курса могут работать некорректно. Что делать в таких случаях? Все просто: отладчик вам поможет!

3. Какое программное обеспечение нужно для того, чтобы создать программу на ассемблере, и где его можно достать?

- Прежде всего, это *текстовый редактор*, как отдельный, например, Akelpad, так и встроенный в какую-нибудь оболочку (например, Far Manager). В принципе, сгодится даже обычный Блокнот, т. к. ассемблерный код — это обычные текстовые файлы. Мы рекомендуем пользоваться встроенным редактором Far Manager (<F4>). Думаем, что не следует заострять внимание на том, как пользоваться данными программами, тем более, что это выходит за рамки настоящей книги.
- Потребуется *ассемблер* — программа, которая переводит ассемблерные инструкции в машинный код. Это может быть MASM.EXE® (ML.EXE) компании Microsoft, TASM.EXE® компании Borland, FASM® или другие. Для программирования на ассемблере под Windows потребуется MASM32®. Скачать все это можно бесплатно на сайте автора книги — <http://Kalashnikoff.ru>. В принципе, большой разницы для наших примеров это пока не имеет, за исключением передачи параметров в командной строке при ассемблировании. Мы будем использовать MASM 6.11 — Macro Assembler® от Microsoft версии 6.11 для программ в *части I* книги и MASM32 — для программ в *части II*, что и вам советую. Если в процессе ассемблирования возникают ошибки, то обращайтесь к *приложению 2* или к нашим экспертам на <http://RFpro.ru>.
- Настоятельно рекомендую иметь *отладчик* (AFD®, SoftIce®, CodeView®). Он необходим для отладки программы и в целом для демонстрации ее работы. Предпочтительно использовать AFD или CodeView для начинающих и SoftIce для уже имеющих опыт программирования.
- В будущем вам, безусловно, понадобится *дизассемблер*, который необходим для перевода машинного кода на язык ассемблера. Автор предпочитает IDA®, как один из самых мощных и удобных в пользовании.

Можно также скачать минимальный, но достаточный для изучения настоящего курса набор программного обеспечения по адресу: <http://www.Kalashnikoff.ru>.

4. Как построены главы книги?

- Ответы на часто задаваемые вопросы.
- Заметки, дополнительные примеры и алгоритмы.
- Объяснение новой темы (теория).
- Примеры программ на ассемблере (практика).

Вы сможете самостоятельно написать простую программу уже после прочтения *главы 1*. Надеюсь, что изучать язык будет интересней, если мы сразу перейдем к практической части, обсуждая параллельно теорию. Попутно отмечу, что данная книга рассчитана, в первую очередь, на людей, которые ни разу не писали про-

граммы ни на ассемблере, ни на каком другом языке программирования. Конечно, если вы уже знакомы с Basic, Pascal, С или каким-либо иным языком, то это только на пользу вам. Тем не менее, все новые термины будут подробно объясняться.

Также следует отметить, что для полного изучения курса необходимы минимальные пользовательские знания операционной системы MS-DOS, т. к. ассемблирование программ из данной книги следует выполнять именно в консоли (**Пуск | Выполнить | cmd**). Однако вы также можете работать в специальных файловых оболочках типа Far Manager, Windows Commander, Total Commander и т. п.

5. Какие темы будут рассмотрены в книге?

- Двоичная и шестнадцатеричная системы счисления.
- Основные команды процессоров Intel 8086, 80286, 80386, 80486.
- 16- и 32-разрядные регистры.
- Основы работы с сопроцессором.
- Сегментация памяти в реальном режиме.
- Расширенная память (XMS-память).
- Прямая работа с видеоадаптером.
- Режимы CGA, EGA, VGA (кратко).
- Управление клавиатурой на уровне прерываний.
- Основные функции BIOS (ПЗУ) и MS-DOS.
- Работа с дисками, каталогами и файлами.
- Управление последовательным портом.
- Высокоуровневая оптимизация программ.
- Структура и особенности программирования в MS-DOS и Windows.
- Не обойдем стороной и технический английский язык, т. к. операторы ассемблера образованы от английских слов.

6. Кому можно задать вопросы, касающиеся материала из данной книги?

На все ваши вопросы по ассемблеру, а также по многим другим темам, ответят наши эксперты на портале профессионалов <http://RFpro.ru>. Стоит отметить, что на упомянутом портале вы сможете:

- установить контакт с начинающими программистами и профессионалами на ассемблере;
- пообщаться в реальном времени с автором данной книги;
- принять участие в реальных встречах, чтобы лично познакомиться с профессионалами и экспертами.

Зарегистрируйтесь прямо сейчас на портале <http://RFpro.ru> и вступайте в наш клуб профессионалов!

В главе 1 мы рассмотрим шестнадцатеричную систему счисления и пример простейшей программы на ассемблере, традиционно называемой "Hello, world!".

Приятного вам изучения!



ЧАСТЬ I

Знакомьтесь: ассемблер



Глава 1

Первая программа

1.1. Шестнадцатеричная система счисления

Для написания программ на ассемблере необходимо разобраться с шестнадцатеричной системой счисления. Ничего сложного в ней нет. Мы используем в жизни десятичную систему. Не сомневаемся, что вы с ней знакомы, поэтому попробуем объяснить шестнадцатеричную систему, проводя аналогию с десятичной.

Итак, в десятичной системе, если мы к какому-нибудь числу справа добавим ноль, то это число увеличится в 10 раз. Например:

$$1 \times 10 = 10$$

$$10 \times 10 = 100$$

$$100 \times 10 = 1000$$

и т. д.

В этой системе мы используем цифры от 0 до 9, т. е. десять разных цифр (существенно, поэтому она и называется десятичной).

В шестнадцатеричной системе мы используем, соответственно, шестнадцать "цифр". Слово "цифр" специально написано в кавычках, т. к. в этой системе используются не только цифры. От 0 до 9 мы считаем так же, как и в десятичной, а вот дальше таким образом: A, B, C, D, E, F. Число F, как не трудно посчитать, будет равно 15 в десятичной системе (табл. 1.1).

Таблица 1.1. Десятичная и шестнадцатеричная системы

Десятичное число	Шестнадцатеричное число	Десятичное число	Шестнадцатеричное число
0	0	26	1A
1	1	27	1B
2	2	28	1C
3	3	29	1D
4	4	30	1E
...
8	8	158	9E

Таблица 1.1 (окончание)

Десятичное число	Шестнадцатеричное число	Десятичное число	Шестнадцатеричное число
9	9	159	9F
10	A	160	A0
11	B	161	A1
12	C	162	A2
13	D
14	E	254	FE
15	F	255	FF
16	10	256	100
17	11	257	101
...

Таким образом, если мы к какому-нибудь числу в шестнадцатеричной системе добавим справа ноль, то это число увеличится в 16 раз (пример 1.1).

Пример 1.1

$$1 \times 16 = 10$$

$$10 \times 16 = 100$$

$$100 \times 16 = 1000$$

и т. д.

Вы смогли отличить в примере 1.1 шестнадцатеричные числа от десятичных? А из этого ряда: 10, 12, 45, 64, 12, 8, 19? Это могут быть как шестнадцатеричные числа, так и десятичные. Для того чтобы не было путаницы, а компьютер и программист смогли бы однозначно отличить одни числа от других, в ассемблере принято после шестнадцатеричного числа ставить символ *h* или *H* (от англ. *hexadecimal* — шестнадцатеричное), который для краткости часто называют просто *hex*. После десятичного числа, как правило, ничего не ставят. Так как числа от 0 до 9 в обеих системах имеют одинаковые значения, то числа, записанные как 5 и 5*h*, — одно и то же. Таким образом, корректная запись чисел из примера 1 будет следующей (примеры 1.2 и 1.3).

Пример 1.2. Корректная форма записи чисел

$$1 \times 16 = 10h$$

$$10h \times 16 = 100h$$

$$100h \times 16 = 1000h$$

Пример 1.3. Другой вариант записи чисел

$1h \times 10h = 10h$
 $10h \times 10h = 100h$
 $100h \times 10h = 1000h$

Для чего нужна шестнадцатеричная система и в каких случаях она применяется — мы рассмотрим в следующих главах. А в данный момент для нашего примера программы, который будет рассмотрен далее, нам необходимо знать о существовании шестнадцатеричных чисел.

Итак, настала пора подвести промежуточный итог. Шестнадцатеричная система счисления состоит из 10 цифр (от 0 до 9) и 6 букв латинского алфавита (A, B, C, D, E, F). Если к какому-нибудь числу в шестнадцатеричной системе добавить справа ноль, то это число увеличится в 16 раз. Очень важно уяснить принцип шестнадцатеричной системы счисления, т. к. мы будем постоянно использовать ее при написании наших программ на ассемблере.

Теперь немного о том, как будут строиться примеры на ассемблере в данной книге. Не совсем удобно приводить их сплошным текстом, поэтому сперва будет идти сам код программы с пронумерованными строками, а сразу же после него — объяснения и примечания. Примерно так, как показано в листинге 1.1.

Листинг 1.1. Пример записи ассемблерных инструкций, применяемой в книге

```
...
(01)    mov ah,9
(02)    mov al,8
...
(15)    mov dl,5Ah
...
...
```

Обратите внимание, что номера строк ставятся только в книге, и при наборе программ в текстовом редакторе эти номера ставить НЕ нужно! Номера строк ставятся для того, чтобы удобно было давать объяснения к каждой строке: в строке (01) мы делаем то-то, а в строке (15) — то-то.

Несмотря на то, что на компакт-диске, прилагаемом к книге, имеются набранные и готовые для ассемблирования программы, мы рекомендуем все-таки первое время набирать их самостоятельно. Это ускорит запоминание операторов, а также облегчит привыкание к самому языку.

И еще момент. Строчные и ПРОПИСНЫЕ символы программой-ассемблером не различаются. Записи вида:

```
mov ah,9
и
MOV AH,9
```

воспринимаются одинаково. Можно, конечно, заставить ассемблер различать регистр, но мы пока этого делать не будем. Для удобства чтения программы лучше всего операторы вводить строчными буквами, а названия подпрограмм и меток начинать с прописной.

1.2. Наша первая программа

Итак, переходим к нашей первой программе (\001\prog01.asm) (листинг 1.2).

Листинг 1.2. Наша первая программа на ассемблере

```
(01) CSEG segment  
(02) org 100h  
(03)  
(04) Begin:  
(05)  
(06)     mov ah,9  
(07)     mov dx,offset Message  
(08)     int 21h  
(09)  
(10)    int 20h  
(11)  
(12) Message db 'Hello, world!$'  
(13) CSEG ends  
(14) end Begin
```

Еще раз обратим внимание: когда вы будете перепечатывать примеры программ, то номера строк ставить не нужно!

В скобках указывается имя файла из архива файлов-приложений (в данном случае — \001\prog01.asm, где 001 — каталог, prog01.asm — имя ассемблерного файла в DOS-формате).

Прежде чем пытаться ассемблировать, прочтите данную главу до конца!

Для того чтобы объяснить все операторы из листинга 1.2, нам потребуется несколько глав. Поэтому описание некоторых команд мы на данном этапе опустим. Просто считайте, что так должно быть. В ближайшее время мы рассмотрим эти операторы подробно. Итак, строки с номерами (01), (02) и (13) вы игнорируете. Строки (03), (05), (09) и (11) остаются пустыми. Это делается для наглядности и удобства программиста при просмотре и анализе кода. Программа-ассемблер пустые строки опускает.

Теперь перейдем к рассмотрению остальных операторов. Со строки (04) начинается код программы. Это метка, указывающая ассемблеру на начало кода. В строке (14) стоят операторы end Begin (Begin — начало; end — конец). Это конец программы. Вообще вместо слова Begin можно было бы использовать любое

другое. Например, `Start`. В таком случае, нам пришлось бы и завершать программу оператором `End Start` (14).

Строки (06)–(08) выводят на экран сообщение "Hello, world!". Здесь придется вкратце рассказать о регистрах процессора (более подробно эту тему мы рассмотрим в последующих главах).

Регистр процессора — это специально отведенная память для хранения какого-нибудь числа. Например, если мы хотим сложить два числа, то в математике запишем так:

$$A = 5$$

$$B = 8$$

$$C = A + B$$

A , B и C — это своего рода регистры (если говорить о компьютере), в которых могут храниться некоторые данные. $A = 5$ следует читать как: "присваиваем A число 5".

Для присвоения регистру какого-нибудь значения в ассемблере существует оператор `mov` (от англ. *move* — в данном случае "загрузить"). Строку (06) следует читать так: "загружаем в регистр `ah` число 9" (проще говоря, присваиваем `ah` число 9). Далее рассмотрим, зачем это необходимо. В строке (07) загружаем в регистр `dx` адрес сообщения для вывода (в данном примере это будет строка "Hello, world!\$"). Затем, в строке (08) вызываем прерывание MS-DOS, которое и выведет нашу строку на экран. Прерывания будут подробно рассматриваться в последующих главах, мы же пока коснемся только самых элементарных вещей.

1.3. Введение в прерывания

Прерывание MS-DOS — это своего рода подпрограмма (часть MS-DOS), которая находится постоянно в памяти и может вызываться в любое время из любой программы. Рассмотрим высказанное на примере (листинг 1.3).

Сразу стоит отметить, что в ассемблере после точки с запятой располагаются *комментарии*. Комментарии будут опускаться MASM/TASM при ассемблировании. Примеры комментариев:

```
;это комментарий
```

```
mov ah, 9 ;это комментарий
```

В комментарии программист вставляет замечания по программе, которые помогают сориентироваться в коде.

Листинг 1.3. Программа (алгоритм) сложения двух чисел

НачалоПрограммы

```
A=5 ;в переменную А заносим значение 5
B=8 ;в переменную В значение 8
```

ВызовПодпрограммы Addition

```
;теперь С равно 13
```

```
A=10 ;то же самое, только другие числа
```

B=25

ВызовПодпрограммы Addition

; теперь С равно 35

КонецПрограммы

; выходим из программы

...

Подпрограмма Addition

C = A + B

ВозвратИзПодпрограммы

; возвращаемся в то место, откуда вызывали

КонецПодпрограммы

В данном примере мы дважды вызвали подпрограмму (процедуру) `Addition`, которая произвела сложение двух чисел, переданных ей в переменных `A` и `B`. Результат математического действия сохраняется в переменной `C`. Когда вызывается подпрограмма, компьютер запоминает, с какого места она была вызвана, и после того, как процедура отработала, возвращается в то место, откуда она вызывалась. Таким образом, можно вызывать подпрограммы неопределенное количество раз с любого участка основной программы.

При выполнении строки (08) (см. листинг 1.2) мы вызываем подпрограмму (в данном случае это называется прерыванием), которая выводит на экран строку. Для этого мы, собственно, и помещаем нужные значения в регистры, т. е. готовим для прерывания необходимые параметры. Всю работу (вывод строки, перемещение курсора) берет на себя эта процедура. Строку (08) следует читать так: "вызываем двадцать первое прерывание" (`int` от англ. `interrupt` — прерывание). Обратите внимание, что после числа 21 стоит буква `h`. Это, как мы уже знаем, шестнадцатеричное число (33 в десятичной системе). Конечно, нам ничего не мешает заменить строку `int 21h` строкой `int 33`. Программа будет работать корректно. Но в ассемблере принято указывать номера прерываний в шестнадцатеричной системе, да и все отладчики работают с этой системой.

В строке (10) мы, как вы уже догадались, вызываем прерывание `20h`. Для его вызова не нужно указывать какие-либо значения в регистрах. Оно выполняет только одну задачу — выход из программы (выход в DOS). В результате выполнения прерывания `20h` программа вернется туда, откуда ее запускали (загружали, вызывали). Например, в Norton Commander или DOS Navigator. Это что-то вроде оператора `exit` в некоторых языках высокого уровня.

Строка (12) содержит сообщение для вывода. Первое слово (`message` — сообщение) — название этого сообщения. Оно может быть любым (например, `mess` или `string` и пр.). Обратите внимание на строку (07), в которой мы загружаем в регистр `dx` адрес этого сообщения.

Можно создать еще одну строку, которую назовем Mess2. Затем, начиная со строки (09), вставим в нашу программу следующие команды:

```
...
(09)    mov ah,9
(10)   mov dx,offset Mess2
(11)   int 21h
(12)   int 20h
(13)  Message db 'Hello, world!$'
(14)  Mess2 db 'Это я!$'
(15) CSEG ends
(16) end Begin
```

Уверены, вы поняли, что именно произойдет.

Обратите внимание на последний символ в строках Message и Mess2 — \$. Он указывает на конец выводимой строки. Если мы его уберем, то прерывание 21h продолжит вывод до тех пор, пока не встретится где-нибудь в памяти тот самый символ \$. На экране, помимо нашей строки, мы увидим "мусор" — разные символы, которых в строке вовсе нет.

Теперь ассемблируйте программу. Как это сделать — написано в *приложении 1*. Заметьте, что мы создаем пока только СОМ-файлы, а не EXE! Для того чтобы получить СОМ-файл, нужно указать определенные параметры ассемблеру (MASM/TASM) в командной строке. Пример получения СОМ-файла с помощью Macro Assembler версии 6.11 и результат выполнения программы приведен на рис. 1.1. При возникновении ошибок в процессе ассемблирования обращайтесь к *приложению 2*, где рассматриваются типичные ошибки при ассемблировании программ.

Если у вас есть отладчик (AFD, CodeView), то можно (и даже нужно!) запустить эту программу под его управлением. Это поможет вам лучше понять структуру и принцип работы ассемблера, а также продемонстрирует реальную работу написанной нами программы.

The screenshot shows a Microsoft Macro Assembler window titled '{C:\Мои документы\Book\Enclosures\Файлы-приложения\001} - Far'. The status bar indicates the time as 19:46. The window displays the following text:

```
C:\...\Enclosures\Файлы-приложения\001>ml.exe Prog01.asm /AT
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

Assembling: Prog01.asm

Microsoft (R) Segmented Executable Linker Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: Prog01.obj/t
Run File [Prog01.com]: "Prog01.com"
List File [nul.map]: NUL
Libraries [.lib]:
Definitions File [nul.def]:
C:\...\Enclosures\Файлы-приложения\001>PROG01.COM
Hello, world?

C:\...\Enclosures\Файлы-приложения\001>
```

The bottom status bar shows file navigation icons: 1 Помог, 2 Распак, 3 ЭрхКом, 4 Редакр., 5 Копир, 6 Перенр., 7 Удален, 8 Сохран, 9 Последн.

Рис. 1.1. Ассемблирование и результат выполнения программы Prog01.com

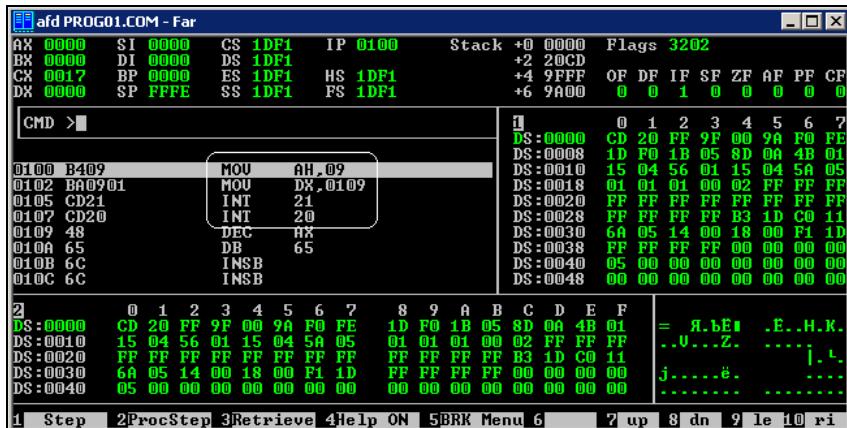


Рис. 1.2. Вид программы в отладчике AFD Pro

На рис. 1.2 показано, как эта программа выглядит в отладчике AFD Pro. Пока не обращайте особого внимания на различие между реальным кодом, набранным руками, и тем, как эта программа отображается в отладчике. Подробно работу отладчика мы рассмотрим в последующих главах.

1.4. Резюме

Целью данной главы не было разобраться подробно с каждым оператором. Это невозможно, если вы не обладаете базовыми знаниями. Но, прочитав 3—4 главы, вы поймете принцип и структуру программы на ассемблере.

Может быть, ассемблер вам показался чрезвычайно сложным, но это, поверьте, только с первого взгляда. Вы должны научиться строить алгоритм программы на ассемблере в голове, а для этого нужно будет самостоятельно написать несколько программ, опираясь на информацию из данной книги. Будем постепенно учиться мыслить структурой ассемблера, составлять алгоритмы, программы, используя операторы языка. После изучения очередной главы вы будете чувствовать, что постепенно начинаете осваивать ассемблер, будет становиться все проще и проще.

Например, если вы знакомы с Бейсиком, то, ставя перед собой задачу написать программу, выводящую 10 слов "Привет", вы будете использовать операторы FOR, NEXT, PRINT и пр., которые тут же появятся в ваших мыслях. Вы строите определенный алгоритм программы из этих операторов, который в какой-то степени применим только к Бейсику. То же самое и с ассемблером. При постановке задачи написать ту или иную программу вы мысленно создаете алгоритм, который применим к ассемблеру и только, т. к. языков, похожих на ассемблер, просто не существует. Наша задача — научить вас создавать в уме алгоритмы, применимые к ассемблеру, т. е. образно говоря, научить "мыслить на ассемблере".

* * *

В главе 2 мы подробно рассмотрим регистры процессора и напишем еще одну простую программу.



Глава 2

Регистры процессора

2.1. Введение в регистры микропроцессоров 8086—80186

Регистр, как мы уже говорили ранее, — это специально отведенная память для временного хранения каких-то данных. Микропроцессоры 8086—80186 имеют 14 регистров. В главе 1 мы познакомились с двумя из них: ah и dx. В табл. 2.1, 2.3 и 2.4 приведен перечень всех регистров, кроме ip и регистра флагов, которые будут рассмотрены отдельно.

2.1.1. Регистры данных

Регистры данных могут использоваться программистом по своему усмотрению (за исключением некоторых случаев). В них можно хранить любые данные: числа, адреса и пр. В верхнем ряду табл. 2.1 находятся 32-разрядные регистры, которые могут хранить числа от 0 до 4 294 967 295 (0FFFFFFFh). Их мы будем рассматривать позже. Во втором ряду — 16-разрядные, которые могут хранить числа от 0 до 65 535 или от 0h до FFFFh в шестнадцатеричной системе, что одно и то же.

В следующей строке расположен ряд 8-разрядных регистров: ah, al, bh, bl, ch, cl, dh, dl. В эти регистры можно загружать максимальное число 255 (FFh). Это так называемые половинки (старшая или младшая) 16-разрядных регистров.

Таблица 2.1. Регистры данных

EAX				EBX			
ax		bx		cx		dx	
ah	al	bh	bl	ch	cl	dh	dl
Аккумулятор		База		Счетчик		Регистр данных	

Мы уже изучили оператор `mov`, который предназначен для загрузки числа в регистр. Чтобы присвоить, к примеру, регистру al число 35h, нам необходимо записать так:

```
mov al, 35h
```

в регистру `ax` — число 346Ah, так:

```
mov ax, 346Ah
```

Если мы попытаемся загрузить большее число, чем может содержать регистр, то при асSEMBЛИРОВании программы произойдет ошибка. Например, следующие записи будут ошибочны:

<code>mov ah, 123h</code>	→ максимум FFh
<code>mov bx, 12345h</code>	→ максимум FFFFh
<code>mov dl, 100h</code>	→ максимум FFh

Здесь надо отметить, что если шестнадцатеричное число начинается не с цифры (12h), а с буквы (C5h), то перед таким числом ставится ноль: 0C5h. Это необходимо для того, чтобы программа-ассемблер могла отличить, где шестнадцатеричное число, а где название переменной или метки. Далее мы рассмотрим это на примере.

Допустим, процессор выполняет команду `mov ax, 1234h`. В этом случае в регистр `ah` загружается число 12h, а в регистр `al` — 34h. То есть `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh` и `dl` — это младшие (**Low**) или старшие (**High**) половинки 16-разрядных регистров (табл. 2.2).

Таблица 2.2. Результаты выполнения различных команд

Команда	Результат
<code>mov ax, 1234h</code>	<code>ax = 1234h, ah = 12h, al = 34h</code>
<code>mov bx, 5678h</code>	<code>bx = 5678h, bh = 56h, bl = 78h</code>
<code>mov cx, 9ABCCh</code>	<code>cx = 9ABCCh, ch = 9Ah, cl = 0BCCh</code>
<code>mov dx, 0DEF0h</code>	<code>dx = 0DEF0h, dh = 0DEh, dl = 0F0h</code>

2.1.2. Регистры-указатели

Регистры `si` (индекс источника) и `di` (индекс приемника) используются в строковых операциях. Регистры `bp` и `sp` задействуются при работе со стеком (табл. 2.3). Мы подробно их рассмотрим на примерах в следующих главах.

Таблица 2.3. Регистры-указатели

si	Di	bp	sp
Индекс источника	Индекс приемника	Регистры для работы со стеком	

2.1.3. Сегментные регистры

Сегментные регистры (табл. 2.4) необходимы для обращения к тому или иному сегменту памяти (например, видеобуферу). Сегментация памяти — довольно сложная и объемная тема, которую также будем рассматривать в следующих главах.

Таблица 2.4. Сегментные регистры

CS	DS	ES	SS
Регистр кода	Регистр данных	Дополнительный регистр	Регистр стека

2.2. Команды сложения и вычитания

Для выполнения арифметических операций сложения и вычитания в ассемблере существуют следующие операторы: add, sub, inc, dec.

2.2.1. Оператор add

Формат оператора add показан в табл. 2.5. Впоследствии мы всегда будем оформлять новые команды в подобные таблицы. В столбце **Команда** будет описана новая команда и ее применение. В столбце **Назначение** — что выполняет или для чего служит данная команда, а в столбце **Процессор** — модель (тип) процессора, начиная с которой команда поддерживается. В столбце **Перевод** будет указано, от какого английского слова образовано название оператора, и дан перевод этого слова.

Таблица 2.5. Оператор add

Команда	Перевод	Назначение	Процессор
add приемник, источник	Addition — сложение	Сложение	8086

В данном примере оператор поддерживается процессором 8086, но работать команда будет, естественно, и на более современных процессорах (80286, 80386, 80486, Pentium и т. д.).

Команда add производит сложение двух чисел (листинг 2.1).

Листинг 2.1. Примеры использования оператора add

```

mov al,10      ;загружаем в регистр al число 10
add al,15      ;al = 25; al — приемник, 15 — источник
mov ax,25000   ;загружаем в регистр ax число 25000
add ax,10000   ;ax = 35000; ax — приемник, 10000 — источник
mov cx,200     ;загружаем в регистр cx число 200
mov bx,760     ;a в регистр bx — 760
add cx,bx      ;cx = 960, bx = 760 (bx не меняется); cx — приемник,
                ;bx — источник

```

2.2.2. Оператор sub

Команда sub производит вычитание двух чисел (табл. 2.6, листинг 2.2).

Таблица 2.6. Оператор sub

Команда	Перевод	Назначение	Процессор
sub приемник, источник	Subtraction — вычитание	Вычитание	8086

Листинг 2.2. Примеры использования оператора `sub`

```
mov al,10
sub al,7           ;al = 3; al — приемник, 7 — источник
mov ax,25000
sub ax,10000       ;ax = 15000; ax — приемник, 10000 — источник
mov cx,100
mov bx,15
sub cx,bx
```

Это интересно

Следует отметить, что ассемблер — максимально быстрый язык. Можно посчитать, сколько раз за одну секунду процессор сможет сложить два любых числа от 0 до 65 535.

Каждая команда процессора выполняется определенное количество тактов. Когда говорят, что тактовая частота процессора 100 МГц, то это значит, что за секунду проходит 100 миллионов тактов. Чтобы компьютер сложил два числа, ему нужно выполнить следующие команды:

```
...
mov ax,2700
mov bx,15000
add ax,bx
...
```

В результате выполнения данных инструкций в регистре `ax` будет число 17 700, а в регистре `bx` — 15 000. Команда `add ax,bx` выполняется за один такт на процессоре 80486. Получается, что компьютер 486 DX2-66 МГц за одну секунду сложит два любых числа от 0 до 0FFFFh 66 миллионов (!) раз!

2.2.3. Оператор `inc`

Формат оператора `inc` представлен в табл. 2.7.

Таблица 2.7. Оператор `inc`

Команда	Перевод	Назначение	Процессор
<code>inc приемник</code>	<code>Increment</code> — инкремент	Увеличение на единицу	8086

Команда `inc` увеличивает на единицу содержимое приемника (регистра или ячейки памяти). Она эквивалентна команде:

`add источник, 1`

только выполняется быстрее на старых компьютерах (до 80486) и занимает меньше байтов (листинг 2.3).

Листинг 2.3. Примеры использования оператора inc

```

mov al,15
inc al          ;теперь al = 16 (эквивалентна add al,1)
mov dh,39h
inc dh          ;dh = 3Ah (эквивалентна add dh,1)
mov cl,4Fh
inc cl          ;cl = 50h (эквивалентна add cl,1)

```

2.2.4. Оператор dec

Формат оператора `dec` представлен в табл. 2.8.

Таблица 2.8. Оператор dec

Команда	Перевод	Назначение	Процессор
<code>dec приемник</code>	Decrement — декремент	Уменьшение на единицу	8086

Команда `dec` уменьшает на единицу содержимое приемника (листинг 2.4). Она эквивалентна команде:

```
sub источник, 1
```

Листинг 2.4. Примеры использования оператора dec

```

mov al,15
dec al          ;теперь al = 14
mov dh,3Ah
dec dh          ;dh = 39h
mov cl,50h
dec cl          ;cl = 4Fh

```

2.3. Программа для практики

Рассмотрим одну небольшую программу, которая выводит на экран сообщение и ждет, когда пользователь нажмет любую клавишу. После чего возвращается в DOS.

Работать с клавиатурой позволяет прерывание BIOS (ПЗУ) `16h`, которое можно вызывать даже до загрузки операционной системы, в то время как прерывания `20h`, `21h` и пр. доступны только после загрузки `IO.SYS/MSDOS.SYS` — определенной части ОС MS-DOS.

Чтобы заставить программу ждать нажатия пользователем любой клавиши, следует вызвать функцию `10h` прерывания `16h`. Вот как это выглядит на практике:

```

mov ah,10h ;в ah всегда указывается номер функции
int 16h    ;вызываем прерывание 16h — сервис работы с клавиатурой BIOS (ПЗУ)

```

После нажатия любой клавиши компьютер продолжит выполнять программу, а регистр `ax` будет содержать код клавиши, которую нажал пользователь.

Следующая программа (`\002\prog02.asm`) выводит на экран сообщение и ждет нажатия любой клавиши, что равнозначно команде `PAUSE` в BAT-файлах (листинг 2.5).

Листинг 2.5. Программа для практики

```
(01) CSEG segment
(02) org 100h
(03) Start:
(04)
(05)     mov ah,9
(06)     mov dx,offset String
(07)     int 21h
(08)
(09)     mov ah,10h
(10)    int 16h
(11)
(12)    int 20h
(13)
(14) String db 'Нажмите любую клавишу...$'
(15) CSEG ends
(16) end Start
```

Строки с номерами (01), (02) и (15) пока опускаем. В строках (05)—(07), как вы уже знаете, производится вывод строки на экран. Затем (строки (09), (10)) программа ждет нажатия клавиши. И наконец, строка (12) завершает работу нашей программы.

Мы уже изучили операторы `inc`, `dec`, `add` и `sub`. Вы можете поэкспериментировать (лучше в отладчике) с числами. Например, вот так:

```
...
mov ah,0Fh
inc ah
int 16h
...
```

Это позволит вам лучше запомнить новые операторы.

* * *

В главе 3 рассмотрим двоичную систему счисления, основы сегментации памяти и сегментные регистры. Напишем интересную программу.



Глава 3

Сегментация памяти в реальном режиме

В данной главе мы рассмотрим основополагающие принципы программирования на языке ассемблера. Необходимо тщательно разобраться в каждом предложении, уяснить двоичную систему счисления и понять принцип сегментации памяти в реальном режиме. Мы также рассмотрим операторы ассемблера, которые не затрагивали в примерах из предыдущих глав. Сразу отмечу, что это одна из самых сложных глав данной книги. Автор попытался объяснить все как можно проще, избегая сложных определений и терминов. Если что-то не поняли — не пугайтесь! Со временем все станет на свои места. Если вы полностью разберетесь с материалом данной главы, то считайте, что базу ассемблера вы изучили. Начиная с главы 4, будем изучать язык намного интенсивней.

Для того чтобы лучше понять сегментацию памяти, нам нужно воспользоваться отладчиком. Лучше использовать в работе два отладчика: CodeView (CV.EXE) и AFD Pro (AFD.EXE). Допустим, вы написали программу на ассемблере и назвали ее prog03.asm. Сассемблировав, вы получили файл prog03.com. Тогда, чтобы запустить программу под отладчиком CodeView/AFD, необходимо набрать в командной строке MS-DOS следующее:

CV.EXE prog03.com

либо:

AFD.EXE prog03.com

Итак, вдохните глубже и — вперед!

3.1. Двоичная система счисления. Бит и байт

Рассмотрим, как в памяти компьютера хранятся данные. Вообще, как компьютер может хранить, например, слово "диск"? Главный принцип — намагничивание и размагничивание одной дорожки (назовем это так). Одна микросхема памяти — это, грубо говоря, огромное количество дорожек (примерно как на магнитофонной кассете). Сейчас попробуем разобраться.

Предположим, что:

Ноль будет обозначаться как 0000 (четыре нуля),

Один 0001,

Два 0010 (т. е. правую единицу меняем на ноль,
а вторую устанавливаем в 1).

Далее так:

Три	0011
Четыре	0100
Пять	0101
Шесть	0110
Семь	0111
Восемь	1000
Девять	1001

и т. д.

"Нули" и "единицы" — это так называемые биты. Один бит, как вы уже заметили, может иметь только два значения — 0 или 1, т. е. размагнечена или намагнечена та или иная дорожка ("0" и "1" — это условное обозначение). Если внимательно посмотреть, то можно обнаружить, что каждый следующий установленный бит, начиная справа, увеличивает число в два раза: 0001 в нашем примере — один; 0010 — два; 0100 — четыре; 1000 — восемь и т. д. Это и есть двоичная форма представления данных. Чтобы обозначить числа от 0 до 9, нам нужно четыре бита (хоть они и не будут до конца использованы; можно было бы продолжить: десять — 1010, одиннадцать — 1011, ..., пятнадцать — 1111).

Компьютер хранит данные в памяти именно так. Для обозначения какого-нибудь символа (цифры, буквы, запятой, точки и др.) компьютер использует определенное количество бит. Компьютер "распознает" 256 (от 0 до 255) различных символов по их коду. Этого достаточно, чтобы вместить все цифры (0—9), буквы латинского алфавита (a—z, A—Z), русского (а—я, А—Я) и др. (см. *приложение 3*). Для представления символа с максимально возможным кодом (255) нужно 8 бит. Эти 8 бит называются байтом. Таким образом, один любой символ — это всегда 1 байт (табл. 3.1).

Таблица 3.1. Один байт с кодом символа "Z"

0	1	0	1	1	0	1	0
P	H	P	H	H	P	H	P

ПРИМЕЧАНИЕ

Символы "H" и "P" в таблице обозначают "намагнично" или "размагнично" соответственно.

Можно элементарно проверить. Создайте в текстовом редакторе файл с любым именем и напечатайте в нем один символ, например, "M", но не нажимайте клавишу <Enter>. Если вы посмотрите его размер, то файл будет равен 1 байту. Если ваш редактор позволяет смотреть файлы в шестнадцатеричном формате, то вы сможете узнать и код сохраненного символа. В данном случае — большая буква "M" имеет код 4Dh в шестнадцатеричной системе, которую мы уже знаем, или 1001101 в двоичной. Таким образом, слово "диск" будет занимать 4 байта или $4 \times 8 = 32$ бита. Как вы уже поняли, компьютер хранит в памяти не сами буквы (символы) этого слова, а последовательность "единичек" и "ноликов".

"В таком случае, почему на экране мы видим набор символов (текст, предложения, слова), а не "единички-нолики"?" — спросите вы. Чтобы удовлетворить ваше любопытство, забежим немного вперед и отметим, что всю работу по выводу самого символа (а не битов) на экран выполняет видеокарта (видеоадаптер), которая находится в вашем компьютере. И если бы ее не было, то мы, естественно, на экране ничего бы не увидели.

В ассемблере после двоичного числа всегда должен стоять символ `b`. Это нужно для того, чтобы в процессе обработки нашего файла ассемблер-программа смогла различать десятичные, шестнадцатеричные и двоичные числа. Например: `10` — это десять, `10h` — это шестнадцать, а `10b` — это два. Таким образом, в регистры можно загружать двоичные, десятичные и шестнадцатеричные числа. Например:

```
...
mov ax,20
mov bh,10100b
mov cl,14h
...
```

В результате в регистрах `ax`, `bh` и `cl` будет находиться одно и то же число, только загружаем мы его, используя разные системы счисления. В компьютере же оно будет храниться в двоичном формате (как в регистре `bh`).

Итак, подведем итог. В компьютере вся информация хранится в двоичном формате (двоичной системе) примерно в таком виде: `10101110 10010010 01111010 11100101` (естественно, без пробелов; для наглядности мы разделили байты). *Восемь бит — это один байт*. Один символ занимает один байт, т. е. восемь бит. По идеи, ничего сложного нет. Очень важно уяснить данную тему, т. к. мы будем постоянно пользоваться двоичной системой, и вам необходимо знать ее на "отлично". В принципе, даже если что-то не совсем понятно, то — не отчаивайтесь! Со временем все станет на свои места.

3.1.1. Как перевести двоичное число в десятичное

Чтобы перевести двоичное число в десятичное, надо сложить двойки в степенях, показатели которых соответствуют позициям единиц в двоичном числе.

Например, возьмем число 20. В двоичной системе оно имеет следующий вид: `10100b`.

Итак, начнем слева направо, считая от 4 до 0. Число в нулевой степени всегда равно единице:

$$10100b = 2^4 + 0 + 2^2 + 0 + 0 = 16 + 8 = 20$$

либо

$$10100b = 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1 = 16 + 0 + 4 + 0 + 0 = 20.$$

3.1.2. Как перевести десятичное число в двоичное

Можно делить его на два, записывая остаток справа налево:

$$20/2 = 10, \text{ остаток } 0;$$

$$10/2 = 5, \text{ остаток } 0;$$

$5/2 = 2$, остаток 1;
 $2/2 = 1$, остаток 0;
 $1/2 = 0$, остаток 1.

В результате получаем: $10100b = 20$.

3.1.3. Как перевести шестнадцатеричное число в десятичное

В шестнадцатеричной системе номер позиции цифры в числе соответствует степени, в которую надо возвести число 16:

$$8Ah = 8 \times 16 + 10 (0Ah) = 138.$$

В настоящий момент существует множество калькуляторов, которые могут считать и переводить числа в разных системах счисления. Например, программа Калькулятор в инженерном виде в Windows. Очень удобен калькулятор и в DOS Navigator. Если он у вас есть, то отпадает необходимость в ручном переводе одной системы в другую, что, естественно, упростит работу. Однако знать этот принцип крайне важно!

3.2. Сегментация памяти в реальном режиме

Возьмем следующее предложение: "Изучаем сегменты памяти". Теперь давайте посчитаем, на каком месте стоит буква "ы" в слове "сегменты" от начала предложения, включая пробелы... На шестнадцатом. Подчеркнем, что мы считали от начала предложения.

Теперь немного усложним задачу и разобьем предложение, как показано в примере 3.1 (символом "_" обозначен пробел).

Пример 3.1

0000:	Изучаем_
0010:	сегменты_
0020:	памяти
0030:	

В слове "Изучаем" символ "И" стоит на нулевом месте; символ "з" на первом, "у" на втором и т. д. В данном случае мы считаем буквы, начиная с нулевой позиции и используя два числа. Назовем их *сегментом и смещением*. Тогда символ "ч" будет иметь следующий адрес: 0000:0003, т. е. сегмент 0000, смещение 0003.

В слове "сегменты" будем считать буквы, начиная с десятого сегмента, но с нулевого смещения. Тогда символ "н" будет иметь следующий адрес: 0010:0005, т. е. пятый символ, начиная с десятой позиции: 0010 — сегмент, 0005 — смещение.

В слове "память" считаем буквы, начиная с сегмента 0020 и также с нулевой позиции. Таким образом, символ "а" будет иметь адрес 0020:0001, т. е. сегмент 0020, смещение 0001.

Итак, мы выяснили, что для того, чтобы найти адрес нужного символа, необходимы два числа: сам сегмент и смещение внутри этого сегмента. В ассемблере сегменты хранятся в сегментных регистрах: cs, ds, ss, es (см. разд. 2.1.3), а смещения могут храниться в других (но не во всех):

- регистр cs служит для хранения сегмента кода программы (code segment — сегмент кода);
- регистр ds — для хранения сегмента данных (data segment — сегмент данных);
- регистр ss — для хранения сегмента стека (stack segment — сегмент стека);
- регистр es — дополнительный сегментный регистр, который может хранить адрес любого другого сегмента (например, видеобуфера).

Давайте попробуем загрузить в пару регистров es:di сегмент и смещение буквы "м" в слове "памяти" из примера 3.1. Вот как это будет выглядеть на ассемблере (пример 3.2).

Пример 3.2

```
...
(1)    mov ax,0020
(2)    mov es,ax
(3)    mov di,2
...

```

Теперь в регистре es находится сегмент с номером 20, а в регистре di — смещение к букве (символу) "м" в слове "памяти". Проверьте, пожалуйста...

В этом месте стоит отметить, что загрузка числа (номера любого сегмента) напрямую в сегментный регистр запрещена. Поэтому мы в строке (1) сперва загрузили номер сегмента в ax, а в строке (2) поместили число 20 из регистра ax в es.

```
mov ds,15      ;ошибка!
mov ss,34h     ;ошибка!
```

Когда мы загружаем программу в память, она автоматически располагается в первом свободном сегменте. В файлах типа COM все сегментные регистры по умолчанию инициализируются для этого сегмента (устанавливаются значения, равные тому сегменту, в который загружена программа). Это можно проверить при помощи отладчика. Если, например, мы загружаем программу типа COM в память, и компьютер находит первый свободный сегмент с номером 5674h, то сегментные регистры будут иметь следующие значения:

```
cs = 5674h
ds = 5674h
ss = 5674h
es = 5674h
```

Иными словами: cs = ds = ss = es = 5674h.

Код программы типа COM должен начинаться со смещения 100h. Для этого мы, собственно, и ставили в наших прошлых примерах программ оператор org 100h,

указывая ассемблеру при асSEMBЛИровании использовать смещение 100h от начала сегмента, в который загружена наша программа (позже мы рассмотрим, почему так). Сегментные регистры, как уже упоминалось, автоматически принимают значение того сегмента, в который загрузилась наша программа. Пара регистров cs:ip задает текущий адрес кода.

3.2.1. Исследование программы в отладчике

Теперь рассмотрим, как все это происходит, на конкретном примере (листинг 3.1).

Листинг 3.1. Исследование программы в отладчике

```
(01) CSEG segment
(02) org 100h
(03) _start:
(04)     mov ah,9
(05)     mov dx,offset String
(06)     int 21h
(07)     int 20h
(08) String db 'Test message$'
(09) CSEG ends
(10) end _start
```

Итак, строки (01) и (09) описывают сегмент:

- CSEG (даем имя сегменту) segment (оператор ассемблера, указывающий, что имя CSEG — это название сегмента);
- CSEG ends (END Segment — конец сегмента) указывает ассемблеру на конец сегмента.

Строка (02) сообщает о том, что код программы будет располагаться, начиная со смещения 100h. По этому адресу в память всегда загружаются программы типа СОМ.

Запускаем программу из листинга 3.1 под отладчиком AFD Pro. Допустим, она загрузилась в свободный сегмент 1DF1h (рис. 3.1). Первая команда в строке (04) будет располагаться по такому адресу:

1DF1h:0100h (т. е. cs = 1DF1h, a ip = 0100h)

Обратите внимание на регистры cs и ip!

Перейдем к следующей команде. Для этого в отладчике AFD нажмите клавишу <F1>, в CodeView — <F8>, в другом — посмотрите, какая клавиша нужна; будет написано что-то вроде <F8>+<Step> или <F7>+<Trace>. Теперь вы видите, что изменились следующие регистры (рис. 3.2):

- ax = 0900h (точнее, ah = 09h, a al = 0, т. к. мы загрузили командой mov ah, 9 число 9 в регистр ah, при этом не трогая al. Если бы al был равен, скажем, 15h, то после выполнения данной команды в ax находилось бы число 0915h);
- ip = 102h (т. е. указывает на адрес следующей команды). Из этого можно сделать вывод, что команда mov ah, 9 занимает 2 байта: 102h – 100h = 2.

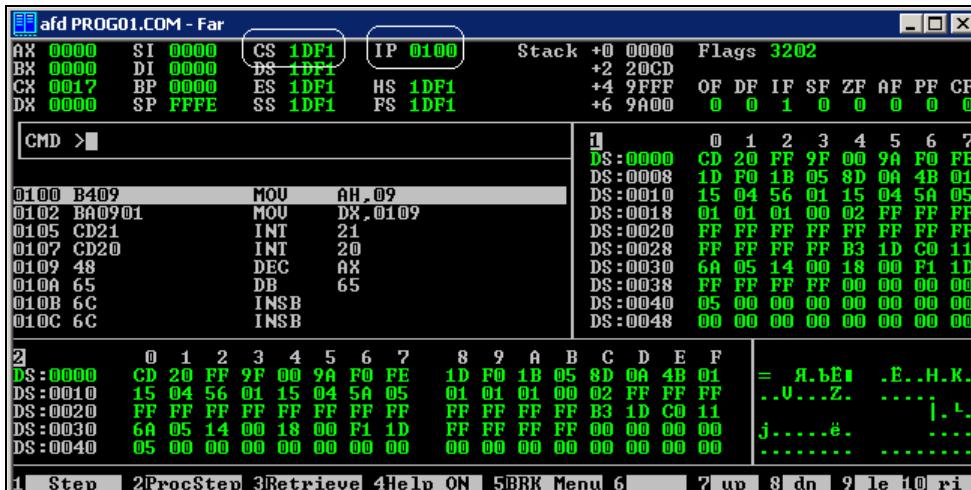


Рис. 3.1. Вид программы из листинга 3.1 в отладчике AFD Pro

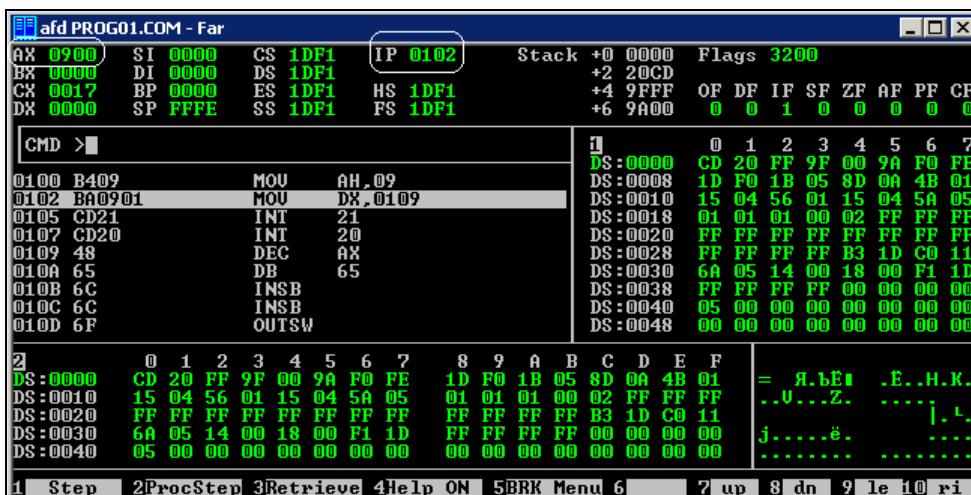


Рис. 3.2. В отладчике выполнена команда mov ah, 9

Следующая команда (нажимаем клавишу <F8>/<F1>) изменяет регистры dx и ip. Теперь dx указывает на смещение строки "Test message\$" относительно начала сегмента, т. е. 109h, а ip равняется 105h (адрес следующей команды). Нетрудно посчитать, что команда mov dx,offset String занимает 3 байта (105h – 102h = 3) (рис. 3.3).

Обратите внимание, что в ассемблере мы пишем:

```
mov dx,offset String
```

а в отладчике видим следующее:

```
mov dx,109 ;109 – шестнадцатеричное число, но CodeView и многие другие  
;отладчики символ 'h' не ставят. Это надо иметь в виду.
```

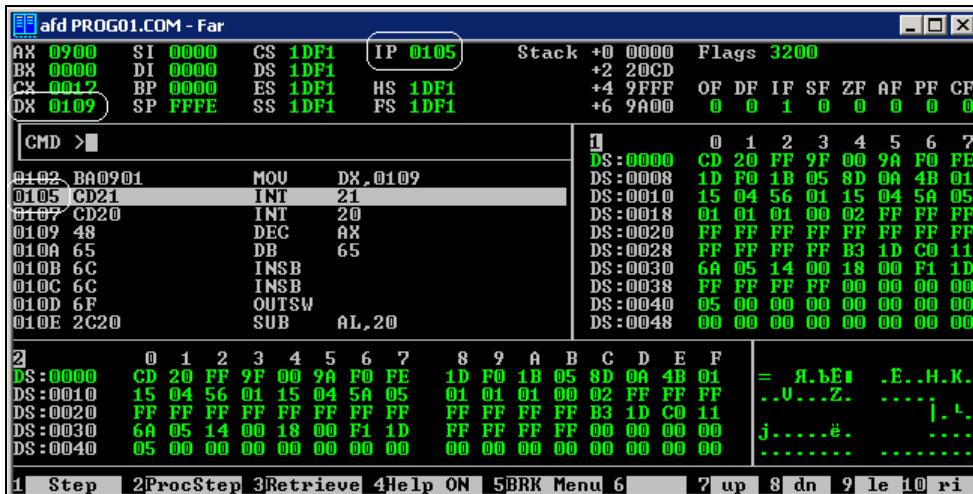


Рис. 3.3. В отладчике выполнена команда `mov dx, 0109`

Почему так происходит? Дело в том, что в процессе обработки нашего файла, программа-ассемблер (MASM/TASM) подставляет вместо `offset String` реальный адрес строки с именем `String` в памяти (ее смещение). Можно, конечно, записать сразу:

`mov dx,109h`

Данная программа будет работать корректно. Но для этого нам нужно высчитать самим этот адрес. Попробуйте вставить следующие команды, начиная со строки (07), в листинг 3.1:

```

...
(07)    int 20h
(08)    int 20h
(09) String db 'Test message$'
(10) CSEG ends
(11) end _start

```

Просто продублируем команду `int 20h` (хотя, как вы уже знаете, выполнение программы прекратится на строке (07)). Теперь асSEMBЛИРУЙТЕ программу заново. Запускайте ее под отладчиком. В качестве демонстрации воспользуемся отладчиком CodeView, при этом программа может загрузиться в другой сегмент, т. к. отладчики имеют разные размеры и, соответственно, занимают разное количество оперативной памяти. Пусть теперь сегмент, в который загрузилась наша программа, будет равен `0A09h`. Выполняйте шаг за шагом программу до тех пор, пока не дойдете до загрузки смещения строки в регистр `dx`. Вы увидите, что в `dx` загружается не `109h`, а другое число. Подумайте, почему так происходит.

А мы пока подробней рассмотрим, что находится в пределах окна отладчика CodeView.

В окне **Memory** (Память) отладчика CodeView (у AFD нечто подобное) вы увидите примерно следующее (табл. 3.2 и рис. 3.4).

Таблица 3.2. Стока отладчика CodeView

1		2	3	4
0A09	:	0000	CD 20 FF 9F 00 9A F0 FE	= я.

Рассмотрим строку отладчика:

- позиция 1 (0A09) — сегмент, в который загрузилась наша программа (может быть любым);
- позиция 2 (0000) — смещение в данном сегменте (сегмент и смещение отделяются двоеточием (:));
- позиция 3 (CD 20 FF ... F0 FE) — код в шестнадцатеричной системе, который располагается, начиная с адреса 0A09:0000;
- позиция 4 (= я.) — код в ASCII (ниже рассмотрим), соответствующий шестнадцатеричным числам с правой стороны.

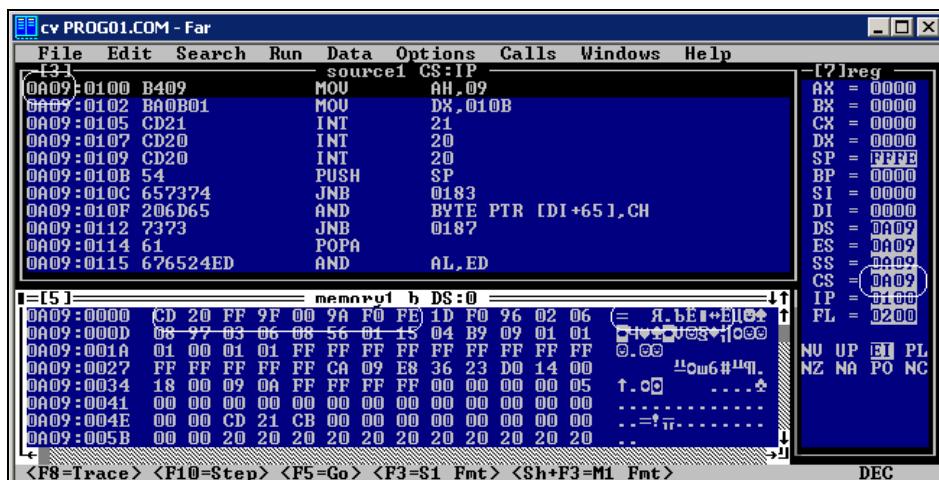


Рис. 3.4. Сегмент, смещение, шестнадцатеричный и двоичный коды программы из листинга 3.1 в отладчике CodeView

В позиции 2 (смещение) введите значение, которое будет находиться в регистре dx после выполнения строки (5). После этого в позиции 4 вы увидите строку Test message\$, а в позиции 3 — коды символов Test message\$ в шестнадцатеричной системе... Так вот что загружается в dx (рис. 3.5)! Это не что иное, как АДРЕС (смещение) нашей строки в сегменте!

Итак, мы загрузили в dx адрес строки в сегменте CSEG (строки (01) и (09) из листинга 3.1). Теперь переходим к следующей команде: int 21h. Вызываем прерывание DOS с функцией 9 (mov ah, 9) и адресом строки в dx (mov dx, offset String).

Как уже упоминалось раньше, при вызове прерываний в программах в ah заносится номер функции. Эти номера желательно запоминать (хотя бы часто используя

зумемые), чтобы постоянно не искать в справочниках, какие действия выполняет та или иная функция.

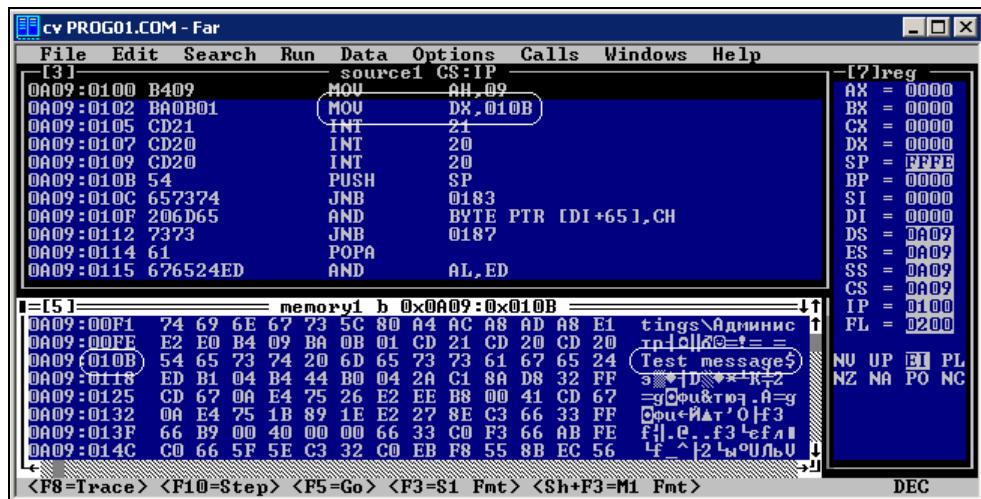


Рис. 3.5. Стока Test message\$ в памяти

3.3. Наше первое прерывание

Функция 09h прерывания 21h выводит строку на экран, адрес которой указан в регистре dx. Изобразим это в табл. 3.3. Далее всегда будем описывать функции с помощью таблиц.

В столбце **Вход** мы указываем, в какие регистры что загружать перед вызовом прерывания, а в столбце **Выход** — что возвращает функция. Сравните эту таблицу с листингом 3.1.

Таблица 3.3. Функция 09h прерывания 21h — вывод строки символов на экран в текущую позицию курсора

Вход	Выход
<code>ah = 09h</code> <code>dx = адрес ASCII-строки символов, заканчивающийся символом \$</code>	Ничего

3.3.1. Что такое ASCII?

Вообще, любая строка, состоящая из ASCII-символов, называется ASCII-строкой. ASCII-символы — это символы от 0 до 255 в DOS, куда входят буквы русского и латинского алфавитов, цифры, знаки препинания и пр. (полный список ASCII-символов различных кодировок см. в *приложении 3*).

На этом мы заканчиваем рассматривать сегментацию памяти. Если и остались кое-какие пробелы, то в последующих главах мы их обязательно заполним. Надеемся, что вы без особого труда разобрались в данной теме. По крайней мере, уловили принцип сегментации памяти.

3.4. Программа для практики

Теперь интересная программка (\003\prog03.asm) для практики, которая выводит в левый верхний угол экрана веселую рожицу на синем фоне (листинг 3.2).

Листинг 3.2. Программа для практики

```
(01) CSEG segment
(02) org 100h
(03) _beg:
(04)     mov ax, 0B800h
(05)     mov es, ax
(06)     mov di, 0
(07)
(08)     mov ah, 31
(09)     mov al, 1
(10)     mov es:[di], ax
(11)
(12)     mov ah, 10h
(13)     int 16h
(14)
(15)     int 20h
(16)
(17) CSEG ends
(18) end _beg
```

Многие операторы вы уже знаете. Поэтому рассмотрим только новые.

В данном примере для вывода символа на экран (рис. 3.6), мы используем метод прямого отображения в видеобуфер. Что это такое — будет подробно рассмотрено в следующих главах, т. к. эта тема заслуживает отдельного обсуждения. В строках (04) и (05) загружаем в сегментный регистр `es` число `0B800h`, которое соответствует сегменту дисплея в текстовом режиме (запомните его!). В строке (06) загружаем в регистр `di` ноль. Это будет смещение относительно сегмента `0B800h`. В строках (08) и (09) в регистр `ah` заносится атрибут символа (31 — ярко-белый символ на синем фоне) и в `al` — ASCII-код символа (01 — это "рожица").

В строке (10) заносим по адресу `0B800:0000h` (это первый символ в первой строке дисплея — в левом верхнем углу) атрибут и ASCII-код символа (31 и 01 соответ-

ственno). Обратите внимание на форму записи оператора `mov` в строке (10). Квадратные скобки [] указывают на то, что надо загрузить число не в регистр, а по адресу, который содержится в этом регистре (в данном случае, как уже отмечалось, — это `0B800:0000h`).

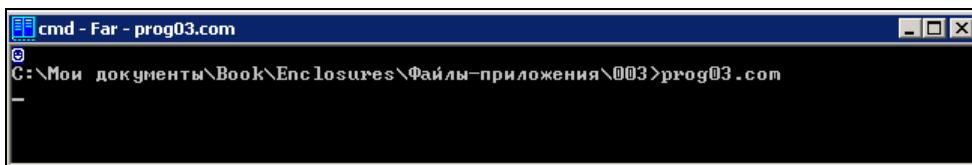


Рис. 3.6. Результат выполнения программы Prog03.com

Можете поэкспериментировать с данным примером. Только не меняйте пока строки (04) и (05). В качестве сегментного регистра оставим `es`, хотя можно, конечно, и `ds` использовать. Более подробно метод прямого отображения в видеобуфер рассмотрим позже. Сейчас нам из приведенной выше программы нужно понять только принцип сегментации на практике.

ЭТО ИНТЕРЕСНО

Ну и напоследок интересный факт. Вывод символа прямым отображением в видеобуфер является самым быстрым из всех возможных способов вывода символов или рисования точки на экране. Выполнение команды в строке (10) занимает 3—5 тактов. Таким образом, на Pentium 100 МГц можно за секунду вывести 20 миллионов (!) символов или чуть меньше точек на экран!

3.5. Подведем итоги

Уважаемые читатели! Вот вы и ознакомились с *частью I* книги.

Итак, давайте подведем итог. В данной части мы рассмотрели следующее:

- шестнадцатеричную систему счисления;
- двоичную систему счисления;
- некоторые регистры микропроцессоров Intel 8086/8088/80186;
- основы сегментации памяти в реальном режиме;
- операторы ассемблера:
 - `org` — с какого места отсчитывать смещение;
 - `mov` — загрузка данных в регистр или память (переменную);
 - `add` — сложение;
 - `sub` — вычитание;
 - `inc` — увеличение на единицу;
 - `int` — вызов прерывания;

- функцию 09h прерывания 21h (вывод строки на экран в текущую позицию курсора);
- функцию 10h прерывания 16h (ожидание нажатия клавиши).

В принципе, довольно много, чтобы начать серьезное и интенсивное обучение языку ассемблера. Если вы усвоили как минимум 75% материала из данной части, то, можно сказать, мы добились того, чего хотели. Если же вы чувствуете, что многое непонятно, то попробуйте прочитать все еще раз с самого начала, воспользуйтесь отладчиком. Мы впоследствии будем еще не раз возвращаться к материалу, изложенному в *части I*. Если вы что-то не поняли сейчас, то, надеемся, поймете со временем. Самое интересное и увлекательное — впереди!

В случае если вопросов по пройденному материалу у вас не возникло, то можете смело приступать к изучению *части II "Усложнляем задачи"*.

Увлекательного вам чтения!



ЧАСТЬ II

Усложняем задачи



Глава 4

Создание циклов

4.1. Еще немного о сегментации памяти

Рассмотрим часть примера из листинга 3.1. Кое-что в этом примере мы не разбирали:

```
(01)    ...
(02)    mov ah,9
(03)    mov dx,offset My_string
(04)    int 21h
(05)    ...
(06) My_string db 'Ура!$'
(07)    ...
```

В строке (03) загружаем в регистр `dx` *адрес* строки в памяти. Обратите внимание на запись: `mov dx,offset My_string`. Вспоминаем, что оператор `mov` загружает в регистр число. Например:

```
mov cx,125
```

В строке (03) мы видим пока еще неизвестный нам оператор `offset`. Что же он делает? И почему нельзя записать таким образом: `mov dx,My_string`?

4.1.2. Введение в адресацию

В переводе с английского "offset" — это смещение. Когда ассемблер (MASM/TASM) обрабатывает нашу программу и доходит до строки (03), он заменяет `offset My_string` на *адрес* (смещение) этой строки в памяти. Если мы запишем `mov dx,My_string` (хотя корректно будет `mov dx,word ptr My_string`, но об этом позже), то в `dx` загрузится не адрес (смещение), а *первые два символа нашей строки* (в данном случае "Ур"). Два символа, потому что `dx` — 16-разрядный регистр, в который можно загрузить два байта. А один символ, как вы уже знаете, всегда занимает один байт. Можно записать и так: `mov dl,My_string` (здесь корректно будет `mov dl,byte ptr My_string`). В этом случае, в `dl` будет находиться символ "У", т. к. `dl` — 8-разрядный регистр и может хранить только один байт.

Несколько слов про записи вида `mov dl,byte ptr My_string` и `mov dx,word ptr My_string`. `Byte` — это байт, `word` — слово (два байта). Посмотрите внимательно

на приведенные ранее строки. Вы заметите, что когда используется 8-разрядный регистр (`dl`), мы пишем `byte`. А когда 16-разрядный (`dx`) — `word`. Это указывает ассемблер-программе, что именно мы хотим загрузить — байт или слово.

Вспомним, что в реальном режиме для формирования адреса используются сегмент и смещение. Данный пример — не исключение. Для формирования адреса строки "Ура!" используется пара регистров `ds` (сегмент) и `dx` (смещение).

Почему же тогда мы ничего не загружаем в `ds`? Дело в том, что при загрузке СОМ-программы в память (а мы пока создаем только такие), все сегментные регистры принимают значение, равное значению сегмента, в который загрузилась наша программа (в том числе и `ds`). Поэтому нет необходимости загружать в `ds` сегмент строки (он уже загружен). Программа типа СОМ всегда занимает один сегмент, в котором должен уместиться код, данные и пр. А размер СОМ-файлов ограничен 64 Кбайт (65 536 байтами). Программы, написанные на "чистом" ассемблере, очень компактны, и 64 Кбайт для них — довольно большой объем.

Приведем несколько примеров. В свое время нами разрабатывалась антивирусная файловая оболочка типа Norton Commander на ассемблере (некоторое ее подобие будем изучать в *части III*). В итоге размер приложения составил всего 36 Кбайт. Несмотря на такой небольшой размер, оболочка выполняла многие функции Norton Commander (хотя некоторых функций Norton Commander в ней не было).

В качестве другого примера можно привести Volcov Commander первых версий. Практически копия Norton Commander, но занимает всего 64 000 байт (в отличие от Norton). Если Volcov Commander писали и не на "чистом" ассемблере, то хотя бы большую часть кода так точно. Да и работает Volcov гораздо быстрее, чем Norton.

Но продолжим изучение. В качестве эксперимента попробуйте перед вызовом прерывания `21h` загрузить в `ds` любое другое число. Например, так:

```
...
mov dx,offset My_string
mov ax,10h
mov ds,ax
mov ah,9
int 21h
...
My_string db 'Hello!$'
...
```

Вы увидите, что программа выведет на экран не строку "Hello!", а какой-то "мусор", хотя в `dx` мы правильно загружаем адрес (смещение) нашей строки, но сегмент-то в `ds` другой (рис. 4.1).

Давайте вспомним, что функция `09h` прерывания `21h` выводит строку, адрес которой задается в регистрах `ds:dx`. На рис. 4.1 отображено состояние программы, при котором мы умышленно загружаем в `ds` число 10. Обратите внимание, что в окне **Memory1** отображены данные, расположенные в памяти по адресу `0A09:010Ch`,

а именно строка `Hello!`, которую должна вывести программа. Это и есть полный адрес выводимой строки, включающий в себя сегмент и смещение. Однако программа перед вызовом прерывания `21h` загружает в `ds` число 10. Следовательно, операционная система начнет выводить символы, находящиеся по адресу `ds:dx = 0010:010Ch`, что является логической ошибкой в программе.

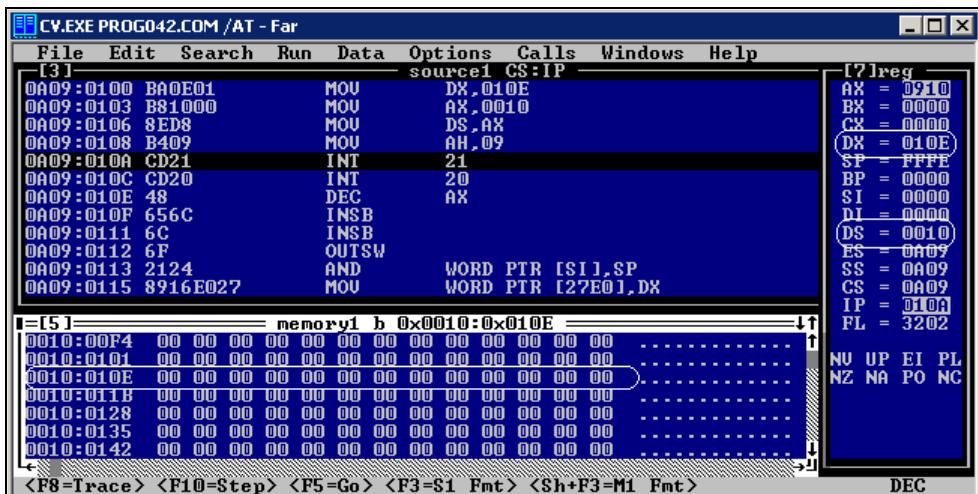


Рис. 4.1. Некорректный вывод строки

Если все же возникает необходимость в выводе строки, расположенной не в том сегменте, в который загружена сама программа, а в другом (такое требуется часто), то после выполнения всех необходимых операций следует восстановить сегментный регистр данных `ds`:

```

...
mov ax,cs
mov ds,ax
...

```

В табл. 4.1 приведено полное описание функции вывода строки на экран.

Таблица 4.1. Функция `09h` прерывания `21h` — вывод строки символов на экран в текущую позицию курсора

Вход	Выход
<code>AH = 09h</code> <code>DS:DX = адрес ASCII-строки символов, заканчивающейся \$</code>	Ничего

С этим также разобрались, хотя возвращаться к сегментным регистрам мы будем еще один раз.

4.2. Создание циклов

Что такое цикл? Допустим, нам нужно выполнить некоторый код программы несколько раз. Возьмем, к примеру, вывод строки функцией 09h прерывания 21h (листинг 4.1).

Листинг 4.1. Выполнение однотипных команд несколько раз

```
...
mov ah,9
mov dx,offset Str
int 21h

mov ah,9
mov dx,offset Str
int 21h

mov ah,9
mov dx,offset Str
int 21h
...

```

Этот участок кода выведет 3 раза на экран некую строку с названием `Str`. Код получается громоздким, читать его неудобно. Размер программы разрастается... Для выполнения подобных примеров используется оператор `loop` (табл. 4.2).

Таблица 4.2. Оператор `loop`

Команда	Перевод	Назначение	Процессор
<code>loop метка</code>	<code>loop — петля</code>	Организация циклов	8086

Количество повторов задается в регистре `CX` (счетчик). В листинге 4.2 показано, как можно использовать этот оператор на практике (изменим программу из листинга 4.1).

Листинг 4.2. Организация цикла

```
...
(01)      mov cx,3

(02) Label_1:
(03)      mov ah,9
(04)      mov dx,offset Str
(05)      int 21h
(06)      loop Label_1
...

```

В строке (01) загружаем в `cx` количество повторов, при этом отсчет будет идти в обратном порядке — от 3 до 0. В строке (02) создаем метку (от англ. *Label* — метка). Далее (строки (03)–(05)) выводим сообщение.

В строке (06) оператор `loop` уменьшает на единицу содержимое регистра `cx` и, если число в нем не стало равно нулю, переходит на метку `Label_1` (02). Итого строка будет выведена на экран три раза. Когда программа перейдет на строку (07), регистр `cx` будет равен нулю. В результате код программы уменьшается почти в три раза по сравнению с примером из листинга 4.1. Строки (02)–(06) являются телом цикла.

4.2.1. Пример высокоуровневой оптимизации

Однако, несмотря на то, что код программы уменьшается, время выполнения увеличивается, т. к. процессору приходится обрабатывать не только строки (03)–(05), но еще и оператор `loop`. В данном случае можно уменьшить тело цикла, т. к. функция `09` прерывания `21h` не изменяет регистров. В листинге 4.3 приведен оптимизированный вариант программы из листинга 4.2.

Листинг 4.3. Оптимизация кода

```
...
(01)      mov ah, 9
(02)      mov dx, offset Str
(03)      mov cx, 3
(04) Label_1:
(05)      int 21h
(06)      loop Label_1
...
...
```

Здесь видно, что загрузка номера функции и смещения строки вынесена за пределы тела цикла. Таким образом, хоть программа и не уменьшится в размере, но будет работать почти в три раза быстрее. Это называется высокоуровневой оптимизацией программ, которую будем постепенно рассматривать в последующих главах.

На рис. 4.2 показано состояние регистров после выполнения цикла, приведенного в листинге 4.3.

Забегая немного вперед, отметим, что сама метка `Label_1`, как видно на рис. 4.1, в отладчике нигде не отображается, а по адресу, на который переходит `loop`, находится инструкция `int 21h`. Следовательно, метки памяти не занимают. Они необходимы только в процессе ассемблирования программы, чтобы ассемблер-программа могла подставить реальное смещение кода.

При подобных оптимизациях следует иметь в виду, что далеко не все функции не изменяют регистры в процессе работы. Например, известная нам функция `10h` прерывания `16h`, которая возвращает в `ax` код нажатой пользователем клавиши, исключает возможность помещения ее в цикл, подобный приведенному в листинге 4.3.

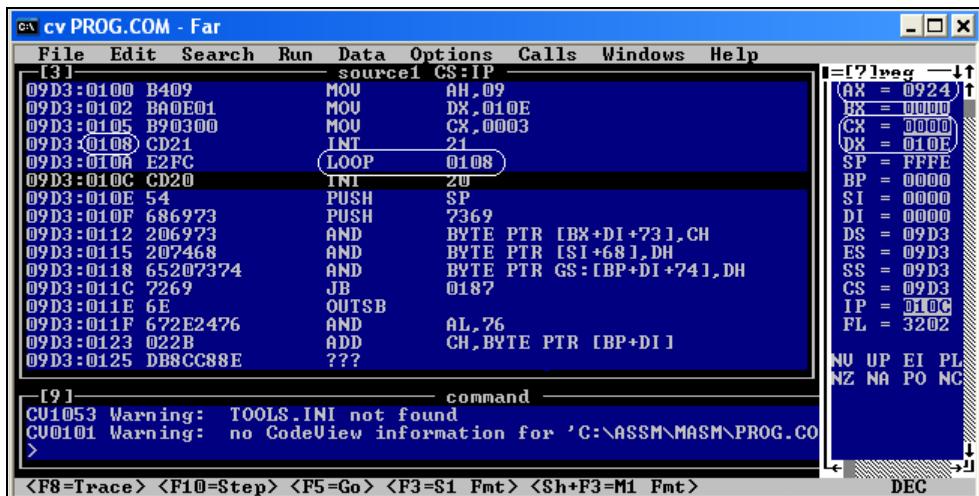


Рис. 4.2. Состояние регистров после выполнения цикла

4.3. Условный и безусловный переходы

Условный переход — это передача управления по другому адресу или на указанную метку, если выполняется определенное условие. Примером условного перехода может служить описанная ранее команда `loop`, которая передает управление на указанную метку только в случае, если регистр `CX` не равен нулю. Если же `CX` содержит ноль, то это является сигналом прекращения выполнения цикла и продолжения работы программы, т. е. в этом случае переход на метку не производится.

Безусловный переход — это передача управления по другому адресу, на указанную метку в программе не зависимо от того, выполняется ли какое-либо условие или нет. Пример безусловного перехода и реализующая его команда приведены далее.

В табл. 4.3 приведено описание команды, выполняющей безусловный переход.

Таблица 4.3. Оператор `jmp`

Команда	Перевод	Назначение	Процессор
<code>jmp метка</code>	<code>Jump</code> — прыжок (в данном случае — переход)	Безусловный переход	8086

Команда `jmp` просто переходит на указанную метку в программе без каких бы то ни было условий (листинг 4.4).

Листинг 4.4. Безусловный переход

```

...
(01)      mov ah,9
(02)      mov dx,offset Str

```

```
(03)      int 21h
(04)      jmp Label_2
(05)
(06)      add cx,12
(07)      dec cx
(08) Label_2:
(09)      int 20h
...
...
```

В результате строки (05)—(07) выполняться не будут. Программа выведет сообщение на экран, а затем инструкция `jmp` заставит программу перейти на строку (08), после чего последняя завершится.

4.3.1. Пример низкоуровневой оптимизации

С помощью оператора `dec` можно создавать циклы, которые будут работать быстрее, чем с классическим применением оператора `loop`. Пример из листинга 4.5 будет работать так же, как программа из листинга 4.2, только чуть-чуть быстрее (на некоторых более старых процессорах).

Листинг 4.5. Пример низкоуровневой оптимизации

```
...
(01)      mov cx,3
(02) Label_1:
(03)      mov ah,9
(04)      mov dx,offset Str
(05)      int 21h
(06)      dec cx
(07)      jnz Label_1
...
...
```

Не обращайте внимания на строку (07). Мы ее рассмотрим позже. Этот пример приведен для того, чтобы показать, что один и тот же прием в ассемблере можно выполнить разными операторами (равно, как и в языках высокого уровня). И чем лучше программист владеет этими операторами, тем компактнее и быстрее программа будет работать. Поэтому и получается, что разные программисты пишут на одном языке, но скорости и размеры программ различаются. В процессе обучения мы также будем учиться оптимизировать программы.

4.4. Программа для практики

Усовершенствуем программу из главы 3, которая выводила в левый верхний угол " рожицу" прямым отображением в видеобуфер (листинг 4.6, \004\prog04.asm).

Листинг 4.6. Программа для практики

```
(01) CSEG segment
(02) org 100h
(03) Begin:
(04)     mov ax,0B800h
(05)     mov es,ax
(06)     mov di,0
(07)     mov al,1
(08)     mov ah,31
(09)     mov cx,2000
(10)
(11) Next_face:
(12)     mov es:[di],ax
(13)     add di,2
(14)     loop Next_face
(15)
(16)     mov ah,10h
(17)     int 16h
(18)     int 20h
(19) CSEG ends
(20) end Begin
```

Прежде чем читать описание программы, попробуйте сами разобраться, что в итоге получится. Поверьте, это принесет вам пользу.

4.4.1. Принцип работы программы

Строки с (01) по (10) и с (15) по (20) вы уже знаете. Рассматриваем только новое. Стока (11) — это метка, "голова" нашего цикла, строка (14) — "хвост". Все, что находится в пределах строк (10)–(14), является телом цикла. Сам же цикл будет повторяться 2000 раз, для чего мы и заносим в `cx` число 2000 (строка (08)).

В строке (12) записываем в видеобуфер по адресу `0B800:DI` число, находящееся в `ax` (атрибут + символ). Итак, первый символ занесли. Далее увеличиваем регистр `di` на 2, для того чтобы перейти к адресу следующего символа.

Почему на 2? Дело в том, что *в видеобуфере один символ занимает 2 байта: сам символ и его атрибут*. Так как символ у нас находится в регистре `al`, а атрибут в `ah`, и мы загрузили уже эти два байта в строке (12), то и увеличиваем `di` (смещение) на 2. `di` теперь указывает на адрес следующего символа. Осталось уменьшить счетчик `cx` на 1 и повторить. Что мы, собственно, и делаем в строке (14).

Обратите внимание на скорость вывода символов при запуске программы.

Еще раз обращаем ваше внимание на то, что программы желательно и необходимо печатать самостоятельно, чтобы быстрее привыкнуть к операторам и освоить ассемблер. Мы уже не один раз заостряли на этом внимание, т. к. это действительно довольно важно при изучении любого языка, и особенно ассемблера.



Глава 5

Подпрограммы

5.1. Исправляем ошибку

Вероятно, те читатели, которые использовали в работе ассемблер TASM, столкнулись с проблемой при ассемблировании программ. TASM выдавал ошибку:

Near jump or call to different CS

В *приложении 2* предлагался вариант решения проблемы путем вставки в код программ строки вида `assume cs:CSEG`, после чего TASM ошибок при ассемблировании больше не выдает.

Так что же это за строка такая?

Дело в том, что оператор `assume` указывает ассемблеру, что необходимо привязать сегментный регистр `cs` к сегменту `CSEG`. Сейчас попробуем разобраться.

MASM ассемблирует прекрасно и без этой строки. Если директива `assume` отсутствует, то MASM просто подразумевает ее по умолчанию и вставляет для себя в процессе ассемблирования автоматически.

Другое дело TASM. Он, встретив в программе строки вида:

```
loop Label_1  
jmp Label_2  
call Procedure
```

не может "понять", с каким сегментным регистром следует связывать метку, и выдает сообщение об ошибке.

Как уже упоминалось, мы пишем СОМ-файлы, в которых существует всего один сегмент (мы называем его `CSEG`). Если вы создадите еще один и назовете его, например, `DSEG`, то компоновщик (`link.exe`), при попытке создать СОМ-файл (и только СОМ-файл!), выдаст ошибку. Чтобы полностью закрыть данную тему, приведем полный вид разбираемой нами строки:

```
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
```

Этим мы "сообщаем" ассемблеру, что сегментные регистры `cs`, `ds`, `es`, `ss` будут привязаны к нашему единственному сегменту `CSEG`. Возможно, это кажется сложным на первый взгляд, но, написав несколько программ, вы полностью поймете принцип. И мы будем стараться помочь вам.

5.2. Подпрограммы

В данном разделе рассмотрим возможности ассемблера при создании подпрограмм.

Мы уже вкратце рассматривали подпрограммы в *части I*. Сейчас затронем эту тему более подробно.

Допустим, нам необходимо написать программу, которая выводит на экран сообщение "Нажмите любую клавишу...", ждет нажатия клавиши, а затем выводит еще одно сообщение: "Вы успешно нажали клавишу!", ждет, пока пользователь нажмет любую клавишу, после чего завершает свою работу.

Что нужно для этого сделать? Вызвать два раза функцию 09h прерывания 21h и столько же функцию 10h прерывания 16h (листинг 5.1).

Листинг 5.1. Неоптимизированный вариант программы

```
...
(01)      mov ah,9
(02)      mov dx,offset Mess1
(03)      int 21h
(04)      mov ah,10h
(05)      int 16h

(06)      mov ah,9
(07)      mov dx,offset Mess2
(08)      int 21h
(09)      mov ah,10h
(10)      int 16h

(11)      int 20h
...
(12) Mess1 db 'Нажмите любую клавишу...$'
(13) Mess2 db 'Вы успешно нажали клавишу!$'
...
```

ПРИМЕЧАНИЕ

В тех местах, где стоит многоточие, опущен код программы. Это специально сделано с той целью, чтобы постоянно не печатать очевидные операторы, тем самым нагружая книгу бесполезным кодом. Так как мы написали и разобрали уже несколько программ, то проблем с пониманием целой структуры программы возникнуть не должно.

Строки (01)—(03) и (06)—(08) выводят сообщения. Они очень похожи, за исключением загрузки в регистр dx различных адресов строк. Вызов же функции 16h в строках (04), (05) и (09), (10) полностью идентичен. Получается, что мы пишем программу с точки зрения ассемблера неэффективно.

Чтобы упростить программу и, тем самым, уменьшить ее размер, воспользуемся оператором `call` (табл. 5.1) и создадим подпрограммы (листинг 5.2).

Таблица 5.1. Оператор `call`

Команда	Перевод	Назначение	Процессор
<code>call метка</code>	Call — вызов	Вызов подпрограммы	8086

Листинг 5.2. Использование оператора `call`

```
...
(01)      mov dx,offset Mess1
(02)      call Out_string
(03)      call Wait_key

(04)      mov dx,offset Mess2
(05)      call Out_string
(06)      call Wait_key

(07)      int 20h
...
(08) Out_string proc
(09)      mov ah,9
(10)      int 21h
(11)      ret
(12) Out_string endp

(13) Wait_key proc
(14)      mov ah,10h
(15)      int 16h
(16)      ret
(17) Wait_key endp
...
(18) Mess1 db 'Нажмите любую клавишу...$'
(19) Mess2 db 'Вы успешно нажали клавишу!$'
...
```

Не обращайте внимания на то, что второй вариант оказался длиннее. Подпрограммы (второе название — процедуры) используются в более сложных программах. Наш же вариант представлен исключительно для изучения.

Итак, что здесь происходит? Полагаем, что особых проблем в понимании кода нет. Однако хотелось бы обратить внимание на некоторые моменты и добавить пояснения.

В строке (01) загружаем в `dx` адрес строки `Mess1`. В строке (02) вызываем подпрограмму, которую мы называли `Out_string`.

Что делает процессор в момент вызова подпрограммы? Он запоминает адрес (смещение) следующей команды (строка (03)) и переходит на метку `Out_string` (строка (08)). Регистр `dx` при этом не изменяется, т. е. в нем сохраняется адрес строки `Mess1` (рис. 5.1).

В строках (09), (10) используется функция 09h прерывания 21h для вывода строки на экран. В строке (11) компьютер восстанавливает сохраненный адрес и переходит на него, в данном случае на строку (03) (`ret` от англ. *return* — возврат). Все, процедура отработала (рис. 5.2)!

The screenshot shows the CodeView debugger interface with the following details:

- Registers:** AX = 0000, BX = 0000, CX = 0000, DX = 011E, SP = FFEC, BP = 0000, SI = 0000, DI = 0000, DS = 09D3, ES = 09D3, SS = 09D3, CS = 09D3, IP = 0114, FL = 3202.
- Stack:** NU UP EI PL NZ NA PO NC.
- Call Stack:** [?]reg
- Assembly View:**

```

File Edit Search Run Data Options Calls Windows Help
source1 CS:IP
I=13] MOU DX,011E
CALL 0114
CALL 0119
MOU DX,0135
CALL 0114
CALL 0119
INT 20
MOU AH,09
INT 21
RET
MOU AH,10
INT 16
RET
LEA SP,WORD PTR [BX+SI-535A]
TEST AL,E2
MOUSW
    
```
- Messages:**
 - CU1053 Warning: TOOLS.INI not found
 - CU0101 Warning: no CodeView information for 'C:\ASSM\MASM\PROG.COM'
- Keyboard Shortcuts:** <F8=Trace> <F10=Step> <F5=Go> <F3=SI Fmt>
- Status Bar:** DEC

Рис. 5.1. Вызов подпрограммы

The screenshot shows the CodeView debugger interface with the following details:

- Registers:** AX = 0924, BX = 0000, CX = 0000, DX = 011E, SP = FFEC, BP = 0000, SI = 0000, DI = 0000, DS = 09D3, ES = 09D3, SS = 09D3, CS = 09D3, IP = 0106, FL = 3202.
- Stack:** NU UP EI PL NZ NA PO NC.
- Call Stack:** [?]reg
- Assembly View:**

```

File Edit Search Run Data Options Calls Windows Help
source1 CS:IP
I=13] MOU DX,011E
CALL 0114
CALL 0119
MOU DX,0135
CALL 0114
CALL 0119
INT 20
MOU AH,09
INT 21
RET
MOU AH,10
INT 16
RET
LEA SP,WORD PTR [BX+SI-535A]
TEST AL,E2
MOUSW
    
```
- Messages:**
 - CU1053 Warning: TOOLS.INI not found
 - CU0101 Warning: no CodeView information for 'C:\ASSM\MASM\PROG.COM'
- Keyboard Shortcuts:** <F8=Trace> <F10=Step> <F5=Go> <F3=SI Fmt>
- Status Bar:** DEC

Рис. 5.2. Процедура 0114h отработала

У вас, вероятно, возник вопрос: "А где это именно (в каком месте, области памяти) компьютер запоминает адрес, куда нужно возвращаться после того, как процедура отработала?" Ответ на этот вопрос не так прост, как кажется. Попробуем ответить на этот вопрос вкратце, т. к. эта тема достойна отдельной главы.

Каждая программа отводит себе некую область памяти именно для подобных целей, которая называется *стеком* (stack). Вот именно там и сохраняется адрес для возврата из подпрограммы. В главе 6 мы подробно рассмотрим стек, его сегмент, а также пару регистров ss:sp, которые отвечают за использование стека.

Надеемся, что разобраться в дальнейшей работе программы (начиная со строки (03)) не составит труда. Единственное, на что следует обратить внимание — оформление процедуры:

```
Out_string proc
```

```
...
```

```
Out_string endp
```

где:

- Out_string — название процедуры;
- proc (procedure) — процедура;
- endp (end procedure) — конец процедуры.

Стоит еще отметить, что из одной подпрограммы можно вызывать другие. Из других — третий. Главное правило — не запутаться в возвратах из подпрограмм, иначе произойдет нарушение работы стека, и компьютер просто "зависнет"! Внимательно следить за работой подпрограмм и использованием стека позволяет отладчик.

5.3. Программа для практики

Давайте попробуем усовершенствовать программу из главы 4, которая заполняла весь экран веселыми рожицами (005\prog05.asm). Усовершенствованный код приведен в листинге 5.3.

Листинг 5.3. Практическое использование подпрограмм

```
(01) CSEG segment
(02) assume CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
(03) org 100h

(04) Start:
(05)     mov ax,0B800h
(06)     mov es,ax
(07)     mov al,1
(08)     mov ah,31
(09)     mov cx,254

(10) Next_screen:
(11)     mov di,0
```

```

(12)      call Out_chars
(13)      inc al
(14)      loop Next_screen

(15)      mov ah,10h
(16)      int 16h

(17)      int 20h

(18) Out_chars proc
(19)      mov dx,cx
(20)      mov cx,2000
(21) Next_face:
(22)      mov es:[di],ax
(23)      add di,2
(24)      loop Next_face

(25)      mov cx,dx
(26)      ret
(27) Out_chars endp

(28) CSEG ends
(29) end Start

```

Строки (01)—(08), (15)—(17) и (28), (29) опускаем. Вопросов по ним быть не должно.

В строке (09) заносим в `cx` число 254, сообщающее, сколько раз будет выполняться основной цикл. Строки (10) и (14) — "голова" и "хвост" нашего основного цикла соответственно. Значение `di` будет меняться во вложенной процедуре, поэтому нам необходимо будет его постоянно аннулировать (строка (11)). В строке (12) вызываем процедуру, выводящую на экран символ, код которого находится в `al` (при первом проходе цикла это будет символ "рожица" с кодом 01). Все! Теперь на экран будет выведен символ с кодом 01. При этом `di` будет равно 2001, поэтому нам и нужно его постоянно обнулять.

Далее увеличим на единицу код символа, который находится в `al`. Во второй раз `al` будет содержать 02 — тоже "рожица", но немного другого вида (строка (13)). Затем уменьшим счетчик на 1 и перейдем к заполнению экрана кодом 02 (строка (14)). И так далее. Всего 254 раза.

Теперь рассмотрим работу самой процедуры. В строке (19) сохраняем содержимое регистра `cx` (просто перенесем его в `dx`), т. к. он будет изменен во вложенном цикле ниже. Строки (21) и (24) — "голова" и "хвост" вложенного цикла, который будет выполняться 2000 раз (именно столько символов вмещается на экране в тек-

стовом режиме 80 на 25). Именно это число заносим в строке (20) в регистр `CX`. Теперь осталось только восстановить `CX` (строка (25)) и выйти из подпрограммы (26).

Итак, у нас здесь два цикла: основной и вложенный. Основной выполняется 254 раза, а вложенный — 2000 раз. Итого: $2000 \times 254 = 508\,000$. Теперь представьте, сколько работы выполняет процессор, и обратите внимание, как быстро работает программа. На современных компьютерах вы не успеете заметить молниеносный вывод всех символов. Очень быстро работает, хотя это далеко и не оптимальный алгоритм.

Для исследования данного примера настоятельно рекомендуем вам воспользоваться помощью отладчика. Лучший вариант в данном случае — AFD.

5.4. Несколько слов об отладчике AFD

AFD, безусловно, не совсем актуален для современных программ по следующим причинам:

- не поддерживает 32-разрядные регистры;
- распознает только инструкции 8086—80186 процессоров, а также сопроцессора 8087;
- не поддерживает форматы PE и NE (Windows).

Зато у него есть преимущества:

- удобен и прост в использовании;
- показывает в одном окне регистры, код, память и др.;
- идеально подходит начинающим программистам для исследования простых программ.

Для изучения работы программ в DOS его больше, чем достаточно. Скачать отладчик AFD можно на сайте <http://www.Kalashnikoff.ru>, приблизительный размер — 64 Кбайта.

Скачивайте, пробуйте, изучайте, экспериментируйте!



Глава 6

Работа со стеком

6.1. Стек

Стек (стэк, stack) — это специально отведенная область памяти для хранения промежуточных (временных) данных.

Теперь попробуем разобраться, что именно это такое. Будем считать, что сегмент "растет" сверху вниз:

```
0000  
0001  
0002  
...  
FFFE  
FFFF
```

То есть мы как бы погружаемся под землю. Таким образом — сверху вниз — выполняется программа, если в ней, безусловно, не встречаются инструкции типа `jmp`, `call` и т. п. Стек же наоборот пополняется снизу вверх. Вершина стека — `0FFFFh`, а низ (дно) — `0000h`. Когда мы вызываем какую-нибудь подпрограмму командой `call`, процессор заносит в стек адрес следующей за командой `call` инструкции. Как только подпрограмма отработала, из стека извлекается сохраненный адрес возврата, после чего происходит переход по этому адресу.

Следить за стеком позволяет пара регистров `ss:sp`. Многие уже, наверное, заметили, что при запуске какой-нибудь СОМ-программы регистр `sp` равен `0FFEh`, а сегментный регистр `ss`, как уже неоднократно упоминалось, равен нашему единственному сегменту `CSEG` (как, собственно, и `cs, ds, es`).

Теперь вам необходимо вооружиться отладчиком. Давайте рассмотрим выше-сказанное на примере. Напечатайте программу из листинга 6.1 в редакторе.

Листинг 6.1. Исследование работы стека

```
CSEG segment  
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG  
org 100h  
  
begin:  
call Our_proc
```

```

int 20h

Our_proc proc

ret
Our_proc endp

CSEG ends
end Begin

```

Запускаем отладчик и сразу смотрим на пару регистров `ss:sp`. `sp=0FFFFEh`, т. е. указываем на вершину стека (рис. 6.1).

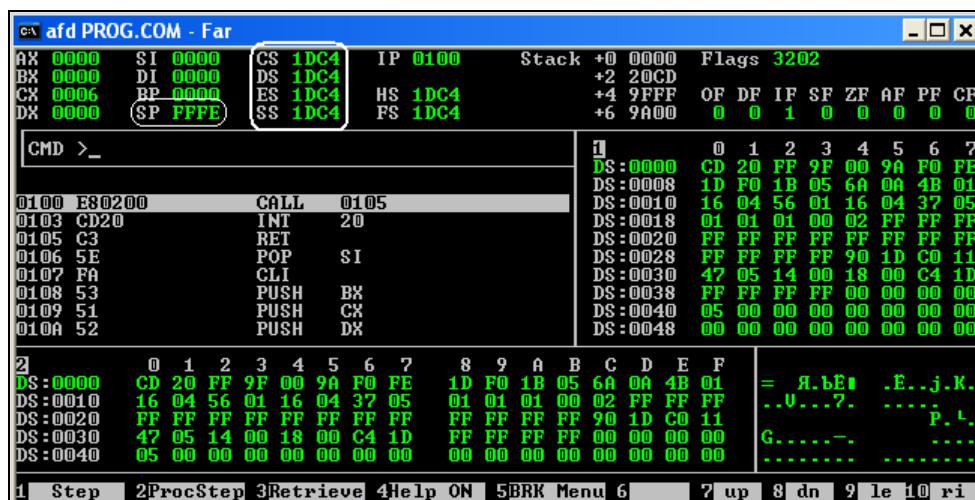


Рис. 6.1. Инициализация стека при старте программы

Теперь заходим в процедуру. Для CodeView следует нажать клавишу <F8>, для AFD — <F1>. Регистр `sp` изменился! Он уменьшился на 2. Компьютер поместил в стек адрес возврата из процедуры на инструкцию `int 20h`. Проще говоря, команда `call` передала управление на метку `Our_proc`, поместив при этом в стек адрес возврата из этой подпрограммы. В нашем случае это число 103 (рис. 6.2).

Нажимаем еще раз <F8>/<F1>. `sp` опять изменился. Но теперь он увеличился на 2, стал равным 0FFFFEh. То есть команда `ret` "вытащила" из стека предварительно сохраненный адрес возврата 0103h и продолжила работу. А по этому адресу как раз и находится следующая за `call` инструкция `int 20h`. В данном случае можно говорить, что стек выровнен. При старте программы он был равен 0FFFFEh и остался равен этому числу перед выходом — 0FFFFEh (рис. 6.3).

Мы рассмотрели один из вариантов использования стека непосредственно процессором. Но на этом мы не остановимся и рассмотрим, какие возможности дает стек программисту и как его можно использовать в своей программе.

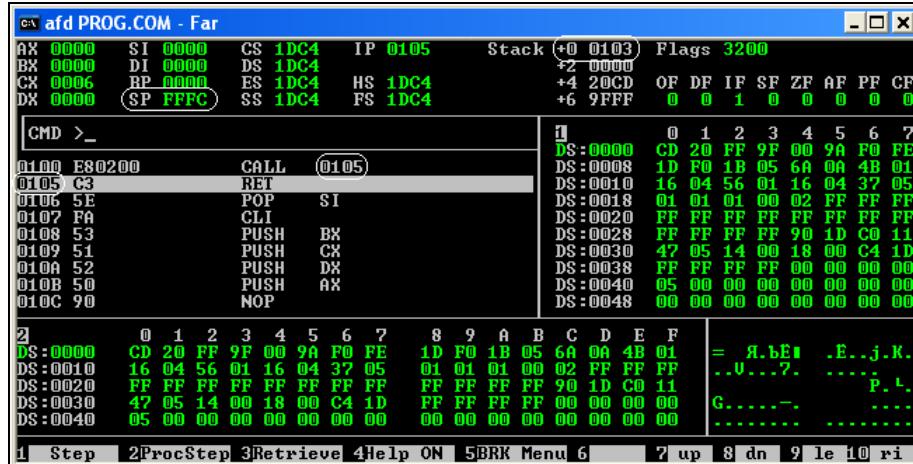


Рис. 6.2. Вызов процедуры с занесением в стек адреса возврата из нее

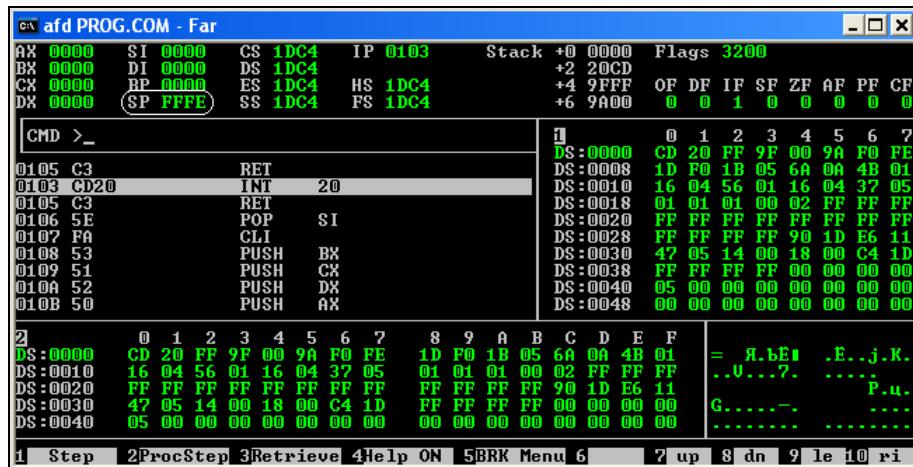


Рис. 6.3. Процедура отработала. Стек выровнен

Прежде всего, разберем два оператора, которые позволяют работать со стеком (табл. 6.1 и 6.2).

Таблица 6.1. Оператор *push*

Команда	Перевод	Назначение	Процессор
push приемник	Push — втолкнуть	Поместить в стек число	8086

Таблица 6.2. Оператор *pop*

Команда	Перевод	Назначение	Процессор
pop приемник	Pop — вытолкнуть	Достать из стека число	8086

Допустим, нам нужно временно сохранить некое число. Например, перед вызовом процедуры, прерывания или циклом. Вот как это будет выглядеть (листинг 6.2).

Листинг 6.2. Временное сохранение числа в стеке

```
...
(01)    mov ax,345h
(02)    push ax
(03)    mov ah,10h
(04)    int 16h
(05)    pop ax
...
...
```

Здесь мы загружаем в `ax` число `345h` (01), сохраняем его (02), ждем нажатия клавиши (03), (04). Функция `10h` прерывания `16h` возвращает в `ax` код нажатой клавиши, т. е. затирает в `ax` число, которое находилось до вызова прерывания. Как только пользователь нажал любую клавишу, выполняется строка (05), которая достанет из стека число `345h` и запишет его в регистр `ax`. В итоге `ax` будет содержать число `345h`, что, как говорится, и требовалось доказать. Однако стоит заметить такой момент. Допустим, мы помещаем в стек содержимое следующих регистров: `ax`, `bx`, `cx`:

```
...
push ax
push bx
push cx
...
...
```

Обратите внимание, что восстанавливать их из стека нужно в обратном порядке:

```
...
pop cx
pop bx
pop ax
...
...
```

Если вы поменяете местами регистры при восстановлении, то ничего страшного не произойдет, только содержать они будут другие числа. Например:

```
...
mov ax,1234h
mov bx,5678h
...
...
```

```
push ax
push bx
pop ax
pop bx
...
...
```

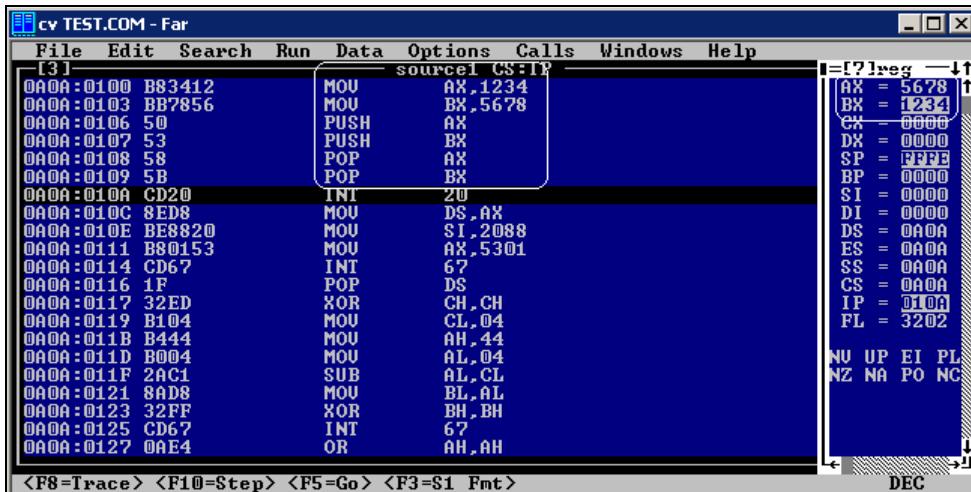


Рис. 6.4. Результат неправильного использования стека

В итоге `ax` будет равен `5678h`, а `bx` — `1234h` (рис. 6.4).

Особую осторожность при работе со стеком следует соблюдать в подпрограммах (листинг 6.3).

Листинг 6.3. Использование стека в подпрограммах

```

...
call Our_proc
int 20h
...

Our_proc proc
    mov ax,15
    push ax
    mov ah,9
    mov dx,offset Str
    int 21h
    ret
Our_proc endp
...

```

Обратите внимание, что мы "забыли" восстановить из стека `ax` в нашей процедуре `Our_proc`. В таком случае компьютер, дойдя до оператора `ret`, вытащит из стека не адрес возврата, а число 15 и передаст управление на инструкцию, которая находится по смещению 15. Что находится по адресу 15 и дальше — не известно. Машина, скорее всего, "зависнет" (см. рис. 6.5 и 6.6).

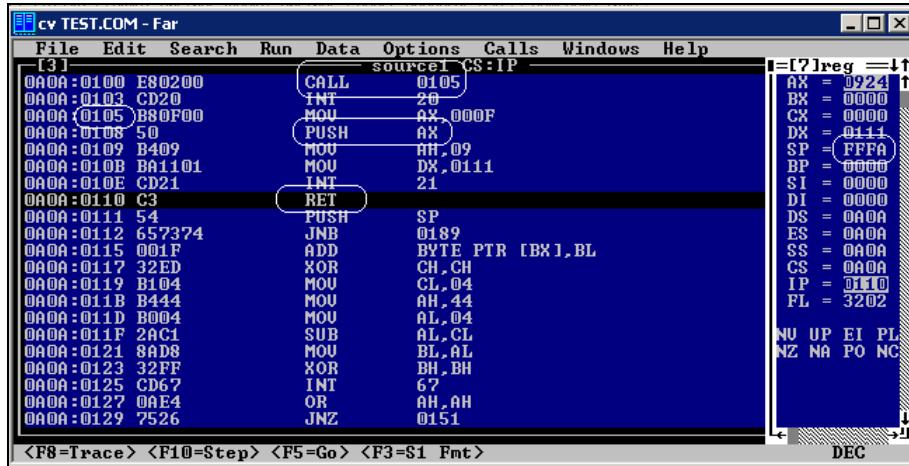


Рис. 6.5. Нарушенем работы стека в подпрограмме

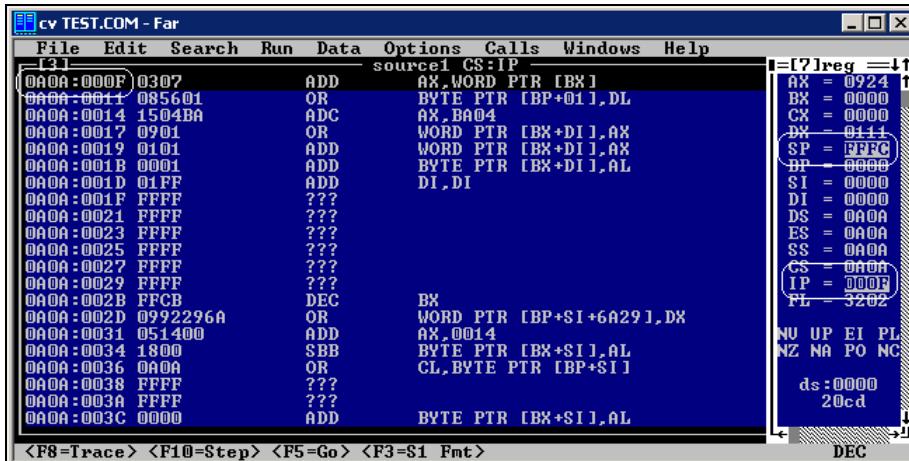


Рис. 6.6. Результат нарушения работы стека в подпрограмме

Следить за стеком (как уже упоминалось) позволяет пара регистров `ss:sp` (сегмент стека : смещение). Программист может менять как сегмент, так и смещение, но при этом следует иметь в виду, что перед сменой регистров `ss` и `sp` необходимо запретить вызов всех аппаратных прерываний, а после изменения — разрешить. Это позволяют сделать операторы `cli` и `sti` (табл. 6.3 и 6.4).

Таблица 6.3. Оператор `cli`

Команда	Перевод	Назначение	Процессор
<code>cli</code>	<code>Clear interrupt</code> — запретить прерывания	Запретить прерывания	8086

Таблица 6.4. Оператор *sti*

Команда	Перевод	Назначение	Процессор
<i>sti</i>	<i>Restore interrupt</i> — восстановить	Разрешить прерывания	8086

Сразу отметим, что запрет вызова аппаратных прерываний замертье "вешает" компьютер в глазах пользователя! Восстановление же — приводит его к нормальной работе. После команды *cli* всегда должна идти *sti*. Следует избегать запрета вызова аппаратных прерываний на длительное время.

Для чего нужно запрещать прерывания? Скажем об этом в двух словах, т. к. механизму работы прерываний необходимо посвятить целый раздел. Если что-то не понятно — просто опустите, особо не заостряйте на этом внимания.

После создания программы и ее запуска на выполнение работает не только она одна. Простейший пример — таймер, который автоматически вызывается примерно 18,2 раз в секунду для обновления часов таймера. Компьютер всегда что-то делает! Даже тогда, когда, например, ждет от пользователя нажатия клавиши.

Что происходит, когда вызывается прерывание, например, от таймера? Примерно то же, что и при вызове процедуры. Компьютер запоминает в стеке адрес текущей команды, а также все регистры, которые изменят прерывание, и переходит на адрес прерывания, по которому находится процедура обработки этого прерывания (например, процедура обработки таймера, которая обновит показания часов/минут/секунд). Затем, когда процедура отработает, компьютер восстановит из стека адрес возврата и все сохраненные регистры, и наша программа продолжит выполнение, даже "не подозревая" о том, что кто-то ее прерывал в процессе работы. Все это происходит очень быстро и для пользователя совсем незаметно. Рассмотрим случай, когда мы меняем стековые регистры, при этом предварительно не запрещаем вызов аппаратных прерываний:

```
...
(01)    mov ax,100h
(02)    mov ss,ax
(03)    mov sp,200h
...

```

Допустим, вызов прерывания таймера пришелся на тот момент, когда наша программа выполнила команду в строке (02). В этом случае *ss* равен 100h, а в *sp* еще не занесено число 200h. Получается так, что сегмент стека именно тот, который нам нужен, а смещение остается прежним, пусть это будет, допустим, *sp*=0FFEh. В итоге, *ss*=100h, а *sp*=0FFEh. Процессор и сохранит по этому адресу данные. Какой при этом код программы или какие данные будут затерты — не известно. Мы ведь хотели сделать *ss*=100h, а *sp*=200h! Хорошо, если две команды успели выполниться перед вызовом прерывания. Хорошо также, если по адресу 0100h:0FFEh ничего нет (память свободна). А если есть? Тогда компьютер, скорее всего, опять же "зависнет".

Отсюда вытекают три важных правила при работе со стековыми регистрами:

- прежде чем менять регистры *ss:sp*, необходимо запретить вызов всех аппаратных прерываний командой *cli*, а затем, после смены, разрешить их командой

sti. Однако в современных компьютерах (Pentium) запрещать прерывания необязательно, т. к. это автоматически делает процессор. Но если вы запустите эту программу на более старой модели, то очень высока вероятность того, что она просто "зависнет";

- ss:sp нужно устанавливать на свободную область памяти. При этом следует убедиться, что код не утратил работоспособности;
- не следует забывать о таком понятии, как переполнение стека. Мы знаем, что после загрузки СОМ-программы в память, ss равен сегменту, куда загрузилась программа, а sp = 0FFEh. Код программы начинается с адреса 100h (org 100h). Вершина стека — конец сегмента. Если наша программа занимает, скажем, 2000h байт, то можем установить sp в 2200h. В этом случае мы отводим 100h (именно сто) байт для стека (т. к. программа загружается в память по адресу 100h (org 100h), то к 2000h нужно прибавить 100h). Стек, как вы помните, расстет снизу вверх. Если мы переполним стек (например, поместим более 100h данных), то затрется часть нашей программы снизу. Об этом следует помнить!

В листинге 6.4 приведен полный пример корректного изменения регистров стека.

Листинг 6.4. Корректное изменение регистров стека

```
...
cli
mov ax,0B900h
mov ss,ax
mov sp,100h
sti
...
```

Вы, вероятно, спросите: "А зачем вообще менять регистры ss:sp?"

Дело в том, что это иногда нужно делать в резидентных программах либо перед освобождением памяти, изначально отведенной под стек, для загрузки программы или оверлея (вспомогательных процедур, расположенных в отдельном файле), а также во многих других случаях, которые мы будем позже рассматривать.

6.2. Программа для практики

6.2.1. Оператор *nop*

Прежде чем перейти к программе, рассмотрим новый оператор (табл. 6.5).

Таблица 6.5. Оператор *nop*

Команда	Перевод	Назначение	Процессор
Nop	No operand — нет операнда	Ничего не делает	8086

Этот оператор абсолютно ничего не делает, не меняет содержимое регистров и флаги, но занимает 1 байт. Его обычно используют для резервирования места либо для того, чтобы "забыть" ненужный код, когда исходник на ассемблере отсутствует. Например, в случае, когда программа перед стартом проверяет версию операционной системы, а версия, которая установлена на вашем компьютере, не соответствует требуемой программой. Для этого данным оператором "забывают" участок кода, который проверяет версию ОС. Все это позволяет сделать редактор Hacker's View, который можно загрузить с сайта: <http://www.Kalashnikoff.ru>.

Запомните машинный код данной команды: 90h. Если вы в любом текстовом редакторе, который позволяет редактировать файлы в шестнадцатеричном формате (Volcov Commander, Hacker's View), вручную введете приведенные ниже строки, то получится коротенькая программа, которая абсолютно ничего не делает:

```
90h  
90h  
0CDh  
20h
```

6.2.2. Хитрая программа

Рассмотрим одну очень интересную и хитрую программу (листинг 6.5). Ни на одном другом языке программирования нельзя создать подобную (\006\prog06.asm).

Листинг 6.5. Программа для практики

```
(01) CSEG segment  
(02) assume cs:CSEG, es:CSEG, ds:CSEG, ss:CSEG  
(03) org 100h  
  
(04) Begin:  
(05)     mov sp,offset Lab_1  
(06)     mov ax,9090h  
(07)     push ax  
(08)     int 20h  
  
(09) Lab_1:  
(10)     mov ah,9  
(11)     mov dx,offset Mess  
(12)     int 21h  
  
(13)     int 20h  
  
(14) Mess db 'А все-таки она выводится! $'  
  
(15) CSEG ends  
(16) end Begin
```

То, что вы видите — "обман зрения". На первый взгляд, программа что-то делает с регистром `sp`, а затем завершает работу. Строки (09)–(12) вообще не будут работать. Но это глубокое заблуждение! Попробуйте запустить ее под отладчиком. Вы увидите, что CodeView, TurboDebugger, AFD будут выдавать какой-то "мусор": непонятные операторы, сообщения типа "программа завершилась", хотя до команды `int 20h` дело не дошло (рис. 6.7). Но если запустить ее просто из DOS, то строка появится на экране, т. е. программа будет работать корректно, при этом выводя строку на экран (рис. 6.8)! Данный пример — типичный случай "заламывания рук" многим отладчикам. И вы уже можете это делать!

Почему так происходит? Ответ один: *указанные выше отладчики используют стек пользовательской программы* (т. е. той программы, которая работает под отладчиком).

Ваша задача — разобрать "по полочкам" программу.

- Что происходит?
- Почему строка выводится?
- Почему отладчик работает неверно?

Вопросов море. И вам необходимо дать ответ на них. Разобравшись с программой *самостоятельно*, вы почувствуете силу и неограниченные возможности ассемблера. Не бойтесь экспериментировать! Компьютер будет часто зависать, но это не главное! Ваши мучения приведут вас к пониманию работы процессора!

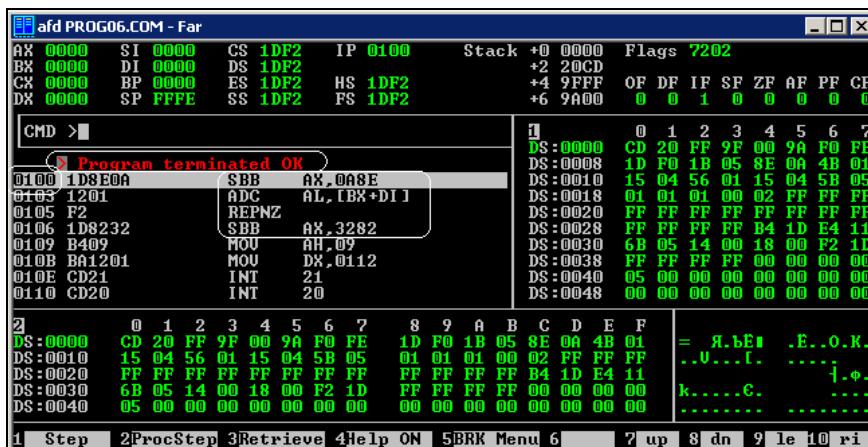


Рис. 6.7. Некорректная работа отладчика с нашей программой

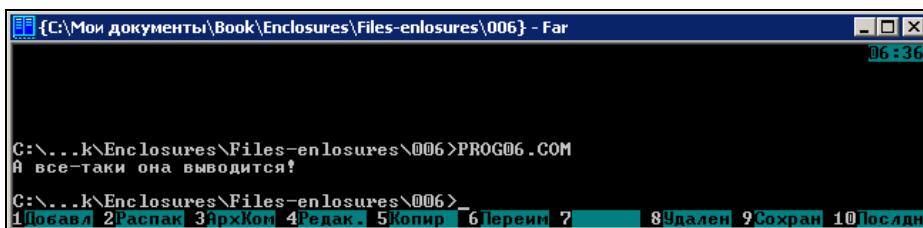


Рис. 6.8. Корректная работа нашей программы без отладчика



Глава 7

Операторы сравнения

7.1. Разбор программы из главы 6

В начале данной главы рассмотрим прошлую программу. Итак, в листинге 7.1 показано решение.

Листинг 7.1. Программа из главы 6

```
...
(01)    mov sp,offset Lab_1
(02)    mov ax,9090h
(03)    push ax
(04)    int 20h

(05) Lab_1:
(06)    mov ah,9
(07)    mov dx,offset Mess
(08)    int 21h
(09)    int 20h
...
...
```

Что же все-таки происходит?

Как мы узнали из главы 6, команда `nop` имеет машинный код `90h`. Посмотрите на строку (02). В ней мы загружаем в регистр `ax` два числа `90h` (или два оператора `nop`). В `sp` заносим адрес метки `Lab_1` (01). Теперь *стек находится внутри нашей программы*, а, точнее, вершина стека на метке `Lab_1`. Стоит отметить, что сама метка памяти не занимает! Она нужна только для MASM/TASM, которые, дойдя до строки (05), запомнят адрес (смещение) метки в программе (что равно `109h`).

После выполнения строки (01) посмотрите в отладчике, какое число находится в `sp`, а также на код, который расположен по этому адресу. Вы увидите, что по адресу `109h` находится команда `mov ah,09`, т. е. физически метки `Lab_1` в памяти не существует.

Теперь внимание! В строке (03) мы "толкаем" в стек два числа `90h`, т. е. две команды `nop`. Что происходит дальше? Стек, как вы уже знаете, "растет" снизу вверх.

После помещения в стек числа `sp` не увеличивается, а уменьшается, что отображено стрелкой на рис. 7.1. Команда `int 20h` занимает 2 байта. Это можно наблюдать в отладчике или в Hacker's View. Машинный код `int 20h` равен `0CDh`, `20h`. Теперь вы знаете, как "вручную" набрать простейшую COM-программу в редакторе. Если есть редактор, позволяющий смотреть шестнадцатеричные числа (например, Volcov Commander), то введите в создаваемом файле `CD` и `20`. Сохраните файл как COM (например, `prog.com`). Его размер должен быть 2 байта. Можете смело запускать получившийся COM-файл.

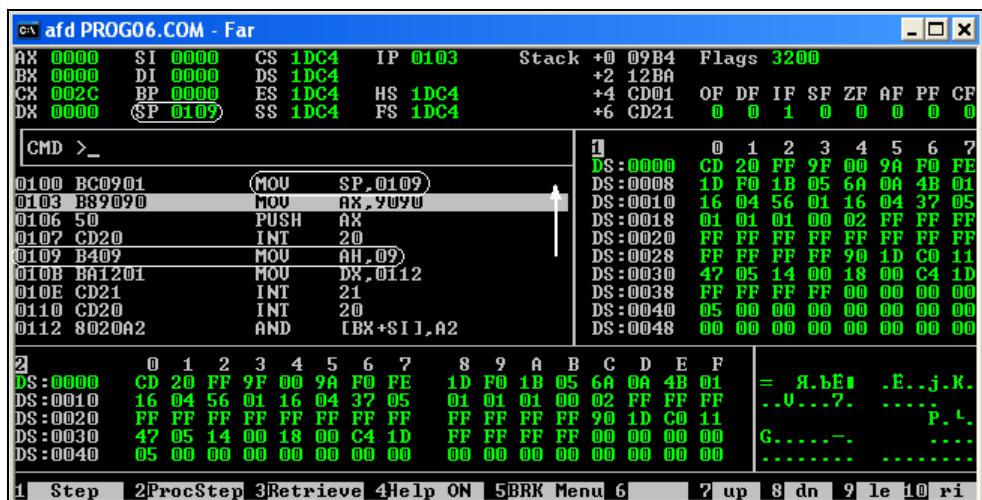


Рис. 7.1. Программа в отладчике AFD Pro

Итак, `int 20h` занимает 2 байта. Командой `push ax` (03) мы заносим в стек два числа `90h`, которые процессор воспримет как два оператора `nop`. Так как смещение в регистре `sp` указывает на `int 20h`, то эта команда будет затерта двумя операторами `nop`. Все! Вместо `int 20h` (завершение работы программы) процессор выполнит два оператора `nop`, которые ничего не делают! После этого выведет строку на экран. На рис. 7.1 показан результат работы этой программы в отладчике.

Обратите внимание, как легко можно менять код собственной программы в ассемблере буквально "на лету".

Эта простая программа наглядно демонстрирует работу стека.

Теперь рассмотрим, почему отладчики CodeView, AFD, Turbo Debugger работают некорректно. Внимательно прочтите следующую фразу: вышеупомянутые отладчики используют стек *нашей* программы. Говоря простым языком, отладчик, загрузив к себе в память нашу программу, использует тот же стек (те же регистры `ss:sp`), что и наша программа. Отладчику ведь тоже надо хранить где-то свои данные. Например, после выполнения одной команды нашей программы, запомнить адрес следующей. Одно дело на экране, но другое — в стеке отладчика, который по сути является *нашим*! Поэтому, переместив стек в область кода написанной нами

программы, мы тем самым переносим и стек отладчика, который, в свою очередь, сохраняет в нем свои данные. Но мы знаем, что заносить в стек можно не более двух байтов, иначе стек будет переполнен, т. к. мы очень уж мало места отвели под стек. Проблема отладчика в том, что он об этом "не знает", да и заносит в стек довольно много информации, которая попросту затирает нашу программу.

7.2. Оператор сравнения

В предыдущих главах мы рассматривали функцию `10h` прерывания `16h`, которая ждет от пользователя нажатия клавиши. Теперь настало время научиться проверять, какую же именно клавишу нажал пользователь.

Вышеуказанная функция возвращает код нажатой клавиши в регистр `ax`. Как же нам ее проверить? Для этого рассмотрим новый оператор сравнения `cmp` (табл. 7.1 и листинг 7.2).

Таблица 7.1. Оператор сравнения cmp

Команда	Перевод	Назначение	Процессор
<code>cmp приемник, источник</code>	Compare — сравнить	Сравнение	8086

Листинг 7.2. Пример использования оператора cmp

```
...
(01)    mov ah,10h
(02)    int 16h
(03)    cmp al,'Y'
(04)    jz Yes_key
...
(05) Yes_key:
...

```

На строке (02) компьютер остановится и будет ждать до тех пор, пока пользователь не нажмет какую-нибудь клавишу. После этого прерывание `10h` поместит в `ax` код клавиши, и выполнение программы продолжится. В строке (03) мы проверяем, нажата ли клавиша <Y>. Обратите внимание на запись:

`cmp al,'Y'`

Нам не обязательно знать ASCII-код клавиши. Вместо этого в одинарных кавычках просто указываем саму клавишу. При ассемблировании программы MASM/TASM автоматически заменит ее соответствующим кодом. Если вы запустите программу под отладчиком, то увидите, что код будет не

`cmp al,'Y'`

а

`cmp al,59h`

Теперь мы знаем код клавиши <Y>! Естественно, можно сразу записать и `cmp al, 59h`. Программа будет работать правильно.

Таким образом, мы узнали, что в ассемблере можно записывать не только двоичные, десятичные и шестнадцатеричные числа, но и просто символы (листинг 7.3).

Листинг 7.3. Пример загрузки символов в регистры

```
...
mov al,'2'
mov dh,'d'
...
```

В результате `al` будет содержать `32h`, а `dh` — `64h`. Все просто!

В строке (04) наша программа перейдет на метку `Yes_key` (05), если нажата клавиша <Y>. Оператор `jz` (от англ. *jump if zero* — переход, если флаг нуля установлен) выполняет переход на указанную метку, если флаг нуля установлен.

Мы подошли вплотную к изучению *регистра флагов*. Полностью мы их в данной главе не будем рассматривать, а коснемся лишь того, что нужно для проверки нажатой клавиши.

Флаг нуля устанавливается в единицу, если предыдущая команда сравнения была верной (листинги 7.4 и 7.5).

Листинг 7.4. Установка флага нуля

```
...
mov ax,15
cmp ax,15
jz Our_lab
mov ah,3

Our_lab:
...
```

Во фрагменте кода из листинга 7.4 флаг нуля будет установлен, и процессор перейдет на указанную метку `Our_lab`. Команда `mov ah, 3` не будет выполняться.

Листинг 7.5. Флаг нуля не установлен

```
...
mov ax,15
cmp ax,16
jz Our_lab
mov ah,3

Our_lab:
...
```

В участке кода из листинга 7.5 процессор не перейдет на метку `Our_lab`, т. к. команда сравнения `cmp ah, 16` не будет верна. Выполнится инструкция `mov ah, 3` и, естественно, все, что идет после данной инструкции.

Допустим, нам нужно подождать, пока пользователь нажмет какую-нибудь клавишу. И если это `<Ф>` или `<ф>`, то перейти на указанную метку. В противном случае, запросить клавишу снова. Обратите внимание, что коды ПРОПИСНЫХ и строчных букв различаются! В листинге 7.6 приведен пример.

Листинг 7.6. Проверка нажатия клавиши `<Ф>` или `<ф>`

```
...
Next_key:
    mov ah, 10h
    int 16h
    cmp al, 'Ф'
    jz F_pressed
    cmp al, 'ф'
    jz F_pressed
    jmp Next_key

F_pressed:
    mov ah, 9
    mov dx, offset Mess
    int 21h
    int 20h
    Mess db 'Вы нажали <Ф> или <ф>! ! !$'
...

```

Стоит отдельно отметить, что команды `jz` и `je` (от англ. *jump if equal* — переход, если равно) выполняют одну и ту же функцию. То есть корректной и идентичной листингу 7.6 будет запись, приведенная в листинге 7.7.

Листинг 7.7. Использование оператора `je` вместо `jz`

```
...
int 16h
cmp al, 'Ф'
je F_pressed
...

```

Команда `je` переведется программой-ассемблером в `jz`. Это легко проверить, если после ассемблирования запустить программу под отладчиком. Разницы между `je` и `jz` нет никакой!

7.3. Понятия условного и безусловного переходов

Команда `jmp` (которую мы уже не раз рассматривали) называется командой *безусловного перехода*. То есть при любом условии компьютер перейдет на указанную метку. Для тех, кто знаком немного с Бейсиком: эта команда эквивалентна `goto`.

```
goto 20 — переход на строку 20
```

Команды вида `je` и `jz` являются командами условного перехода. То есть компьютер перейдет на метку *только*, если какое-то условие выполняется или не выполняется. В нашем случае — если нажата клавиша, которую мы проверяем: `cmp al, 'F'`. В Бейсике это выглядит примерно так:

```
if Key = "F" then goto 20
```

Иногда мы пишем, что *компьютер* сделает то-то или то-то. На самом деле, грамотнее говорить, что *процессор* выполнит то или иное действие. Мы посылаем команды не компьютеру (компьютер — это общее понятие), а процессору, который и выполняет наши команды, причем команды ассемблера и только. Программа, написанная на любом другом языке высокого уровня (например, Бейсике, Паскале, Си, Ада и пр.), переводится в машинный код. Проблема в том, что ни один язык высокого уровня не способен создать из команд своего языка оптимальный машинный код. Ассемблер — это создание машинного кода "вручную". А что может быть лучше ручной работы?

7.4. Расширенные коды ASCII

Теперь рассмотрим таблицу некоторых кодов символов расширенного кода ASCII.

Что такое расширенный код ASCII? Мы писали, что код клавиши, получаемый функцией `10h` прерывания `16h`, возвращается в регистр `ah`. Но в приведенных ранее примерах мы проверяли почему-то только содержимое регистра `al`:

```
cmp al, 'Y'
```

Дело в том, что кроме символов (A...Я/A...Z), цифр (0,..., 9) и других символов существуют еще другие клавиши, такие как, например, `<F1>`—`<F12>`. Мы не можем записать так, проверяя, нажата ли клавиша `<F1>`:

```
cmp al, 'F1'
```

В процессе ассемблирования произойдет ошибка, потому что '`F1`' — 2 байта, а регистр `al` может хранить только 1 байт. Для этого используются расширенные коды ASCII. При этом, если мы на запрос нашей программы нажмем клавишу `<F1>`, то в `al` помещается 0, а в `ah` — расширенный код.

В табл. 7.2 приведены расширенные ASCII-коды некоторых часто используемых клавиш.

Таблица 7.2. Расширенные коды ASCII некоторых клавиш

Клавиши	Коды ASCII
<F1>, ..., <F10>	3Bh, ..., 44h
<Alt>+<F1>, ..., <Alt>+<F10>	68h, ..., 71h
<Shift>+<F1>, ..., <Shift>+<10>	54h, ..., 5Dh
<Ctrl>+<F1>, ..., <Ctrl>+<F10>	5Eh, ..., 67h

Из таблицы видно, что клавиша <F1> имеет код 3Bh, <F2> — 3Ch, <F3> — 3Dh и т. д.

В листинге 7.8 приведена выдержка из программы, в которой мы проверим нажатие комбинации клавиш <Shift>+<F4> (в данном случае нужно нажать клавишу <Shift> и, не отпуская ее, клавишу <F4>).

Листинг 7.8. Проверка расширенного кода ASCII

```
...
(01) No_ext:
(02)     mov ah,10h
(03)     int 16h

(04)     cmp al,0
(05)     jnz No_ext

(06)     cmp ah,57h
(07)     je Shift_f4

(08)     jmp No_ext

(09) Shift_f4:
...
...
```

Мы уже выяснили, что при нажатии клавиши типа <F1>, <Alt>+<F1> и т. п. в al помещается 0, а в ah — расширенный код. В строках (04)—(07) мы это и проверяем. В строке (05) переходим на метку No_ext, если пользователь нажал клавишу, код которой не расширенный (например: <A>, <ф>, <Пробел>, <Enter> и т. п.). То есть мы как бы просто проигнорируем нажатую клавишу и "попросим" пользователя нажать другую. В строке (06) проверяем, нажата ли именно комбинация клавиш <Shift>+<F4>. Если же пользователь нажал <Shift>+<F4>, то, как не трудно догадатьсяся, программа перейдет на метку Shift_f4 — строка (09) (метку Shift_f4, естественно, можно было назвать по-другому). Если пользователь нажал какую-либо иную клавишу, имеющую расширенный код, то программа опять-таки вернется на метку No_ext (08).

Итак, программа продолжит работу, только если пользователь нажмет комбинацию клавиш <Shift>+<F4>.

В табл. 7.3 приведены коды других часто используемых клавиш, но не расширенные, т. е. эти коды будут загружаться функцией 10h прерывания 16h в регистр al.

Таблица 7.3. Коды часто используемых клавиш

Клавиша	Код
<Enter>	0Dh (13)
<ESC>	1Bh (27)
<Пробел> (<Spacebar>)	20h (32)
<Tab>	09h (9)

ПРИМЕЧАНИЕ

В скобках указаны десятичные числа.

Эти коды желательно запомнить. Полный перечень кодов ASCII находится в *приложении 3*.

7.5. Программа для практики

В данном разделе мы рассмотрим программу (\007\prog07.asm), описание которой будет приведено в главе 8. Ваша задача — постараться разобрать ее самостоятельно (листинг 7.9).

Листинг 7.9. Программа для практики

```
CSEG segment
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
org 100h
Begin:
    call Wait_key
    cmp al,27
    je Quit_prog
    cmp al,0
    je Begin

    call Out_char
    jmp Begin

Quit_prog:
    mov al,32
```

```
call Out_char
int 20h

; === Подпрограммы ===
; --- Wait_key ---
Wait_key proc
    mov ah,10h
    int 16h
    ret
Wait_key endp

; --- Out_char ---
Out_char proc
    push cx
    push ax
    push es

    push ax
    mov ax,0B800h
    mov es,ax
    mov di,0
    mov cx,2000
    pop ax
    mov ah,31
Next_sym:
    mov es:[di],ax
    inc di
    inc di
    loop Next_sym

    pop es
    pop ax
    pop cx
    ret
Out_char endp

CSEG ends
end Begin
```

Пожалуйста, внимательно набирайте программу! Если что-то не работает, то ищите опечатку в вашем наборе. Набранную ассемблерную программу (как и все

файлы-приложения) можно найти на прилагаемом к книге компакт-диске или на сайте <http://www.Kalashnikoff.ru>. А если возникнут вопросы, то вы в любое время без тени сомнения можете задать их нашим экспертам на портале <http://RFpro.ru>.

Это интересно

Если посмотреть возможности языка ассемблера, то получается, что языки высокого уровня могут делать практически то же самое и даже лучше.

Зачем нужен ассемблер? Что он может делать, что не сможет сделать любой другой язык? Ведь неинтересно изучать то, что не понимаешь!

И в самом деле, уважаемые читатели! Чем же ассемблер лучше других языков, какие у него достоинства и недостатки? Попробуем все их перечислить.

Достоинства:

- программа, написанная на ассемблере, максимально быстро работает (в 50—200 раз быстрее программы на Бейсике и в 3—20 раз быстрее программы на С++);
- код программы максимально компактен;
- позволяет делать то, что ни один язык высокого уровня не способен сделать.

Недостатки:

- больше времени затрачивается на написание программы;
- код (листинг) длиннее, чем в других языках;
- в какой-то степени сложнее других языков.

Однако вы в дальнейшем поймете, что стоит один раз написать процедуры (например, вывод окна заданных размеров на экран, получение строки символов и пр.), а затем вызывать их из других программ. В итоге, времени у вас займет не настолько больше, чем писать, например, на Паскале. Наибольший эффект достигается при комбинировании двух языков: Паскаля и ассемблера или С и ассемблера. Это становится особенно актуально при программировании под Windows.

Кстати, о Windows. Программирование под Win32 на ассемблере мало чем отличается от программирования на языках высокого уровня. Если в DOS скорость работы зависит от использования языка и профессионализма программиста, то в Windows скорость зависит в основном от операционной системы.

Многие нам возразят: зачем, мол, учитывать скорость, считать такты, оптимизировать программы, уменьшать код, если и так большинство пользователей работает за современными компьютерами, скорость работы которых очень и очень высока, и разница в несколько тысяч тактов на глаз не заметна?

Перечислим некоторые аспекты, когда требуется знание ассемблера.

- Ассемблер часто используется для написания драйверов или ядра операционной системы, где размер программы важен, а скорость выполнения отдельных участков кода очень критична, и программисты учитывают буквально каждый бит и каждый такт.
- Существуют задачи, реализация которых без применения ассемблера невозможна, т. к. языки высокого уровня просто бесполезны и не отвечают необходимым требованиям.
- Ассемблер применяют и в случаях, когда нужно исследовать другие программы, исходный код которых отсутствует. При этом, абсолютно не важно, на каком языке была написана та или иная программа. Готовый код дизассемблируется и исследуется программистами.
- Знание ассемблера также требуется, когда необходимо написать компактную программу и прошить ПЗУ. Это часто применяется в компаниях, которые занимаются разработкой систем безопасности.

Именно в таких случаях требуются высококвалифицированные программисты на ассемблере.



Глава 8

Учимся работать с файлами

8.1. Программа из прошлой главы

В принципе, ничего сложного в программе из главы 7 не было. Единственное, на чем был сделан акцент, — это на переводе шестнадцатеричных чисел в десятичные. Задача-минимум — запомнить некоторые часто используемые шестнадцатеричные, десятичные числа и соответствующие им клавиши. Например:

- 20h — 32, клавиша <Пробел>;
- 100h — 256, круглое число;
- 1Bh — 27, клавиша <Esc>;
- 21h — 33, сервис MS-DOS

и т. п.

Теперь рассмотрим выдержки из программы с комментариями (листинг 8.1).

Листинг 8.1. Обзор программы из главы 7

```
...
(01)    call Wait_key      ;ждем нажатия клавиши...
(02)    cmp al,27          ;это <Esc>?
;Если да — то на метку Quit_prog (quit — выход, prog (program) — программа)
(03)    je Quit_prog
(04)    cmp al,0            ;код клавиши расширенный? (<F1>-<F12> и т. п.)
(05)    je Begin            ;да — повторим запрос...
;Вызываем процедуру вывода нажатой клавиши на экран
(06)    call Out_char
(07)    jmp Begin           ;ждем дальше...

;Метка, на которую придет программа в случае нажатия клавиши <Esc>
(08) Quit_prog:
(09)    mov al,32            ;помещаем в al <пробел>
;Вызываем процедуру вывода символа в al (в данном случае — пробела).
;Здесь мы как бы "обманываем" процедуру Out_char, которая нужна для вывода
;нажатого символа на экран. Мы симулируем нажатие клавиши <Пробел>
```

;и вызываем процедуру.

(10) call Out_char

(11) int 20h ;выходим в DOS...

(12) ...

(13) ; --- Out_char --- ;процедура (комментарий)

(14) **Out_char proc** ;начало процедуры

;Сохраним все регистры, которые будут изменены подпрограммой...

(15) push cx

(16) push ax

(17) push es ;сохраним сегментный регистр

(18) push ax ;сохраним ах, т. к. в нем код нажатой клавиши...

(19) mov ax,0B800h ;установим es на сегмент видеобуфера

(20) mov es,ax

(21) mov di,0 ;di – первый символ первой строки

;Выводим 2000 символов (80 символов в строке * 25 строк)

(22) mov cx,2000

(23) pop ax ;восстановим код клавиши (см. строку 18)...

(24) mov ah,31 ;цвет символа

;Метка для цикла, который выполнится 2000 раз (количество повторов

;задается в строке 22)

(25) **Next_sym:**

;Заносим код клавиши и ее цвет (цвет всегда 31)

(26) mov es:[di],ax

;Увеличиваем указатель на 2 (первый байт – символ, второй байт – цвет)

(27) inc di

(28) inc di

(29) loop Next_sym ;обработка следующего символа

;Восстановим сохраненные регистры и выровняем стек

(30) pop es

(31) pop ax

(32) pop cx

(33) ret ;вернемся из подпрограммы

(34) **Out_char endp**

...

В строке (12) для экономии места опущена процедура ожидания нажатия клавиши. В целом же, программа делает следующее:

- ждет от пользователя нажатия любой клавиши;
- если это расширенный код ASCII (<F1>—<F12>, кнопки со стрелками), то игнорирует ее;
- если это не расширенный ASCII (<A>—<Z>, <0>—<9> и т. п.) — заполняет экран данным символом;
- если нажимаем клавишу <Esc> (27 или 1Bh), то заполняет экран пробелами (mov al, 32) и выходит.

8.2. Основы работы с файлами

В настоящем разделе рассмотрим функции MS-DOS для работы с файлами и общий принцип.

Для того чтобы прочитать содержимое файла, необходимо вначале открыть его. Это позволяет сделать функция 3Dh прерывания 21h (табл. 8.1).

Таблица 8.1. Функция 3Dh прерывания 21h — открытие файла

Вход	Выход
ah = 3Dh al = тип открытия (00 — только чтение, 01 — только запись, 02 — и чтение, и запись) ds : dx = адрес ASCII-строки с именем файла	ax = номер файла jc — ошибка

Итак, на входе al должен содержать тип открытия, т. е. указание на действие, для которого открывается файл:

- только чтение;
- только запись;
- чтение и запись.

Естественно, при открытии файла для чтения и записи (al=02) мы не обязаны прочитать его, а затем что-то записать. Можно только прочитать файл и/или записать что-то в него, а можно вообще ничего не делать. Однако следует иметь в виду, что если мы попытаемся открыть файл с атрибутом "только чтение" ("read-only") для записи (al=2) или для чтения/записи (al=02), то функция вернет ошибку. Код из листинга 8.2 пробует открыть файл command.com из текущего каталога для чтения/записи.

Листинг 8.2. Открытие файла для чтения/записи

```
...
mov ax, 3D02h
mov dx, offset File_name
```

```
int 21h  
...  
  
File_name db 'command.com', 0  
...
```

Обратите внимание, что в регистр `ax` загружаются сразу два числа: `3Dh` и `02h`. Такой метод работает быстрее и занимает меньше байтов, чем если бы мы загружали оба числа по отдельности:

```
...  
mov ah,3Dh  
mov al,02h  
...
```

Строку `File_name db 'command.com', 0` можно размещать в любом месте кода, но только не смешивая непосредственно с кодом. Например, нельзя вставлять данные между операторами ассемблера:

```
...  
mov ax,3D02h  
mov dx,offset File_name  
File_name db 'command.com', 0  
int 21h  
...
```

Обратите внимание, что между командами `mov dx,offset File_name` и `File_name db 'command.com'` не стоит многоточие. В данном случае, программа сасемблируется без ошибок, но зависнет при запуске. Процессор распознает `'command.com'` как набор инструкций, а не строку символов. Эти инструкции, скопеर всего, не имеют никакой логики (рис. 8.1). После команды `mov dx,0106h` (`mov dx,offset File_name`) мы видим шестнадцатеричный код строки `File_name` — `"command.com"`. Как только процессор загрузит в `dx` адрес этой строки, он начнет выполнять следующую за ней команду `aprl`, затем `insw` и т. д., что, безусловно, приведет к "зависанию" компьютера.

Из этого делаем вывод, что процессор "не отличает" данные от кода. Он просто выполняет последовательность байт, которые ему указал программист. Поэтому на плечи программиста ложится задача отслеживать, чтобы данные не попали в область кода, и процессор не начал их выполнять.

Вернемся к функциям работы с файлами. При открытии файла можно указывать не только его имя, но и полный путь к нему:

```
File_name db 'C:\ASSM\command.com', 0
```

Если имя диска или путь к нему опущены, то операционная система будет пытаться открыть файл на текущем диске в текущем каталоге. Если же указан полный путь, то файл-система попытается открыть файл только по указанному пути. ПРОПИСНЫЕ и строчные символы значения не имеют. Можно записать и так:

```
My_file db 'a:myfile.doc', 0
```

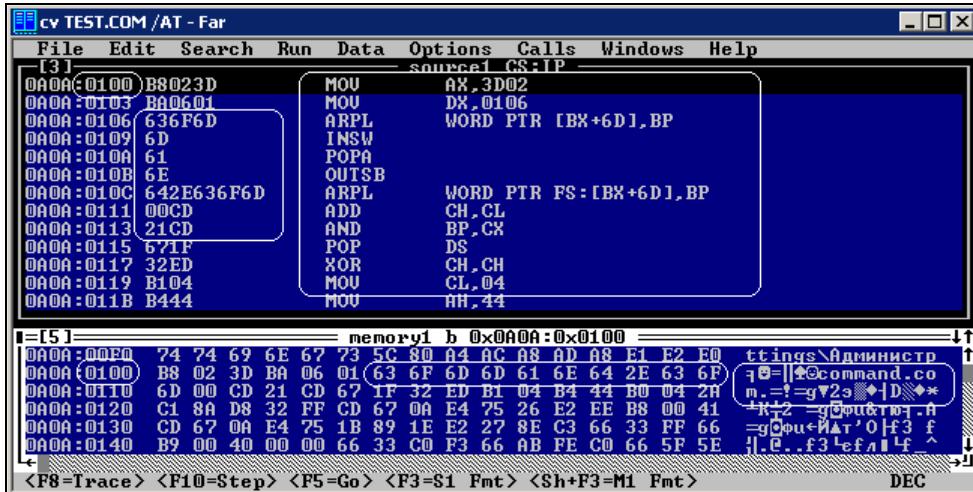


Рис. 8.1. Недопустимость перемешивания данных с кодом

В данном случае ОС попытается открыть файл myfile.doc, который должен быть расположен на диске А: в текущем каталоге.

Теперь о том, что возвращает функция. Вот код:

```
...
mov ax, 3D00h
mov dx, offset Just_file
int 21h
...
Just_file db 'file', 0
...
```

Допустим, в текущем каталоге файл с именем file (см. последнюю строку в приведенном выше примере) не был найден. Тогда функция 3Dh устанавливает в единицу флаг переноса (вспомните схожую ситуацию с флагом нуля). Если же файл все-таки найден и успешно открыт, то флаг переноса сбрасывается (становится равным нулю).

Для проверки состояния флага переноса используются операторы jc (от англ. *jump if carry* — переход, если установлен флаг переноса) и jnc (от англ. *jump if not carry* — переход, если флаг переноса не установлен):

```
...
int 21h
jc Error
Ok:
...
Error:
...
```

или так:

```
...
int 21h
jnc Ok
```

Error:

```
...
```

Ok:

```
...
```

Естественно, вместо меток `ok` (порядок) и `Error` (ошибка) можно задавать любые другие имена. По аналогии с `je` и `jne`, можно сделать вывод, что `jc` и `jnc` — команды условного перехода.

Все функции прерывания 21h устанавливают в единицу флаг переноса, если произошла ошибка, и сбрасывают его, если ошибки не было. Однако ошибка при попытке открыть файл может возникать и в следующих случаях:

- файл не найден. Имя файла или путь к нему не найдены;
- файл уже открыт какой-то программой. Файл не может быть одновременно открыт двумя и более программами (процессами);
- открыто максимально возможное количество файлов. Максимальное количество одновременно открытых файлов указывается в переменной `FILES=xx` файла `config.sys`, где `xx` — число не более 99. MS-DOS резервирует для каждого файла определенное количество байтов памяти;
- попытка открыть файл с атрибутом "только чтение" для чтения и/или записи. Прежде необходимо снять этот атрибут, а затем уже открывать файл для чтения/записи.

В листинге 8.3 приведен код, который открывает файл и выводит сообщение о том, открыт файл или нет.

Листинг 8.3. Открытие файла

```
...
mov ax,3D00h
mov dx,offset File_name
int 21h
jc Bad_file

mov dx,offset Mess1
Quit_prog:
    mov ah,9
    int 21h

    int 20h

Bad_file:
    mov dx,offset Mess2
```

```

jmp Quit_prog
...
File_name db 'c:\assm\masm\binr\ml.exe',0
Mess1 db 'Файл успешно открыт!$'
Mess2 db 'Не удалось открыть файл!$'
...

```

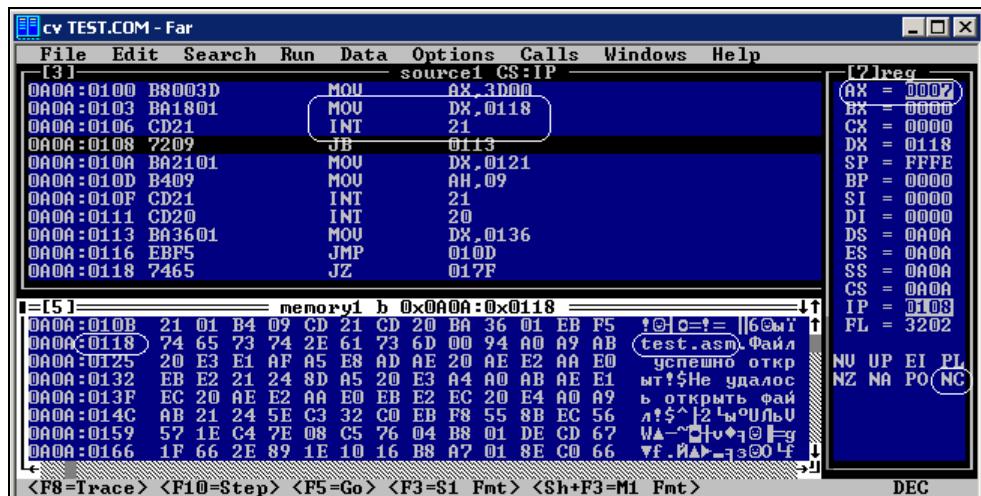


Рис. 8.2. Успешное открытие файла

При успешном открытии файла в `ax` возвращается уникальный идентификационный номер файла. В дальнейшем, при обращении к данному файлу, будет указываться *не его имя, а этот номер*. Флаг переноса сброшен. После вызова функции `3Dh` необходимо сохранить номер файла, переданный ею в регистре `ax!` Состояние регистров после открытия файла в отладчике отображено на рис. 8.2.

После того как программист закончил работу с файлом (записал или прочитал что-нибудь), файл необходимо закрыть функцией `3Eh` прерывания `21h` (табл. 8.2).

Таблица 8.2. Функция `3Eh` прерывания `21h` — закрытие файла

Вход	Выход
<code>ah = 3Eh</code> <code>bx = номер файла</code>	Ничего

Все данные, которые мы записывали в файл, на самом деле не записываются в тот же момент на диск. Они хранятся в памяти до тех пор, пока файл не будет закрыт. Только после этого сбрасываются все дисковые буферы, и файл сохраняется на диске.

Не забывайте закрывать файл! В любом случае, при завершении работы программы, операционная система автоматически закрывает все открытые этой программой файлы, освобождает занимаемую ими память и отведенные ей блоки памяти, а также сбрасывает дисковые буферы (т. е. происходит запись данных, оставшихся в кэш-памяти, на диск). Далее приведен участок кода, закрывающий открытый файл, номер которого находится в переменной Handle:

```
...
mov ah,3Eh
mov bx,Handle
int 21h      ;Файл закрыт
...
```

Обратите внимание на запись `mov bx,Handle`. Здесь Handle — это переменная, в которую необходимо будет занести номер файла после открытия. Переменные мы подробно рассмотрим в следующих главах, а сейчас коснемся только того, как создать переменную Handle. Вот пример:

```
Handle dw 0
```

Здесь мы резервируем два байта для хранения некоторых данных. В нашем случае — для хранения номера файла. В листинге 8.4 приведен фрагмент программы, который открывает файл для чтения, сохраняет номер файла в переменной, а затем закрывает файл.

Листинг 8.4. Корректное открытие и закрытие файла

```
...
mov ax,3D00h
mov dx,offset File_name
int 21h
jc Error
mov Handle,ax
;файл открыт успешно...

mov ah,3Eh
mov bx, Handle
int 21h
;файл закрыт

Error:
int 20h
...
Handle dw 0
...
```

Для чтения информации из файла используется функция 3Fh, а для записи в файл — 40h. При этом bx должен содержать тот самый номер файла — Handle,

`cx` — количество читаемых или записываемых байтов, `ds:dx` — адрес буфера для чтения/записи. Пример чтения файла в память с использованием функции `40h` приведен в разд. 8.3.

8.3. Программа для практики

Чтобы полностью понять приведенный далее пример, придется рассмотреть много нового. В данном случае мы поступим так же, как и в главе 1: о некоторых операторах (командах) скажем позже.

Итак, в листинге 8.5 приведен образец чтения файла (до `65 000 (0FDE8h)` байт) в память, а точнее, в наш сегмент (`(008\prog08.asm)`).

Листинг 8.5. Чтение файла в память

```
CSEG segment
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
org 100h

;Начало
Begin: mov ax,3D00h
        mov dx,offset File_name
        int 21h
        jc Error_file

        mov Handle,ax
        mov bx,ax
        mov ah,3Fh
        mov cx,0FDE8h
        mov dx,offset Buffer
        int 21h

        mov ah,3Eh
        mov bx,Handle
        int 21h

        mov dx,offset Mess_ok
Out_prog:
        mov ah,9
        int 21h

        int 20h

Error_file:
        mov dx,offset Mess_error
```

```
jmp Out_prog

;==== Переменные ====
Handle dw 0
Mess_ok db 'Файл загружен в память! Смотрите в отладчике!$'
Mess_error db 'Не удалось открыть файл '

;Будем читать этот файл:
File_name db 'c:\msdos.sys',0,'!$'

Buffer equ $

CSEG ends
end Begin
```

Из этого примера вы узнаете очень много. Чтобы досконально разобраться в этой программе, необходимо, как обычно, воспользоваться отладчиком. В нем внимательно изучайте дамп (содержимое) памяти, на который указывает пара регистров `ds:dx`.



Глава 9

Работа с файлами

9.1. Программа из прошлой главы

В листинге 9.1 приведены выдержки программы из главы 8 с комментариями.

Листинг 9.1. Программа из главы 8

```
...
;Опустим код CSEG и пр.

(01) Begin: mov ax,3D00h      ;будем открывать файл для чтения
;Обратите внимание, как мы записываем операторы (сразу за меткой).

(02)    mov dx,offset File_name      ;ds:dx указывают на путь к файлу
(03)    int 21h                      ;открываем
(04)    jc Error_file
;Если произошла ошибка (нет такого файла, слишком много открытых файлов,
;ошибка чтения), то переходим на метку Error_file

;Запомним номер файла в переменной Handle.
;Для того чтобы прочитать файл, нужно в bx указать его номер, полученный
;после открытия, который находится в ах. Загрузка числа в один регистр из
;другого (а тем более, если используется ах) происходит быстрее, чем
;из памяти (переменной). Поэтому загружаем из ах, а не из переменной. Хотя
;запись mov bx,Handle не будет ошибочной.

(05)    mov Handle,ax      ;запомним номер файла в переменной Handle
(06)    mov bx,ax
(07)    mov ah,3Fh          ;функция 3Fh – чтение файла
(08)    mov cx,0FDE8h       ;будем читать 0FDE8h = 65000 байт
;ds:dx должен указывать на буфер в памяти для чтения
(09)    mov dx,offset Buffer
(10)    int 21h            ;все готово. Читаем...

(11)    mov ah,3Eh          ;закрываем файл
```

;Номер файла должен находиться в bx. Но т. к. регистр bx менялся, то
;загрузим его с нашей переменной (Handle)

(12) mov bx,Handle
(13) int 21h ;закрываем файл

;загрузим в dx строку с сообщением о том, что все в порядке.

(14) mov dx,offset Mess_ok

(15) Out_prog: mov ah,9 ;функция 09h – вывод строки на экран
(16) int 21h ;выводим строку

(17) int 20h ;выходим из программы

;Загрузим в dx строку с сообщением о том, что не смогли открыть файл...

(18) Error_file: mov dx,offset Mess_error

;...и пойдем на метку Out_prog (зачем нам дублировать код, если он уже есть?)

(19) jmp Out_prog

;==== Данные ===

(20) Handle dw 0 ;резерв 2 байта для нашей переменной

(21) Mess_ok db 'Файл загружен! Смотрите в отладчике!\$'

;Строки (22)–(23) рассмотрим ниже

(22) Mess_error db 'Не удалось открыть (найти) файл '

(23) File_name db 'c:\msdos.sys',0,'!\$'

(24) Buffer equ \$

...

Запомните оператор \$, который нами впервые используется в строке (24). При ассемблировании нашей программы ассемблер заменит этот знак адресом, по которому он расположен (листинг 9.2).

Листинг 9.2. Использование оператора \$

```
(01) CSEG segment
(02) assume cs:CSEG
(03) org 100h

(04) Begin:
(05)     My_lab equ $
(06)     My_lab2 equ $+2
(07)     mov bx,offset My_lab
```

```
(08)      mov dx,offset My_lab2
(09)      int 20h

(10) CSEG ends
(11) end Begin
```

Строки (5) и (6) места в памяти не занимают (как и метки). Ассемблер-программа запомнит, что метка `My_lab` находится по адресу 100h (`org 100h`), а метка `My_lab2` — 102h. На рис. 9.1 показан фрагмент приведенной выше программы, который иллюстрирует использование оператора \$.

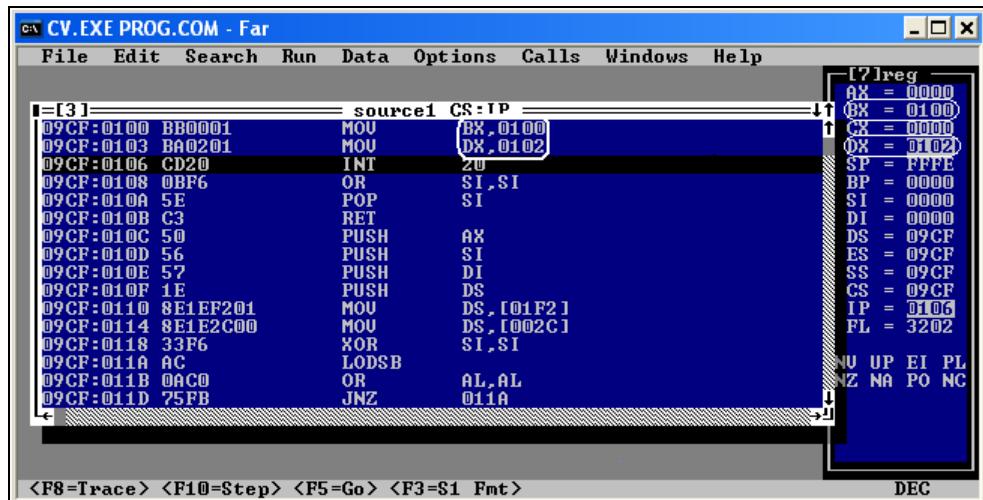


Рис. 9.1. Использование оператора \$

В программе из главы 8 мы размещаем метку `Buffer` в конце кода. Таким образом, команда `mov dx, offset Buffer` занесет в `dx` последний свободный байт в нашем сегменте `CSEG`. По этому адресу мы и будем загружать программу в память. В отладчике это хорошо видно.

В строке (22) находится переменная с сообщением о том, что файл открыть невозможно. Однако она не заканчивается символом \$, как обычно. Это значит, что функция 09h прерывания 21h продолжит выводить символы до тех пор, пока в памяти не встретится этот символ. Если мы посмотрим на строку (23), то увидим, что она как раз заканчивается знаком \$. Это значит, что при выводе символов из строки (22) функция выведет также и те символы, которые находятся в строке (23).

```
(22) Mess_error db 'Не удалось открыть (найти) файл '
(23) File_name db 'c:\msdos.sys',0,'!$'
```

Если файл не был найден, функция 09h выведет на экран буквально следующее (рис. 9.2):

Не удалось открыть (найти) файл c:\msdos.sys !

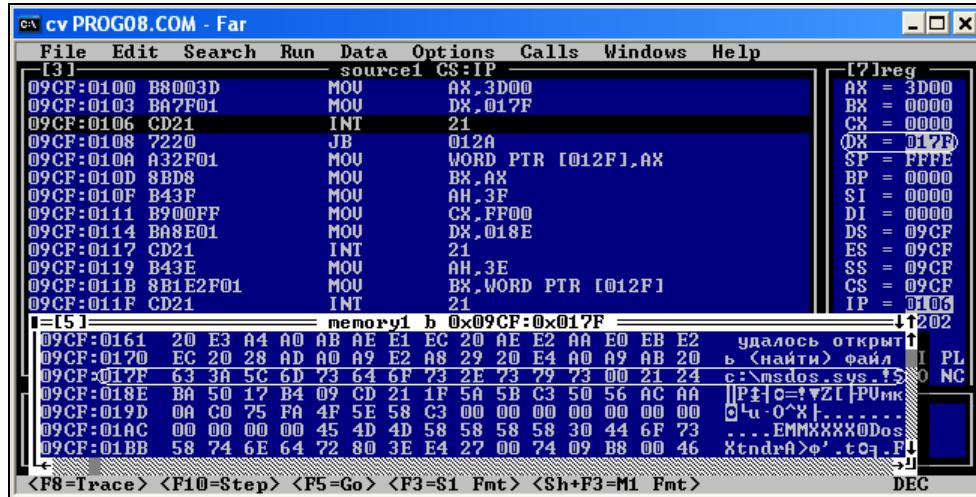


Рис. 9.2. Имя открываемого файла

Символ '0' после `c:\msdos.sys` будет отображен как пробел.

А для чего нужен '0' в строке (23)? При открытии файла пара регистров `ds:dx` должна содержать ссылку на имя файла. Стока должна завершаться символом '0'. Если этот символ убрать, то функция, скорее всего, вернет ошибку. Ведь файла с именем `c:\msdos.sys!$` не существует!

Можем сделать и так, конечно. Только в этом случае мы потеряем байты:

```
Mess_error db 'Не удалось открыть (найти) файл c:\msdos.sys!$'
File_name db 'c:\msdos.sys',0
```

Обращаем ваше внимание, что мы будем постоянно использовать данный метод в наших примерах.

9.2. Программа для практики

Теперь подробнее рассмотрим работу с файлами. В листинге 9.3 приведен пример (`\009\Prog09.asm`).

Листинг 9.3. Открытие, чтение и вывод файла на экран

```
CSEG segment
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
org 100h
; ----- Начало -----
Begin:
    mov dx,offset File_name
    call Open_file
    jc Error_file
```

```
; ----- Открыли файл -----
mov bx,ax
mov ah,3Fh
mov cx,offset Finish-100h
mov dx,offset Begin
int 21h
; ----- Прочитали файл -----
call Close_file

; ----- Выводим сообщение -----
mov ah,9
mov dx,offset Mess_ok
int 21h
ret

; ----- Не смогли найти файл -----
Error_file:
mov ah,2
mov dl,7
int 21h
ret

; Процедуры
; --- Открытие файла ---
Open_file proc
    cmp Handle,0FFFFh
    jne Quit_open
    mov ax,3D00h
    int 21h
    mov Handle,ax
    ret
Quit_open:
    stc
    ret
    Handle dw 0FFFFh
Open_file endp

; --- Закрытие файла ---
Close_file proc
    mov ah,3Eh
```

```
    mov bx,Handle
    int 21h
    ret
Close_file endp

; Данные
File_name db 'Prog09.com',0
Mess_ok db 'Все нормально!', 0Ah, 0Dh, '$'

Finish equ $

CSEG ends
end Begin
```

ВНИМАНИЕ!

Этот файл нужно обязательно сохранить как Prog09.asm!

Эта программа также с подвохом. Попробуйте самостоятельно разобраться с принципом ее работы. Отладчик в данном случае мало поможет, поэтому вам нужно будет разбирать программу без его помощи.

Принцип работы данной программы будет подробно рассмотрен в главе 10.



ЧАСТЬ III

**Файловая оболочка,
вирус, резидент**



Глава 10

Введение в вирусологию. Обработчик прерываний

10.1. Программа из прошлой главы

В главе 9 наша программа перезагружала сама себя (загружалась поверх своего кода) в оперативной памяти. Мы не зря просили вас присвоить ей имя именно Prog09.asm. Если программа не найдет файл с данным именем, то она просто завершит работу, при этом издаст звуковой сигнал. Ничего сложного в ней нет. Данная программа наглядно показывает, что в ассемблере можно проделывать невероятные вещи, которые языкам высокого уровня попросту недоступны. Далее приведен код программы с некоторыми пояснениями ее работы (листинг 10.1).

Листинг 10.1. Программа Prog09.asm из главы 9

```
;Программа выполняет запись самой себя в то место памяти, куда она
;уже загружена
CSEG segment
assume CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
org 100h

Begin:
;Открываем файл с помощью спецпроцедуры (см. ниже).
mov dx,offset File_name
call Open_file          ;Открываем файл с именем Prog09.com
jc Error_file           ;Переходим на метку Error_file при неудаче
mov bx,ax                ;Сохраняем идентификатор файла

;Чтение файла
mov ah,3Fh
;Загружаем длину нашей программы (количество читаемых байтов) в регистр cx...
mov cx,offset Finish-100h
mov dx,offset Begin      ;И читаем файл в память,
int 21h                  ;начиная с метки Begin.

call Close_file          ;Закрываем файл с помощью спецпроцедуры
```

```
; Выводим сообщение об удачном завершении
mov ah,9
mov dx,offset Mess_ok
int 21h
ret

; Если файл с указанным именем не нашли (File_name db 'Prog09.com',0),
; то выдаем звуковой сигнал и выходим
Error_file:
mov ah,2
mov dl,7
int 21h
ret

;Процедуры
;Процедура открытия файла для чтения
Open_file proc
cmp Handle,0FFFFh      ;Выясняем, открыт ли файл
jne Quit_open           ;И если не открыт — открываем его
mov ax,3D00h
int 21h
mov Handle,ax
ret

Quit_open:
stc      ;Устанавливаем флаг переноса в 1, необходимый
ret      ;для подтверждения факта открытия файла (для jc)

Handle dw 0FFFFh
Open_file endp

;Процедура закрытия файла
Close_file proc
mov ah,3Eh
mov bx,Handle
int 21h
ret
Close_file endp

File_name db 'prog09.com',0

;0Ah,0Dh — переход в начало следующей строки
```

```

Mess_ok db 'Все нормально!', 0Ah, 0Dh, '$'

Finish equ $ ;Признак (адрес) конца кода программы

CSEG ends
end Begin

```

На рис. 10.1 приведен фрагмент кода в отладчике CodeView перед тем, как программа прочитает себя в память. Здесь следует обратить внимание на пару регистров *ds:dx*, которые указывают функции 3Fh, в какое место памяти следует читать данные из файла.

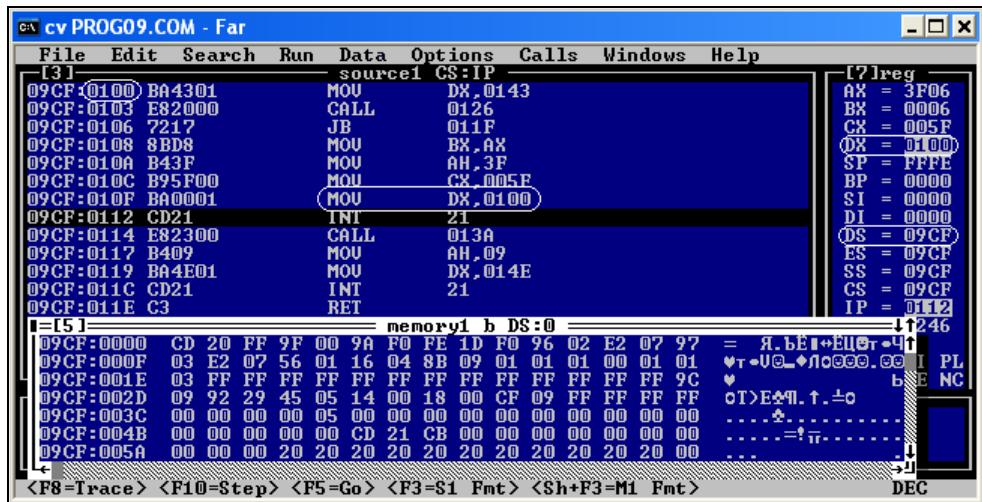


Рис. 10.1. Перечитывание программы поверх себя

Дополнительно следует особо отметить действие команды *ret* в следующих строках:

```

...
mov ah,9
mov dx,offset Mess_ok
int 21h
ret      ;Что делает ret здесь?
...

```

В данном случае все просто. Полагаем, вы наверняка заметили, что отладчик переходит по адресу *cs:0000h* на команду *int 20h*, которая и завершает работу (рис. 10.2 и 10.3).

Использование инструкции *ret* для завершения программы предпочтительней, т. к. мы экономим один байт. Однако это работает только в том случае, если программист перед выполнением финальной инструкции *ret* выровнял стек, т. е. вернулся в исходное состояние регистра *ss:sp*, если они изменились.

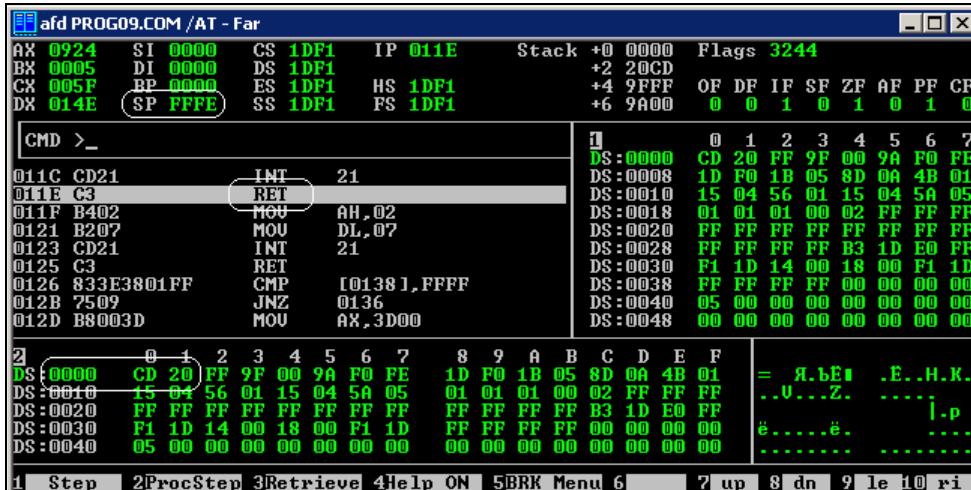


Рис. 10.2. Перед выполнением последней инструкции ret

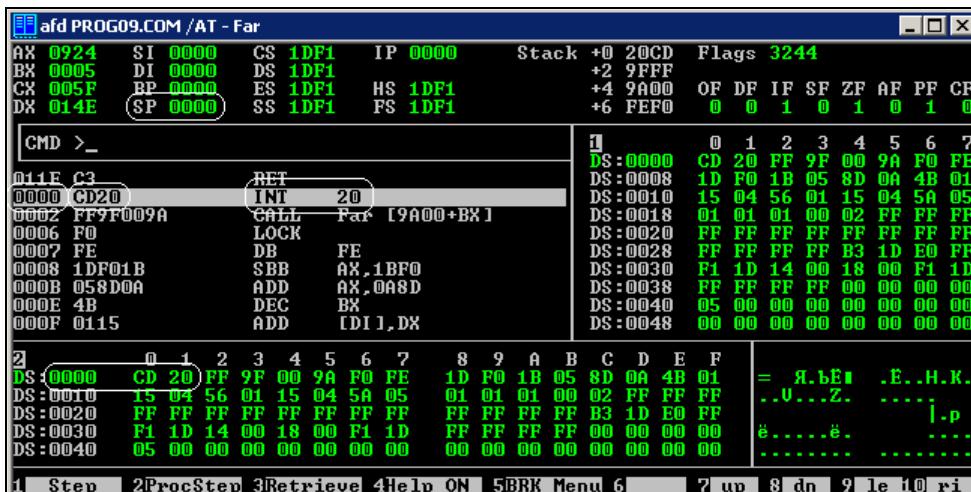


Рис. 10.3. Инструкция int 20h — выход в DOS

Вы, конечно, помните, что при загрузке COM-программы вершина стека устанавливается на предпоследний байт нашего сегмента, т. е. sp=0FFF Eh. При выполнении команды "ближний call" sp уменьшается на 2, т. е. становится равным 0FFFC h. Это свидетельствует о том, что при вызове подпрограммы процессор заносит в стек адрес возврата. Поэтому и не происходит путаницы при работе: выходить ли в DOS или вернуться из процедуры. Так или иначе, выход в DOS командой ret в COM-программах происходит только в том случае, если sp=0FFF Eh, и программа сама не затерла первые 2 байта по смещению 0000h (т. е. команду int 20h) того сегмента, куда она загрузилась изначально.

10.2. Вирус

Вирус — это обычная программа, написанная на ассемблере, Pascal, C, Visual Basic и прочих языках программирования. Эта программа может сама размножаться, заражая другие программы без участия программиста, написавшего этот вирус.

Писать вирус на ассемблере — милое дело. Компьютерные вирусы подобны настоящим вирусам. Как больной человек может заразить другого, так и компьютерный вирус способен заражать "здоровые" файлы. Зараженный файл может заразить другой и т. д. Если вовремя не "вызвать доктора", то вирус заразит все файлы, а также может уничтожить часть или всю информацию на жестком диске (все зависит от того, что и при каких условиях выполняет программа-вирус).

Прежде чем перейти к изучению структуры вируса, мы хотели бы предупредить вас об уголовной ответственности за распространение вирусоподобных программ.

Кто пишет вирусы?

- Прежде всего, это программисты, которые только-только изучили ассемблер, и не видят иной возможности применить свои знания, кроме как написать какой-нибудь вирус и тайно передать его своему соседу.
- Представим ситуацию: у фирмы X есть конкурент — фирма Y. Как сделать так, чтобы на время фирме Y доставить хлопот? А почему бы их компьютеры не заразить вирусом "собственного производства", который такого-то числа или при таких-то обстоятельствах удалял бы всю информацию на винчестере? Пишется вирус сотрудником фирмы X, а затем заносится в компьютер к конкуренту. И все! Как правило, заражаются компьютеры не только фирмы Y, но и другие (сотрудник переписал зараженный файл и принес к себе домой; этот же файл переписал его знакомый, файл распространился по Интернету и т. д.).
- Вирусы могут написать также люди, желающие завладеть секретной информацией пользователей. Например, клавиатурный шпион, перехватывающий нажатие клавиш пользователем и отправляющий эту информацию по указанному адресу. Что это может быть за информация? Логин и пароль от кошелька Yandex.Деньги, регистрационные данные электронной почты, сайтов и т. д. В этом случае автор вируса получает несанкционированный доступ к необходимой ему информации посетителей Интернета.

Последнее время появилось множество вирусов, написанных на языках высокого уровня и работающих под управлением интерпретатора. К таким языкам можно отнести, в частности, Visual Basic for Applications (VBA). Такие вирусы пишутся за считанные минуты и могут быть легко обнаружены пользователем и удалены без помощи антивирусных программ. Безусловно, они не идут ни в какое сравнение с вирусами, написанными на ассемблере. В этой книге под словом "вирус" мы будем подразумевать только вирусы, которые написаны на чистом ассемблере.

Естественно, существуют антивирусы (DrWeb, NOD32, Norton Antivirus, Антивирус Касперского и пр.). Однако они способны вылечить файл, который знаком автору того или иного антивируса. Если, допустим, мы напишем свой вирус, то уверяю вас, что ни один антивирус его не обнаружит. Максимум — антивирус напишет сообщение о возможном (!) заражении файла, но вылечить его не сможет.

10.2.1. Структура и принцип работы вируса

Что должен делать вирус?

Прежде всего — определить, загружен ли он вместе с каким-нибудь файлом или это его первая загрузка. Затем найти первый попавшийся файл, проверить, не заражен ли уже файл этим вирусом. Если заражен, то искать следующий файл, а если нет — то заразить его. Все это происходит очень быстро и для пользователя практически незаметно.

Какой объем памяти занимает вирус?

Все зависит от фантазии программиста, от того, что он хочет сделать с компьютером/файлами, его профессионализма и языка, на котором пишется вирус. Средний размер вируса — 500—600 байт. Хотя есть умельцы, у которых вирус занимает 100—150 байт, а есть и такие, у которых простейший вирус занимает 1,5—3 и более килобайт.

Что может вирус?

Все, что угодно. Например, удалить всю информацию с винчестера за 0,5—1 секунду по принципу работы программы FDISK, т. е. очистить таблицу разделов жесткого диска, обнулить таблицу размещения файлов (FAT — File Allocation Table), физически испортить винчестер и монитор! Все зависит от фантазии и профессионализма программиста.

Какой вирус мы будем изучать?

Мы будем рассматривать нерезидентный вирус, заражающий СОМ-файлы (самый простой).

Что будет делать вирус?

Ничего, кроме размножения.

Как оформляется вирус?

Точно так же, как обычные СОМ-файлы, созданные нами ранее (листинг 10.2).

Листинг 10.2. Структура вируса

```
CSEG segment  
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG  
org 100h  
  
Start:  
  
...      ;Здесь находится тело вируса.  
  
CSEG ends  
end Start
```

Это типичный СОМ-файл, структура которого вам уже известна. Можно создать и EXE-файл, однако его размер будет больше.

В следующих разделах мы будем подробнее рассматривать работу вируса на практике.

10.3. Резидент

Приступаем к самому сложному, но очень интересному разделу. Будет действительно сложно. Но мы уверены, что вы без особого труда разберетесь.

ВНИМАНИЕ!

Резидентные программы могут работать некорректно, если их запускать из файловых оболочек, использующих API-функции. Таковой, например, является FAR Manager. Если резидентные программы из данной книги действительно работают некорректно с некоторыми оболочками, то запускайте их из командной строки или из DOS Navigator.

Резидентная программа (резидент) — программа, которая постоянно находится в памяти. Примером резидента является драйвер мыши, Norton Guide, всевозможные антивирусы, которые следят за тем, что делает та или иная программа и сообщают о ее действиях пользователю и пр. Естественно, любая резидентная программа всегда занимает часть оперативной памяти (в DOS — часть основной памяти).

Как правило, резидентная программа должна перехватывать то или иное прерывание, с тем, чтобы программист или другие программы могли обратиться к ней. Давайте подробнее рассмотрим прерывания (немного мы затрагивали данную тему в предыдущих главах).

10.3.1. Подробней о прерываниях

Прерывание — это своего рода процедура (подпрограмма), которая имеет не название (например, `print_string`), а номер. Всего существует 256 прерываний. Некоторые номера зарезервированы BIOS (ПЗУ) компьютера. Например, `16h`:

```
...
mov ah,10h
int 16h
...
```

Или операционной системой (MS-DOS):

```
...
mov ah,9
mov dx,offset String
int 21h
...
```

Тем не менее, ничто не мешает программисту перехватить, скажем, прерывание `21h` и таким образом проконтролировать, кто и что делает с ним. Например, вызовем функцию `09h` прерывания `21h`:

```
...
mov ah,9
```

```

mov dx,offset Our_string
int 21h

...
Our_string db 'Привет!$'
...

```

В результате операционная система выведет на экран сообщение "Привет!". Мы же можем перехватить прерывание 21h, и если какая-то программа попытается вывести на экран некую строку, то мы в регистры ds:dx подставим адрес нашей строки. Теперь функция 09h прерывания 21h будет выводить на экран нашу и только нашу строку. Это можно сделать следующим образом (листинг 10.3).

Листинг 10.3. Часть нашего обработчика прерывания 21h

```

...
cmp ah,9
je Out_str
Go_21h:
...
;Здесь передаем управление предыдущему (оригинальному) обработчику 21h.
...
Out_str:
push cs
pop ds
mov dx,offset My_string
jmp Go_21h
...
My_string db "Моя строка$"
...

```

В приведенном примере проверяем, вызывается ли функция вывода строки на экран 09h прерывания 21h или какая-либо другая. Если вызывается иная функция, то мы просто передаем управление оригинальному обработчику. В противном случае загружаем в ds:dx адрес некоторой нашей строки и опять-таки передаем управление оригинальному обработчику прерывания 21h. Что произойдет, полагаем, догадаться несложно.

Обработчик прерывания — это процедура, постоянно (или временно) находящаяся в памяти. Обработчик прерывания первым получает управление, выполняет некоторые действия, а затем передает управление оригинальному обработчику (т. е. процедуре, которая уже находилась в памяти до загрузки нашего обработчика). Оригинальный обработчик может также выполнить некоторые действия, а затем передать управление другому обработчику и т. д.

Вообще оригинальные (скажем так, первичные) обработчики MS-DOS прерываний 20h—2Fh находятся в файлах IO.SYS/MSDOS.SYS. До того момента, пока они не загрузились в память, эти прерывания, грубо говоря, "пустые". То есть при попытке вызвать одно из прерываний DOS (начиная с 20h) до загрузки указанных выше программ либо ничего не произойдет, либо компьютер просто "зависнет".

Прерывания делятся на программные и аппаратные по способу их вызова.

- *Программные прерывания* вызывает непосредственно программа при помощи команды int (отсюда и название — программные), например, для того чтобы получить код клавиши, которую нажмет пользователь (`mov ah, 0/int 16h`), или вывести некоторую строку на экран (`mov ah, 9/int 21h`).
- *Аппаратные прерывания* вызываются самостоятельно процессором (аппаратной компьютера) при возникновении каких-либо событий. При этом процессор прекращает выполнение текущей программы, сохраняет в стеке регистры ss, sp и флаги, вызывает соответствующее прерывание, а затем восстанавливает сохраненные регистры и продолжает выполнение текущей программы. Например: при нажатии и отпускании какой-либо клавиши пользователем вызывается прерывание ПЗУ 09h. Или прерывание таймера (также ПЗУ) 1Ch, вызываемое автоматически примерно 18,2 раза в секунду.

Однако процессор не всегда продолжает выполнение текущей программы после обработки аппаратного прерывания. В некоторых случаях (например, если программа выполнила деление на ноль) продолжение выполнения программы невозможно. Процессор вызывает соответствующее прерывание, которое выводит на экран сообщение "Divide overflow" (или подобное ему) и останавливает систему.

Программы могут недолго заблокировать вызов аппаратных прерываний (кроме немаскируемых, типа "деления на ноль") с помощью команды cli в момент выполнения критических участков кода. К таким участкам можно отнести, к примеру, смену векторов прерываний напрямую в таблице векторов прерываний или смену стековых регистров. После выполнения необходимых действий программа должна снова разрешать вызов аппаратных прерываний при помощи команды sti. Если программист "забыл" разблокировать вызов аппаратных прерываний командой sti после их блокировки cli, то пользователь не сможет работать с компьютером, т. к. аппаратные прерывания (в том числе и 09h — управление клавиатурой) будут заблокированы, что фактически приведет к "зависанию" компьютера. Однако программные прерывания можно вызывать всегда.

В системе Windows используется новое поколение готовых подпрограмм, которые называются WinAPI. Принцип работы WinAPI очень похож на работу прерываний. Однако исследовать и понять работу прерываний проще, поэтому мы с них и начнем. Полученные знания понадобятся вам при программировании под Windows.

10.4. Первый обработчик прерывания

Теперь перейдем непосредственно к обработчику, его код расположен в листинге 10.4 (010\prog10.asm) (обращайте внимание на описания после точки с запятой).

Листинг 10.4. Обработчик прерывания

```
CSEG segment
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
org 100h
Start:
;Переходим на метку инициализации. Нам нужно будет перехватить прерывание 21h,
;а также оставить программу резидентной в памяти
jmp Init
```

;Ниже идет, собственно, код обработчика прерывания 21h (он будет резидентным).
;После того как программа выйдет, процедура Int_21h_proc останется в памяти
;и будет контролировать функцию 09h прерывания 21h.

;Мы выделим код обработчика **полужирным шрифтом**.

```
Int_21h_proc proc
cmp ah,9           ;Проверим: это функция 09h?
je Ok_09
```

;Если нет, перейдем на оригинальный обработчик прерывания 21h.
;Все. На метку Ok_09 программа уже не вернется
jmp dword ptr cs:[Int_21h_vect]

```
Ok_09:
push ds             ;Сохраним регистры
push dx
push cs             ;Адрес строки должен быть в ds:dx
pop ds
```

;Выводим нашу строку (My_string) вместо той, которую должна была вывести
;программа, вызывающая прерывание 21h
mov dx,offset My_string
pushf ;Эта инструкция здесь необходима...
call dword ptr cs:[Int_21h_vect]

```
pop dx              ;Восстановим использованные регистры
pop ds
iret                ;Продолжим работу (выйдем из прерывания)
;Программа, выводящая строку, считает, что на экран было выведено
;ее сообщение. Но на самом деле это не так!
```

;Переменная для хранения оригинального адреса обработчика 21h

```
Int_21h_vect dd ?
```

```
My_string db 'Моя строка!$'
```

```
int_21h_proc endp
```

```
;Со следующей метки нашей программы уже не будет в памяти (это нерезидентная  
;часть). Она затрется сразу после выхода (после вызова прерывания 27h)
```

```
Init:
```

```
;Установим наш обработчик (Int_21h_proc) (адрес нашего обработчика)
```

```
;на прерывание 21h. Это позволяет сделать функция 25h прерывания 21h.
```

```
;Но прежде нам нужно запомнить оригинальный адрес этого прерывания.
```

```
;Для этого используется функция 35h прерывания 21h:
```

```
;ah содержит номер функции
```

```
mov ah,35h
```

```
;al указывает номер прерывания, адрес (или вектор) которого нужно получить  
mov al,21h
```

```
int 21h
```

```
;Теперь в es:bx адрес (вектор) прерывания 21h (es – сегмент, bx – смещение)
```

```
;Обратите внимание на форму записи
```

```
mov word ptr Int_21h_vect,bx
```

```
mov word ptr Int_21h_vect+2,es
```

```
;Итак, адрес сохранили. Теперь перехватываем прерывание:
```

```
mov ax,2521h
```

```
mov dx,offset Int_21h_proc ;ds:dx должны указывать на наш обработчик  
; (т. е. Int_21h_proc)
```

```
int 21h
```

```
;Все! Теперь, если какая-либо программа вызовет 21h, то вначале компьютер
```

```
; попадет на наш обработчик (Int_21h_proc). Что осталось? Завершить программу,
```

```
; оставив ее резидентной в памяти (чтобы никто не затер наш обработчик).
```

```
;Иначе компьютер просто зависнет.).
```

```
mov dx,offset Init
```

```
int 27h
```

```
;Прерывание 27h выходит в DOS (как 20h), при этом оставив нашу программу  
;резидентной. dx должен указывать на последний байт, оставшийся в памяти  
;(это как раз метка Init). То есть в памяти остается от 0000h до адреса,  
;по которому находится метка Init.
```

```
CSEG ends
```

```
end Start
```

10.4.1. Новые операторы и функции прерываний

Рассмотрим новые прерывания и операторы, используемые в обработчике (табл. 10.1—10.3).

Таблица 10.1. Функция $35h$ прерывания $21h$: получить адрес (вектор) прерывания

Вход	Выход
$ah = 35h$	$es =$ сегментный адрес вектора прерывания $bx =$ его смещение

Таблица 10.2. Функция $25h$ прерывания $21h$: установить вектор прерывания

Вход	Выход
$ah = 25h$ $ds =$ сегмент нашего обработчика $dx =$ его смещение	Ничего

Таблица 10.3. Прерывание $27h$: оставить программу резидентной в памяти

Вход	Выход
$ds : dx =$ последний байт, оставляемый в памяти	Ничего (выход в DOS)

10.5. Работа с флагами процессора

Познакомимся с двумя новыми операторами (табл. 10.4 и 10.5).

Таблица 10.4. Оператор $pushf$

Команда	Перевод	Назначение	Процессор
$pushf$	Push flags — втолкнуть флаги	Сохранить флаги	8086

Таблица 10.5. Оператор $popf$

Команда	Перевод	Назначение	Процессор
$popf$	Pop flags — вытолкнуть флаги	Восстановить флаги	8086

Рассмотрим еще раз работу флагов процессора на конкретном примере:

```
...
(01)    cmp ax,23
(02)    je Ok_ax
...
```

Здесь, после выполнения строки (01), установится либо сбросится флаг нуля. В строке (02) проверяем этот самый флаг. Если он установлен, значит, команда

сравнения верна (т. е. $ax = 23$). Если же не установлен, то ax не равен 23. Рассмотрим, как это отображается в отладчиках AFD и Code View (особенно обращайте внимание на то, как в этих отладчиках отображается флаг нуля и прочие флаги) (рис. 10.4 и 10.5).

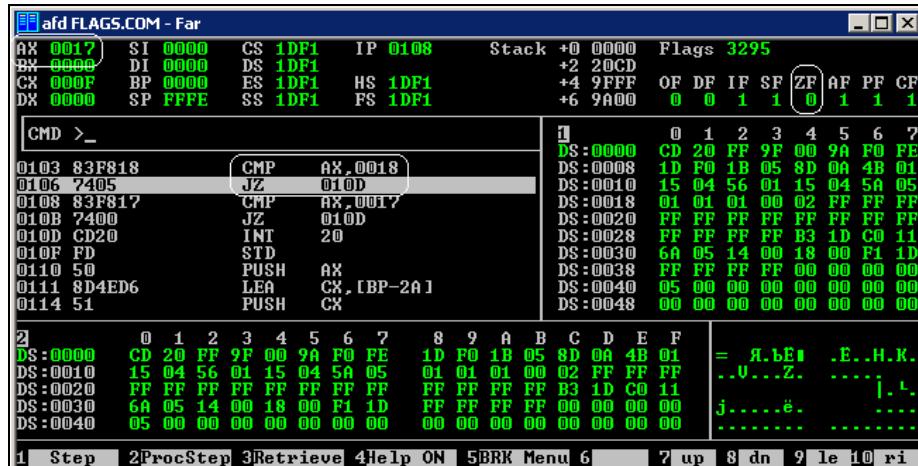


Рис. 10.4. AFD: Флаг нуля ZF (Zero Flag) сброшен

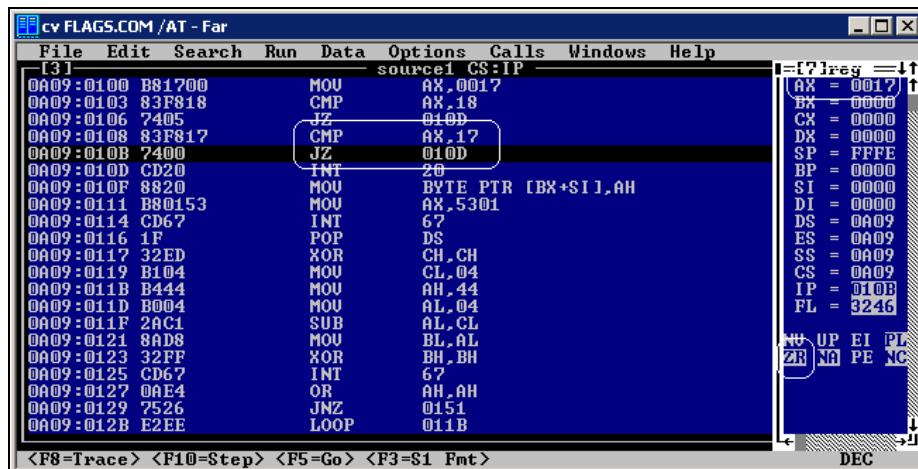


Рис. 10.5. Code View: Флаг нуля ZR (ZeRo) установлен

Теперь обратите внимание на самое начало процедуры `Int_21h_proc` листинга 10.4. Мы видим следующие команды:

```
...
cmp ah, 9
je Ok_09
...
...
```

В результате выполнения команды сравнения будет изменен флаг нуля. Однако нет необходимости сохранять флаги в стеке в нашем обработчике. Дело в том, что при вызове любого прерывания (будь то программный или аппаратный вызов) в стеке сохраняются не только регистры `cs:ip`, но и флаги. Таким образом, после выполнения команды `iret` процессор восстановит указанные выше регистры и флаги.

Тем не менее, все остальные регистры должны быть сохранены, если процедура обработки их изменяет. Иначе это может привести к довольно-таки серьезным изменениям в ходе работы программы, вызвавшей прерывание `21h`. Для этого, собственно, мы и сохраняем в стеке регистры `ds` и `dx`. Обратите внимание на то, что в обработчике прерывания (в нашем случае — `Int_21h_proc`) необходимо сохранять *все* используемые регистры, кроме `cs:ip` и регистра флагов.

Теперь рассмотрим следующую строку:

```
;Переменная для хранения оригинального адреса обработчика 21h
Int_21h_vect dd ?
```

Данная переменная может хранить двойное слово (Define Double word — определить двойное слово — четыре байта). Вспоминаем, что один 16-разрядный регистр занимает 2 байта (одно слово) (`dx`, `ax`, `es`, `ss` и т. д., но не `ah`, `dl`, `bh`, ...) — это 8-разрядные регистры, которые занимают 1 байт). Если мы хотим загрузить в переменную `Int_21h_vect` слово (2 байта), то нам необходимо указать это следующим образом:

```
mov word ptr Int_21h_vect,ax
```

Если же мы хотим загрузить 1 байт, то запишем так:

```
mov byte ptr Int_21h_vect,ah
```

Вспоминаем, что "word" по-английски — это "слово", а `byte` — "байт".

10.5.1. Как проверить работу программы?

В данном случае нам понадобится отладчик, который способен заходить "внутрь" прерывания `21h`. Например, AFD, TD, SoftIce и пр., но не CodeView. Если у вас нет AFD, то его можно загрузить с сайта <http://www.Kalashnikoff.ru>. Затем создайте простейшую программу, которая будет выводить на экран некоторую строку путем вызова функции `09h` прерывания `21h`. Например, так, как это показано в листинге 10.5 (`\010\test10.asm`).

Листинг 10.5. Программа для проверки работы прерываний

```
CSEG segment
assume CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
org 100h
Begin:
    mov ah,9
    mov dx,offset String
```

```

int 21h

int 20h

String db 'My string.$'

CSEG ends
end Begin

```

Запустите вначале программу из листинга 10.4. После того как она завершит свою работу, запустите программу из листинга 10.5. Обратите внимание, что происходит. Затем запустите программу из листинга 10.5 в отладчике. Смело заходите "внутрь" прерывания 21h и обратите внимание на код. На рис. 10.6 отображено состояние программы test10.com, которая только что выполнила команду int 21h.

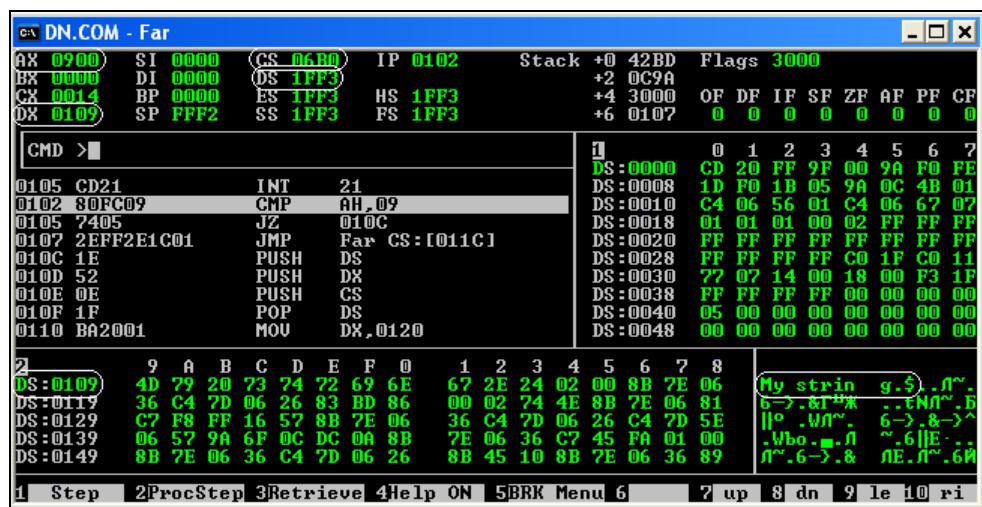


Рис. 10.6. Программа test10.com выполнила команду int 21h

Обратите внимание на текущую команду (выделенную светлым фоном) и следующие за ней. Это как раз и будет процедура обработки 21h-го прерывания Int_21h_proc из программы данной главы prog10.com, которая и заменяет пару регистров ds:dx. На рис. 10.7 приведен фрагмент кода, заменяющий адрес выводимой строки на экран.

На рис. 10.6 и 10.7 четко видно, что процедура int_21h_proc выполняет следующие действия:

- сохраняет изменяемые регистры;
- заменяет ds:dx, которые теперь содержат смещение строки "Моя строка!";
- вызывает оригинальный обработчик прерывания 21h, которое и выведет строку;
- восстанавливает сохраненные регистры;

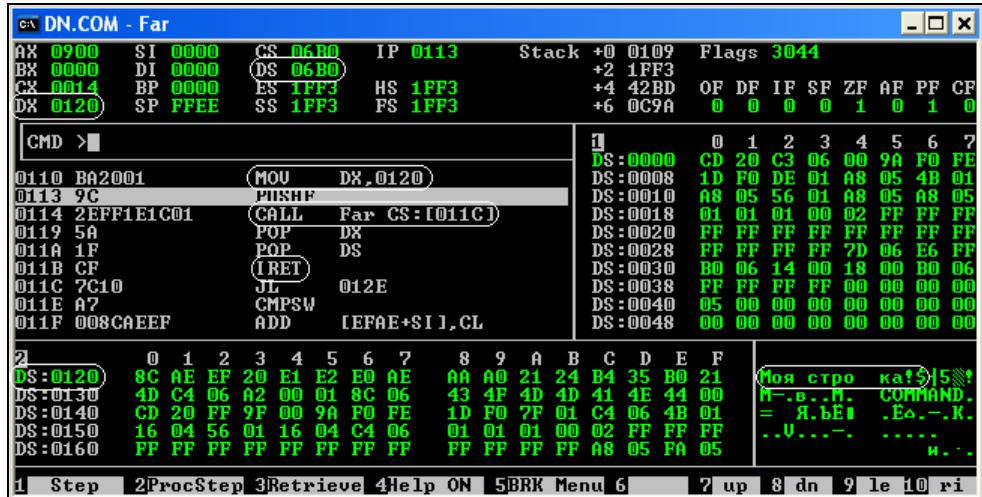


Рис. 10.7. Подмена выводимой строки

- выходит из обработчика прерывания, при этом оператор `iret` передаст управление на `test10.com`, сразу после команды `int 21h`.

Вот таким образом и будут фильтроваться вызовы прерывания `21h` *всех* программ. Именно так, с помощью резидентных программ, и можно контролировать почти все процессы, происходящие в компьютере.

* * *

В главе 11 начнем писать оболочку типа Norton Commander.



Глава 11

Управление видеоадаптером

Как вы уже заметили, писать программы на ассемблере не так-то быстро, как, например, на Бейсике. Вы уже знаете, сколько нужно напечатать символов для того, чтобы просто вывести строку на экран. На Бейсике это можно сделать так:

```
print "Строка"
```

Ассемблер — трудоемкий язык, но очень гибкий, мощный и интересный. Именно потому, что на нем довольно тяжело писать программы, у многих пропадает желание изучать его. Более того, те программисты, которые попробовали выучить ассемблер и не смогли ничего понять, начинают говорить, что ассемблер — устаревший язык, и никто на нем уже не пишет. Но зачем же писать сложные программы на чистом ассемблере, если можно его смешивать с языками высокого уровня, например, с Паскалем или С? Более того, знание ассемблера очень пригодится программисту в случае изучения кода других программ, вирусов, написания антивирусов и пр. Ведь исходный код скомпилированной программы можно посмотреть только в машинных кодах, которые дизассемблер и отладчик переводят на язык, понятный человеку, т. е. на ассемблер.

11.1. Оболочка

В данной главе начинаем рассматривать довольно большую тему: создание оболочки типа Norton Commander (Windows Commander, FAR Manager и т. п.) с поддержкой длинных имен файлов и возможностью использования XMS-памяти. Цель главы — не написать полноценную оболочку, а подвести вас к тому, чтобы вы ее написали сами. Мы будем рассматривать подпрограммы, прерывания, вывод на экран, рисование окон, команды 8086—80486 процессоров, сопроцессор, работу с файлами и дисками, XMS-память, клавиатуру и пр. После того как мы рассмотрим необходимые функции (команды, прерывания), вы сможете без труда написать собственную оболочку, которая будет не хуже, чем известнейший Norton Commander. Правда, работать она будет существенно быстрее и иметь меньший объем кода.

Наша оболочка будет выполнять только простейшие функции. Почему?

Дело в том, что ассемблерный код многофункциональной оболочки будет занимать более 300 Кбайт и разобраться в нем совсем непросто.

Из данного раздела вы также узнаете, как работают программы с файлами, что находится "внутри" FAR Manager, а также много чего интересного. Вы поймете, что написание программы — это большой и тяжелый, но очень интересный труд.

Для того чтобы написать хорошую оболочку на ассемблере, нам будет вполне достаточно СОМ-файла (т. е. размер оболочки не будет превышать 64 Кбайт). "Хватит ли нам?" — спросите вы. Безусловно. Вспомните предыдущие версии Volcov Commander, которые имеют формат СОМ и не превышают 64 Кбайт. Все зависит от того, насколько хорошо вы умеете писать, оптимизировать программы, сокращать код. Все это будем изучать.

Итак, рассмотрим основу нашей оболочки (листинг 11.1).

Листинг 11.1. "Скелет" нашей будущей оболочки

```
CSEG segment
assume cs:CSEG, ds:CSEG, es:CSEG, ss:CSEG
org 100h

Begin_shell:

; Проверим видеорежим
call Check_video

; Выведем сообщение-приветствие
mov ah,9
mov dx,offset Mess_about
int 21h

; Вызовем главную процедуру
call Main_proc

; Сюда мы вернемся, только когда пользователь решит выйти из программы.
int 20h

; Здесь будут идти процедуры в произвольном порядке
; Главная процедура, где будут происходить все действия
include main.asm

; Процедуры работы с дисплеем:
include display.asm

; Процедуры работы с файлами:
include files.asm

; Процедуры работы с клавиатурой:
include keyboard.asm
```

```
; Тексты, сообщения:  
include messages.asm  
  
; И другие. Со временем добавим...  
  
CSEG ends  
Begin_shell endp
```

Сохраните этот файл под именем Sshell11.asm (как это сделать — см. далее). Дабы не запутаться, давайте договоримся, что Sshell — это сокращение Super shell (так мы нескромно назовем нашу оболочку), а цифра указывает на главу, к которой относится этот файл (в данном случае — главу 11).

Вы видите здесь новый оператор (директиву): `include`. Эта директива именно программы-ассемблера (MASM/TASM): она не будет занимать места в ассемблированной программе. Директивой `include` очень удобно пользоваться в случае, если код программы довольно большой. Далее рассмотрим это на примере.

Что нужно сделать теперь? В каталоге, где будет находиться ассемблерный головной файл (приведен выше), необходимо создать еще пять файлов (пока что пустых, с нулевой длиной): main.asm, display.asm, files.asm, keyboard.asm, messages.asm.

Как создать и сохранить все необходимые ассемблерные файлы? Нажмите в оболочке (Far Manager, Norton Commander и т. п.) комбинацию клавиш `<Shift>+<F4>`. Введите соответствующее имя (перечислены выше). Нажмите клавишу `<Enter>`, а затем `<F2>` и `<Esc>`. Должно все получиться.

Если у вас нет архива файлов-приложений в готовом для ассемблирования DOS-формате, то вы можете скачать их в упакованном виде с сайта <http://www.Kalashnikoff.ru/>.

Итак, начало готово. Можно нашу оболочку ассемблировать. Обратите внимание, что среди файлов есть Sshell11.asm. Это наш головной файл. Его и нужно будет ассемблировать.

ВНИМАНИЕ!

НЕ пытайтесь ассемблировать файлы main.asm, display.asm, files.asm, keyboard.asm, messages.asm. Ничего не получится! Они сами автоматически ассемблируются при запуске ассемблера с такими параметрами (если у вас MASM 6.11):

ML.EXE Sshell11.asm /AT

или с такими (если у вас TASM):

TASM.EXE Sshell11.asm

TLINK.EXE Sshell11.obj /t /x

Более подробно об ассемблировании программ см. в приложении 1.

Должен получиться файл Sshell11.com. Запустите и посмотрите, как он работает. Затем внимательно изучите все файлы с расширением ASM. Обратите внимание на оформление программ, на отступы и т. д.

11.2. Управление видеокартой

За управление дисплеем и видеокартой отвечает прерывание 10h — набор функций BIOS (ПЗУ) (табл. 11.1).

Таблица 11.1. Функция 0Fh прерывания 10h: получать текущее состояние дисплея

Вход	Выход
ah = 0Fh	ah = текущий видеорежим ah = число текстовых колонок на экране bh = номер активной видеостраницы

Дисплей может работать в двух режимах: текстовом и графическом. В свою очередь, текстовые и графические режимы могут разделяться на подрежимы. Скажем, текстовый режим 40×25, текстовый режим 80×25 (наиболее удобный и часто используемый, например, в Norton Commander), графический режим четырехцветный (CGA) (в старых играх типа Digger), графический шестнадцатицветный и пр. В табл. 11.2 приведен список некоторых видеорежимов.

Таблица 11.2. Некоторые видеорежимы

№ режима	Текст/графика	Количество символов/точек
Режимы CGA+		
01h	Текст, 16 цветов	40 × 25
03h	Текст, 16 цветов	80 × 25
04h	Графика, 4 цвета	320 × 200
06h	Графика, 2 цвета	640 × 200
Режимы VGA+		
11h	Графика, 2 цвета	640 × 480
12h	Графика, 16 цветов	640 × 480
13h	Графика, 256 цветов	320 × 200

Существуют также дополнительные режимы sVGA с более высоким разрешением и количеством цветов (например, 1024×768, 64K цветов). Эти режимы используются только, если видеоадаптер поддерживает спецификацию (или стандарт) VBE (как правило, все современные компьютеры ее поддерживают). Дополнительную информацию о текстовых и графических режимах CGA-, EGA-адаптеров можно найти в программе helpasm.exe, которая доступна на сайте <http://www.Kalashnikoff.ru/>.

Следует дополнительно отметить, что в настоящий момент очень редко используются графические режимы CGA- и EGA-мониторов.

Вернемся к программе. Как видно из функции:

```
...
mov ah,0Fh
int 10h
cmp al,3
je Ok_video
...
```

мы проверяем, установлен ли текущий режим 3 (т. е. текстовый 80×25). Если так, то проверяем текущую видеостраницу. Если нет, то установим его функцией 0h прерывания 10h:

```
...
mov ax,3      ;Функция установки видеорежима
int 10h      ;Устанавливаем текстовый режим 80x25 и заодно чистим экран
...
```

При выполнении данной функции происходит очистка экрана, даже если текущий режим был 3.

Видеокарта, как правило, имеет достаточно памяти для размещения более одной видеостраницы (если у вас современный sVGA монитор, то можно смело утверждать, что ваша видеокарта имеет необходимый объем памяти для размещения 8 текстовых видеостраниц).

Для чего нужны дополнительные видеостраницы? Обычно нет необходимости переключаться на другие страницы (первую, вторую и т. д.). Обращение, как правило, всегда происходит к нулевой странице. Однако есть некоторые программы, которые используют первую и вторую видеостраницы и "забывают" переключаться на нулевую перед выходом. В нашей оболочке мы будем выводить все символы прямым отображением в видеобуфер на нулевую страницу (самый оптимальный алгоритм). И если текущей будет первая или вторая страница, то пользователь, естественно, не увидит ничего, что вывела наша оболочка. Чтобы такого не случилось, программисту следует проверить, установлена ли нулевая страница или нет. Кстати, многие оболочки этого не делают.

Здесь стоит еще добавить, что при выводе строки средствами MS-DOS символы выводятся на текущую видеостраницу в текущую позицию курсора. Однако при использовании метода прямого отображения в видеобуфер программист должен указать номер страницы, на которую следует выводить символы. Поэтому при вызове функции 09h прерывания 21h (вывод строки на экран средствами операционной системы) программисту не следует заботиться о том, какая видеостраница текущая. Операционная система сама определяет это.

Для выбора текущей видеостраницы используется функция 05h прерывания 10h (табл. 11.3).

Таблица 11.3. Функция 05h прерывания 10h:
установить текущую видеостраницу

Вход	Выход
<code>ah = 05h</code> <code>al = номер видеостраницы</code>	Ничего

Попробуйте поэкспериментировать с данными прерываниями.

Несколько дополнений:

- как правило, все страницы дисплея (кроме нулевой) чистые, если некоторая программа не заносила в них данные;
- при переключении режима дисплея с помощью функции 00h прерывания 10h очищается не только нулевая, но и все видеостраницы;
- смена видеостраницы происходит мгновенно. Это особенно полезно, если необходимо вывести на экран сложный и объемный рисунок. При этом можно, например, на текущую нулевую страницу вывести надпись вида "Подождите немного", на первой строить сам рисунок, а затем на нее переключиться. Создается эффект моментального вывода изображения на экран. Этот прием раньше часто использовали в старых играх.

* * *

На этом данная глава заканчивается. В главе 12 мы начнем рассматривать первую резидентную программу.



Глава 12

Повторная загрузка резидента

12.1. Резидент

В данной главе рассмотрим один из вариантов применения резидента. Но сперва изучим файлы-приложения `resid12.asm` и `test12.com`.

Загрузите файл `resid12.com` в память, а затем запустите `test12.com`.

`Resid12.com` оставляет в памяти процедуру, которая выводит на экран строку `ASCII` (`ASCII`-строка, заканчивающаяся символом 0). Причем процедура `Int_10h_Proc` перехватывает прерывание `10h` (это прерывание BIOS (ПЗУ)). Нечто похожее рассматривалось в *главе 10*. В данном случае мы добавляем еще одну функцию (`88h`) к прерыванию `10h`, которой, естественно, может воспользоваться любая другая программа, если будет знать, что такая функция существует. Для того чтобы вызвать функцию `88h`, нам нужно вызвать прерывание `10h`, а в регистры загрузить необходимые числа (данные). А именно:

`ah = 88h` — номер нашей функции;

`ds:si` = адрес строки, которую нужно вывести на экран (`ds` — сегмент, `si` — смещение).

Обратите еще раз внимание, что мы в `ds` (сегмент) ничего не загружаем, только в `si` (смещение). Заметьте, что при старте любого СОМ-файла сегментные регистры `cs`, `ds`, `es`, `ss` равны нашему единственному сегменту, в который загрузилась программа. Сегмент может быть любым. Все зависит от того, сколько резидентных программ уже загружено в память, каков объем ядра ОС и пр.

12.2. Проверка на повторную загрузку резидента

Первое, что делает программа, — переходит на метку инициализации. При инициализации обычно настраиваются необходимые регистры, перехватываются прерывания и пр. То есть происходит "подготовительный процесс" перед установкой резидентной программы.

Перво-наперво проверим, загружен ли уже наш резидент в память или нет. Как это сделать? Так как мы перехватили прерывание, то можем из него сделать "отклик". Вызываем прерывание `10h`, заносим в регистр `ax` "магическое число" `8899h`. Можно любое другое, главное — чтобы не конфликтовало с какой-нибудь функцией

данного прерывания. Понятно, что в ah заносится 88h, а в al — 99h. Функции 88h у прерывания 10h не существует (не дошли пока разработчики до этого числа). Это можно с уверенностью предположить на 99,9%. Так как номер прерывания 10h еще не задействован, то произойдет немедленный выход из прерывания 10h (регистры, как правило, не меняются), что легко проверить в отладчике. Следовательно, вызвав прерывание 10h с числом 8899h в ah и получив также ответ 8899h в том же ah, мы уверены, что наш резидент еще не загружен. На рис. 12.1 изображено окно программы в состоянии после вызова прерывания 10h и получения отклика от резидента.

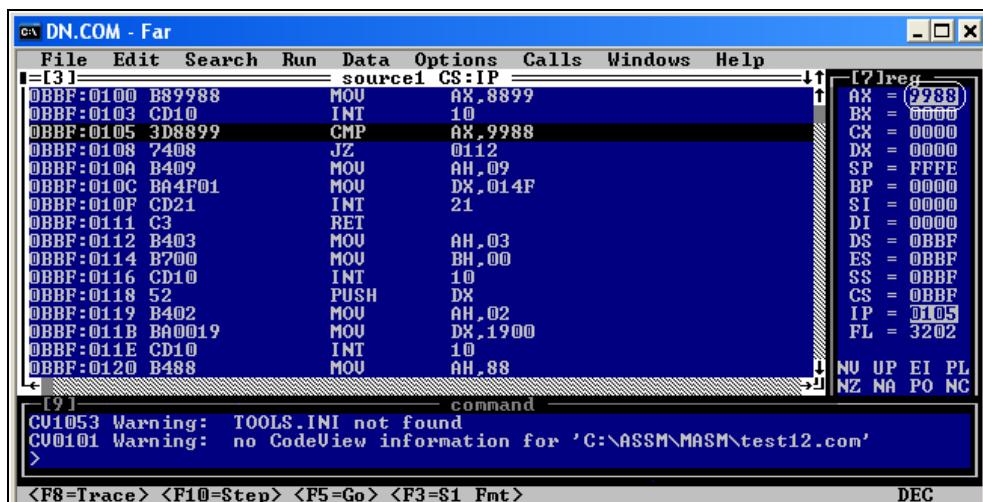


Рис. 12.1. Проверка на присутствие резидента в памяти: отзыв получен!

Теперь о нашем обработчике прерывания 10h. Если файл resid12.com уже находится в памяти, перехватив прерывание 10h, то, процедура обработки прежде всего должна проверить, вызывают ли ее с числом 8899h в ah. Если так, то нужно будет послать какой-нибудь ответ. Какой? В нашем примере мы меняем местами содержимое регистров ah и al и немедленно выходим из прерывания. Проще говоря, возвращаемся в нашу программу, которая, в свою очередь, проверяет, вернулось ли число 9988h. Здесь важно запомнить, что если нашего резидента нет в памяти, то вернется 8899h в ah, причем BIOS регистры не изменит, т. к. такой функции просто не существует. Если же в ah вернулось 9988h, то, значит, наша программа уже загружена. То есть процедура Int_10h_proc поменяла местами регистры (а что еще может это сделать?). Обратите внимание на первые строки процедуры, которые и меняют местами ah и al (листинг 12.1).

Листинг 12.1. Часть обработчика прерывания 10h программы resid12.com

```
...
Int_10h_proc proc
```

```

pushf           ;Сохраним флаги в стеке, т. к. они поменяются...

cmp ax,8899h   ;Проверим на повторную загрузку в память
jne Next_test  ;Если не проверка, то смотрим дальше...

;Меняем местами ah и al — признак того, что мы в памяти,
;что-то вроде ответного сигнала.
xchg ah,al

popf            ;Выровняем стек
iret             ;Выйдем из прерывания (вернемся в нашу программу)
;ах при возврате будет равен 9988h!!!
...

```

Познакомимся с новым оператором `xchg` (табл. 12.1 и листинг 12.2).

Таблица 12.1. Оператор `xchg`

Команда	Перевод	Назначение	Процессор
<code>xchg источник, приемник</code>	Exchange — обменять	Обмен регистров	8086

Листинг 12.2. Использование оператора `xchg`

```

...
mov ax,10h
mov bx,15h
xchg ax,bx      ;Теперь ax=15h, bx=10h
...

```

Как вам уже известно, регистр `cs` (от англ. *code segment* — сегмент кода) всегда содержит номер сегмента, в котором находится наша программа, а `ip` (от англ. *instruction pointer* — указатель инструкций) — смещение. Допустим, процедура обработки прерывания `10h` расположена по адресу `0010:0400h`, а наша программа загрузилась в сегмент `1234h`. Тогда получаем:

```

;cs:ip = 1234:0100h
...
[1234:0100h] mov ax,8899h
;После выполнения данной инструкции cs:ip=1234:0103h

[1234:0103h] int 10h
;Теперь cs:ip = сегменту/смещению адреса (вектора) прерывания 10h,
;т. е. 0010:0400h

[1234:0105h] mov bx,10
;Работаем дальше после того, как прерывание завершило свою работу...
...

```

Выход из прерывания осуществляется с помощью команды `iret`, в отличие от выхода из ближней процедуры — `ret`. Это видно из приведенного выше примера обработчика прерывания 10h.

Очень просто все выглядит на практике. Попробуйте запустить наш резидент (resid12.com) в отладчике. Внимательно следите за состоянием пары регистров `cs:ip`. Вы заметите, что они постоянно меняются. Обратите особое внимание на значения, которые они принимают при вызове прерываний (для этого нужно зайти внутрь прерываний; в AFD — <F1>, в CodeView — <F8>) (рис. 12.2).

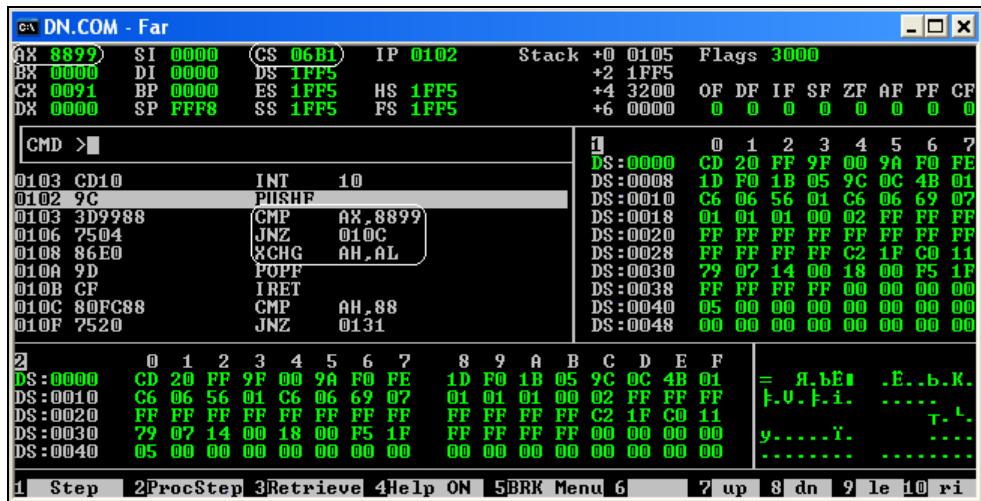


Рис. 12.2. Резидент посыпает отклик

12.3. Команды работы со строками

Рассмотрим некоторые новые инструкции, встречающиеся в нашем резиденте: `stos`, `lod`, `rep`. Это очень мощные и быстрые команды ассемблера. Они предназначены для работы с массивами данных (строки, данные любого типа, числа и пр.).

Инструкция `lod` загружает в регистр `ax/al` число, которое находится по адресу, указанному в регистрах `ds:si`, при этом `si` автоматически увеличивается на единицу или на двойку.

Почему мы написали `ax/al`? Дело в том, что эта инструкция имеет две разновидности: `lodsb` и `lodsw`. `lodsb` (`b` — byte, байт) загружает в `al`, а `lodsw` (`w` — word, слово) — в `ax` (листинг 12.3, рис. 12.3 и 12.4).

Листинг 12.3. Использование операторов `lodsb` и `lodsw`

```
...
mov si,offset String ; si указывает на начало String, т. е. на '1'
lodsb
```

;теперь в al символ '1' (31h); si = si + 1, т. е. si теперь указывает на '2'

lodsw

;теперь ax содержит '32' (3332h); si = si + 2, si указывает на '45'

...

String db '12345'

...

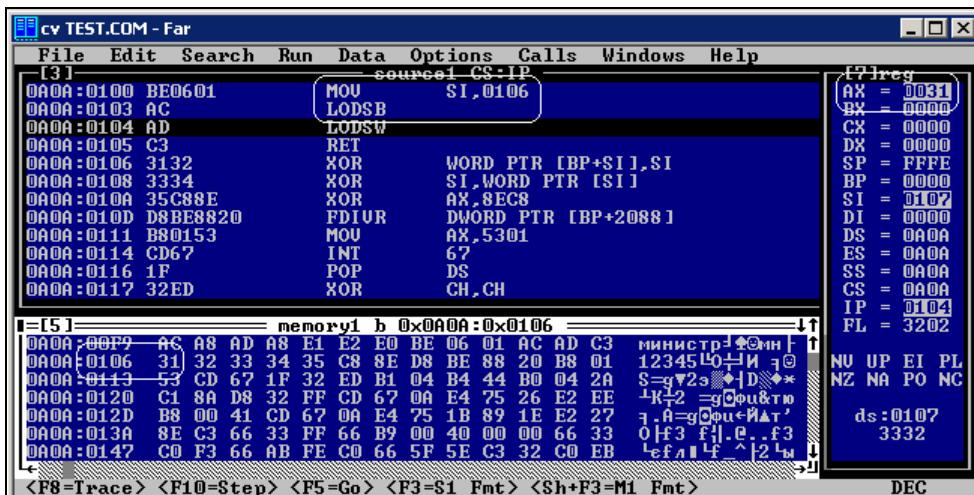


Рис. 12.3. Результат выполнения команды lodsb

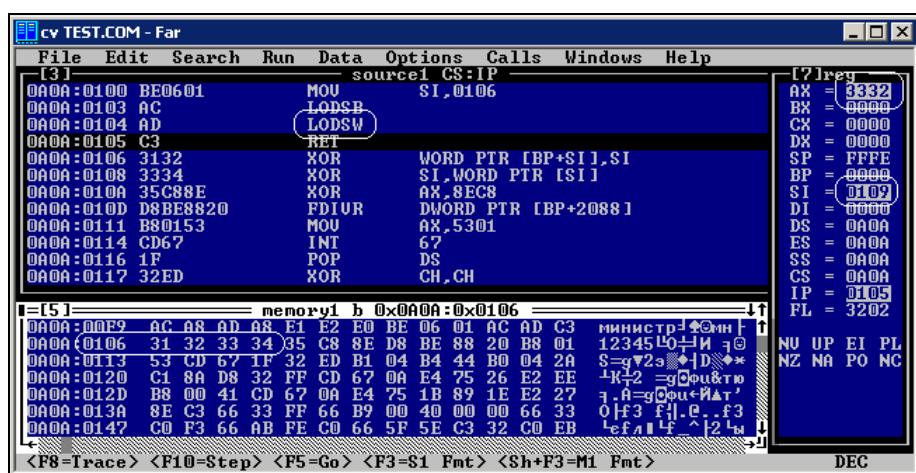


Рис. 12.4. Результат выполнения команды lodsw

Принцип таков: если последний символ в инструкции `lod`s — `b`, то загружается один байт в `al`, и `si` увеличивается на 1. Если же последний символ `w`, то загружаются два байта (слово) в `ax`, и `si` увеличивается на два. Здесь могут возникнуть два вопроса:

1. Почему не загружаем ничего в `ds`?
2. Почему после команды `lodsw` в `ax` содержится '32', а не '23', что было бы вполне логично?

ПРИМЕЧАНИЕ

Когда число указывается в кавычках ('32'), то это значит, что оно занимает 2 байта. Первый байт занимает число 3, а второй — 2.

Ответ на первый вопрос должен быть понятен: если `String` находится в том же сегменте, в котором наша программа, и сегментный регистр не менялся в процессе работы программы, то в него ничего загружать не надо. В нем и так находится нужный сегмент. Ведь при запуске СОМ-программы все сегментные регистры равны нашему единственному сегменту.

Ответ на второй вопрос заслуживает подробного рассмотрения.

Дело в том, что в компьютере двух- и более байтовые числа хранятся в обратном порядке. Путаница не возникает, если к двухбайтовому числу мы обращаемся как к двухбайтовому, а к четырехбайтовому (к двойному слову) обращаемся как к четырехбайтовому. В листинге 12.4 приведены примеры.

Листинг 12.4. Загрузка чисел в переменные

```
...
Handle dw 1234h
;Изначально присваиваем переменной значение 1234h. В памяти это число
;расположится в таком порядке: 3412h. Проверьте в отладчике...
...
mov ax,Handle           ;ax=1234h
mov al,byte ptr Handle ;al=34h
mov al,byte ptr Handle+1 ;al=12h
...
```

Byte `ptr`, как вы уже знаете, обозначает, что мы хотим загрузить 1 байт с переменной двухбайтового типа: `Handle dw 1234h` (`dw` от англ. *define word* — определить слово (два байта)) — рис. 12.5—12.7.

Исходя из вышесказанного, рассмотрим, почему мы при сохранении вектора прерывания `10h` заносим вначале смещение, а затем сегмент, хотя логичнее было бы наоборот (листинг 12.5).

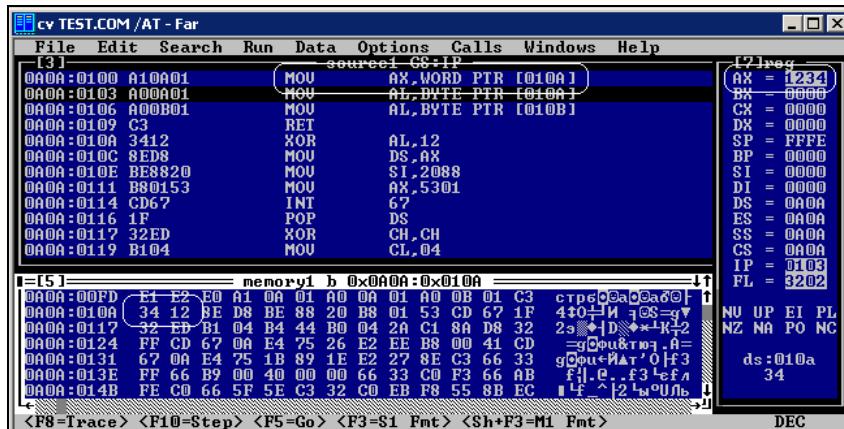


Рис. 12.5. Загрузка из переменной слова в регистр ах

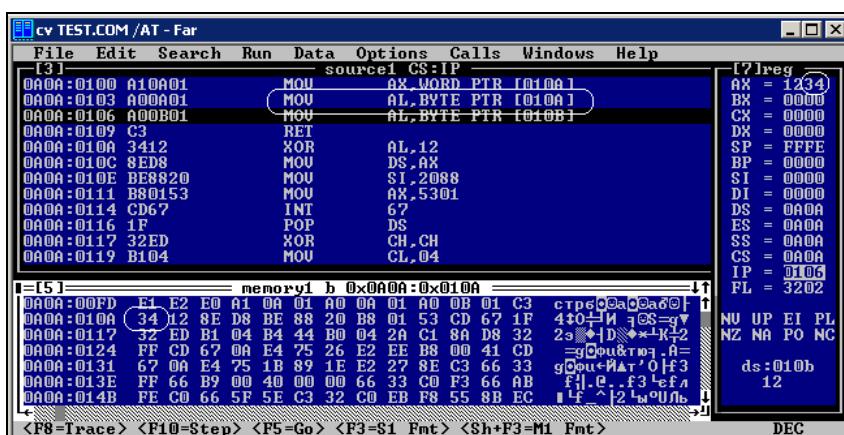


Рис. 12.6. Загрузка из переменной байта в регистр а1

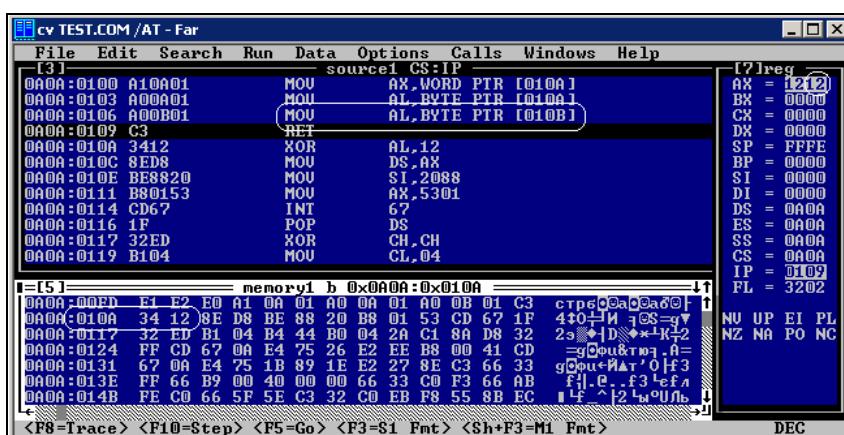


Рис. 12.7. Загрузка из переменной байта в регистр а1

Листинг 12.5. Сохранение адреса вектора прерывания в переменной

```

...
mov ax,3510h      ;получим адрес (вектор) прерывания 10h.
int 21h          ;теперь es содержит сегмент, а bx — смещение...

;сохраним сперва смещение
mov word ptr Int_10h_vect,bx
;a затем сегмент, учитывая, что данные хранятся в обратном порядке
mov word ptr Int_10h_vect+2,es
...
;переменная для хранения двух слов (четыре байта)
Int_10h_vect dd ?
...

```

word ptr указывает на то, что нужно занести слово в переменную Int_10h_vect. Обратите внимание, что данная переменная имеет тип dd (от англ. *define double word* — определить двойное слово). Но мы-то заносим одно слово (es или bx)! Для этого и указываем ассемблеру на то, что заносится слово в переменную, которая может хранить двойное слово.

Итак, команда lodsb загружает в al однобайтовое число, находящееся по адресу ds:si (сегмент:смещение). В принципе, данная команда аналогична следующей паре инструкций:

```

...
mov al,ds:[si]
inc si           ;(или add si,1)
...

```

Только работает она гораздо быстрее, да и занимает меньше байтов. По аналогии: команда lodsw загружает в ax двухбайтовое число, расположенное также по адресу ds:si. Она эквивалентна паре инструкций:

```

...
mov ax,ds:[si]
add si,2
...

```

Как правило, подобные команды (lodsb/lodsw) применяются при работе в цикле для чтения значений из строки (или другого массива данных). *Массив данных* — цепочка символов (байтов), расположенных последовательно друг за другом. Например, следующая строка является своего рода массивом данных:

Data_array db 'Это массив данных'

Рассмотрим еще пару подобных команд, которые используются в нашем резиденте: stosb/stosw. stosb заносит однобайтовое число из al по адресу es:di, а stosw — двухбайтовое число по тому же адресу (листинг 12.6, рис. 12.8).

Листинг 12.6. Использование команды stosw

```
...
mov di,offset Data_array
mov ax,2030h
stosw
...
Data_array dw ?
;теперь в этой переменной находится число 2030h, что равносильно
;команде: mov Data_array,2030h
...

```

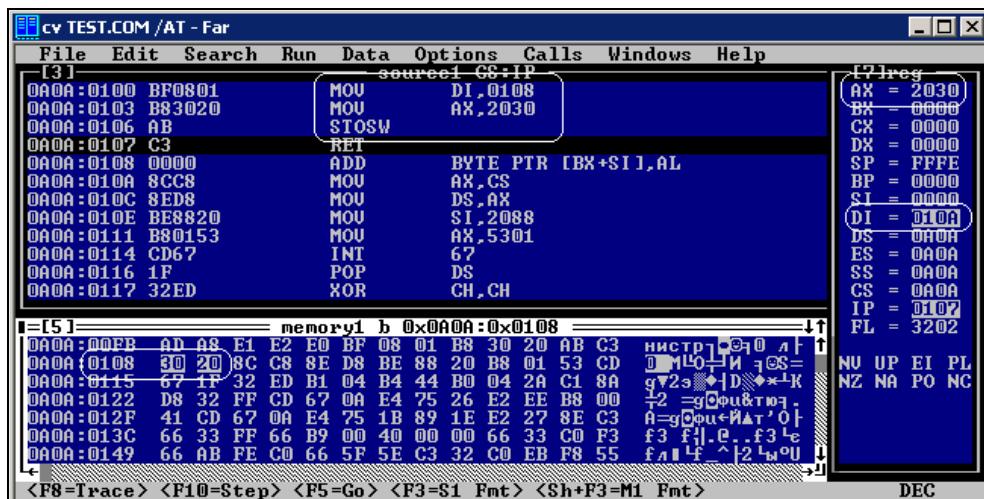


Рис. 12.8. Результат выполнения команды stosw

Например, занесем в левый верхний угол экрана нашу "рожицу", используя данные команды. Вот как это выглядит (листинг 12.7).

Листинг 12.7. Вывод "рожицы" на экран с помощью оператора stosw

```

...
mov ax,0B800h
mov es,ax
mov di,0
mov ah,07h      ;атрибут символа (белый на черном)
mov al,01h      ;(сам символ - "рожица")
stows           ;заносим, что эквивалентно: mov es:[di],ax
...

```

Как мы знаем, видеобуфер имеет следующую структуру:

СИМВОЛ : АТРИБУТ СИМВОЛ : АТРИБУТ ...

Обратите внимание на приведенный выше пример. Вы видите, что в ah мы загружаем атрибут, а в al — символ. Получается, что сперва заносится атрибут (т. к. ah — старшая (левая) половинка), а затем символ, что и доказывает расположение данных в компьютере в обратном порядке. Получится, что при переносе ax в сегмент видеобуфера символ и атрибут поменяются местами!

Рассмотрим префикс rep, который часто используется вместе со строковыми командами. Допустим, нам надо очистить экран. Это можно сделать с помощью команды stosw, просто забив экран пробелами (листинг 12.8).

Листинг 12.8. Очистка экрана с помощью loop

```
...
mov ax,0B800h
mov es,ax
mov di,0          ;как обычно, с левого верхнего угла
mov cx,2000       ;80x25=2000 символов на экране
mov ax,0720h      ;07 — атрибут, 20h — пробел

Next_sym:         ;заносим посимвольно
stosw
loop Next_sym
;теперь экран чистый
...
```

Выглядит вроде и нормально, хотя этот пример имеет ряд недостатков:

1. Использование loop вместо префикса rep существенно замедляет работу программы.
2. Код программы больше, чем при использовании того же rep.
3. Необходимо заводить метку (в нашем случае — Next_sym).

Попробуем реализовать такой же алгоритм, но с использованием префикса rep (листинг 12.9).

Листинг 12.9. Очистка экрана с помощью rep

```
...
mov ax,0B800h
mov es,ax
mov di,0
mov cx,2000      ;cx — counter, счетчик
mov ax,0720h      ;атрибуты/символ
rep stosw        ;выводим пробел столько раз, сколько указано в cx.
...
```

Все, экран чистый. В данном случае очистка экрана происходит мгновенно. Этот процесс не заметен даже на компьютерах IBM PC/XT. После выполнения последней команды — `rep stosw` — `cx` будет содержать 0.

ВНИМАНИЕ!

Оператор `rep` (от англ. *repeat* — повтор) неотделим от `cx`, как и `loop` от `cx`!

Подведем итог:

- одна команда `stosw` записывает только два символа, находящихся в `ax` по адресу `es:di`;
- команда `rep stosw` запишет два символа, находящихся в `ax` по адресу `es:di` столько раз, сколько находится в регистре `cx`;
- строковые команды предпочтительней использовать в паре с `rep`, где это возможно.

Все просто. При исследовании работы строковых команд желательно воспользоваться отладчиком и следить за изменением регистров `cx`, `si`, `di`.

12.4. Использование `xor` и `sub` для быстрого обнуления регистров

Рассмотрим одну маленькую хитрость, которую часто применяют программисты на ассемблере: использование команд `xor` и `sub` для быстрого обнуления регистров (рис. 12.9):

```
xor ax,ax      ; равносильно mov ax,0
```

А можно и так (рис. 12.10):

```
sub ax,ax      ; вычтем из ax ax
```

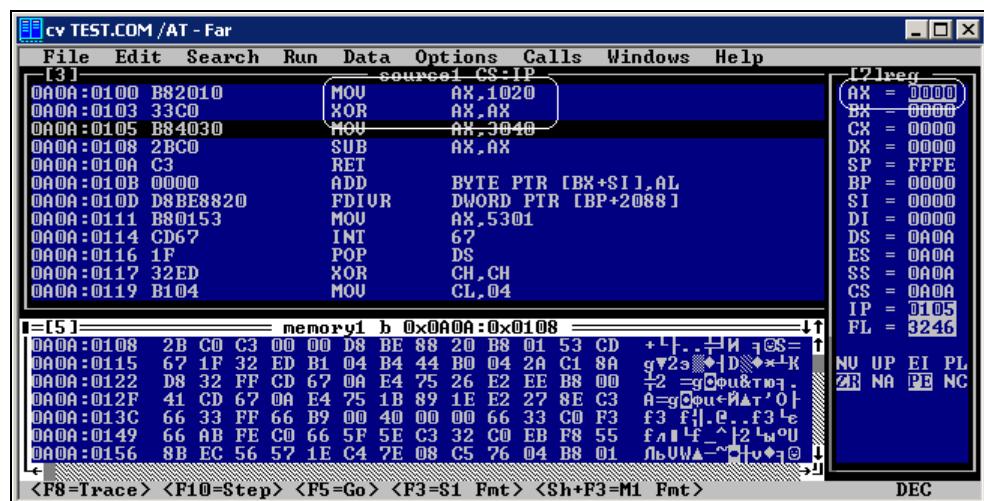


Рис. 12.9. Обнуление регистра с помощью команды `xor`

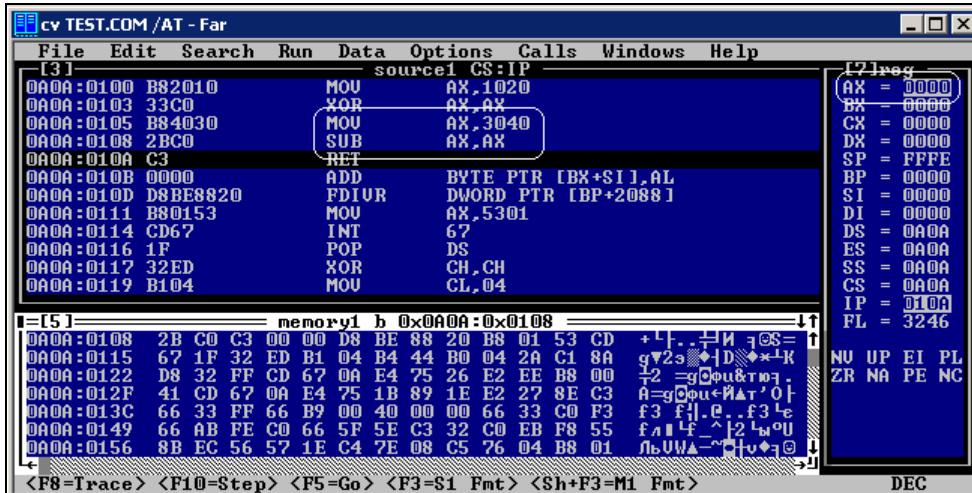


Рис. 12.10. Обнуление регистра с помощью команды sub

Эти команды (`xor ax,ax`/`sub ax,ax`) выполняются быстрее команд типа `mov ax, 0`. Поэтому не удивляйтесь, если где-то в наших примерах или в других программах встретите нечто подобное.

`xor` — исключающее ИЛИ. Это логическая команда. Подобные инструкции мы рассмотрим позже. Пока вам нужно уяснить, что такими операторами, как `xor` и `sub`, можно быстро и просто аннулировать регистры.

12.5. Задание для освоения информации из данной главы

Досконально исследуйте программы `resid12.asm` и `test12.asm` в отладчике (лучше в AFD): это позволит вам понять принцип работы резидентных программ и строковых команд.



Глава 13

Поиск и считывание файлов: вирус

Эта глава небольшая, но информативная. Из нее вы узнаете, как поместить вирус временно в память, как передать ему управление, как найти очередной "файл-жертву" и многое другое.

13.1. Теория

В данной главе рассмотрим следующее:

- как вирус "прикрепляется" к файлу, не нарушая его работоспособности;
- что должен сделать вирус в первую очередь;
- как передать управление вирусу из чужой программы.

Первое, что хотелось бы отметить, — это то, что наш вирус будет заражать только СОМ-файлы.

Как вы уже знаете, файлы типа СОМ загружаются в первый свободный сегмент, и их код располагается по смещению 100h. Следовательно, нам нужно будет сохранить в теле нашего вируса первые три байта "файла-жертвы", записать вирус в "хвост" файла и, вместо сохраненных трех байтов, установить команду jmp на начало кода нашего вируса (т. е. передать ему первому управление).

После того как вирус отработает, следует восстановить сохраненные три байта (записать их в память по адресу 100h) и передать им управление. На практике все будет понятно. Получается примерно так, как представлено в листингах 13.1 и 13.2.

Листинг 13.1. Программа до заражения (следите за адресами)

```
; код программы до заражения, расположенный по адресу 100h
[1234:0100h] mov ax,34
[1234:0103h] mov dx,15      ;здесь может быть все, что угодно...
[1234:0106h] add ax,bx
[1234:0108h] ...и т. д...
...
[1234:0500h] int 20h      ;последний байт программы
```

```
[1234:0502h] ---
;далее идет уже память, не занятая под код или данные "файла-жертвы"...
...
```

Листинг 13.2. Программа после заражения

```
; "прыгаем" на начало нашего вируса (заменили байты здесь)
[1234:0100h] jmp 0502h

[1234:0103h] mov dx,15      ;а это байты "файла-жертвы"
[1234:0106h] add ax,bx
...
[1234:0500h] int 20h

[1234:0502h] ---          ;а здесь уже начинается код нашего вируса.
...
;здесь тело вируса. Делаем, что хотим...

;== После того, как вирус отработал ==
;Восстановим первые два байта "файла-жертвы"
[1234:0700h] mov word ptr cs:[0100h], First_bytes_1

;Восстановим третий байт "файла-жертвы"...
[1234:0705h] mov byte ptr cs:[0102h], First_bytes_2

;Теперь по адресу 100h первые два байта не jmp 502h, а mov ax,34
;(т. е. оригинальный код программы). Вспоминаем из прошлых глав о том,
;что в ассемблере можно менять код программы "на лету"...
;Все это меняется только в памяти, а не на диске!
```

```
;...перейдем по адресу 100h, т. е. передадим управление программе
[1234:0709h] jmp 0100h
...
```

Сравните два участка кода: незараженной программы и зараженной. Теоретически вопросов возникнуть не должно...

13.2. Практика

Теперь приступим к практической части. Откройте файл virus13.asm. Первая строка — .286 — указывает ассемблеру, что будем использовать инструкции (команды, операторы) не только процессора 8086, но и 80286. То есть на компьютере 8086 наш вирус уже работать не будет! С первого же байта перейдем на метку `Init` (инициализации нашего вируса).

Сразу возникает проблема: при поиске файла функцией DOS мы затираем DTA "программы-жертвы". И тут же возникает вопрос: что такое DTA и для чего оно нужно?

Полную информацию об этом можно найти в программе HELPASSM, которую рекомендуется скачать с нашего сайта <http://Kalashnikoff.ru>. В этой главе мы рассмотрим только то, что нас интересует для написания вируса.

Как вы помните, все COM-программы начинаются с адреса 100h (org 100h). Что же находится в памяти от 0 до 100h? Там расположен PSP (Program Segment Prefix, префикс программного сегмента), который создается автоматически операционной системой после загрузки программы в память, но до передачи ей управления.

По адресу 80h находится по умолчанию *DTA* (Disk Transfer Area, область обмена дисковыми данными). В DTA записывается информация в процессе выполнения функций поиска файлов (4Eh и 4Fh прерывания 21h). Все вроде бы и ничего, но проблема в том, что по этому адресу (80h) располагается изначально командная строка. Например:

```
FORMAT.COM C:/S/U
```

По адресу cs:0080h будет находиться *L_C:/S/U*, где *L* — длина командной строки, а *_* — символ пробела. Для того чтобы удостовериться, запустите отладчик CodeView следующим образом:

```
CV.EXE FORMAT C:/S/U
```

Затем посмотрите, что будет находиться по адресу cs:0080h (рис. 13.1).

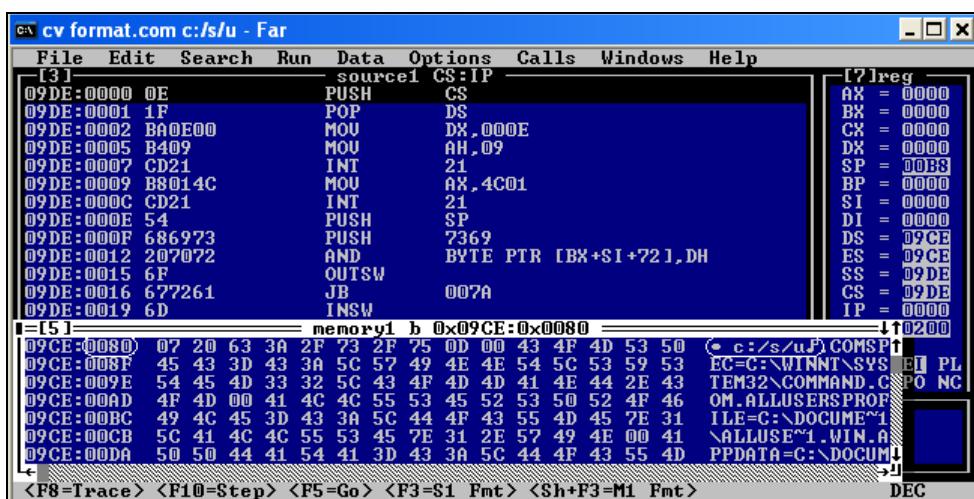


Рис. 13.1. Командная строка в PSP

Для чего мы рассматриваем PSP? Дело в том, что когда мы попробуем искать первый файл, то затрем командную строку (*L_C:/S/U*). Получается, что "программа-жертва", к которой мы "подцепились", не сможет прочитать те параметры,

которые ей передал пользователь. В данном случае — это L/_S/U. Существуют два способа обойти это:

- сохранить PSP программы перед поиском файла. А затем, когда наш вирус отработал, восстановить его;
- установить DTA на другую область памяти, а затем восстановить его. Это позволяет сделать функция 1Ah прерывания 21h (табл. 13.1).

Таблица 13.1. Функция 1Ah прерывания 21h: изменить адрес DTA

Вход	Выход
ah = 1Ah ds : dx = адрес DTA для установки	Ничего

Мы выберем второй способ, т. к. он гораздо проще и не требует дополнительных затрат на перенос области DTA в другое место памяти и дальнейшего его восстановления.

Однако возникает одна проблема: код нашей программы "теряется в адресах". То есть мы занесем наш сассемблированный код в конец программы, при этом смещения все поменяются. Например:

```
mov dx, offset String
```

Ассемблер занесет в dx смещение строки `String` в памяти. Фактически — после ассемблирования — это будет выглядеть так:

```
mov dx,125h
```

Какое именно число загружаем в регистр dx — не важно. Главное, что в этом регистре будет находиться смещение (адрес) строки в памяти, отсчитываемое от 0. Но мы-то запишем код нашего вируса в конец программы, включая все строки и прочие области данных! Получается, что строка в памяти будет находиться по одному адресу, а в регистры будет загружаться совсем другой адрес!

В листинге 13.3 приведен полный пример с указанием адресов сегментов и смещений, в который может загрузиться наша программа.

Листинг 13.3. Расположение строки в памяти

```
...
[1234:0100h] mov dx,400h
; В неассемблированном варианте это выглядит, как mov dx,offset String.
; То есть ассемблер заменит offset String на адрес (смещение) этой строки
; в памяти, начиная с нуля.
...
[1234:0400h] 'Строка'
; А вот и строка, которая расположится по адресу 400h. Она может находиться
; и по любому другому адресу, но сути это не меняет.
...

```

Теперь представим, что "файл-жертва" занимает 100h байт, а мы записываем наш код в конец файла. Получается, что строка будет находиться по такому адресу: 1234:0400h + 100h = 1234:0500h. Хорошо было бы, если бы все файлы имели одинаковую длину. Но один файл может занимать 100 байт, а другой 23 000 байт! В итоге, обращаясь к своей строке в зараженной программе, мы получаем следующее (листинг 13.4).

Листинг 13.4. Перенос строки на 100h дальше в процессе заражения

```
...
;1234:0200h потому, что 100h байт занимает "файл-жертва", а мы у него
;в "хвосте" ...
[1234:0200h] mov dx, 0400h
;Код "файла-жертвы"
...
[1234:0400h] --- ;Здесь все еще расположен код "файла-жертвы"...
...
[1234:0500h] 'Строка' ;Вот, где будет находиться строка!
...

```

Можно, конечно, перед заражением получить длину "файла-жертвы" и затем заменить код `mov dx, 400h` на `mov dx, 500h` напрямую, в процессе заражения программы. Но что делать, если таких ссылок много? Представляете, до каких размеров разрастется наш вирус? Мы поступим иначе: просто возьмем и перенесем вирус (и только вирус!) с "хвоста" "файла-жертвы" в свободный сегмент со смещением 100h. В листингах 13.5 и 13.6 показано, что получится (обратите внимание на сегменты и смещения в квадратных скобках).

Листинг 13.5. До перемещения

```
...
;Код "файла-жертвы"...
...
[1234:0200h] mov dx, 400h ;мы в "хвосте" программы
...
[1234:0500h] 'Строка'
...
```

Листинг 13.6. После перемещения

```
...
;перебросили код вируса в сегмент 5678h, по смещению 0100h
[5678:0100h] mov dx, 400h
...
[5678:0400h] 'Строка' ;строка встала на свое место (смещение) !
...
```

Еще вопрос: где гарантия того, что в сегменте, куда будем перемещать код вируса, не будет находиться код или данные другой программы?

Предлагаем временно (т. е. на тот момент, пока работает вирус) переслать код нашей программы-вируса в адрес 7-й страницы дисплея (отсчет с нуля!). Видеокарта имеет достаточно памяти для размещения восьми видеостраниц в текстовом режиме 80×25 символов (режим 03). Эти страницы, кроме нулевой, почти никогда не используются программами. Более того, известны точные сегменты этих страниц (смещения в них всегда нулевые для первого символа). Перечень этих сегментов приведен в табл. 13.2.

Таблица 13.2. Адреса видеостраниц

Сегмент	№ видеостраницы
0B800h	00
0B900h	01
0BA00h	02
0BB00h	03
0BC00h	04
0BD00h	05
0BE00h	06
0BF00h	07

Теперь давайте посчитаем размер одной видеостраницы. Хватит ли нам места для того, чтобы разместить на ней код вируса?

Наш вирус будет занимать не более 300—400 байт. Возьмем режим 03: в одной строке 80 символов, строк на экране 25. Один символ занимает 2 байта (атрибут/смещение). Получаем: $80 \times 25 \times 2 = 4000$ байт (точнее, 4096 байт — идет выравнивание на границу). Хватит ли нам этого? Конечно, хватит! Даже, если бы не хватало, мы могли бы использовать две, три, четыре страницы.

Сразу же за меткой `Init` пересылаем код вируса в область седьмой видеостраницы. Код нашего вируса на экране не будет отображаться, т. к. обычно текущей бывает нулевая страница. Хотя можете проверить, какая страница текущая перед перемещением...

Для перемещения кода вируса очень удобно воспользоваться еще одной строковой командой: `movs`.

13.3. Команда пересылки данных `movs`

Оператор `movs` предназначен для работы с массивами данных. По аналогии с уже рассмотренными нами подобными операторами, вам не составит труда разобраться с новым оператором. Принцип его работы полностью соответствует команде `stos`, которую мы рассматривали в предыдущих главах. Его описание приведено в табл. 13.3.

Таблица 13.3. Оператор movs

Команда	Перевод	Назначение	Процессор
movs	Move string — скопировать строку	Копирование строки (массива)	8086

При этом ds:si указывает на то, откуда брать данные, es:di куда их копировать, а cx — количество пересылаемых байтов/слов (листинги 13.7, 13.8 и рис. 13.2, 13.3).

Листинг 13.7. Пример использования оператора movs

```
...
mov cx, 10          ;количество пересылаемых байтов
mov si, offset Str1 ;откуда будем брать
mov di, offset Str2 ;куда копировать
rep movsb           ;пересылаем побайтно, т. к. movsb.

;Теперь Str1 = Str2

...
Str1 db '0123456789'
Str2 db '9876543210'
...
```

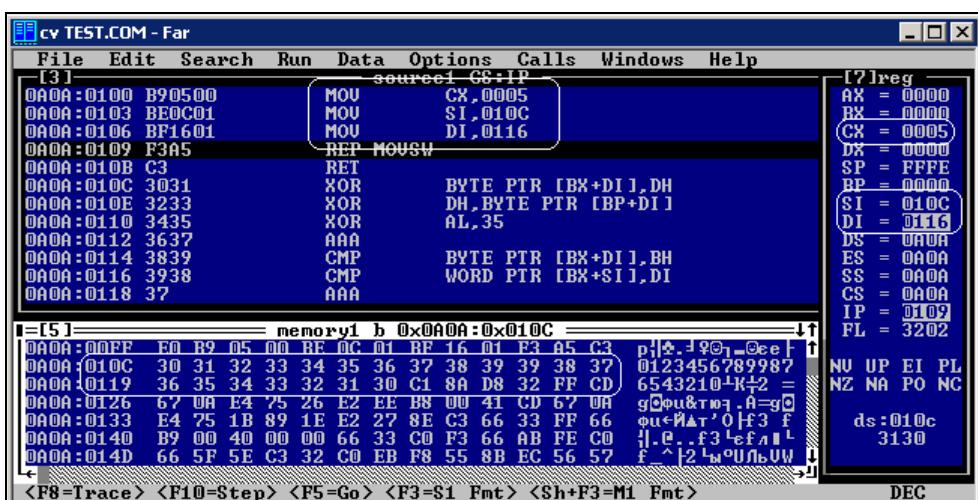


Рис. 13.2. Состояние регистров и переменных перед копированием

Листинг 13.8. Другой пример использования оператора movs

```
...
mov cx,5          ;количество пересылаемых слов (два байта)
mov si,offset Str1 ;откуда будем брать
mov di,offset Str2 ;куда копировать
rep movsw         ;пересыпаем пословно (по два байта), т. к. movsw.

;Теперь Str1 = Str2
...
Str1 db '0123456789'
Str2 db '9876543210'
...
```

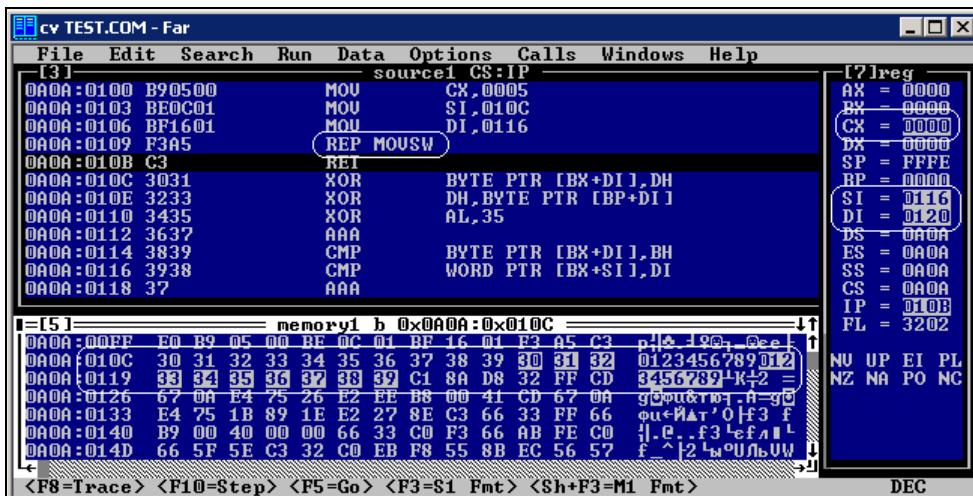


Рис. 13.3. Результат выполнения команды movs

13.4. Передача управления программе, расположенной в другом сегменте

Итак, теперь код нашего вируса расположен в двух местах в памяти:

- сразу за "программой-жертвой";
- в области 7-й страницы (0BF00:0100h).

Нам осталось "прыгнуть" на адрес 0BF00:ip. Как известно, cs:ip всегда показывают текущую операцию (адрес текущей операции). Обратите внимание, как мы "прыгаем":

```
jmp dword ptr cs:[Off_move]
```

Посмотрите, что содержит переменная `Off_move`, а также посмотрите в отладчике, что будет происходить с регистрами `cs:ip`.

Остается только добавить, что, начиная с метки `Lab_jmp`, наша программа работает уже в области 7-й видеостраницы. На рис. 13.4 показано окно отладчика AFD с программой `virus13.com` после выполнения приведенной выше команды.

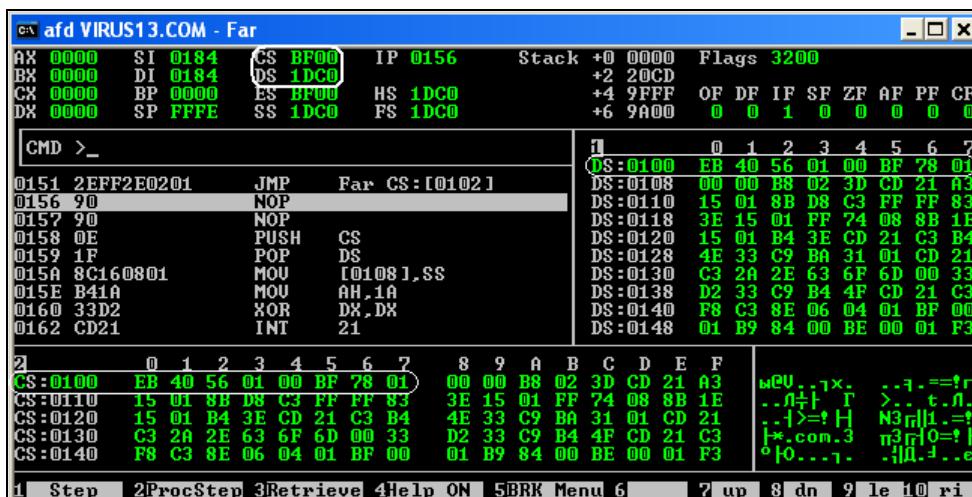


Рис. 13.4. Virus13.com в двух местах памяти

13.5. Поиск файлов

Установим DTA в область 7-й видеостраницы со смещения 0. Сюда будет записываться информация о найденных файлах для заражения. Теперь можно попробовать найти первый файл с расширением СОМ в текущем каталоге. Для этого используется функция `4Eh` прерывания `21h` (табл. 13.4).

Таблица 13.4. Функция `4Eh` прерывания `21h`: поиск первого файла/каталога

Вход	Выход
<p><code>ah = 4Eh</code></p> <p><code>cx</code> = атрибуты для поиска файла: у нас — 0 (обычные, не <code>read-only</code> и <code>hidden!</code>)</p> <p><code>ds:dx</code> = маска для поиска.</p> <p>Например:</p> <p><code>*.*</code></p> <p><code>*.exe</code></p> <p><code>comm????.c??</code></p> <p><code>c:*.asm</code></p>	<p><code>cf = 1</code>: нет такого файла (или ошибка)</p> <p><code>cf = 0</code>: DTA заполняется информацией о найденном файле</p>

`cf` — это флаг переноса. Если мы пишем `cf=1`, то это значит, что флаг переноса установлен (равен 1), а если `cf=0`, то сброшен (равен 0). Флаг переноса используется DOS, как правило, для индикации ошибки после выполнения функции. В данном случае, если функция `4Eh` установила флаг переноса, то это значит, что файлы, удовлетворяющие условию (маске поиска), не были найдены. Если флаг переноса сброшен (равен нулю), то в DTA заносится информация о файле. Ее мы рассмотрим в последующих главах. Если же `cf=0` (сброшен флаг переноса), то с найденным файлом можно выполнить какие-либо действия.

Для того чтобы найти последующие файлы, удовлетворяющие нашему условию (маске поиска), необходимо воспользоваться функцией `4Fh` прерывания `21h` (табл. 13.5).

Таблица 13.5. Функция `4Fh` прерывания `21h`: поиск следующих файлов/каталогов

Вход	Выход
<code>ah = 4Fh</code> <code>cx</code> = атрибуты для поиска файла: у нас — 0 (обычные, не <code>read-only</code> и <code>hidden!</code>) <code>ds:dx</code> = маска для поиска (как у функции <code>4Eh</code>)	<code>cf = 1</code> : нет больше файлов, удовлетворяющих условию (маске поиска) <code>cf = 0</code> : DTA заполняется информацией о найденном файле

Как видите, функция `4Fh` прерывания `21h` аналогична функции `4Eh`.

Обращаем ваше внимание, что вирус у нас пока неработоспособный. То есть он ничего не заражает вообще! Можете спокойно его ассемблировать и запускать. Однако на экране ничего не будет отображено в процессе работы программы. Лучше ее сразу смотреть под отладчиком.

Дальше все просто! В файле-приложении вполне достаточно описаний для того, чтобы понять принцип работы программы.



Глава 14

Вывод окна в центре экрана

Эта глава довольно-таки сложная и, возможно, покажется вам не совсем интересной. Но то, что мы будем в ней рассматривать, очень важно. Вам придется вспомнить основы математики и немного подумать. Да-да, уважаемые читатели! Чтобы писать программы на ассемблере, нужно иметь хотя бы элементарные знания по математике и способность мыслить математически. Однако не волнуйтесь. Большую часть работы мы для вас сделали, а именно: продумали алгоритм вывода любого окна в центр экрана. Вам осталось только внимательно с ним ознакомиться и разобраться.

Очень внимательно изучите данную главу и прилагаемый к ней файл sshell14.asm!

Теперь давайте договоримся о некоторых определениях. Так как программа у нас будет большая, то для облегчения ее понимания и поиска необходимой процедуры будем указывать в скобках название самой процедуры и файла, где нужно искать то, о чем мы вас попросим. Например: "... Затем мы удаляем файл (`Delete_file`, `files.asm`)..." .

Здесь `Delete_file` — процедура, которая удаляет файл, а `files.asm` — файл, в котором нужно ее искать.

14.1. Модели памяти

14.1.1. Почему мы пишем только файлы типа СОМ?

Потому, что модель памяти Windows (FLAT) очень похожа на модель памяти СОМ-файлов, которые мы сейчас рассматриваем (TINY).

14.1.2. Что такое модель памяти и какие модели бывают?

Модель памяти определяет структуру и способ расположения кода и данных программы. Модели памяти могут быть следующими:

- TINY — "крошечная" модель, где код программы, ее данные и стек располагаются в одном сегменте, при этом используется 16-битная адресация (т. е. размер

программы не может превышать 64 Кбайт). Эта модель памяти используется в файлах типа СОМ, она компактна и очень удобна для написания небольших программ на ассемблере (например, резидентов). Все предыдущие наши примеры имеют именно такую модель памяти;

- **SMALL** — "маленькая" модель, где код размещается в одном сегменте, а данные и стек — в другом;
- **COMPACT** — "компактная" модель, где код программы размещается в одном сегменте, а данные могут занимать один сегмент и более. В этом случае обращение к данным происходит с указанием сегмента и смещения;
- **MEDIUM** — "средняя" модель, где код размещается в нескольких сегментах, а данные — в одном. Следовательно, к данным можно обращаться, используя только смещение, а к коду (вызов подпрограмм, безусловные переходы и т. п.) — не только смещение, но и сегмент;
- **LARGE** и **HUGE** — "большая" и "огромная" модели, где и код, и данные могут занимать несколько сегментов;
- **FLAT** — "плоская" модель, где код, данные и стек размещаются в одном сегменте (как и в модели TINY), но при этом используется 32-битная адресация. Таким образом, один сегмент может иметь размер до 4 294 967 296 байт (до 4 Гбайт). Подобная модель используется, как правило, в программах, написанных под Windows.

Модель памяти задается директивой `.model`. Приведенная в листинге 14.1 программа типа СОМ оформлена с использованием этой директивы.

Листинг 14.1. Использование директивы `.model`

```
;Указываем программе-ассемблеру тип модели памяти (TINY).
;Директиву ASSUME в таком случае можно опустить.
.model TINY
csEG segment ;Однако имя сегмента и счетчик (ORG) задаются.
org 100h
```

Begin:

```
    mov ah,9
    mov dx,offset Message
    int 21h

    ret
```

```
Message db 'Hello, world!$'
```

```
;Данные и стек расположены в том же сегменте, что и код. Вершина стека
;изначально равна 0FFEh того сегмента, куда загрузилась программа.
```

```
CSEG ends
end Begin
```

14.2. Оболочка SuperShell

14.2.1. Управление курсором

Сразу заглядываем в файл main.asm. Нам необходимо скрыть курсор (`Hide_cursor`, `display.asm`). Прежде чем это делать, необходимо запомнить его текущую позицию. Это позволяет сделать функция `03h` прерывания `10h` (табл. 14.1).

Таблица 14.1. Функция *03h* прерывания *10h*:
получить текущую позицию курсора

Вход	Выход
$ah = 3$ $bh =$ видеостраница, с которой получить текущую позицию курсора	$dx =$ текущая позиция курсора (dh — строка, dl — колонка)

Как видите, `bh` должен указывать на видеостраницу, с которой нужно сосчитать позицию курсора. Как уже вам известно, видеокарта имеет достаточно памяти для размещения 8 видеостраниц. Каждая видеостраница имеет свой курсор. Но мы-то будем отображать все на нулевой странице, следовательно, необходимо в `bh` загрузить 0. На выходе `dx` будет содержать текущую позицию курсора: `dh` — строку, а `dl` — колонку. Причем отсчет начинается с нуля (рис. 14.1).

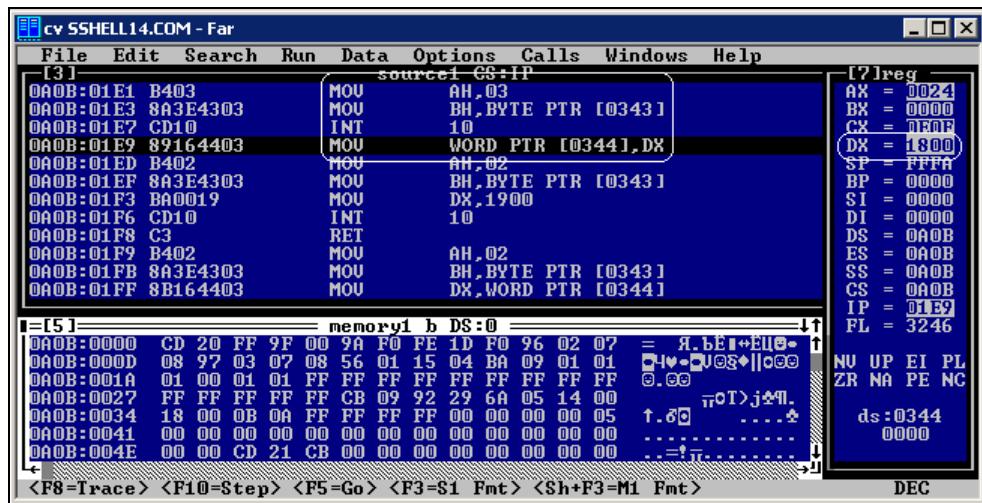


Рис. 14.1. Получение текущей позиции курсора

После того как получили текущую позицию курсора, мы можем его спрятать. Самый распространенный метод — установка курсора за границу экрана. В нашем случае — установка на 25-ю строку. Режим-то у нас будет 3 (25 строк, 80 колонок).

Так как отсчет идет с нуля, то последняя позиция курсора (самая нижняя строка) — 24. Следовательно, установив курсор на строку 25 (19h), мы его просто "прячем"! Установить курсор на нужную позицию на экране позволяет функция 02h прерывания 10h (табл. 14.2).

Таблица 14.2. Функция 02h прерывания 10h: позиционирование курсора

Вход	Выход
<code>ah = 2</code> <code>bh = видеостраница, на которой будем устанавливать курсор</code> <code>dh = строка, dl = столбец для установки</code>	Ничего

14.2.2. Операторы работы со стеком процессора 80286+

После того как мы спрятали курсор, можем переходить к сохранению текущего содержания нулевой видеостраницы. Как вы знаете, все уважающие себя оболочки сохраняют и, при необходимости, показывают пользователю то, что изображено на экране. Например, Norton Commander или Far Manager, которые отображают пользовательский экран при нажатии комбинации клавиш <Ctrl>+<O>.

Как сохранить то, что находится на экране в данный момент, а затем, при необходимости, показать это пользователю?

Мы уже упоминали о том, что у видеокарты есть достаточно памяти для размещения 8 видеостраниц. Почему бы нам не скопировать содержимое пользовательской, нулевой страницы в первую? Вот мы это и делаем (`Save_mainscr`, `display.asm`). Здесь мы встречаем новые операторы: `pusha` и `popa` (`pusha` от англ. *push all* — втолкнуть все; `popa` от англ. *pop all* — вытолкнуть все). Оператор `pusha` толкает в стек регистры в следующем порядке: `ax, cx, dx, bx, sp, bp, si` и `di`, и, соответственно, оператор `popa` выталкивает их из стека в обратном порядке: `di, si, bp, sp, bx, dx, cx` и `ax`.

На рис. 14.2 показано состояние стека после того, как в нем были сохранены перечисленные выше регистры командой `pusha`.

Данные операторы используются только процессором 286+. Когда пишут, например, что такой-то оператор используется в процессоре 386+, то это значит, что он будет работать на процессорах 80386, 80486, Pentium и т. д., но никак не на 8086 (PC/XT) и 80286 (PC/AT). В нашем случае команды `pusha` и `popa` будут работать на 80286 и последующих (более современных) процессорах. Для этого в самом начале файла `sshell.asm` мы устанавливаем директиву `.286`. Эта директива указывает ассемблеру (MASM, TASM), что в данной программе могут встречаться команды (инструкции) не только процессора 8086, но и 80286.

"Почему же мы раньше ничего подобного не писали?" — спросит внимательный читатель.

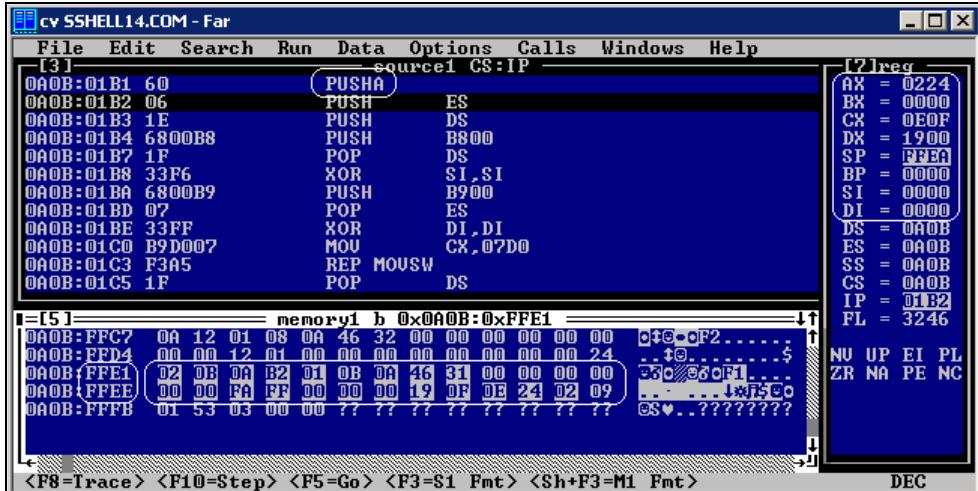


Рис. 14.2. Результат работы команды `pusha`

Если ничего подобного не указывать (как мы делали до сих пор), то ассемблер (MASM, TASM) считает, что в программе используются только инструкции процессора 8086.

Безусловно, использование команд `pusha` и `popa` удобно в тех случаях, когда нам нужно сохранить в стеке более одного регистра. Они уменьшают размер программы, однако в то же время необходимо следить за тем, чтобы не произошло переполнения стека.

Обратите также внимание, как мы заносим в сегментный регистр число:

```
...  
push 0B800h  
pop es
```

Это получается на 1 байт короче, чем если бы мы делали следующим образом:

```
...  
mov ax, 0B800h  
mov es, ax  
...
```

Более того, в этом случае мы не затрагиваем регистр `ax`, что иногда бывает полезно.

Команды вида `push 1234h` работают также на процессорах 80286+, а если быть точным, то и на процессорах 80186+. Но процессоры 80186 не получили большого распространения, о них вообще мало кто знает, поскольку через несколько месяцев после их выпуска появились 80286.

Все остальное в данной процедуре (`Save_mainscr`, `display.asm`) должно быть понятно. Как и в процедуре восстановления экрана (`Restore mainscr`, `display.asm`).

14.3. Процедура рисования рамки (окна)

14.3.1. Прямое отображение в видеобуфер

Курсор спрятали, экран пользовательский сохранили. Сейчас можно и окно рисовать. Будем это делать, не прибегая ни к каким прерываниям, т. к. они работают очень медленно. Воспользуемся выводом символов на экран путем прямого отображения в видеобуфер. С данным методом мы уже сталкивались, когда пытались выводить на экран "рожицу". Теперь рассмотрим его более подробно.

Метод прямого отображения символов в видеобуфер — вывод символов на экран, путем записи их напрямую в сегменты видеобуфера (в память видеокарты), минуя сервис прерываний. Стоит отметить, что это самый быстрый способ вывода символов на экран. Быстрее не существует. Здесь все зависит от профессионализма программиста и — как следствие — алгоритма вывода, скорости его выполнения.

Рассмотрим вкратце модели мониторов, многие из которых уже ушли в прошлое, и только старые и добрые игрушки периодически напоминают об их былом существовании. Вот ряд от старых к современным:

- MDA (Monochrome Display Adapter — монохромный видеоадаптер);
- Hercules (улучшенный графический адаптер "Геркулес");
- CGA (Color Graphics Adapter — цветной графический адаптер);
- EGA (Enhanced Graphics Adapter — улучшенный графический адаптер);
- VGA (Video Graphics Array — видеографическая матрица);
- MCGA (Multi-Color Graphics Adapter — подобие VGA);
- sVGA (Super Video Graphics Array — то, что в настоящий момент используют в работе).

Из истории

При попытке вывода символа в мониторе CGA и подобных ему возникает так называемый эффект "снега". Эта недоработка (ошибка?) была исправлена только в EGA. "Снег" не возникал, если символы выводились при помощи прерываний 10h или 21h.

Однако эти прерывания использовали процедуру, которая проверяла состояние хода луча монитора (символы выводились в тот момент, когда луч совершал обратный ход), в связи с чем скорость вывода символов существенно замедлялась. Не знаем, есть ли в настоящих "ПЗУшках" и в прерывании 21h подобная процедура или нет. Главное — при выводе символов через прерывания скорость резко снижается. Полагая, что никто из вас не использует сейчас CGA-монитор, будем выводить символы прямым отображением в видеобуфер. Повторим еще раз: это очень быстро даже на процессорах 8086. Очистка экрана в PC/XT происходит мгновенно. И уж тем более на процессорах 80286+...

Для процедуры рисования окна (а может и не только для нее) заведем две переменные: `Height_X` и `Width_Y`. Пока временно. В дальнейшем научимся передавать данные в стеке. Переменная `Height_X` содержит высоту окна, а `Width_Y` — его ширину. `Num_attr` — цвет рамки (`Main_proc`, `main.asm`). Можно вызывать процедуру `Draw_frame`, которая находится в файле `display.asm`.

14.3.2. Процедура *Draw_frame*

Вот сейчас и рассмотрим самое главное. Итак, многофункциональная процедура *Draw_frame* (рисование рамки любого размера в центре экрана).

Перед вызовом данной процедуры необходимо занести некоторые числа в соответствующие переменные (main.asm), а именно:

- Height_X — высота окна;
- Width_Y — ширина окна;
- Num_attr — атрибуты окна.

Пока все. Дальше будем изучать — добавим еще переменных.

Итак, занесли. Вызываем процедуру *Draw_frame* (display.asm).

Теперь внимание! Перед тем, как вывести рамку в центр экрана, нужно произвести некоторые расчеты. Для этого необходимо разделить высоту рамки на 2 и вычесть из полученного числа середину высоты. Вот так:

```
...
(1)    mov ax,Height_X
(2)    shr al,1
(3)    mov dh,11
(4)    sub dh,al
...
...
```

В строке (1) заносим в *ax* высоту рамки, которую указали перед вызовом процедуры. Затем, в строке (2), пока еще неизвестный нам оператор *shr* сдвигает все биты в регистре *al* на 1 вправо.

Что делает оператор *shr*? Вот пример:

```
mov ax,16      ;ax = 10000b, т. е. 16
shr ax,1       ;ax = 1000b, т. е. 8
shr ax,1       ;ax = 100b, т. е. 4
shr ax,1       ;ax = 10b, т. е. 2
shr ax,1       ;ax = 1b, т. е. 1
```

Таким образом, сдвиг битов вправо на единицу делит число на 2. На двойку — в четыре раза и т. д. Это гораздо быстрее, чем если бы мы пользовались оператором *div* — деление (который, кстати, мы еще не рассматривали). В строке (3) заносим в *dh* середину экрана по горизонтали, т. е. 11. А затем вычитаем из 11 полученный результат.

Например, мы выводим рамку высотой в 5 строк. Вот как производим вычисления:

```
ax = 10
ax = ax/2 = 5
ax = 11-5 = 6
```

Таким образом, вывод начнем со строки 6. Все то же самое мы делаем с колонками (вертикально). Посмотрите в прилагаемом примере. В принципе, если что-то пока не совсем понятно, то не заостряйте внимание на этом: со временем придет опыт и более точное понимание.

Как только мы вычислили координаты для вывода окна в центре экрана, мы переводим высоту/ширину в линейный адрес.

Что такое линейный адрес и зачем он нужен?

Как вы уже знаете, то, что находится на экране, расположено по адресу от 0B800:0000h до 0B800:1000h (нулевая страница). Видеокарта не знает, что такое колонка и строка. Для нее все данные расположены в линейном массиве, т. е. от 0 до 1000h. Так как один символ занимает 2 байта (символ и атрибут), то одна строка режима 3 занимает 160 байт ($80 \text{ строк} \times 2 = 160$). Для того чтобы отобразить символ в первой строке (точнее, во второй, т. к. отсчет начинается с нуля), нам нужно выводить его по смещению 161, сегмента 0B800h. Следовательно, если мы указываем строку/столбец, то нам необходимо перевести эти числа в линейный массив, "понятный" видеокарте. Вот пример (разбираем внимательно!):

```
;dh обычно указывает на строку
mov dh,0
;a dl — на столбец (это мы и будем делать во всех наших процедурах)
mov dl,15
;таким образом, мы будем выводить в нулевой строке
;(т. е. самой верхней на экране), 15 столбце.
```

Переводим 2 числа (строка и колонка) в линейный массив (переменная Linear будет содержать в итоге адрес для вывода символа на экран):

```
;Умножаем на 2 адрес строки, т. к. один символ на экране занимает
;два байта: символ + его атрибут
Linear = dl*2
Linear = Linear + dh*160
```

Теперь переменная Linear равна 30. Вот он, линейный адрес: 0B800:001Eh (1Eh=30). Еще пример:

mov dh,8	;Строка 8
mov dl,56	;Столбец 56
Linear = dl*2	;Теперь Linear = 112
Linear = Linear + dh*160	;Теперь Linear = 1392 или 570h

И еще произведем вычисления для последнего символа на экране (правый нижний угол):

$$\text{Linear} = (24 \times 160 + 80 \times 2) - 2 = 0F9Eh.$$

Здесь:

- 80 — количество столбцов на экране, которое нужно умножить на 2 (учитывая не только символ, но и его атрибут).
- 24 — количество строк на экране, начиная с 0. Так как в одной строке 80 символов (начиная с 1): $80 \times 2 = 160$, то, для того чтобы получить адрес самого последнего (24-го) ряда, нужно 24 умножить на 160.

Затем все это складываем (адрес 24-й строки + адрес последнего символа – 2). Еще раз:

$$(24\text{-я строка} \times 160 + 80\text{-й символ} \times 2) - 2$$

или

$$(24 \times 160 + 80 \times 2) - 2 = 3998 \text{ или } 0F9Eh.$$

14.4. Практика

Теперь рассмотрим, как это выглядит на практике. Итак, процедура (`Get_linear`, `display.asm`) (листинг 14.2).

Листинг 14.2. Процедура `Get_linear`

```
...
;умножаем dl на 2 (dl=dl*2). Просто сдвинем биты на 1 влево (по принципу
;деления; см. предыдущий раздел).
shl dl,1
mov al,dh      ;в al — ряд,
mov bl,160     ;который нужно умножить на 160
mul bl        ;умножаем: al (ряд) * 160; результат — в ax
```

;Здесь видим новый оператор `mul` — деление, который рассмотрим позже.

```
mov di, ax      ;результат умножения — в di
xor dh, dh      ;аннулируем dh
add di,dx      ;теперь в di линейный адрес в видеобуфере.
```

...

14.5. Новые операторы

В табл. 14.3—14.5 приведено описание новых операторов. В листингах 14.3—14.5 показаны примеры использования этих операторов, а на рис. 14.3—14.5 — результаты их работы.

Таблица 14.3. Оператор `shl`

Команда	Перевод	Назначение	Процессор
<code>shl источник, приемник</code>	Shift left — сдвиг влево	Сдвиг бит	8086

Листинг 14.3. Пример использования оператора `shl`

```
mov ax,100b
shl ax,1      ;Теперь в ax число 1000b
```

Таблица 14.4. Оператор `shr`

Команда	Перевод	Назначение	Процессор
<code>shr источник, приемник</code>	Shift right — сдвиг вправо	Сдвиг бит	8086

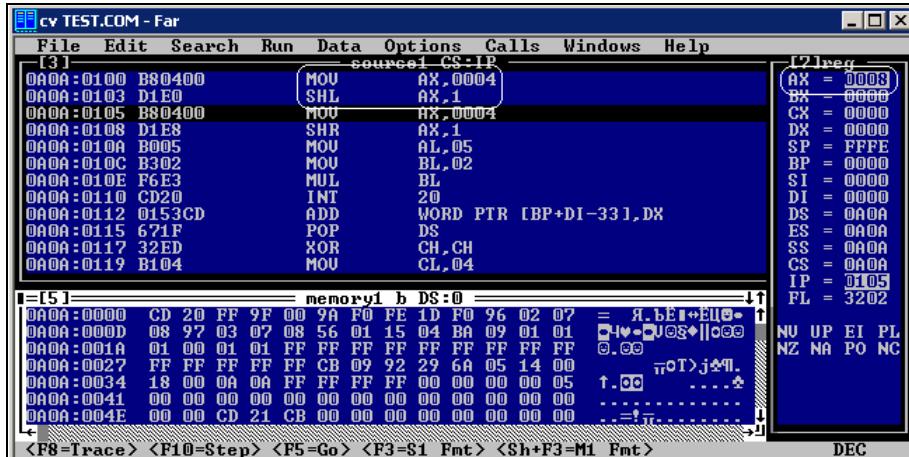


Рис. 14.3. Результат работы оператора shl

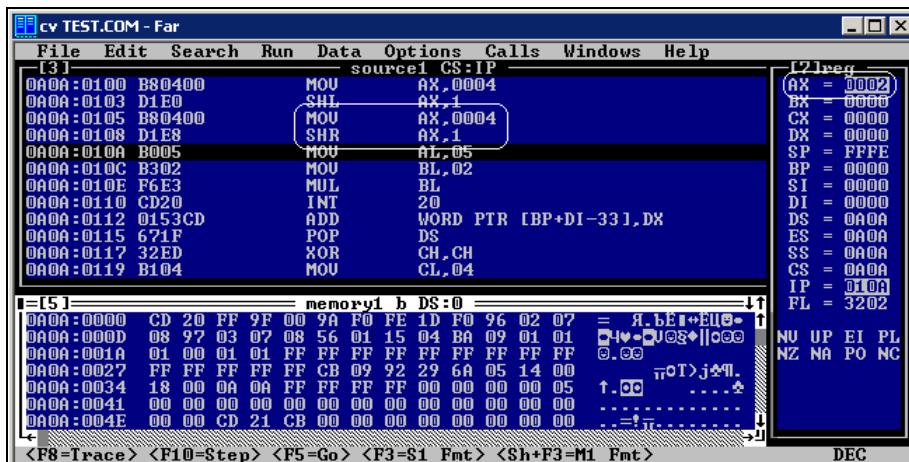


Рис. 14.4. Результат работы оператора shr

Листинг 14.4. Пример использования оператора shr

```
mov ax,100b
shr ax,1           ;Теперь в ах число 10b
```

Таблица 14.5. Оператор mul

Команда	Перевод	Назначение	Процессор
mul bl	Multiplex — умножение	Умножение al на bl	8086

Умножаем al на bl, помещая результат в ах.

Листинг 14.5. Пример использования оператора `mul`

```
mov al,5  
mov bl,2  
mul bl ;Теперь вах число 10
```

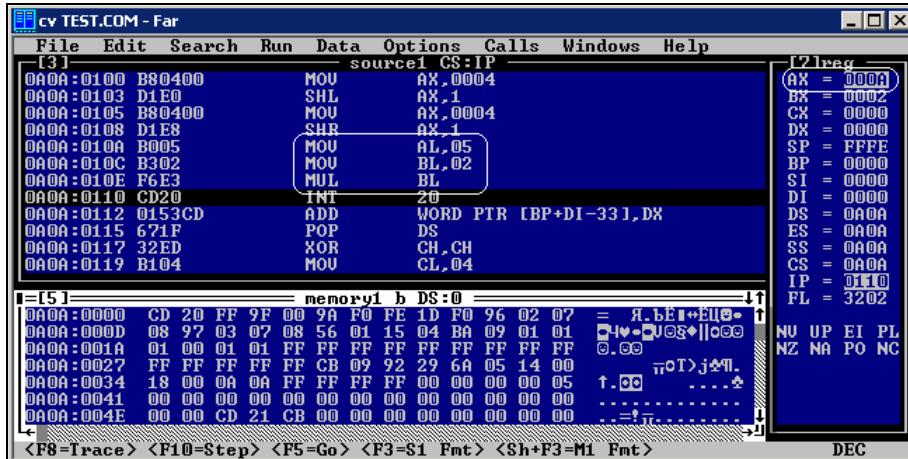


Рис. 14.5. Результат работы оператора `mul b1`

Ну вот, собственно, и все. Остальное должно быть понятно из комментариев к программе.



Глава 15

Обработка аппаратных прерываний

В этой главе рассмотрим "многофункциональный" резидент. Будет сложно, но интересно. Наша резидентная программа выполняет следующие действия:

1. Записывает в файл содержимое текстового экрана (то, что в момент нажатия определенных клавиш находится в области экрана).
2. Меняет на экране все символы "A" и "a" на "O" и "o" соответственно.
3. Передает неверные данные о файлах оболочкам Norton Commander, Volkov Commander, Dos Navigator.

И все это умеет один резидент, размер которого всего 746 байт!

15.1. Теория

Разобраться в инициализации резидента (метка `Init`) труда не составит. Говоря в двух словах, мы делаем следующее: проверяем на повторную загрузку в память путем отправления нашего "позвывного" (в данном случае — `9889h` в `ax`) и получения или неполучения "отклика" (число `8998h` также в `ax`). Если после вызова прерывания `21h` с числом `9889h` в аккумулятор возвращается `8998h`, то наш резидент уже в памяти (он-то и меняет местами `ah` и `al`). Обратите внимание на первые четыре строки процедуры `Int_21h_proc`, которые как раз меняют местами `ah/al`, но делают это только в том случае, если наш резидент уже загружен в память.

Проведем эксперимент. Перенесем приведенные ниже строки после установки всех прерываний, но перед вызовом `int 27h` (листинг 15.1).

Листинг 15.1. Проверка на повторную загрузку резидента в память

```
...
mov ax, 9889h
int 21h
cmp ax, 8998h
jne Set_resident
...
...
```

В этом случае наша программа выдаст сообщение о том, что она уже загружена в память, хотя и загружается первый раз. Следовательно, наш обработчик `Int_21h_proc`

будет находиться в памяти и контролировать прерывание 21h еще до выхода! Таким образом, наш резидент активизируется сразу после того, как мы установили на него вектор прерывания (а мы это сделали, используя функцию 25h прерывания 21h) (листинг 15.2).

Листинг 15.2. Сохранение и установка вектора прерывания

```
...
;Установка обработчика 21h
mov ax, 3521h
int 21h          ;получим и сохраним адрес (вектор) прерывания 21h
;вначале младшее слово (смещение)
mov word ptr Int_21h_vect,bx
;затем старшее (сегмент)
mov word ptr Int_21h_vect+2,es

mov ax, 2521h
mov dx,offset Int_21h_proc
```

;Завершаем, оставив резидентную часть в памяти

```
int 21h
...
```

Еще одним подтверждением того, что резидент активизируется до выполнения команды int 27h, является тот факт, что даже в отладчике мы увидим результат его работы. Символы начнут заменяться уже после установки процедуры Int_21h_proc на прерывание 21h (рис. 15.1).

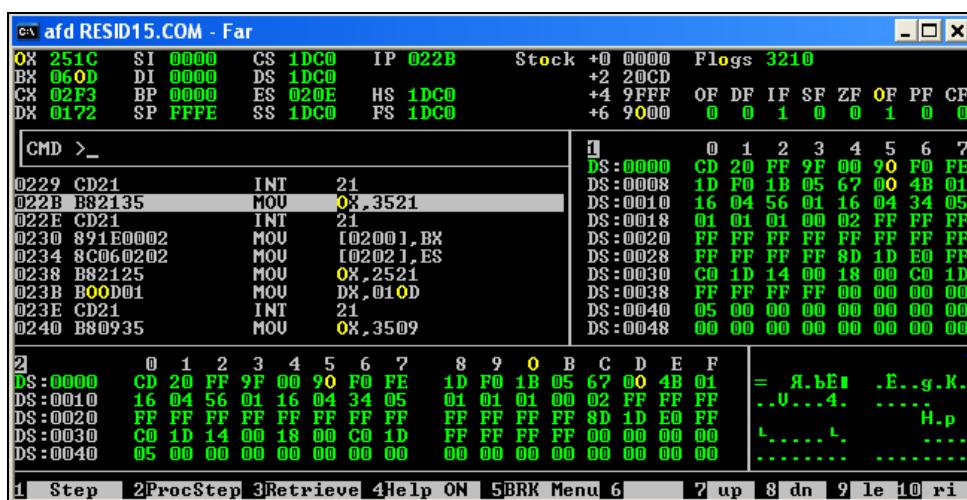


Рис. 15.1. Активизация резидента в процессе отладки

Для того чтобы резидентная часть программы продолжала корректно работать после выхода, нам нужно оставить ее резидентной в памяти. Если мы после установки прерываний выйдем в DOS, используя `int 20h`, то компьютер просто "зависнет"! Память, где находятся наши обработчики (`Int_05h_proc`, `Int_21h_proc`, `Int_1Ch_proc`), освободится, и на ее место загрузятся совсем другие программы. После чего адрес прерывания `21h` будет указывать на место в памяти, где должна находиться процедура обработки этого прерывания (`Int_21h_proc`). Но процедуры обработки прерывания в данном месте памяти больше не существует, и никому неизвестно, что там теперь находится вместо нее. Поэтому компьютер "зависнет" или перестанет работать должным образом, если какая-либо иная программа или же сам процессор попытается вызвать одно из перечисленных выше прерываний.

15.1.1. Сохранение предыдущего вектора прерывания

Для чего следует сохранять старый вектор `21h`?

DOS, при смене вектора прерывания, нигде не запоминает его прежний адрес. Если программист его самостоятельно не сохранит, то наша процедура, получив управление в первый раз, отработает и не будет "знать", что делать дальше, кроме как просто вернуться из прерывания и передать дальнейшее управление той программе, которая вызывала это прерывание.

Например, обработчик перехватывает прерывание `21h` и следит за тем, вызывается ли функция `3Dh` (открытие файла) некоторой программой. Если так, то резидент должен сохранить в своей области имя открываемого файла, а затем передать управление на адрес старого обработчика `21h`, который и откроет файл. В случае, если вызывается другая функция, отличная от `3Dh`, то следует также немедленно передать управление предыдущему обработчику, который и выполнит необходимые действия. Адрес прежнего обработчика может указывать как на ядро DOS, так и на другой резидент, перехвативший прерывание `21h` и загрузившийся ранее нашего. Получается своеобразная цепочка, каждое звено которой (резидентные программы) должно в обязательном порядке сохранять адрес прежнего обработчика прерывания `21h`, кроме, разумеется, самого ядра DOS, являющегося начальным звеном и выполняющим необходимые действия.

Для сохранения предыдущего адреса того или иного прерывания необходимо завести отдельные 32-разрядные переменные, способные хранить два слова (сегмент и смещение):

```
Int_21h_vect dd ?
```

или

```
Int_1Ch_vect dd ?
```

и т. п.

Здесь `dd` означает "define double word" — определить переменную с двойным словом.

15.1.2. Способы передачи управления на прежний адрес прерывания

Существуют 2 стандартных способа передачи управления на прежний адрес (вектор) прерывания.

Первый способ

```
jmp dword ptr cs:[Int_21h_vect]
```

dword ptr означает, что надо "прыгнуть", используя не только смещение, но и сегмент (сегмент:смещение), которые находятся в переменной Int_21h_vect.

cs: указывает на то, что переменная находится в текущем сегменте. А cs: всегда указывает на текущий сегмент, т. е. на тот, в котором выполняется код программы в настоящий момент. Если мы опустим cs:, то процессор по умолчанию будет работать так:

```
jmp dword ptr ds:[Int_21h_vect]
```

А это неверно, т. к. ds будет содержать отличное от cs значение (см. в отладчике файл-приложение).

Переход в другой сегмент происходит в тот момент, когда мы, например, вызываем некоторое прерывание (как в нашем примере resid15.asm). При вызове прерывания из сегментных регистров меняется только cs, а прочие регистры остаются прежними, сохраняют те значения, которые были перед вызовом прерывания (подробнее об этом скажем далее).

Квадратные скобки ([]) указывают на то, что нужно "прыгнуть" на тот адрес, который находится в переменной Int_21h_vect, хотя в данном случае квадратные скобки можно было и опустить.

Команды вида jmp dword ptr ... называются "дальним jmp с переходом в другой сегмент".

Второй способ

```
call dword ptr cs:[Int_21h_vect]
```

Здесь рассуждаем, как и в первом случае. Но есть некоторые отличия.

Вы привыкли к тому, что call используется для вызова процедур внутри нашего сегмента. Такие процедуры называют близкими. В этом случае вызывается дальнняя процедура, расположенная в другом сегменте. Принципы вызова процедур следующие:

- вызов ближней (near) процедуры. В данном случае в стек автоматически помещается смещение следующей за командой call инструкции. После того как ближняя процедура отработала, из стека извлекается сохраненный адрес того места в памяти, куда необходимо вернуться, чтобы продолжить выполнение программы. Ближняя процедура должна находиться в том же сегменте, в котором находится команда call, ее вызывающая. Пример вызова ближней процедуры:

```
call Near_proc
```

- вызов дальней (`far`) процедуры. В этом случае процессор автоматически помещает в стек не только смещение, но и сегмент для возврата. Компьютер, дойдя в процедуре до инструкции возврата, извлечет из стека сохраненные сегмент и смещение и передаст управление по этому адресу. Дальняя процедура может находиться в любом другом сегменте, отличном от того, откуда она вызывается. Пример вызова процедуры, расположенной в другом сегменте:

```
call dword ptr cs:[Far_proc]
```

В данном случае сегмент и смещение вызываемой процедуры находятся в переменной `Far_proc`, которая расположена в текущем сегменте, т. е. в том, в котором выполняется в настоящий момент указанная выше команда `call`.

15.2. Инструкции `ret` и `retf`

15.2.1. Оператор `ret`

Пример работы оператора `ret` приведен в листинге 15.3.

Листинг 15.3. Вызов ближней процедуры

```
;Предположим, что стек пустой (ss=1234h, sp=0FFFFh).
...
(1) [1234:0100h]    mov ax,0A0Bh
(2) [1234:0103h]    call Our_proc      ;Вызываем процедуру Our_proc
(3) [1234:0105h]    mov dx,123h

...
(4) [1234:0200h]    Our_proc proc
(5) [1234:0200h]    mov dx,offset Message

...
;Выходим из процедуры (переходим на адрес 1234:0105h)
(6) [1234:0250h]    ret
(7) [1234:0250h]    Our_proc endp
...
```

Давайте предположим, что строка (1) расположилась по адресу 1234:0100h, а процедура `Our_proc` находится по адресу 1234:0200h (см. адреса в скобках). Когда программа дойдет до строки (2), сегмент `cs` будет содержать адрес текущего сегмента (1234h), а смещение `ip` — адрес следующей команды (0105h).

В момент выполнения строки (2) в стек заносится текущее состояние регистра `ip` (и только!), т. е. число `0105h`. Затем процессор переходит на адрес `0200h` (на метку нашей процедуры). Как видите, директива `Our_proc proc` не занимает памяти: она нужна только ассемблер-программам.

Начинает работать процедура `Our_proc`, при этом адрес возврата (в нашем примере — `0105h`) находится в стеке. Мы уже обращали ваше внимание, что в процедурах следует очень тщательно следить за стеком, т. к. если мы в подпрограмме занесем в стек какое-нибудь число и впоследствии, перед выходом из процедуры, его не извлечем оттуда, то процессор, дойдя до инструкции `ret`, вытащит из стека не адрес возврата, а то число, которое находится на вершине стека.

Итак, процедура отработала. Процессор дошел до команды `ret`. Команда `ret` достает последнее число, находящееся на вершине стека в данный момент, и перейдет по этому адресу. Если мы со стеком ничего не намудрили (ничего в нем не оставили и ничего "лишнего" не вытащили), то `ret` достанет именно число `0105h`, т. е. то, что сохранила инструкция `call`, и перейдет по этому адресу.

Важно!

При вызове ближней процедуры в стеке сохраняется только смещение следующей за процедурой инструкции (в примере из листинга 15.3 — только `0105h`).

15.2.2. Оператор `retf`

При вызове дальней процедуры в стеке сохраняется не только смещение, но и сегмент для возврата. Например, мы точно знаем, что по адресу `3456:0400h` находится процедура вывода строки на экран. Проблема в том, как определить сегмент этой процедуры, отличный от того, в котором находится наша программа (не `1234h`, а `3456h`). Для этого занесем в некую переменную (пусть это будет `New_proc`) два слова — `3456h` и `0400h` — адрес процедуры `New_proc`, включая сегмент и смещение. Затем передаем управление этой процедуре (листинг 15.4).

Листинг 15.4. Вызов дальней подпрограммы

```
...
;Заносим сперва смещение
mov word ptr [New_proc],0400h
;Затем сегмент
mov word ptr [New_proc+2],3456h
;Вызываем процедуру
call dword ptr [New_proc]
...
New_proc dd ?      ;Переменная для хранения адреса дальней процедуры (2 байта)
...
```

В листинге 15.4 мы вначале заносим в переменную `New_proc` смещение, а затем сегмент процедуры (вспоминаем о том, что в компьютере данные хранятся в об-

ратном порядке). Если после команды `call` следует `dword ptr`, то в стек заносится не только смещение, но и сегмент.

Возникает вопрос: как компьютер различает, когда нужно извлечь из стека только смещение, а когда смещение и сегмент? Ответ простой: существуют две разновидности инструкции `ret`: `ret` и `retf`. Инструкция `retf` (от англ. *return far* — дальний возврат) извлекает из стека не только смещение, но и сегмент (в отличие от команды `ret`, достающей смещение). Если мы вызовем дальнюю процедуру (`call dword ptr [Far_proc]`), а выйдем из нее, используя `ret`, то компьютер просто "зависнет". В листингах 15.5 и 15.6 приведены примеры.

Листинг 15.5. Вызов ближней процедуры

```
...
[1234:0200h] call Near_proc      ;Вызов ближней процедуры
...
;Ближняя процедура Near_proc (находится в том же сегменте, что и
;и основная программа)
[1234:4569h] Near_proc proc
...
;Правильный выход из ближней процедуры (процедуры, которая располагается
;в том же сегменте, что и программа, ее вызывающая)
[1234:6789h] ret
[1234:6789h] Near_proc endp
...
...
```

Листинг 15.6. Вызов дальней процедуры

```
...
[1234:0200h] call dword ptr cs:[New_proc]
...
;Дальняя процедура New_proc (находится в другом сегменте)
[3456:0300h] New_proc proc
...
;Правильный выход из дальней процедуры (процедуры, которая находится
;в другом сегменте, в отличие от программы, ее вызывающей)
[3456:0534h] retf
[3456:0534h] New_proc endp
...
```

Одна и та же процедура не может быть вызвана двумя разными способами. Если это дальнняя процедура, то вызвать ее стандартными способами, как ближнюю, не получится, даже если она будет находиться в том же сегменте, в котором выполняется код ее вызова.

Возможно, это покажется вам на первый взгляд сложным, но, поверьте, все на самом деле очень просто. Более того, путаницы не возникает с вызовами и возвратами. Нужен, прежде всего, опыт работы с ассемблером.

15.3. Механизм работы аппаратных прерываний. Оператор *iret*

"Ага! — скажет читатель. — Что-то мы тут упустили! А почему тогда перед `call dword ptr [Int_21h_vect]` мы заносим в стек командой `pushf` регистр флагов, но впоследствии его не достаем?"

```
...
pushf          ;Заносим в стек флаги
call dword ptr [Int_21h_vect] ;Вызываем дальнюю процедуру
...
```

Зачем нужно заносить флаги в стек командой `pushf` перед вызовом оригинального обработчика прерывания? Попробуем дать ответ на этот вопрос. Сперва рассмотрим, для чего вообще вызываются прерывания, и что происходит в этот момент.

Существуют два вида прерываний: *программные* и *аппаратные*. Программные прерывания вызываются программами при помощи инструкции `int`, что мы уже неоднократно делали. К таким прерываниям можно отнести, в частности, `21h`, `16h`, `10h` и пр.

В настоящий момент нас больше интересуют *аппаратные* прерывания, которые вызываются автоматически аппаратурой (процессором) в тех или иных ситуациях. Например, при делении на ноль, обновлении таймера, нажатии пользователем клавиши и пр. В листинге 15.7 приведен фрагмент кода обработчика прерывания `09h` (клавиатура) и некоторой программы, выполняемой в данный момент (обращайте внимание на сегменты).

Листинг 15.7. Обработчик прерывания `09h`

```
;Обработчик прерывания 09h, расположенный где-то в памяти, который
;активизируется при нажатии и отпускании какой-нибудь клавиши:
...
[0900:0050h] mov al,bl
;Здесь неважно, какой код находится. Главное то, что сегмент другой,
;отличный от того, в котором находится наша программа (см. ниже)
[0900:0052h] ...
[0900:0345h] iret
...
;Это часть нашей программы, которая в данный момент выполняется:
...
(1) [1234:0200h] mov ax,Num_Regax
```

```
(2) [1234:0205h]    cmp ax,17
(3) [1234:0208h]    jne Not_equal
...

```

Аппаратное прерывание 09h вызывается всегда, когда пользователь нажимает или отпускает какую-нибудь клавишу, даже тогда, когда процессор чем-то занят (например, копирует файл или форматирует диск).

Допустим, пользователь нажал клавишу в тот момент, когда наша программа выполнила строку (2). Что произойдет? На первый взгляд кажется, что ничего особенного. Процессор занесет клавишу в буфер, которая будет храниться там до того момента, пока DOS ее оттуда не запросит и не выполнит определенные действия. Однако на самом деле процессор проделает уйму работы за считанные доли миллисекунд. Рассмотрим это подробней.

У нас в регистрах находятся определенные числа. Более того, после выполнения строки (2) изменится регистр флагов. В строке (3) мы проверяем состояние флага нуля, который сигнализирует нам в данном случае о том, равен ли ax 17. Наша программа находится в момент вызова прерывания в сегменте 1234h, смещение 0208h (адрес следующей команды).

Как сделать так, чтобы передать управление прерыванию 09h, которое выполнит свою работу (занесет в буфер клавиатуры код клавиши, которую программы могут затем получить при помощи функции 10h прерывания 16h), при этом вернуться в то место, с которого произошло прерывание, не нарушив работы нашей программы?

Процессору нужно запомнить номер текущего сегмента (cs), смещение следующей команды (оно всегда в ip), а также состояние флагов в стеке. Этого достаточно для того, чтобы вернуться назад после того, как прерывание 09h отработает, не нарушив при этом ход выполнения нашей программы. Заметьте, что при вызове прерывания регистры не сохраняются, кроме флагов и cs:ip! В случае необходимости сохранять регистры должно то прерывание, которое получило управление (в случае нажатия клавиши — 09h).

После того как процессор сохранил регистры cs:ip и флаги в стеке, он передает управление обработчику аппаратного прерывания (в нашем случае — 09h), адрес которого находится в определенном месте памяти (где именно находятся адреса прерываний, мы рассмотрим позже). Что значит "передает управление"? Да просто "прыгает" на определенный адрес. Это будет называться "дальним безусловным переходом", т. к. мы прыгаем не только на смещение внутри сегмента (как в случаях, которые мы рассматривали уже, например, jmp Init в разд. 15.1.2), а на сегмент и смещение.

Итак, управление получило прерывание 09h. Если в процессе его работы меняются какие-то регистры, то оно должно их предварительно сохранить, иначе наша программа, после отработки прерывания 09h, получит совсем другие значения в регистрах.

Например, если 09h изменит ax, предварительно не сохранив его, а затем, соответственно, не восстановив, то при выходе из данного прерывания мы получим, что ax не равен Num_Regax (1).

Отсюда жесткое правило: если вы пишете свой обработчик, то обязательно сохраняйте все регистры, которые он меняет. Если вы написали обработчик, например, прерывания 21h, и после загрузки его в память компьютер "зависает" или ведет себя не так, как хотелось бы, то ищите ошибку в вашем резиденте. Возможно, вы забыли сохранить тот или иной регистр в стеке. Хотя причин, по которым "зависает" компьютер, может быть очень много...

Итак, прерывание 09h (т. е. обработчик прерывания 09h) отработало: процессор дошел до инструкции `iret` (от англ. *interrupt return* — возврат прерывания). Эта инструкция отличается от `ret` тем, что при ее выполнении процессор извлечет из стека сегмент (`cs`), смещение (`ip`) и регистр флагов, вместо одного смещения (`ip`) (как `ret`). Вот и вся разница между `ret` и `iret`.

Резюмируем:

- `ret` достает из стека только смещение для возврата; процедура должна находиться в том же сегменте, из которого ее вызывают (ближняя процедура — `near` (по умолчанию));
- `retf` достает из стека сегмент и смещение; процедура может находиться в любом сегменте, независимо от того, откуда ее вызывают (далняя процедура — `far` или `dword ptr`);
- `iret` достает из стека сегмент, смещение и адрес флагов. Используется для выхода из прерываний.

Команда `iret` вытаскивает из стека адрес возврата (`cs:ip`). В примере из листинга 15.7 `cs=1234h, ip=0208h`, а в регистре флагов установлен флаг нуля, т. е. его значение 1. Затем управление передается на этот адрес. Наша программа продолжает работать, не догадываясь даже о том, что кто-то ее прервал. Естественно, все эти процедуры выполняются чрезвычайно быстро.

Теперь ответ на главный вопрос: почему перед вызовом прерывания командой вида `call dword ptr cs:[Int_21h_vect]` мы заносим в стек регистр флагов командой `pushf` и впоследствии его не достаем?

Ответ прост. Стоит только посмотреть на отличие оператора `retf` от `iret` (см. ранее). При передаче управления (вызове) прерывания командой `call dword ptr ...` процессор заносит в стек только сегмент (`cs`) и смещение (`ip`) следующей за командой `call` инструкции. А `iret` извлечет из стека сегмент, смещение и флаги. Но флаги ведь не заносятся командой `call dword ptr!` Мы их заносим сами, "вручную". Иначе произойдет нарушение работы стека, и компьютер "зависнет". Как мы уже убедились, за стеком нужно следить очень внимательно!

15.4. Практика

Что делает наш обработчик прерывания 21h?

Он умышленно передает неверные данные программе, которая ищет файлы в каталоге, используя функции 4Eh и 4Fh. Подробнее эти функции мы рассмотрим, когда будем писать оболочку. Здесь все вкратце.

Функция 4Eh ищет первый файл или каталог на диске, и, если какой-нибудь файл найден, заносит в DTA информацию о нем, которая содержит, в частности:

- имя и расширение файла;
- размер файла;
- дату и время создания файла;
- атрибуты файла.

То же самое делает функция 4Fh. Ее отличие только в том, что она ищет второй и последующие файлы. Так устроена MS-DOS.

Наш резидент контролирует прерывание 21h. Если вызывается одна из упомянутых выше функций, то резидент подменяет информацию, которая находится в DTA после вызова прерывания 21h (листинг 15.8).

Листинг 15.8. Проверка вызовов прерывания 21h

```
...
;Проверяем: вызывает ли какая-то программа функцию 4Eh или 4Fh (поиск файлов)
cmp ah,4Eh
je Do_not

cmp ah,4Fh
je Do_not

;Если вызывается другая функция, то просто передадим управление оригинальному
;обработчику 21h командой " дальний jmp". Здесь заносить в стек флаги не нужно,
;т. к. мы больше не вернемся в наш обработчик. Уходим навсегда...
Go_21h:
jmp dword ptr cs:[Int_21h_vect]

;Итак, кто-то вызывает функцию 4Fh или 4Eh...
Do_not:
pushf
call dword ptr cs:[Int_21h_vect]
...
```

Прежде чем менять информацию о найденных файлах в DTA, нам нужно вызвать прерывание 21h самим, для того чтобы оно занесло необходимые данные в эту область. Мы, конечно, можем самостоятельно записать туда что угодно, не вызывая указанное выше прерывание, но это будет просто и неинтересно.

Обратите внимание на две команды, следующие за меткой Do_not в листинге 15.8. Здесь мы вызываем прерывание 21h. Но каким образом! Ведь данный фрагмент кода — это и есть наш обработчик прерывания 21h. Следовательно, если мы вызовем прерывание стандартной командой int 21h, то она передаст управление на наш обработчик, что зациклит процедуру и "повесит" компьютер. Нам это

абсолютно не надо! Поэтому мы вызываем прежний обработчик прерывания 21h, который использовался до загрузки нашего резидента. Это может быть как другой резидент, так и само ядро DOS. Получается своего рода "фильтр" прерывания 21h. Программист может "фильтровать" те или иные функции прерывания и делать с ними все, что посчитает нужным!

Итак, передали управление прежнему обработчику. Он нам вернул информацию о найденном файле в DTA. Что дальше?

"Но мы не знаем, где в памяти находится этот DTA!" — скажете вы. Верно, не знаем. Он может быть где угодно. Получить адрес DTA позволяет функция 2Fh прерывания 21h. Но обратите внимание, как мы теперь вызываем 21h:

```
int 21h
```

Почему так? Вызвав 21h стандартной командой int 21h, процессор попадет на начало нашего обработчика, в самое начало процедуры Int_21h_proc. Рассуждаем так: в этот момент в регистре ah у нас находится число 2Fh. Теперь посмотрите внимательно, что произойдет, если наша процедура Int_21h_proc получит управление с находящимся в ah числом 2Fh.

А если она же получит управление с числом 4Fh в ah? Вот именно! Поэтому мы, в случае вызова прерывания 21h с числом 4Eh или 4Fh, вызываем напрямую прежний обработчик командой call dword ptr cs:[Int_21h_vect], а с любым другим числом — наш обработчик командой int 21h. Зачем так делать? Дело в том, что второй вариант занимает меньше байтов, да и работает быстрее...

Адрес DTA получили. Теперь необходимо получить случайное число. Это можно сделать, вызвав функцию 2Ch прерывания 21h, которая вернет в определенных регистрах текущее время. В файле-приложении вы найдете подробное описание.

15.5. Логические команды процессора

Пришло время рассмотреть *логические команды процессора*, которые используются в файле-приложении. Логических команд всего несколько: and, or, xor.

Проще всего объяснить их действие на примерах. Главное — понять принцип. Мы еще не раз будем использовать логические операторы. Это одна из быстрых и простых вещей ассемблера, в отличие от языков высокого уровня.

15.5.1. Оператор or

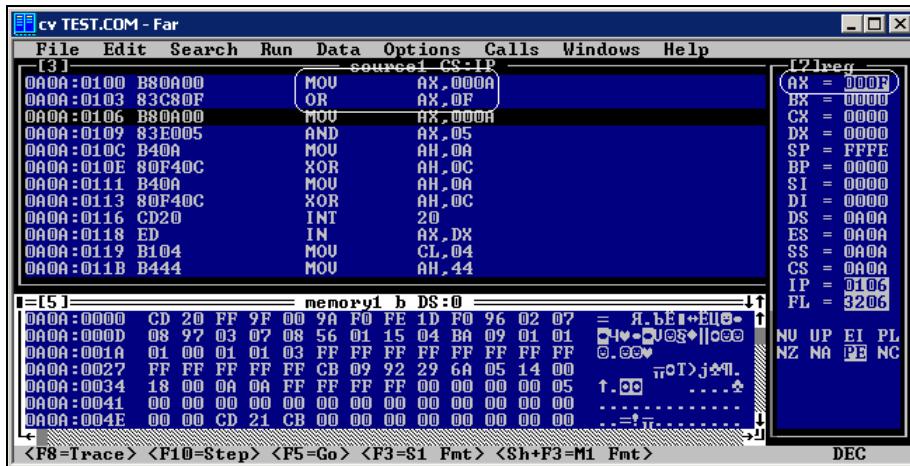
Оператор or (ИЛИ) служит для включения определенных битов (не байтов!) в регистре, переменной или в памяти (табл. 15.1 и рис. 15.2).

```
mov    ax,1010b
or     ax,1111b
```

;Теперь ax=1111b=0Fh. Таким образом мы включили (установили) первые четыре бита.

Таблица 15.1. Результат работы команды `or`

Первый бит	Второй бит	Результат работы <code>or</code>
1	1	1
1	0	1
0	0	0

Рис. 15.2. Результат выполнения команды `or`

```
mov ah,1000b
or ah,1001b
```

; Теперь ah=1001b. Таким образом мы установили нулевой бит, а третий был
; уже установлен до нас, но он остался неизмененным!

15.5.2. Оператор `and`

Оператор `and` (И) служит для исключения битов (табл. 15.2, рис. 15.3).

```
mov ax,1010b
and ax,0101b
```

; Теперь ax=0000b=0h. Таким образом мы исключили первый и третий биты
; (отсчет справа налево, начиная с нуля).

Таблица 15.2. Результат работы команды `and`

Первый бит	Второй бит	Результат <code>and</code>
1	1	1
1	0	0
0	0	0

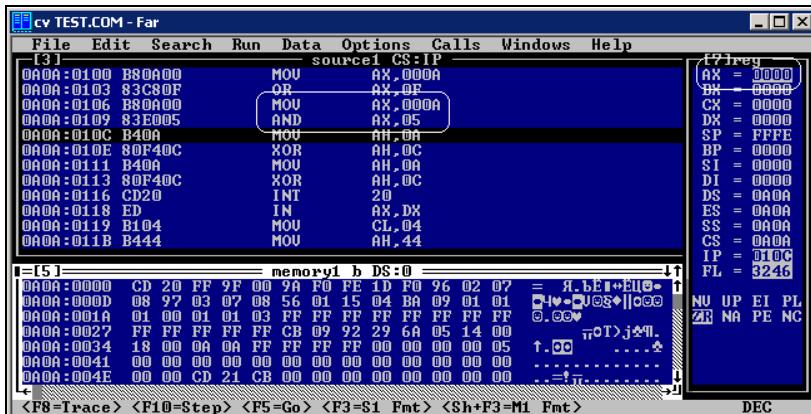


Рис. 15.3. Результат выполнения команды and

```
mov ah,1001b
and ah,0001b
```

;Теперь ah=0001b. Таким образом мы исключили третий бит.

15.5.3. Оператор xor

Оператор xor (исключающее ИЛИ) используется в основном для кодирования данных (табл. 15.3, рис. 15.4).

```
mov ah,1010b
xor ah,1100b ;Теперь ah=0110b=06h
```

Таблица 15.3. Результат работы команды xor

Первый бит	Второй бит	Результат xor
1	1	0
1	0	1
0	0	0

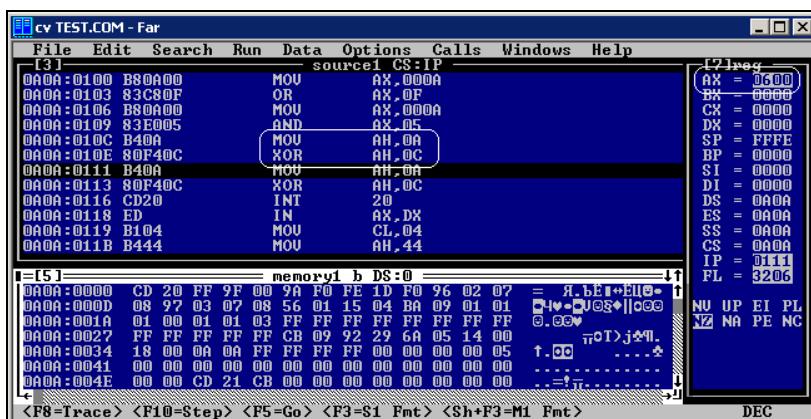


Рис. 15.4. Результат выполнения команды xor

Оператор xor очень удобен также для включения/исключения одного бита:

```
mov al,0      ;al=0
xor al,1      ;al=1
xor al,1      ;al=0
xor al,1      ;al=1 и т. д.
```

Именно этим оператором мы быстро и красиво аннулировали регистры в предыдущих главах.

15.6. Аппаратные прерывания нашего резидента

Кроме аппаратных прерываний, в процедуре обработки прерывания 21h мы пока ничего рассматривать не будем. В файле-приложении есть описания, которых вполне достаточно для понимания работы программы.

15.6.1. Аппаратное прерывание 05h

Прерывание 05h активизируется (т. е. процессор передает управление процедуре обработки прерывания 05h) при нажатии комбинации клавиш <Shift>+<Print Screen>. В этот момент содержимое текстового экрана выводится на принтер (что изначально и делает оригинальный обработчик BIOS). Теперь вы знаете, какое прерывание направляет содержимое экрана на принтер. Ничего не мешает программисту установить свой обработчик на прерывание 05h. Мы это и делаем в нашем резиденте (процедура Int_05h_proc).

После загрузки резидента в память пользователь нажимает комбинацию клавиш <Shift>+<Print Screen> и... содержимое экрана выводится не на принтер, а сохраняется в текстовый файл в текущем каталоге. Имя файла — Screen.txt. Это работает наша процедура! Абсолютно ничего сложного в ней нет. Далее мы рассмотрим в двух словах принцип работы нашего обработчика прерывания 05h. Дополнительную информацию вы найдете в файле-приложении.

Как вы уже знаете, один символ на экране занимает 2 байта: символ и его атрибут. Атрибуты-то нам не надо сохранять в текстовом файле! Поэтому их нужно "отсеять", оставив только символы. Для временного хранения символов мы используем память первой видеостраницы. Затем, вызвав функцию 40h прерывания 21h, мы сохраняем отделенные от атрибутов символы в текстовый файл Screen.txt. Все! Можно смотреть этот файл в любом текстовом редакторе...

15.6.2. Аппаратное прерывание 09h

Как уже упоминалось, это прерывание вызывается в случае, если пользователь нажал или отпустил какую-нибудь клавишу. В компьютере существуют порты ввода/вывода, которые сейчас мы подробно рассматривать не будем. При нажатии или отпускании клавиши в порт 60h заносится ее скан-код. Чтобы получить из порта число, нужно пользоваться оператором in.

Следует отличать ASCII-код от скан-кода. Одна из функций прерывания 09h — преобразование скан-кода в ASCII-код. Для чего это нужно — рассмотрим позже. Сейчас главное — уловить принцип работы разных прерываний (листинг 15.9).

Листинг 15.9. Обработчик прерывания 09h

```
...
(1) Int_09h_proc proc
(2)     pusha
(3)     in al,60h
(4)     cmp al,58h
(5)     jne No_F12

(6)     xor cs:Num_status,1
(7) No_F12:
(8)     popa
(9)     jmp dword ptr cs:[Int_09h_vect]

(10) Int_09h_vect dd ?
(11) Int_09h_proc endp
...
...
```

Скан-код клавиши <F12> — 58h (полный список скан-кодов клавиш см. в *приложении 3*). Мы вначале получаем из порта 60h скан-код нажатой клавиши. Если это <F12>, то инвертируем нулевой бит некой переменной Num_status и передаем управление оригинальному обработчику 09h. Если мы не передадим управление оригинальному обработчику, то пользователь не сможет работать с клавиатурой, т. к. прерывание 09h заносит в буфер клавиатуры ASCII-код нажатой клавиши. Но мы-то этого в нашем обработчике не делаем! Можно, конечно, самому написать подобную процедуру, но мы не ставили задачу сделать это в данной главе.

Что же касается "тайнственной" переменной Num_status, мы рассмотрим ее чуть позже...

15.6.3. Аппаратное прерывание 1Ch

Это прерывание примечательно тем, что оно само по себе вызывается примерно 18,2 раза в секунду. Если мы напишем какую-нибудь процедуру и установим на нее вектор прерывания 1Ch, то она будет автоматически вызываться указанное количество раз.

В нашем случае процедура Int_1Ch_proc "просматривает" содержимое нулевой страницы и заменяет символы "A" и "a" на "O" и "o". Причем все это происходит с периодичностью 18,2 раза в секунду. Начало данной процедуры такое:

```
...
cmp cs:Num_status,0
```

```
jnz Go_1Ch  
...  
Bот она, эта "тайная" переменная Num_status. Как видите, процедура  
Int_1Ch_proc проверяет, равна ли она нулю или нет. И если не равна, то переходит  
на метку Go_1Ch. А там:  
...  
Go_1Ch:  
jmp dword ptr cs:[Int_1Ch_vect]  
...
```

Просто передача на оригинальный обработчик 1Ch. Если переменная Num_status равна единице, то наш обработчик моментально передаст управление на старый вектор, т. е. символы на экране заменяться не будут. Теперь вспомним про прерывание 09h, которое меняло переменную Num_status при нажатии клавиши <F12>. Причем меняло командой xor. А что именно делает xor — мы рассматривали ранее в этой главе...

15.7. Резюме

Итак, ваша задача сейчас — не разобрать программу полностью, а понять принцип работы прерываний, причем на довольно-таки сложном примере. Но если вы досконально разберетесь с материалом данной главы, то в дальнейшем вам будет очень просто постигать новое.



Глава 16

Принципы работы отладчиков

В этой главе мы рассмотрим принципы работы отладчиков на примере AFD Pro. Данная тема вынесена в отдельную главу, т. к. это действительно важно. И вот почему:

- отладчик для программиста — это основной и очень важный инструмент, т. к. тестирование написанных на ассемблере программ осуществляется только с его помощью. Без отладчика найти ошибку в программе очень и очень проблематично;
- при необходимости исследовать чужую программу помочь профессионального программиста со знанием ассемблера просто необходима. Иного пути, кроме как дизассемблировать программу или запустить ее под отладчиком, просто не существует.

Чтобы продолжить изучение ассемблера, вам необходимо вооружиться отладчиком AFD Pro. Если до настоящего момента вы его не установили, то мы настоятельно рекомендуем это сделать сейчас. Скачать программу можно с сайта <http://www.Kalashnikoff.ru>.

16.1. Как работает отладчик

Вы задумывались над тем, как отладчик выполняет программу? Можно ли его обмануть? Естественно, уважаемые читатели! Обмануть отладчик на ассемблере очень просто. Мы с вами уже знаем два способа:

- перенести стек в тело программы;
- прочитать программу заново с диска и записать в память по месту ее загрузки.

Однако второй способ мы до сих пор подробно не рассмотрели. Поэтому сделаем это прямо сейчас.

Почему же отладчик работал неверно, если программа перечитывала саму себя в память? Давайте разберемся сперва с работой самого отладчика.

16.1.1. Прерывание 03h

Вы уже знаете, что существует ряд аппаратных прерываний, выполняющих те или иные функции при возникновении некоторых ситуаций. Например:

- при нажатии комбинации клавиш <Shift>+<Print Screen> вызывается прерывание 05h;

- при нажатии или отпускании любой клавиши вызывается прерывание 09h;
- прерывание же 1Ch вообще вызывается автоматически примерно 18,2 раза в секунду.

Все эти прерывания и их обработку мы подробно рассмотрели в *главе 15*.

Вообще номера прерываний от 00 до 1Fh "обслуживаются" BIOS (ПЗУ). Все остальные доступны программисту или операционной системе (программные прерывания). Например, MS-DOS использует номера от 20h до 2Fh (int 20h — выход из программы; int 21h — комплекс процедур и т. д.).

Как понять фразу "обслуживаются ПЗУ"? Это значит, что обработчики этих прерываний находятся в области ПЗУ — постоянном запоминающем устройстве, в то время как обработчики 20h—OFFh находятся в ОЗУ — оперативном запоминающем устройстве (т. е. в той области памяти, которая теряется при выключении/перезагрузке компьютера). Естественно, мы можем перехватить как прерывания ПЗУ (от 00 до 1Fh), так и все остальные, что мы уже делали.

Чем же примечательно прерывание 03h?

- Во-первых, оно используется для отладки программ, в частности, для работы AFD и CodeView.
- Во-вторых, машинный код этой команды занимает всего один байт — 0CCh, чего не скажешь о вызове других прерываний. Например, int 20h—0CDh 20h, т. е. два байта.
- В-третьих, обработчик прерывания 03h изначально содержит всего одну инструкцию: iret, т. е. при вызове данного прерывания происходит моментальный возврат. Можете, кстати, это проверить, принудительно вызвав данное прерывание в любом месте вашей программы. Ничего не произойдет.

ПРИМЕЧАНИЕ

Чтобы принудительно вызвать прерывание 03h, нужно записать в любом месте нашей программы int 3.

В процессе отладки программы отладчик перехватывает прерывание 03h. Проще говоря, устанавливает вектор (адрес) данного прерывания на некую свою процедуру. Выполняя одну команду (когда пользователь нажимает клавишу <F1> или <F2>), отладчик просто сохраняет следующий за текущей командой байт и вместо него вписывает 0CCh, т. е. int 3. Естественно, что на экран отладчик выводит измененную инструкцию в ее нормальном, первозданном виде, а не int 3. Рассмотрим это на примере. Возьмем такую простейшую программу, которая выводит один символ "Q" на экран в текущую позицию курсора (листинг 16.1).

Листинг 16.1. Вывод символа в текущую позицию курсора

```
cseg segment
assume cs:cseg, ds:cseg, es:cseg, ss:cseg
org 100h

Begin:
    mov ah,2
```

```

mov dl,'Q'
int 21h
ret

cseg ends
end Begin

```

В табл. 16.1 показана эта программа в том виде, как ее отображает отладчик.

Таблица 16.1. Шаг первый: программа только что загрузилась

Смещение	Инструкция ассемблера	Машинный код	Что на экране
0100h	Mov ah,2	0B402h	MOV AH,02
0102h	Mov dl,'Q'	0B251h	MOV DL,51
0104h	Int 21h	0CD21h	INT 21
0106h	ret	0C3h	RET

ПРИМЕЧАНИЕ

Курсивным начертанием выделены текущие команды, т. е. команды, которые выполнятся после нажатия клавиши <F1>/<F2> в AFD.

В колонке **Смещение** указано, по какому смещению находится в памяти команда. В колонке **Инструкция ассемблера** — реальные ассемблерные команды, расположенные по соответствующему смещению. Реальные ассемблерные команды — команды, которые на самом деле находятся в памяти по соответствующему смещению (читайте дальше — будет понятно). В колонке **Машинный код** — машинный код команды в шестнадцатеричной системе из колонки **Инструкция ассемблера**. Колонка **Что на экране** отражает то, что отладчик показывает нам на экране, а также оригинальный код команд.

ПРИМЕЧАНИЕ

Посмотреть машинные коды соответствующих команд ассемблера можно в любом отладчике, а лучше — в Hacker's View.

Итак, запускаем нашу программу под отладчиком AFD. Команда `mov ah,2` расположится по адресу 100h, а `mov dl,'Q'` — 102h (см. табл. 16.1). Естественно, что отладчик сразу не даст программе работать, а просто загрузит ее в память и выведет на экран инструкции ассемблера.

Допустим, пользователь нажимает клавишу <F2>. AFD запоминает 1 байт по адресу 102h (первый байт следующей за `mov ah,2` команды; в нашем случае — 0B2h), записывает на его место 0CCh, т. е. команду `int 3`, и выполняет инструкцию `mov ah,2`. После этого процессор выполнит не команду `mov dl,'Q'`, а `int 3` — вызовет прерывание 03h, адрес которого указывает на определенную процедуру обработки AFD (отладчик-то перехватывает это прерывание непосредственно после

загрузки!). Когда пользователь первый раз нажимает клавишу <F1>, получается следующее (табл. 16.2).

Таблица 16.2. Нажали клавишу <F2> первый раз

Смещение	Инструкция ассемблера	Машинный код	Что на экране	
0100h	mov ah, 2	0B402h	MOV AH, 02	0B402h
0102h	int 3	0CCh	MOV DL, 51	0B251h
0103h	push cx	51h	—	—
0104h	int 21h	0CD21h	INT 21	0CD21h
0106h	ret	0C3h	RET	0C3h

Что делает процедура обработки прерывания 03h отладчика?

Допустим, текущая команда находится по адресу 0102h (см. табл. 16.2). Пользователь нажимает <F1>/<F2>. Прерывание 03h делает следующее:

- Сохраняет все изменяемые отладчиком регистры в памяти.
- Восстанавливает сохраненный байт (0B2h) по адресу 102h.
- Высчитывает количество байтов следующей команды (`mov dl, 'Q'` = 2 байта = 0B251h).
- Получив адрес следующей за `mov dl, 'Q'` команды (у нас — 104h), заносит по этому адресу число 0CCh (т. е. `int 3`), предварительно сохранив затертый байт (у нас — 0CDh) в своей переменной.
- Выполняется инструкция `mov dl, 'Q'`, а за ней же сразу `int 3`, которая и передаст управление отладчику (процедуре обработки прерывания 03h).
- Процедура обработки 03h изменяет кое-что на экране.
- Выполняет некоторые другие действия (все зависит от конкретного отладчика).
- И ждет от пользователя дальнейших указаний.

Когда пользователь второй раз нажмет клавишу <F2>, получится следующее (табл. 16.3).

Таблица 16.3. Пользователь нажал клавишу <F2> второй раз

Смещение	Инструкция ассемблера	Машинный код	Что на экране	
0100h	mov ah, 2	0B402h	MOV AH, 02	0B402h
0102h	mov dl, 'Q'	0B251h	MOV DL, 51	0B251h
0104h	int 3	0CCh	INT 21	0CD21h
0105h	and bx, ax	21C3h	RET	0C3h
0107h	"мусор"	"мусор"	"мусор"	"мусор"

Обратите внимание, какой машинный код находится по адресу 0105h—21C3h. Перед ним, по адресу 0104h, находится код 0CCh (`int 3`).

Как вы уже, видимо, обратили внимание, команда ассемблера может занимать 1, 2 и более байтов. Например, `mov ah, 2` — `0B402h` — 2 байта; `ret` — `0C3h` — 1 байт. Если мы "вручную" в процессе работы рассматриваемой нами программы заменим `0B4h`, скажем, на `0C3h`, то вместо `mov ah, 2` получим инструкцию `ret` (машинный код которой `0C3h`). Но команда `ret` однобайтовая, в отличие от `mov ah, 2`. Куда же денется второй байт команды `mov ah, 2` (`02h`)?

Процессор просто поймет `02h` как `add dh, [CD51+BP+SI]`. Но поскольку данная команда занимает 4 байта, то процессор "присоединит" к `02h` еще и `0B2h`, `51h`, `0CDh`. Все, код начал путаться... Вот именно это и происходит по смещению `105h` в табл. 16.3. Процессор распознал `21h` как инструкцию `and`. Но после `and` должны идти приемник и источник (например, `and ax, 11b`). Следовательно, процессор возьмет следующий за `21h` байт, которым будет `0C3h`. `21C3h` как раз и есть `and bx, ax!` Отсюда и появляется по смещению `103h` какой-то `push cx` в табл. 16.2.

Вывод: если вы "на лету" меняете код программы, имейте в виду, что однобайтовую команду нужно менять на однобайтовую, двухбайтовую на двухбайтовую и т. д. Можно также менять две однобайтовые на одну двухбайтовую и пр. А можно, изменив всего 1 байт, сделать всю программу (или ее часть) другой. Например, в Hacker's View вы видите один код, а выполняет программа совсем другие действия.

Почему же тогда отладчик выполняет программу корректно, даже после того, как код "искривился"? Дело в том, что обработчик прерывания `03h` отладчика восстанавливает код, который был изменен. После нажатия клавиши `<F2>` третий раз мы получим буквально следующее (табл. 16.4).

Таблица 16.4. Пользователь нажал клавишу <F2> третий раз

Смещение	Инструкция ассемблера	Машинный код	Что на экране	
0100h	<code>mov ah, 2</code>	<code>0B402h</code>	<code>MOV AH, 02</code>	<code>0B402h</code>
0102h	<code>mov dl, 'Q'</code>	<code>0B251h</code>	<code>MOV DL, 51</code>	<code>0B251h</code>
0104h	<code>int 21h</code>	<code>0CD21h</code>	<code>INT 21</code>	<code>0CD21h</code>
0105h	<code>int 3</code>	<code>0CCh</code>	<code>RET</code>	<code>0C3h</code>

В данном случае получилось все корректно: команды `ret` и `int 3` занимают по одному байту. В табл. 16.5 показано, что происходит после четвертого нажатия клавиши `<F2>`.

Таблица 16.5. Нажали клавишу <F2> четвертый и последний раз

Смещение	Инструкция ассемблера	Машинный код	Что на экране	
0100h	<code>mov ah, 2</code>	<code>0B402h</code>	<code>MOV ah, 02</code>	<code>0B402h</code>
0102h	<code>mov dl, 'Q'</code>	<code>0B251h</code>	<code>MOV dl, 51</code>	<code>0B251h</code>
0104h	<code>int 21h</code>	<code>0CD21h</code>	<code>INT 21</code>	<code>0CD21h</code>
0105h	<code>ret</code>	<code>0C3h</code>	<code>RET</code>	<code>0C3h</code>

Отладчик сообщает: "Program terminated OK" ("Программа корректно завершилась").

16.2. Способы обойти отладку программы

Как же нам сделать так, чтобы AFD не смог производить отладку нашей программы?

Очень просто. Рассуждаем: AFD перехватывает прерывание 03h, затем вставляет 0CCh (int 3) после каждой команды. После этого процедура обработки прерывания 03h останавливает нашу программу.

Рассмотрим один из способов. Нам нужно записать в самое начало обработчика прерывания 03h команду `iret`. Получить адрес обработчика того или иного прерывания можно при помощи функции 35h прерывания 21h. Однако можно получить и поменять адрес, не прибегая к помощи операционной системы, хотя Microsoft делать так не советует. Но мы все равно рассмотрим такой способ.

16.2.1. Таблица векторов прерываний

Все текущие адреса обработчиков прерываний находятся в так называемой *таблице векторов прерываний*, которая расположена по адресу 0000:0000h. Как вы знаете, адрес обработчика любого прерывания занимает 4 байта: сегмент (2 байта) и смещение (2 байта), причем первые 2 байта — это смещение, а вторые — сегмент (мы уже знаем, что все данные в компьютере хранятся в обратном порядке, включая расположение "сегмент:смещение"). Таким образом, адрес нулевого прерывания будет расположен по адресу 0000:0000h, первого — 0000:0004h, второго — 0000:0008h и т. д. В листинге 16.2 приведен пример чтения адреса обработчика прерывания 21h напрямую из таблицы векторов прерываний.

Листинг 16.2. Получаем адрес вектора прерывания из таблицы векторов прерываний

```
...
xor ax,ax
mov es,ax          ;Аннулируем es
mov bx,es:[21h*4]   ;В bx — смещение
mov es,es:[21h*4+2] ;В es — сегмент

;Сохраним адрес прерывания 21h на будущее
mov Int_21h_offset,bx ;Сохраним смещение
mov Int_21h_segment,es ;Сохраним сегмент

;Вызываем прерывание 21h
mov ah,2
mov dl,'!'
pushf             ;Зачем здесь pushf — вы знаете...
call dword ptr [Int_21h_offset] ;Равносильно int 21h
...
```

```
Int_21h_offset dw ?
Int_21h_segment dw ?
...
```

В целом, пример не нуждается в пояснениях, т. к. все операторы мы уже рассматривали. Однако стоит обратить внимание на следующие строки:

```
mov bx,es:[21h*4]
mov es,es:[21h*4+2]
```

Ассемблер, в отличие от языков высокого уровня, вычислит выражение в квадратных скобках всего один раз, если в них находятся только числа, а не регистры. То есть процессор, выполняя приведенные выше строки, не станет постоянно вычислять, сколько будет $21h*4+2$. Давайте разберемся, как такое возможно.

Для начала вычислим сами: $21h*4+2=134$ ($86h$). Подобные выражения, встречающиеся в программе, вычисляются на стадии ассемблирования, т. е. программой-ассемблером. Запустив программу под отладчиком, мы увидим, что строка `mov es,es:[21h*4+2]` будет отображаться как `mov es,es:[86h]`. Учитывая это, можно смело записывать, например, так:

```
mov ax,(23+5)*3
mov cx,34h+98/2
```

не беспокоясь за то, что подобные выражения будут постоянно вычисляться процессором, тем самым замедляя работу.

А зачем это нужно? Неужели программисту самому сложно посчитать и записать результат? Ответ прост: для удобства и наглядности в процессе написания программы. В нашем случае мы сразу видим, что в `es` заносится сегмент прерывания $21h$:

```
mov es,es:[21h*4+2] ;21h — прерывание; +2 — сегмент.
```

А если бы мы посчитали и сразу записали результат — `mov es,es:[86h]` — то нам пришлось бы тогда при просмотре кода своей программы когда-нибудь в будущем постоянно вычислять, вспоминать, что именно мы загружаем в `es`. Можно, конечно, делать пометки в виде комментариев. Но это уж на ваше усмотрение: кому как удобнее...

Следует иметь в виду, что выражения типа `mov ax,[bx+di]` остаются как есть, т. е. вычисление значения суммы регистров `bx` и `di` будет производиться постоянно, каждый раз при проходе процессором этого участка кода! Откуда знать ассемблер-программе, какие значения находятся в регистрах `bx` и `di` в момент ассемблирования? В данном выражении в `ax` поместится сумма чисел, которые в процессе выполнения этой команды находятся в регистрах `bx` и `di`.

А с какой целью мы перед $[21h*4+2]$ указываем регистр `es`? Конечно, можно опустить `es`, но тогда мы получим не адрес прерывания $21h$, а что-то совсем другое. Регистр `es` указывает на то, что нужно загрузить слово в `es`, которое расположено по смещению $21h*4+2$, причем не из какого-нибудь сегмента, а именно из того, который указан в `es`. Поэтому мы, собственно, и загружали в начале ноль (т. е. сегмент таблицы векторов прерываний) в `es` (листинг 16.3).

Листинг 16.3. Заносим в es сегмент таблицы векторов прерываний

```

...
xor ax,ax
mov es,ax          ;Аннулируем es
...
mov es,es:[21h*4+2] ;В es — сегмент
...

```

16.3. Практика

Теперь вернемся к отладчику. Итак, нам нужно занести в первый байт процедуры обработки прерывания 03h команду `iret`, тем самым "вырубив" отладчик. Прежде получим адрес обработчика прерывания 03h (листинг 16.4).

Листинг 16.4. Получаем вектор прерывания 03 из таблицы векторов прерываний

```

...
xor ax,ax
mov es,ax
mov bx,es:[03h*4]    ;В bx — смещение
mov es,es:[03h*4+2]  ;В es — сегмент
...

```

Теперь занесем `iret` (машиинный код `iret` — `0CFh`) в самое начало обработчика:

```
mov byte ptr es:[bx],0CFh
```

После выполнения этой команды отладчиком, при вызове прерывания 03h произойдет моментальный возврат. Следовательно, отладчик не сможет получить управление после выполнения каждой команды отлаживаемой программы. Поэтому весь остальной код (вплоть до `int 20h`) выполнится до конца без остановок (см. файл `dbg16_01.asm`).

На команде же `int 20h` отладчик остановится и сообщит о нормальном завершении программы, т. к. прерывание 20h также перехватывается им после загрузки, дабы контролировать завершение работы отлаживаемой программы.

Для того чтобы лучше понять работу отладчика в файле-приложении, найдите `"!DEBUG.ASM"`. Что нужно сделать:

1. Изучите описания в данном файле.
2. Получите `"!DEBUG.COM"`.
3. Запустите его в режиме эмуляции DOS (**Пуск | Выполнить | cmd**) или из файловой оболочки типа Far Manager.
4. Запустите файл под отладчиком AFD или CodeView.
5. *Обратите внимание, что после команды `call` следует пор.*
6. В AFD нажмите 4 раза клавишу `<F2>`, до тех пор пока не появится надпись "Program terminated OK".

7. В CodeView нажмите 4 раза клавишу <F10>, до тех пор пока не появится надпись "Process XXXX terminated normally".
8. Выйдите из отладчика.
9. Запустите снова файл под отладчиком. Все внимание на команду `int 3`, если таковая имеется!

Вот вам и доказательства всего вышеизложенного...

Усвоив приведенный в данной главе материал, вы без труда поймете, почему при чтении программы самой себя в память отладчик работает неверно. Причем что именно он делает? Это показано в табл. 16.6.

Таблица 16.6. Замена отладчиком инструкции `ret` на `int 3`

Смещение	Инструкция ассемблера	Машинный код	Что на экране
0120h	<code>int 21h</code>	0CD21h	INT 21
0122h	<code>int 3</code>	0CCh	RET

Здесь приведен фрагмент кода программы, которая считывает себя в память. Предположим, что полный код этой программы в листинге 16.5.

Листинг 16.5. Полный код программы

```
...
mov ah,3Fh
mov bx,Handle
mov cx,offset Finish-100h
mov dx,offset Begin
int 21h      ;Эта строка...
ret          ;...и эта строка приведена в таблице 16.6
...

```

Внимательно посмотрите на колонки **Инструкция ассемблера** и **Что на экране**. Подумайте, что произойдет, если мы колонку **Что на экране** скопируем в колонку **Инструкция ассемблера**?

В этом случае просто затрется `int 3` (0CCh). То же самое происходит, если мы читаем свою программу в память с диска. На диске ведь вместо `int 3` будет `ret`! А как отладчик получит управление, если `int 3` в памяти затирается (перезаписывается)?

На компакт-диске, в каталоге 016, вы найдете файлы типа `dbg16_0?.asm`. Посмотрите их, почитайте описания, запустите под отладчиком. Однако обращаем ваше внимание, что некоторые из этих программ могут зависать при их отладке под ОС Windows 9x.



Глава 17

Заражение файлов вирусом

17.1. Определение текущего смещения выполняемого кода

Как уже упоминалось в прошлых главах, наш вирус теряется в адресах, если он дописывается после кода заражаемого файла. Чтобы избежать этого, мы переносили код вируса в область 7-й видеостраницы, в которой он и выполнялся. Далее приведен один из вариантов определения текущего смещения выполняемого кода. Причем этот вариант работает в любом случае, независимо от того, переместил ли программист код собственной программы куда-либо по другому смещению или нет. Помните, когда мы пытались записывать вирус в хвост файла, мы акцентировали ваше внимание на том, что команды вида `mov dx, offset Message` будут выглядеть в отладчике как `mov dx, 400h`. При перемещении вируса в хвост "программы-жертвы" мы, тем самым, перемещаем строку `Message`. В итоге получается, что по смещению `400h` будет не наша строка, а что-то другое.

Получить же значение регистра `ip` обычной командой `mov` не удается. Поэтому нам следует прибегнуть к более хитрым способам. Например, к способу из листинга 17.1.

Листинг 17.1. Получение значения регистра `ip`

```
...
;Якобы переходим на процедуру, при этом в стеке сохраняется адрес следующей
;за командой call инструкции (т. е. pop ax).
call Label_1
Label_1:
;И сразу же достаем этот адрес из стека.
pop ax
;Теперь в ах находится текущий адрес (смещение).
...
```

Почему команда `call Label_1` всегда заносит в стек текущее смещение, выяснить можно в отладчике (рис. 17.1, 17.2 и 17.3). Тем не менее, мы позже рассмотрим различие `mov dx, offset Message` и `call Label_1`. Мы выясним, почему в первом случае в `dx` заносится константа (постоянное число), а во втором случае — "плаывающее". Так или иначе, считайте, что в приведенном выше примере мы получаем всегда текущее смещение метки `Label_1`.

По этому принципу мы теперь получим текущее смещение в нашем вирусе, тем самым, усложнив задачу и — как следствие — усовершенствовав сам код вируса. Но об этом в следующем разделе.

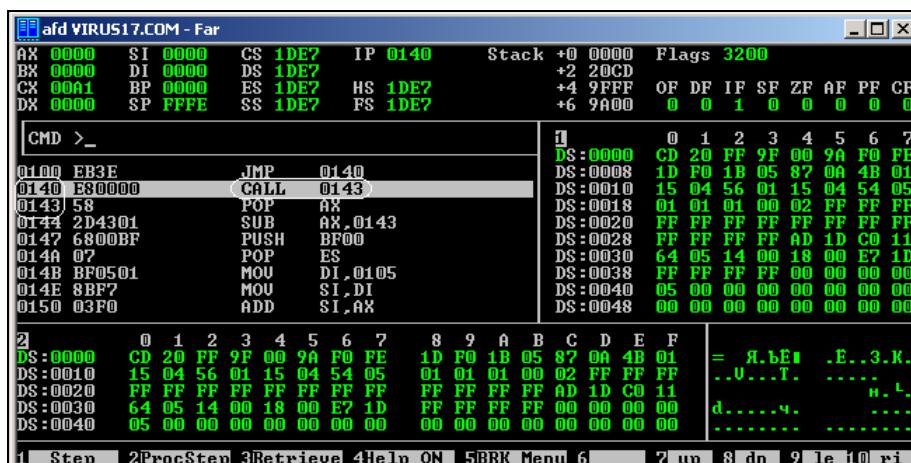


Рис. 17.1. Перед вызовом лжеподпрограммы

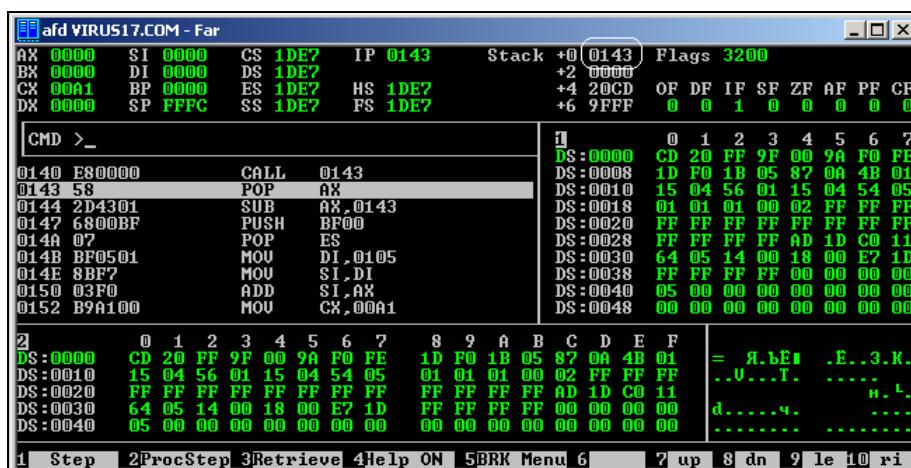


Рис. 17.2. В стеке — смещение следующей команды

17.2. Вирус

Итак, открываем файл virus17.asm. Сразу же за меткой `Init` мы получаем смещение, по которому расположена метка `Get_ip` (см. разд. 17.1). Из этого адреса нужно вычесть смещение метки `Get_ip`.

Допустим, мы запускаем вирус первый раз. То есть все смещения содержат верные значения. Также допустим, что метка `Get_ip` находится по адресу 0203h (листинг 17.2).

Листинг 17.2. Получаем текущее смещение в коде

```
...
(1) [1234:0200h]      call Get_ip
(2) [1234:0203h]  Get_ip:
(3) [1234:0203h]      pop ax
(4) [1234:0204h]      sub ax,offset Get_ip
...
...
```

На рис. 17.3 продемонстрировано, как приведенный выше фрагмент кода выглядит в отладчике в случае, если вирус запускается первый раз.

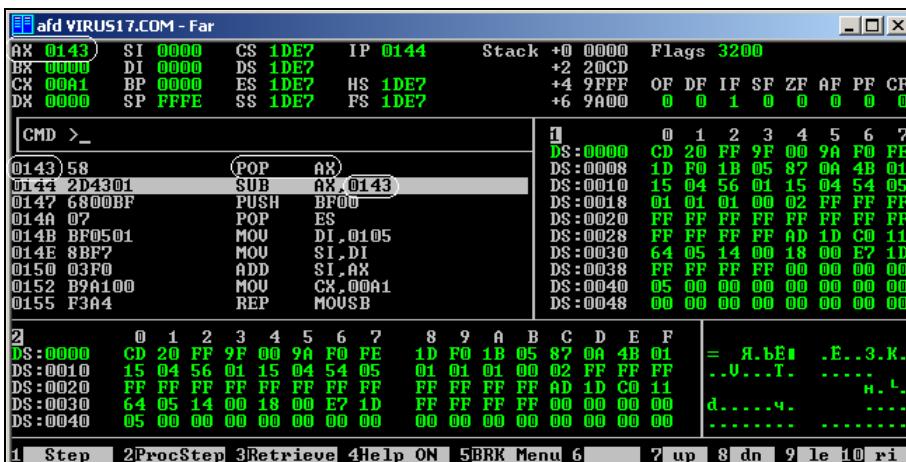


Рис. 17.3. В ax — текущее смещение

Строка (1) заносит в стек число 0203h, а (3) — достает из стека 0203h. Таким образом, мы оставили стек выровненным и получили текущий адрес (смещение) выполняемого кода. Сегмент, как вы знаете, можно получить так: `mov ax, cs`, чего не скажешь о `ip`. *Загрузка и чтение напрямую числа в/из ip не допускается!*

Далее происходит следующее: в строке (4) вычитаем из 203h смещение метки `Get_ip` в памяти. В отладчике строка (4) будет всегда выглядеть так:

```
sub ax,0203h ; sub ax,offset Get_ip → ax=203h-203h=0
```

Теперь представим, что код вируса был перенесен самой программой в процессе заражения в хвост некоторого файла, размером 1000h байт (обратите внимание на смещения в скобках). Тот же самый код будет располагаться по такому адресу (листинг 17.3).

Листинг 17.3. Код перемещен в хвост "файла-жертвы"

```
...
(1) [1234:1200h]      call Get_ip
(2) [1234:1203h]  Get_ip:
(3) [1234:1203h]      pop ax
(4) [1234:1204h]      sub ax,offset Get_ip
...
...
```

Тогда строка (1) заносит в стек число 1203h. Pop ax достает его из стека. Затем выполняется следующее действие:

```
sub ax,203h ;sub ax,offset Get_ip → ax=1203h-203h=1000h
```

Теперь в ax находится размер "файла-жертвы", т. е. 1000h. Получили то, что нам и нужно!

Вполне возможно, что принцип не до конца понятен. В таком случае особо не переживайте, т. к. в главе 20, когда вирус будет полностью работоспособным, вы увидите на практике, как именно подставляются необходимые нам значения. Сейчас просто можете наблюдать за тем, как мы "играем" с адресами.

Итак, теперь в ax размер "файла-жертвы". Чтобы получить реальное смещение того или иного адреса, нам достаточно прибавить к уже имеющемуся в регистре "ненормальному" смещению то, что находится в ax, т. е. длину "файла-жертвы". Вот что мы делаем в вирусе (листинг 17.4).

Листинг 17.4. Вычисление реального смещения

```
...
;Скопируем наш вирус в область 7-й видеостраницы, т. е. сделаем то,
;что уже делали в предыдущей главе, только "правильней".
push 0BF00h
pop es
;es – сегмент, куда будем перемещать код вируса,
mov di,offset Open_file
;di – смещение (адрес самой первой процедуры (см. файл-приложение))

mov si,di
;si должен содержать РЕАЛЬНЫЙ адрес (смещение), т. к. мы пока еще
;в сегменте "файла-жертвы"...
add si,ax
```

```

mov cx,offset Finish-100h
;т. е. cx = длина нашего вируса в байтах

rep movsb      ;Теперь в памяти две копии нашего вируса
...

```

Обратите внимание, как мы получаем адрес первой процедуры в памяти. Понятно, что после директивы `offset` не обязательно должен идти адрес метки или строки, например, для вывода с помощью функции `09h`. А так как метки и директивы (инструкции программы-ассемблера) не занимают памяти, то в `di` мы занесем адрес (смещение) первого байта процедуры `Open_file`, т. е. `mov ax,3D02h` (но не саму команду `mov`, а ее смещение).

Копию кода в 7-й видеостранице мы сделали. Теперь можно и переходить ("прыгать") на нее. Но в данном случае мы не будем использовать оператор `jmp`. Давайте чуть-чуть усложним задачу. Прежде чем разбирать, вспомним об операторах возврата из подпрограмм и прерываний: `ret`, `retf` и `iret` (листинг 17.5).

Листинг 17.5. Переход по другому адресу с помощью `retf`

```

...
(1)    mov bx,offset Lab_return
(2)    add bx,ax
(3)    push cs
(4)    push bx
(5)    retf

(6) Lab_jmp:
...

```

На рис. 17.4 и 17.5 показано, что процессор передает управление копии кода вируса совсем в другой сегмент (следите за регистром `cs`).

Теперь, если вы разобрались с `retf`, то понять принцип работы приведенных выше строк труда не составит. Команда `retf` достает из стека 2 слова (4 байта): смещение и сегмент для возврата в то место, откуда дальняя процедура вызывалась. Мы же имитируем вызов процедуры. Поверьте, процессору все равно, откуда вызывалась процедура и куда нужно возвращаться. Более того, он вообще не следит за тем, вызывалась ли процедура. Однако при пользовании такими методами нужно быть предельно внимательным, очень тщательно все подготовить и обязательно проверить несколько раз.

При вызове дальней процедуры в стек заносится сперва смещение, а затем сегмент для возврата. Делаем это в строках (1)–(4). Учитывая, что в компьютере данные хранятся наоборот, но стек растет снизу вверх, то в памяти сегмент и смещение будут располагаться именно так, как нам нужно. В строках (1), (2) мы

получаем смещение для возврата. К этому смещению и прибавляем размер "файла-жертвы", который находится в `ax`. Теперь в стеке у нас смещение и сегмент метки `Lab_jmp` (точнее, первого оператора, который находится после нее). На основе вышеизложенного постараитесь разобраться, что делает строка (5).

Вопрос: а как нам вернуться назад, в сегмент "файла-жертвы"? Ответ на этот вопрос можно найти в прилагаемом файле для практического изучения.

The screenshot shows the assembly code for the `VIRUS17.COM` file. The assembly window displays the following assembly code:

```

AX 0000  SI 01A6  CS 1DE7  IP 0166  Stack +0 BF00  Flags 3200
BX 0168  DI 01A6  DS 1DE7
CX 0000  BP 0000  ES BF00  HS 1DE7
DX 0000  SP FFF8  SS 1DE7  FS 1DE7

CMD >_
0163 6800BF  PUSH  BF00
0166 53          PUSH  BX
0167 CB          RET   Far
0168 0E          PUSH  CS
0169 1F          POP   DS
016A B41A        MOU   AH,1A
016C 33D2        XOR   DX,DX
016E CD21        INT   21
0170 E8B2FF      CALL  0125

DS:0000  CD 20 FF 9F 00 9A  F0  FE  1D F0 1B 05  87 0A 4B 01
DS:0010  15 04 56 01 15 04 54 05  01 01 01 00  02 FF FF FF
DS:0020  FF FF FF FF FF FF FF  AD 1D C0 11
DS:0030  64 05 14 00 18 00 E7 1D  FF FF FF FF FF 00 00 00 00
DS:0040  05 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00

1 Step 2 ProcStep 3 Retrieve 4 Help ON 5 BRK Menu 6 7 up 8 dn 9 le 10 ri

```

The assembly code consists of several instructions: `PUSH BF00`, `PUSH BX`, `RET Far`, `PUSH CS`, `POP DS`, `MOU AH,1A`, `XOR DX,DX`, `INT 21`, and `CALL 0125`. The stack dump shows the current state of the stack, which includes the pushed values and the return address `0125`.

Рис. 17.4. Пока еще в начальном сегменте

The screenshot shows the assembly code for the `VIRUS17.COM` file, now in a different segment. The assembly window displays the following assembly code:

```

AX 0000  SI 01A6  CS BF00  IP 0168  Stack +0 018D  Flags 3200
BX 0168  DI 01A6  DS 1DE7
CX 0000  BP 0000  ES BF00  HS 1DE7
DX 0000  SP FFFA  SS 1DE7  FS 1DE7

CMD >_
0167 CB          RET   Far
0168 0E          PUSH  CS
0169 1F          POP   DS
016A B41A        MOU   AH,1A
016C 33D2        XOR   DX,DX
016E CD21        INT   21
0170 E8B2FF      CALL  0125
0173 720A        JC    017F
0175 E8C6FF      CALL  013E

DS:0000  CD 20 FF 9F 00 9A  F0  FE  1D F0 1B 05  87 0A 4B 01
DS:0010  15 04 56 01 15 04 54 05  01 01 01 00  02 FF FF FF
DS:0020  FF FF FF FF FF FF FF  AD 1D C0 11
DS:0030  64 05 14 00 18 00 E7 1D  FF FF FF FF FF 00 00 00 00
DS:0040  05 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00

1 Step 2 ProcStep 3 Retrieve 4 Help ON 5 BRK Menu 6 7 up 8 dn 9 le 10 ri

```

The assembly code has changed significantly, reflecting the transition to a new segment. It includes `RET Far`, `PUSH CS`, `POP DS`, `MOU AH,1A`, `XOR DX,DX`, `INT 21`, `CALL 0125`, `JC 017F`, and `CALL 013E`. The stack dump shows the current state of the stack, including the return address `0125` and other stack data.

Рис. 17.5. Уже в новом сегменте кода

17.2.1. Первые байты "файла-жертвы"

Обратите также внимание, что мы создаем массив байтов "файла-жертвы", которые нужно восстанавливать после того, как вирус отработал:

```
First_bytes db 90h, 90h, 90h, 90h, 0CDh, 20h
```

Это — знакомые машинные коды:

- 90h — nop;
- 0CDh — int;
- 20h — 20h.

Из этого получаем программу:

```
nop
nop
nop
nop
int 20h
```

В данный массив будем заносить первые 6 байт "программы-жертвы", вместо которых запишем jmp на начало нашего вируса. Более того, эти байты будут записаны вместе с телом нашего вируса на диск, после "файла-жертвы". Они нужны для того, чтобы впоследствии, когда вирус отработает, восстановить их в памяти по адресу 100h и "прыгнуть" на него. Программа даже и не заподозрит, что кто-то перед тем, как она получила управление, уже что-то делал.

Изначально мы заполняем массив командами nop, а после них — int 20h. Это нужно для того, чтобы при первом запуске вируса корректно завершиться.

Первый запуск вируса — это когда мы только-только его скомпилировали (получили СОМ-файл) и запустили полученную программу (в нашем случае — это будет virus17.com). Она отработает: если найдет подходящий файл, то заразит его, восстановит первые 6 байт и передаст управление на метку 100h, полагая, что восстановила байты "файла-жертвы" и передала ему управление. Но на деле она просто закончит работу. А если мы не восстановим эти байты, но перейдем на метку 100h, тогда вирус будет работать бесконечно. Если же мы заразили какой-то файл и сохранили в переменной First_bytes первые байты зараженного файла, то мы просто передадим ей управление (рис. 17.6 и 17.7).

Если вы заглянули в файл-приложение, то заметили там не строку

```
First_bytes db 90h, 90h, 90h, 90h, 0CDh, 20h
```

a

```
First_bytes db 4 dup (90h), 0CDh, 20h
```

Директива `dup` (от англ. *duplicate*) дублирует число, которое находится в скобках столько раз, сколько указывает стоящая перед ней цифра. Сравните приведенные выше две строки. Однако следует иметь в виду, что этот массив занимает память на диске. Проще говоря, он увеличивает нашу программу на 6 байт.

Пример:

```
Array db 1500 dup ('1')
```

создаст массив `Array` размером 1500 байт, заполнив его по умолчанию цифрой 1, но увеличит программу на указанное количество байтов. Заполнение массива "еди-

ницей", а также любым другим символом, указанным в круглых скобках после `dup`, происходит на стадии ассемблирования.

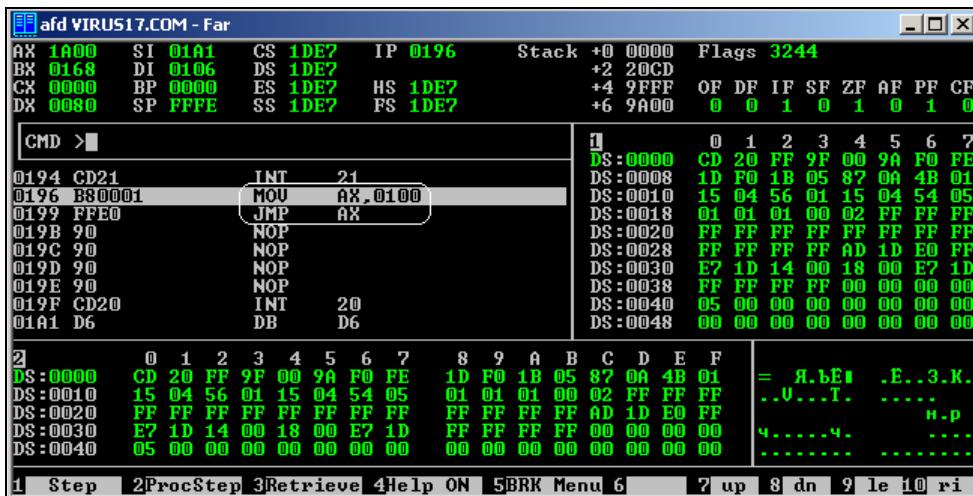


Рис. 17.6. Подготовка к переходу на адрес 100h

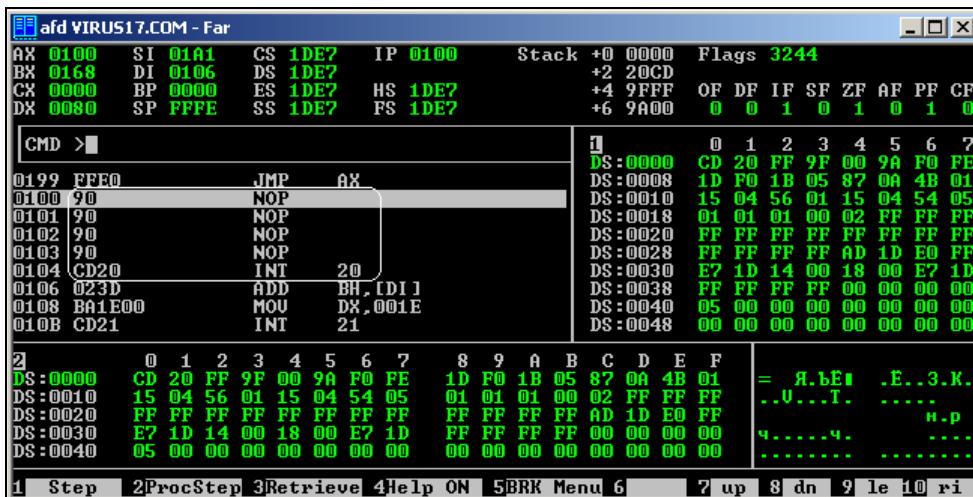


Рис. 17.7. Вот что в конце работы вируса находится по стартовому адресу 100h

17.2.2. Передача управления "файлу-жертве"

Обратите также внимание, как мы переходим на адрес 100h (передаем управление "файлу-жертве" после того, как вирус отработал) (листинг 17.6).

Листинг 17.6. Переход на адрес 100h

```
...
mov ax,100h
jmp ax      ;безусловный переход по адресу, указанному в ах
...
```

С вирусом пока достаточно. В главе 18 мы рассмотрим ошибку, которая была сознательно допущена в настоящей главе, с тем, чтобы вы сами попробовали ее обнаружить, а также продолжим писать оболочку.



Глава 18

Высокоуровневая оптимизация программ

18.1. Пример высокоуровневой оптимизации

Алгоритм перевода колонки/строки в линейный адрес для вывода символов методом прямого отображения в видеобуфер, приведенный в процедуре `Get_linear` файла `display.asm` главы 14, можно немного оптимизировать следующим образом (листинги 18.1 и 18.2).

Листинг 18.1. Неоптимизированный вариант (14 строк)

```
;Из файла display.asm, процедура Get_linear, глава 17
...
push ax      ;сохраним все используемые регистры
push bx
push dx

shl dl,1    ;математика: умножаем dl на 2 (dl=dl*2)
mov al,dh   ;в al - ряд,
mov bl,160  ;который нужно умножить на 160
mul bl     ;умножаем: al(ряд) *160; результат - в ax
mov di,ax   ;результат умножения в di

xor dh,dh  ;аннулируем dh
add di,dx  ;теперь в di линейный адрес в видеобуфере
pop dx     ;восстанавливаем регистры...
pop bx
pop ax
ret
...
```

Листинг 18.2. Оптимизированный вариант (13 строк)

```

...
push ax
push dx
xor ax,ax
xchg dh,al      ;Имеем: dx = dl, а ax = dh
mov di,ax
shl ax,6        ;dh * 64
shl di,4        ;dh * 16
add di,ax
add di,dx
shl di,1        ;di * 2
pop dx
pop ax
ret
...

```

То, что второй вариант короче на 1 строку — это мелочи. Не в этом дело. Он быстрее, т. к. используется команда сдвига `shl`, а не умножения `mul`, да и регистров задействовано меньше — только `ax`.

Далее также приведены формулы вычисления линейного адреса видеобуфера:

$$\text{Линейный адрес} = (\text{COL} \times 80 + \text{RAW}) \times 2.$$

Отсюда:

$$\text{Линейный адрес} = (\text{COL} \times 64 + \text{COL} \times 16 + \text{RAW}) \times 2.$$

Или:

$$\text{Линейный адрес} = ((\text{COL} \text{ shl } 6) + (\text{COL} \text{ shl } 4) + \text{RAW}) \text{ shl } 1.$$

Важно!

Команды `div` (деление) и `mul` (умножение) выполняются гораздо медленнее, чем операции сдвига.

Данный пример наглядно демонстрирует принцип высокого оптимизирования программ. Для того чтобы овладеть им в совершенстве, необходимо знать количество тактов, за которые выполняется та или иная команда.

18.2. Ошибка в главе 17

В главе 17 была умышленно допущена одна ошибка. Заметить эту ошибку могли только в том случае, если разобрали программу полностью.

С какой целью была допущена ошибка? Дело в том, что для того, чтобы узнать что-то новое, приходится столкнуться с чем-то сложным. В данном случае сложность заключалась в том, что программа не работала так, как она должна была ра-

ботать. Вы начинаете думать: почему так? Покопавшись в полученной из той или иной главы информации, изучив файл-приложение, вы вдруг находите ошибку! При этом, безусловно, вы испытываете положительные эмоции. Более того, если вы пытались найти эту ошибку, то это послужило вам хорошей практикой в исследовании чужого кода.

18.3. Оболочка Super Shell

Прежде всего, запустите полученный файл (sshell8.com) и посмотрите, какие задачи он выполняет, как реагирует на нажатие клавиш $<\text{Ctrl}>+<\text{F5}>$ и $<\text{Esc}>$. Теперь будем разбирать...

18.3.1. Передача данных процедуре через стек

Обратите внимание, каким образом теперь вызывается подпрограмма `Draw_frame`.

Этот метод называется передачей данных процедуре через стек. При программировании под Windows вызов всех процедур производится таким способом. В Windows нет понятия "прерывание". Есть понятие "системная функция". Самое главное для вас на данном этапе — понять принцип передачи данных в стеке. Хотя на самом деле все очень просто. Принцип таков: занося в стек, например, 20 байт, процедура, получившая управление с параметрами в стеке, должна самостоятельно их достать, чтобы выровнять стек. Каким образом?

Существует оператор `ret N`, где `N` — количество освобождаемых байтов из стека. В принципе, оператор `ret N` аналогичен `ret`, за исключением того, что он не только выходит из подпрограммы, но перед этим достает из стека определенное количество байтов. Давайте рассмотрим это на простом примере (листинг 18.3).

Листинг 18.3. Передача данных подпрограмме в стеке

```
...
(1)    push 123h
(2)    call Our_pr
(3)    pop ax

...
(4) Our_pr proc
      ...
(5)    ret
(6) Our_pr endp
...
```

Здесь мы вызываем процедуру `Our_pr` (2), предварительно занеся в стек некоторый параметр для данной процедуры (1). Что такое параметр? Например, для вывода

строки на экран с помощью функции `09h` мы должны передать данной функции параметр — адрес строки для вывода.

Процедура `Our_pr` отработает и завершится, при этом в стеке останется переданный ей параметр (число `123h`). Нам нужно вытащить его из стека, т. к. стек у нас остается не выровненным. Мы это и делаем в строке (3).

Вроде все понятно. Но что делать, если мы передаем не один, а 20 параметров? Каждый раз при вызове процедуры доставать их из стека? Громоздко, неудобно и медленно. Более того, нам нужно использовать какой-нибудь регистр, куда можно будет выгружать параметры из стека. Для решения подобных задач и существует оператор `ret n`, где `n` — количество высвобождаемых из стека байтов перед возвратом из подпрограммы. Вот как будет выглядеть приведенный ранее пример с использованием данного оператора (листинг 18.4).

Листинг 18.4. Использование оператора `ret n`

```
...
(1)    push 123h
(2)    call Our_pr
...
(3) Our_pr proc
...
(4)    ret 2
(5) Our_pr endp
...
```

Оператор `ret 2` вытащит из стека прежде всего адрес для возврата, а затем увеличит `sp` на 2, т. е. как бы "искусственно" достанет из стека данные, причем не за действуя никаких регистров.

Для чего нужно использовать метод передачи параметров через стек? Дело в том, что если той или иной процедуре необходимо передавать огромное число параметров, то:

- во-первых, не хватит регистров;
- во-вторых, придется заводить соответствующее количество переменных, где будут храниться параметры;
- в-третьих, это увеличивает размер программы;
- в-четвертых, уменьшит скорость работы.

Конечно, если процедуре нужно передать два-три параметра, то можно (и желательно) передавать их в регистрах. А если десять, как у нас при вызове `Draw_frame?`? В прошлый раз мы заводили три специальные переменные. Но теперь мы существенно усовершенствовали нашу процедуру. Мы уже передаем 10 параметров. Следовательно, будем передавать их в стеке (листинг 18.5).

Листинг 18.5. Передача параметров в стеке для подпрограммы Draw_frame

```
...
push 23          ;высота
push 78          ;ширина
push 1F00h        ;цвет
push offset Mess_head    ;надпись вверху
push 1E00h        ;ее цвет
push offset Mess_down    ;надпись внизу
push 1D00h        ;ее цвет
push 0            ;сообщение внутри рамки
push 0            ;его цвет
push 0            ;копировать ли экран?
call Draw_frame      ;рисуем рамку
...
...
```

Сразу отметим, что команды вида `push 23`, `push 0` толкают в стек 2 байта, а не 1. Теперь считаем количество занесенных в стек байтов:

```
push 23          - 2 байта
push 78          - 2 байта
push 1F00h        - 2 байта
push offset Mess_head    - 2 байта
... и т. д...
```

Итого: **- 20 байт**

Следовательно, процедура `Draw_frame` должна доставать из стека столько же байтов. Таким образом, выход из этой процедуры будет следующим:

```
ret 20
```

Надо также иметь в виду, что при вызове данной процедуры нужно будет всегда заносить в стек 20 байт (10 слов), даже если эти данные ей не нужны. Иначе мы оставляем стек не выровненным!

Обратите внимание, как мы передаем параметры процедуре рисования окна в нашей оболочке (рис. 18.1—18.3).

Возникает вопрос: как нам получить доступ в процедуре к занесенным в стек параметрам?

Для этой цели в ассемблере принято использовать регистр `bp`, который мы редко затрагивали в наших предыдущих примерах. Однако вспомним еще раз принцип работы стека.

Стек растет снизу вверх. Первый параметр, занесенный в стек — последний для нашей процедуры. И наоборот. Давайте сперва рассмотрим все на самом простом примере (листинг 18.6).

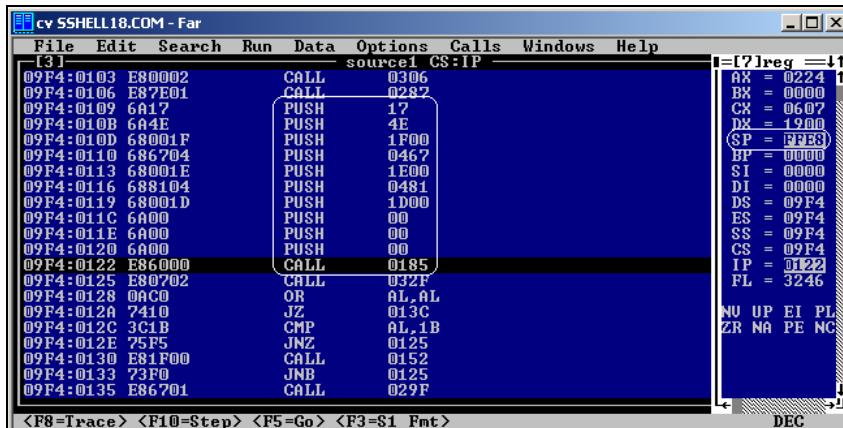


Рис. 18.1. Передача параметров в стеке

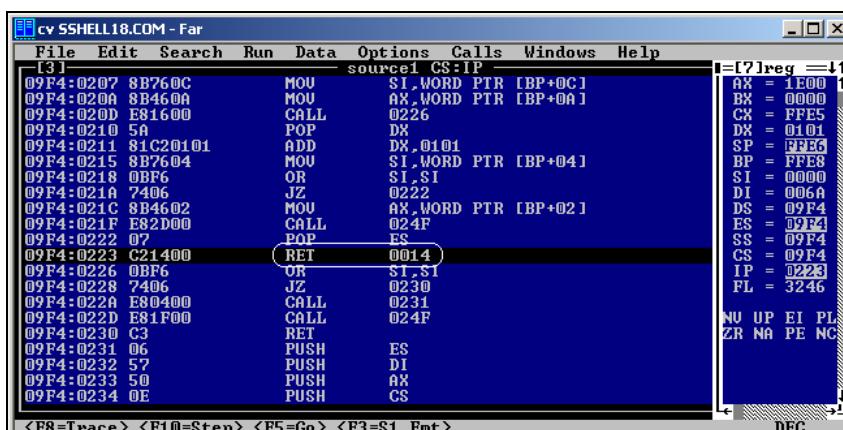


Рис. 18.2. Возврат из подпрограммы и выравнивание стека

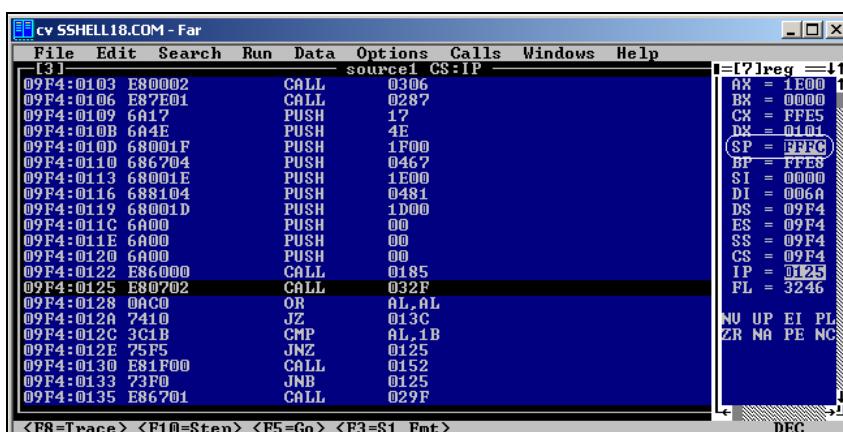


Рис. 18.3. После выполнения подпрограммы стек выровнен

Листинг 18.6. Использование регистра bp

```
...
(1)    push offset Message1
(2)    push offset Message2
(3)    call Print_string
```

...

```
(4) Print_string proc
(5)    mov bp,sp
(6)    add bp,2
(7)    mov ah,9
(8)    mov dx,[bp]
(9)    int 21h
(10)   mov dx,[bp+2]
(11)   int 21h
(12)   ret 4
(13) Print_string endp
```

...

```
(14) Message1 db 'Привет!$'
(15) Message2 db 'Это я!$'
```

...

Здесь мы вывели две строки на экран, используя дважды функцию 09h прерывания 21h. Это, конечно, ужасно, но для теоретического исследования пойдет. Как вы думаете, что произойдет?

Сперва выведется строка "Это я!", а затем — "Привет". Если у вас остался вопрос, почему именно в таком порядке, то еще обратим внимание на следующее: стек растет снизу вверх. Следовательно, первый параметр, занесенный в стек, будет последним. Стока (8) получает адрес Message2, а (10) — Message1. Очень важно помнить это при использовании данного метода! Получение параметров в стеке показано в отладчике на рис. 18.4.

Обратите внимание, что в самом начале процедуры (строки (5), (6)) мы заносим в bp текущее состояние стека, а затем увеличиваем bp на 2. Это необходимо для того, чтобы "перепрыгнуть" адрес возврата, который занесла инструкция в строке (3). Адрес возврата из процедуры сохраняет в стеке команда call после всех занесенных в него параметров. Следовательно, первый параметр и будет адресом возврата. Но он нашей процедуре вовсе не нужен в качестве параметра. Поэтому мы

просто увеличим `bp` на 2, установив его сразу на переданные параметры. Так даже удобней будет считать. И еще: не забывайте выходить из процедуры соответствующим образом, как в строке (12). Мы занесли 4 байта, нам и выйти нужно, используя `ret 4`.

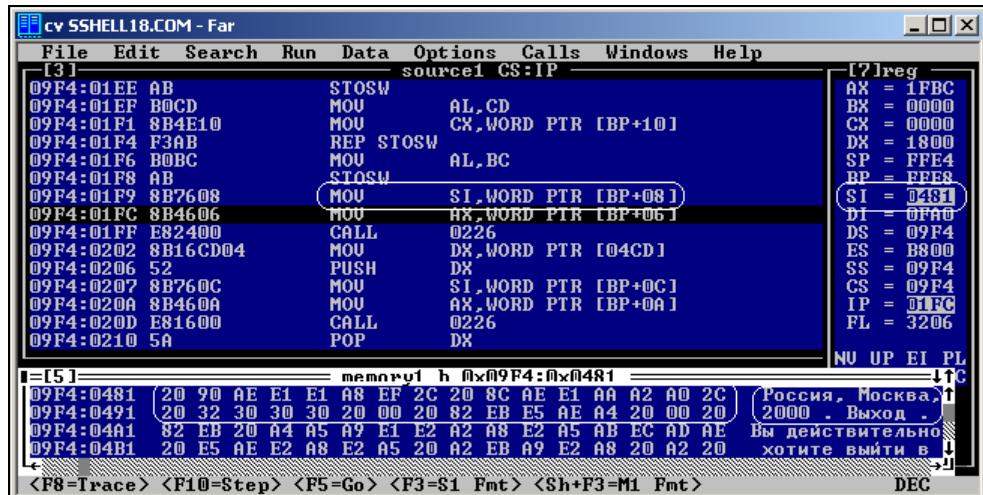


Рис. 18.4. Получение смещения строки из переданных в стеке параметров

"Ну, нет! Это ж как неудобно! Лучше я заведу 10, 50, 100 переменных и буду работать с ними, чем через стек. Пусть это и займет много байтов, зато удобно!" — скажете вы. Однако это далеко не так! Давайте попробуем разобраться дальше.

Для удобства работы со стеком (и не только с ним) используется директива `equ` (от англ. *equivalent* — эквивалент). (Помните `Finish EQU $?`) Вот именно с помощью данной директивы мы можем очень просто получить доступ к параметрам, занесенным в стек. Теперь внимательносмотрите, как в листинге 18.7 используется директива `equ`.

Листинг 18.7. Использование директивы `equ`

```
...
(1)    push offset Message1
(2)    push offset Message2
(3)    call Print_string

...
(4) Print_string proc
(5)    mov bp,sp
(6)    add bp,2
```

```

(7)    mov ah, 9
(8)    mov dx,Mess1
(9)    int 21h
(10)   mov dx,Mess2
(11)   int 21h
(12)   ret 4
(13)  Print_string endp

...
(14) Message1 db 'Привет!$'
(15) Message2 db 'Это я!'
(16) Mess1 equ [bp+2]
(17) Mess2 equ [bp]
...

```

В данном случае строки будут выведены на экран в следующем порядке: "Привет", а затем "Это я". Строки (16), (17) места в памяти не занимают. При ассемблировании MASM/TASM заменит строки

```

mov dx,Mess1
mov dx,Mess2
на
mov dx, [bp+2]
mov dx, [bp]

```

соответственно. То есть стоит один раз позаботиться, а затем все будет понятно и просто. Конечно, вам нужно будет еще немного "руку набить": написать три-четыре подобные программы. Смотрите, как мы делаем подобные вещи в нашей оболочке (data.asm):

```

Height_X    equ [bp+18]      ;высота рамки
Width_Y     equ [bp+16]       ;ширина рамки
Attr        equ [bp+14]       ;атрибуты рамки

```

И т. д.

Получаем же доступ к параметрам следующим образом (`DRAW_FRAME`, `display.asm`):

```

mov ax,Height_X
mov ax,Attr

```

И т. п.

Один раз посчитали, записали и пользуемся!

Так как у программистов часто возникают ошибки при использовании данного метода, то мы очередной раз обращаем ваше внимание на следующее: переменная `Height_X` заносится в стек первой, а достается из стека последней — `Height_X equ [bp+18]`. Возврат из процедуры: `ret 20`, т. к. заносим 20 байт. Ошибки (упущения)

у вас, конечно, будут. Но сразу проверяйте: а правильно ли вы передали и получили параметры в/из стека, а также верно ли вы выходите из процедуры. Проверьте, это очень удобно. Более того, при программировании под Windows мы будем постоянно передавать параметры в стеке, причем очень много! Так что, готовьтесь и привыкайте!

18.3.2. Передача параметров в стеке

Еще несколько слов о данной процедуре. Обратите внимание на параметр `Other`. В нем мы будем передавать параметры, которые требуют проверки на "да" или "нет". Для этого используется всего один бит данной переменной. В нашем примере мы, допустим, договариваемся о том, что нулевой бит (крайний справа) будет указывать на то, стоит ли перед выводом рамки копировать часть экрана или нет (см. далее). В дальнейшем, когда мы добавим функции, будем проверять уже не байты, а биты.

Предположим, что третий бит параметра `Other` указывает на то, выводить подчеркнутую линию или нет. Если третий бит установлен, то выводим. Таким образом, один байт в ассемблере может нести 8 различных параметров (в байте, как мы помним, 8 бит). Для этого и изучали мы в самом начале двоичную систему счисления. Удобно ли использовать 8 бит одного байта для передачи сразу восьми параметров? Не только удобно, но и компактно!

Обратите внимание, как мы заносим в стек смещение строки:

```
push offset Mess_head
;надпись вверху рамки (если 0, то не выводить)
push offset Mess_head
push 1E00h           ;цвет надписи вверху рамки
```

Если в качестве надписи заносим просто ноль, то строка выводиться не будет. Может ли быть такое, что смещение некоторой строки будет нулевым? Вряд ли... Тем не менее, если нам не нужно выводить строку, то ее атрибут все равно надо заносить (просто любое число). Если мы этого не сделаем, то стек останется не выровненным. Так как занесли мы 16 байт, а процедура вытащит 20. Получается нарушение работы стека.

18.3.3. Вычисление длины строки на стадии ассемблирования

Обратите внимание, как просто вычисляется длина строки, в случае, если ее длина известна еще до ассемблирования программы:

```
Mess_quit db 'Ассемблер',0
Mess_quitl equ $-Mess_quit
```

Причем последняя строка занимать памяти не будет! В отладчике строка

```
mov dx,offset Mess_quitl
```

будет выглядеть как

```
mov dx,0009
```

Зачем это нужно? Вспомните ситуацию со строками вида `mov ax,23+5*2`. Это удобно. Более того, добавив что-либо в строку `Mess_quit` или удалив из нее, ассемблер-программа (MASM/TASM) автоматически посчитает ее размер при обработке файла. А иначе нам придется делать это самостоятельно...

18.3.4. Процедуры `Copy_scr / Restore_scr` (`display.asm`)

Прежде чем нарисовать на экране рамку (окно), нам нужно сохранить ту информацию на экране, которая будет потеряна (затерта выводимыми поверх нее символами). Попробуйте нажать в Far Manager клавишу `<F5>`, а затем `<Esc>`. Вы задумывались, каким образом окно появляется на экране, а затем исчезает, восстановив затертые символы. Создается ощущение того, что окно просто располагается поверх чего-то другого. Кажется, мелочь. А это все, уважаемые читатели, надо собственными руками делать. И еще пример: восстановление пользовательского экрана при нажатии комбинации клавиш `<Ctrl>+<O>`.

Для сохранения содержимого экрана перед рисованием окна и для восстановления затираемых данных напишем две процедуры: `Copy_scr` (копирование в буфер) и `Restore_scr` (восстановление) (`display.asm`).

Принцип работы этих подпрограмм таков: получаем `dh` (ряд), с которого следует начать сохранение, причем `dl` (колонка) будет всегда нулевым (так удобнее для программиста). В `ax` заносится количество рядов, которое нужно будет сохранить относительно `dh`. Сперва надо получить линейный адрес (просто вызовем известную нам процедуру `Get_linear`) (листинг 18.8).

Листинг 18.8. Получение линейного адреса

```
...
xor dl,dl          ;Обнулим dl. Теперь dh = ряд, dl = 0
call Get_linear    ;Получим линейный адрес
...
```

После этого необходимо получить количество байтов для сохранения (вспоминаем, как расположены символы в видеокарте). Для этого количество рядов нужно умножить на 160 (листинг 18.9).

Листинг 18.9. Вычисление количества копируемых байтов

```
...
mov bl,160          ;Получим количество байтов, которые нужно копировать
mul bl
mov cx,ax          ;переносим их в cx (будем использовать cx как счетчик)
...
```

Необходимая информация заносится в регистр `al` перед вызовом подпрограммы. `Mul bl` умножает содержимое `bl` на `al`, результат помещается в `ax`, который впоследствии переносим в `cx`. Сохраним все эти данные в переменных для восстановления. Копировать будем целыми рядами, т. е. от начала левого угла экрана до конца (`dl` от 0 до 79; для этого мы и обнуляем `dl` на всякий случай). Копируется столько рядов, сколько будет занимать выводимое окошко.

Теперь все готово. Переносим данные в область второй видеостраницы (первую занимает наша оболочка, вторую — пользовательский экран) (листинг 18.10).

Листинг 18.10. Копируем видеостраницу

```
...
mov si,di          ;ds:si — откуда копируем
xor di,di          ;es:si — куда копируем

;Сохраним полученные значения для восстановления
mov Num_copysi,si
mov Num_copydi,di
mov Num_copycx,cx

push 0B800h        ;Настроим сегментные регистры
pop ds
push 0BA00h
pop es
rep movsb          ;Копируем...
...
```

Восстановление происходит достаточно просто и по тому же принципу, что и сохранение экрана. (См. описание процедуры `Restore_scr.`)

18.3.5. Оператор `scas`

Рассмотрим еще один оператор, позволяющий работать со строками (массивами данных) (табл. 18.1).

Таблица 18.1. Оператор `scas`

Команда	Перевод	Назначение	Процессор
<code>scas</code>	<code>Scan string</code> — сканирование строки	Поиск символа в массиве	8086

Данный оператор имеет две разновидности, подобно `movs` и `stos`, а именно: `scasb` и `scasw`. Оператор `scasb` служит для поиска первого попавшегося байта, а `scasw` — первого попавшегося слова (двух байтов). При этом `es:di` должен содержать адрес строки (листинг 18.11).

Листинг 18.11. Пример использования оператора scas

```
...
;es:di — адрес строки
(1)    mov di,offset String
;cx — максимальное количество сканируемых байтов/слов
(2)    mov cx,offset String_len
;Символ для поиска (9)
(3)    mov al,9
;Ищем первый байт, который находится в al
(4)    repne scasb
...
(5) String db 1,2,3,4,5,6,7,8,9,10,11,12
(6) String_len equ $-String
...
```

В cx заносим длину (количество символов/байтов) строки String, т. е. 12. В al — символ, который нам нужно найти. После выполнения строки (4) di будет указывать на адрес байта, следующего после найденного символа (т. е. на смещение числа 10). Вроде, все понятно, но возникают четыре вопроса:

1. Что такое repne? Мы до сих пор работали только с rep.

Итак, оператор repne (о англ. *repeat if not equal* — повторять, если не равно) сканирует строку (массив) до тех пор, пока символ, расположенный в al/ax, не будет в этой строке (массиве) найден. Данный оператор обычно используется для поиска первого символа, в отличие от оператора repe (от англ. *repeat if equal* — повторять, если равно).

2. Существует ли префикс repe?

Безусловно, существует и выполняет действия, противоположные действиям оператора repne (листинг 18.12).

Листинг 18.12. Пример использования оператора repe

```
...
;es:di — адрес строки
(1)    mov di,offset String
;cx — максимальное количество сканируемых байтов/слов
(2)    mov cx,offset String_len
;Символ для поиска
(3)    mov al,1
;Ищем первый байт, отличный от того, который находится в al
(4)    repe scasb
```

```
...
(5) String db 1,1,1,1,1,6,1,1,1,1,1,1
(6) String_len equ $-String
...
```

В данном случае после выполнения строки (4) регистр `di` будет указывать на адрес следующего за цифрой 6 байта (т. е. 1), а флаг нуля будет установлен. Мы можем проверить, найден ли байт, отличный от загруженного в `al`, или нет, путем выполнения команды `je Label` (переход на метку `Label`, если байт найден). Данный префикс (`repe`) обычно используется для поиска первого символа, не равного тому, который содержится в `al/ax`.

3. Что произойдет, если в листинге 18.11 мы загрузим в `al` число 13?

В таком случае, если после числа 12 в массиве `String` не находится байт с числом 13, будет сброшен флаг нуля, а `di` выйдет за пределы массива `String`.

4. Что произойдет, если в листинге 18.11 мы в `cx` загрузим, например, число 7?

Это будет означать, что `scasb` выполнится 7 раз, при этом не дойдет до числа 9 (которое загружено в `al`). Следовательно, `di` будет указывать на число 8 в массиве `String`, а флаг нуля будет сброшен. Мы можем элементарно проверить это, выполнив команду `jnz Label` (перейдем, если флаг нуля сброшен — значит, символ не найден).

18.3.6. Подсчет длины нефиксированной строки

Мы рассмотрели оператор `scas`, чтобы теперь использовать его в нашей оболочке для подсчета длины строки (`Count_strmid`, `display.asm`). Каким образом?

Признаком окончания строки у нас является ASCII-символ 0. Считать же количество символов в строке необходимо для того, чтобы вывести ее в центре ряда на экране монитора. В дальнейшем мы будем постоянно использовать данную процедуру, поэтому давайте сейчас ее подробнее рассмотрим (листинг 18.13).

Листинг 18.13. Подсчет длины строки

```
...
(1)    push cs          ;es=cs
(2)    pop es
(3)    mov di,si        ;di=si
(4)    xor al,al        ;al=0
(5)    mov cx,0FFFFh    ;Сколько символов перебирать (возьмем максимум)
(6)    repne scasb     ;Ищем 0 в массиве данных...
;0 найден! di указывает на следующий символ за найденным нулем
;si=начало строки
;di=конец строки+1
(7)    sub di,si        ;di=di-si-1 = длина строки
```

```
(8)    dec di
(9)    shr di,1           ;Делим длину на 2
(10)   mov ax,40          ;Делим количество символов в строке на 2 = 40
(11)   sub ax,di          ;ax=40-половина длины строки = нужная колонка
(12)   mov dl,al          ;dl=колонка, с которой следует выводить строку.
...

```

Перед вызовом этой процедуры мы в `si` загружаем адрес строки, в которой следует посчитать количество символов. Так как оператор `scas` работает с парой регистров `es:di`, то нам нужно сперва в `di` загрузить содержимое регистра `si` (строка (3)). Далее аннулируем `al` (4), заносим в `cx` максимальное число. Длину строки мы пока не знаем, поэтому предположим, что строка имеет максимально возможную длину, которую может содержать регистр `cx`. Начался поиск (6)... Итак, нашли символ '`'0'` (7). Мы его просто обязаны найти.

Теперь мы имеем следующее: `si` содержит начальное смещение строки, `di` — конец строки + 1. Чтобы получить длину, нужно из `di` вычесть `si` и еще единицу ((7), (8)):

$$di = di - si - 1 = \text{длина строки}.$$

Обратите внимание, как это выглядит в отладчике (рис. 18.5).

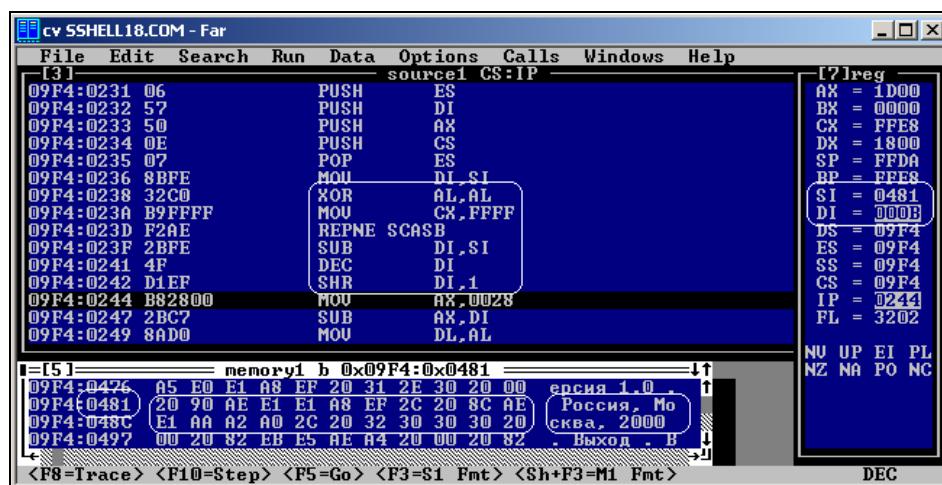


Рис. 18.5. Вычислим длину строки и разделим эту длину на 2. $di = \text{длина строки} / 2$

Затем разделим полученную длину на два (9). Так как на экране в одном ряду 80 символов (режим 3), то 80 делим на два, чтобы получить середину ряда. Из середины ряда вычитаем половину длины строки. Полные формулы:

$di = \text{длина строки},$

$dl = (80/2) - (di/2).$

Теперь `dl` содержит колонку, с которой следует начинать выводить строку. Это и будет центр ряда.

18.3.7. Вывод строки на экран путем прямого отображения в видеобуфер

Пример вывода строки на экран показан в листинге 18.14.

Листинг 18.14. Вывод строки (Print_string, display.asm)

```

...
Print_string proc
(1)    call Get_linear           ;Получаем линейный адрес строки
...
Next_symstr:
(2)    lodsb                  ;Получаем очередной символ строки
(3)    or al,al                ;Это 0 (конец строки?)
(4)    jz Stop_outstr          ;Да — выходим...
;Иначе заносим в видеобуфер атрибут (ah) и символ (al)
(5)    stows
(6)    jmp short Next_Symstr ;Следующий символ...

(7) Stop_outstr:
(8)    ret
Print_string endp
...

```

Перед вызовом данной процедуры мы должны загрузить в пару регистров `ds:si` адрес строки для вывода, в `dx` — координаты для вывода (`dh` — столбец, `dl` — строка (ряд)), а в `ah` — атрибуты выводимой строки.

В самом начале мы вызываем процедуру перевода `dx` в линейный адрес. Для чего, ведь это замедляет работу программы в целом?

- Во-первых, это делается для удобства написания программы. Как вы думаете, было бы удобно постоянно вычислять самим линейный адрес перед тем, как вывести строку, или отдать эту работу отдельной готовой подпрограмме? Безусловно, гораздо проще, если необходимые вычисления произведет компьютер.
- Во-вторых, что было бы удобней при написании кода: указать просто строку/столбец или одно число — линейный адрес, который нужно вычислять программисту? Конечно же, первый вариант предпочтительней.
- В-третьих, скорость работы программы не настолько уж падает. Даже на компьютерах моделей 8086/8088 это абсолютно незаметно. Так что беспокоиться по поводу скорости нет необходимости.

Строка, передаваемая подпрограмме, должна заканчиваться символом 0. Конечно, вы можете придумать другое ограничение строки (например, как у функции `09h` — '\$'). Но вряд ли это будет удобно, т. к. нередко приходится выводить этот символ на экран, а вот вывод числа 0 вообще не имеет смысла.

Понять работу данной процедуры труда не составит. Все элементарно!

На что хотелось бы обратить ваше внимание, так это на то, как у нас организована проверка нажатых клавиш (`MAIN_PROC`, `main.asm`). Также посмотрите процедуру `Quit_prog` в этом же файле. В дальнейшем мы усовершенствуем вывод рамок на экран с запросом "Да"/"Нет". Это диалоговое окно, как правило, аналогично общему.

Интерес представляет также то, каким образом мы показываем пользовательский экран при нажатии комбинации клавиш `<Ctrl>+<F5>` (`MAIN_PROC`, `main.asm`).

18.4. Резюме

По большому счету, вы уже самостоятельно можете написать оболочку типа Far Manager, используя программу HELPASSM. Наша задача теперь заключается в том, чтобы показать вам "подводные камни", некоторые алгоритмы, которые не рассматривались до сих пор, общие понятия при работе с расширенной памятью и вывод десятичных чисел на экран. Этим мы и займемся в следующих главах.

Как видите, мы уже написали несколько мощных процедур (например, `Draw_frame`, `Print_string`), которыми вы можете пользоваться. Просто вставляйте их в собственные программы и вызывайте! Безусловно, довольно много времени уходит на написание самих процедур. Зато как от этого выигрывает скорость работы программы и ее размер!



Глава 19

Создание резидентного шпиона

19.1. Резидент

В данной главе приступаем к изучению очередной резидентной программы. Прежде всего, разберемся, какие функции она выполняет.

Итак, после загрузки в память резидент перехватывает прерывание 21h. Затем отслеживает некоторые действия программ (функций, которые они вызывают) и записывает информацию в свой LOG-файл. Список отслеживаемых функций прерывания 21h приведен в табл. 19.1.

Таблица 19.1. Отслеживаемые функции

Номер функции в ax	Описание функции
ax = 4B00h	Запуск файла на выполнение
ah = 39h	Создание каталога
ah = 3Ah	Удаление каталога
ah = 3Bh	Смена каталога
ah = 3Ch	Создание файла
ax = 3D02h	Открытие файла для чтения/записи
ah = 41h	Удаление файла

По стандартам Microsoft перед вызовом всех этих функций прерывания 21h в регистры ds:dx должно быть загружено имя файла или каталога. Поэтому мы выполняем одни и те же действия, если какая-либо функция вызывается. Примеры корректного вызова этих функций приведены в листингах 19.1 и 19.2.

Листинг 19.1. Создание каталога

```
mov ah,39h  
mov dx,offset Directory  
int 21h
```

Листинг 19.2. Удаление файла

```
mov ah, 41h
mov dx, offset File
int 21h
```

Исключение составляет функция 4B00h. Для этой функции, помимо указания смещения на файл в регистрах ds:dx, необходимо еще дополнительно хорошо подготовиться. Запуск программ на выполнение мы будем рассматривать в последующих главах, т. к. это довольно объемная тема и требует подробного рассмотрения.

Теперь рассмотрим использование этих функций на примере получения атрибутов файла.

В операционных системах MS-DOS и Windows файл может иметь следующие атрибуты (табл. 19.2).

Таблица 19.2. Атрибуты файлов в MS-DOS

Атрибут	Описание
000001b	Только чтение
000010b	Скрытый
000100b	Системный
001000b	Метка тома
010000b	Подкаталог
100000b	Архивный

Существенное отличие в атрибутах файлов в Windows и MS-DOS заключается в том, что в Windows атрибуты более расширенные, в частности: сжатый, шифрованный, индексируемый и т. п.

Атрибуты файла могут комбинироваться, но не все. Например:

- 100001b — "только чтение" и "архивный";
- 000110b — "системный" и "скрытый".

Однако не следует указывать атрибуты, которые явно противоречат друг другу. Например, "подкаталог" и "метка тома" вместе.

На рис. 19.1 приведен участок кода, который создает файл в текущем каталоге с помощью функции 3Ch. При этом смещение имени создаваемого файла находится в регистрах ds:dx, а атрибуты — в cx.

В этом примере атрибуты создаваемого файла будут "архивный" и "скрытый". В двоичной системе это выглядит как 100010b, а в шестнадцатеричной — 22h.

Обратите внимание, что существует такой атрибут, как *"метка тома"*. Метка диска (тома) — это не что иное, как обычный файл с соответствующим атрибутом и имеющий нулевую длину. Аналогично и с *"файлами"*, имеющими атрибут *"подкаталог"*.

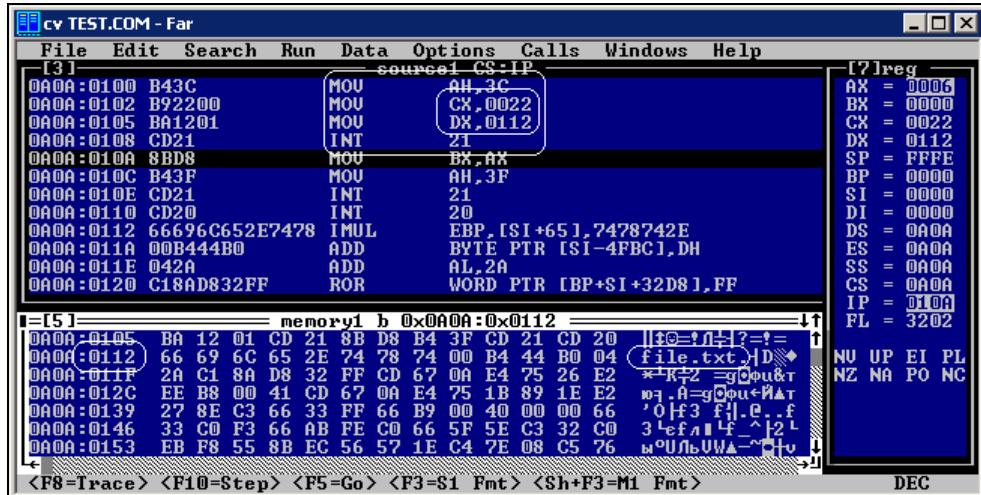


Рис. 19.1. Создание файла

Что именно делает наш резидент?

Программы типа COM начинаются со смещения 100h. От 0 до 0FFh находится PSP (Prefix Segment Program). Некоторые поля PSP вы уже знаете. Например, по смещению 0 находится команда int 20h, а по смещению 80h — DTA и изначально командная строка.

Оставаясь в памяти, резидент сохраняет с собой также PSP (т. е. "лишние" 100h байт). Получается, что 100h байт памяти постоянно зарезервировано. Вы уже, возможно, заметили, что наши предыдущие резиденты занимали памяти больше, чем их реальный размер на диске. Это-то как раз и связано с тем, что в памяти вместе с кодом резидента остается PSP программы (плюс еще строка окружения, но об этом позже).

Существует несколько способов избежать размещения лишних байтов в памяти:

1. Сделать так, чтобы программа в памяти начиналась не со смещения 100h, а со смещения 0. Однако здесь возможны некоторые проблемы. Об этом позже.
2. Разместить в нашей свободной области стек на то время, пока работают наши обработчики прерываний.
3. Сделать в области PSP буфер, т. е. задействовать эту память для временного хранения данных, которые не превышают 100h байт.

Мы будем использовать третий способ, хотя иногда удобнее пользоваться вторым и даже комбинировать оба этих метода. Для какой цели нам нужен будет этот буфер, сейчас и рассмотрим.

Как уже упоминалось, мы перехватываем указанные выше функции прерывания 21h и фиксируем действия программ, которые вызывают данное прерывание в нашем LOG-файле (c:\report.txt). Теперь обратим внимание на то, что именно делает обработчик прерывания 21h (листинг 19.3).

Листинг 19.3. Проверка вызова функций прерывания 21h

```
...
cmp ax,4B00h      ;Это запуск программы?
je Start_f

cmp ah,39h        ;Это создание каталога?
je Create_dir

...
;Если что-то другое, то передадим управление оригинальному обработчику
;прерывания 21h ...
jmp short Go_21h

Start_f:
;Строка для записи в наш LOG-файл.
mov si,offset Start_filemess
call Move_string      ;Готовим строку и записываем ее в файл...

jmp short Go_21h      ;Передадим управление прерыванию 21h...

Create_dir:           ;То же самое для остальных...
mov si,offset Create_dirmess
call Move_string

jmp short Go_21h
...
```

В приведенном фрагменте кода нашего резидента мы вначале выясняем, запускает ли какая-то программа на выполнение или нет. Затем проверяем другие функции, которые могут вызываться какой-либо программой.

Допустим, некто пытается создать каталог при помощи функции 39h. Тогда мы передаем управление на метку Create_dir (создание каталога). Это видно из приведенного выше примера. Затем в si заносится адрес строки Create_dirmess:

```
...
Create_dirmess dw Create_dirmess1
db 'Создание каталога ---> '
Create_dirmess1 equ $-Create_dirmess-2
...
```

Как видите, первые два байта адреса, который заносится в si, содержат длину строки. Обратите внимание, как просто мы получаем эту длину: переменную

Create_dirmess1 программа-ассемблер будет использовать исключительно для своих целей, а значит, в код программы она не попадет. Рассмотрим это подробнее:

```
Create_dirmess1 equ $-Create_dirmess-2
```

\$ (текущее смещение) минус смещение Create_dirmess минус два (т. к. первые два байта и будут указывать на длину строки). Временная переменная Create_dirmess1 будет равна длине строки. Пожалуйста, внимательно изучите приведенные выше примеры. Очень важно понять этот принцип.

Как только занесли в si адрес строки, вызываем процедуру Move_string. При этом в bx будет находиться сегмент строки, который передала некая программа.

В целом, принцип такой: процедура Move_string заносит по смещению 0 сегмента, в котором находится наш резидент (т. е. в область PSP), соответствующую строку (например, "Создание каталога ---> "), после нее заносится имя файла, а именно то, что находится в паре регистров ds:dx на входе в прерывание 21h. Это может быть как имя файла, так и полный путь к нему. Например, диск:\каталог\имя_файла — c:\My_dir\file.ext. Вот именно это мы и записываем по смещению 0 сегмента, в котором находится наша программа.

Допустим, какая-то программа пытается создать каталог c:\assm\directory, тогда в LOG-файл запишется приведенная ниже строка:

```
Создание каталога ---> c:\assm\directory
```

Резидент достаточно прост, хотя в нем существуют команды, которые мы еще не рассматривали. В описаниях файла-приложения достаточно информации, чтобы разобраться в том, что именно делает та или иная функция или команда. Вы уже сами можете написать подобный резидент, опираясь на информацию из этой главы.

19.2. Что нужно вам вынести из этой главы?

Необходимо разобрать резидент и понять его работу. Вы должны проанализировать алгоритмы, если, конечно, хотите выучить ассемблер. Мы лишь приведем общие понятия работы резидентной программы.

Мы перехватываем прерывание 21h, помещая его адрес в прерывание 03h. Это значит, что наш резидент невозможно будет просмотреть под многими отладчиками. Обратите внимание, как мы вызываем прерывание 21h в нашем обработчике.

В принципе, настоящий резидент довольно прост. Вам достаточно разобраться с описанием в файле-приложении.

Все то, что осталось за кадром, мы подробнее рассмотрим в последующих главах на других сходных примерах.



Глава 20

Финальная версия вируса

Теперь наш вирус полностью работоспособен. Будьте осторожны! Случайно заразив нужный файл, вам придется провести некоторое время, чтобы вылечить его вручную. Так что, будьте предельно внимательны!

Несколько слов о программе. (Обязательно прочитайте, прежде чем запускать вирус!)

- Вирус нерезидентный. Это значит, что заражать он будет программы только в случае, если запустить зараженный СОМ-файл. После того как вирус отработал, он из памяти удаляется.
- Вирус не делает ничего, кроме как заражает файлы.
- Ни один антивирус не вылечит файл, зараженный нашим вирусом! Так что, не надейтесь ни на Dr. Web, ни на что-либо еще.
- Вирус заражает только СОМ-файлы в текущем каталоге. Что это значит?

Допустим, в каталоге C:\assm есть следующие файлы:

- test.com;
- dn.com;
- nc.exe.

Причем test.com уже заражен нашим вирусом. Запуская test.com на выполнение, он заражает dn.com. Запустив затем dn.com, вирус ничего не заразит, т. к. в данном каталоге больше нет незараженных нашим вирусом СОМ-файлов.

Но если у вас в переменной PATH (в autoexec.bat) стоит путь к каталогу c:\assm, то, запустив test.com из текущего каталога, скажем, c:\nc, наш вирус заразит первый встречный СОМ-файл в каталоге c:\nc. Будьте осторожны!!!

Если все же произошла беда, и вы не знаете, как удалить вирус, заразивший весь компьютер, то см. главу 24, где приведен код антивируса. Хотя, если перед тем, как запускать virus20.com на выполнение, вы прочтете описания к нему и разберетесь во всем, то вам не составит труда восстановить зараженный файл.

- СОМ-файлы, расположенные в каталоге C:\Windows\Command, после заражения отказываются работать корректно. С чем это связано — не совсем понятно. Причем после сжатия файлов программой PKLITE, командная строка по-прежнему определяется неверно. Скорее всего, эти файлы подсчитывают контрольную сумму или еще что-то... Для экспериментов придется искать другие СОМ-файлы.

- Как вы думаете, зачем мы изучали работу отладчика в главе 16? Дело в том, что если вы будете исследовать работу вируса в отладчике, то столкнетесь с одной проблемой. Записав вирус в файл с помощью отладчика, а затем, запустив файл на выполнение, вы с удивлением заметите, что этот зараженный файл работать не будет. Почему? Вспомните, что отладчик вставляет `int 3 (0CCh)`. При перезаписи же программой самой себя под управлением отладчика, программа перезапишет и `int 3`, вставленный этим отладчиком. Поэтому об этом забывать не следует.

Еще такой момент: в главе 24 мы напишем резидентный антивирус против нашего вируса. Однако корректно лечить зараженные файлы он будет только в том случае, если вы не измените в нашем вирусе ни единого байта!

20.1. Вирус

20.1.1. Альтернативы `ret`, `call` и `jmp`

Вопрос: можно ли вместо `ret` и `call` использовать `jmp`, а вместо `jmp — ret?` Безусловно. Более того, в некоторых случаях просто невозможно обойтись без использования нестандартных команд.

В табл. 20.1 приведены альтернативы стандартным операторам ассемблера.

Таблица 20.1. Альтернативы стандартным операторам

Команда	Альтернатива	Описание
<code>ret</code>	<code>pop ax</code> <code>jmp ax</code>	Вытащим из стека адрес возврата и перейдем на него
<code>call proced</code>	<code>move bx, offset lab_ret</code> <code>push bx</code> <code>jmp proced</code> <code>lab_ret:</code>	Занесем в стек адрес возврата и "прыгнем" на метку процедуры
<code>jmp label1</code>	<code>push offset label1</code> <code>ret</code>	Занесем в стек адрес метки и сымитируем выход из процедуры

Все это описывалось по той причине, что теперь переход на метку инициализации вируса (`Init`) осуществляется следующим образом:

```
...
push offset Init      ;3 байта
ret                  ;1 байт
...
```

Дело в том, что, используя нестандартный метод, нам гораздо удобнее менять адрес метки `Init` в зараженных файлах. Читая дальше, вы поймете, почему именно так, а не иначе.

20.1.2. Заражение файла

Заражение файла происходит с помощью процедуры `Infect_file`. Опустим поиск файлов, будем считать, что подходящий файл уже найден. Осталось только проверить его длину (листинг 20.1).

Листинг 20.1. Проверка размера заражаемого файла

```
...
mov ax,cs:[1Ch]      ;Получим второе слово длины заражаемого файла
or ax,ax             ;Если оно не равно 0, то выходим...
jnz Error_infect    ;... это значит, что размер файла больше 64 Кбайт
mov bp,cs:[1Ah]      ;Получим младшее слово (т. е. размер файла)
...
...
```

Для чего нам нужно производить проверку размера файла перед его заражением, если известно, что СОМ-файлы не могут превышать 64 Кбайт?

Дело в том, что операционная система определяет, какой тип файла запускается не по его расширению, а по первым двум байтам содержимого файла. Если это "mz" или "zm", то файл EXE, иначе — СОМ. Это можно легко проверить, если посмотреть в текстовом редакторе на первые два символа файла. Ничто не мешает любому пользователю переименовать файл, например, `test.exe` длиной 450 Кбайт в `test.com`. ОС все равно определит, что это EXE-файл, по первым двум байтам.

Типичным примером может служить файл `command.com`, который в MS-DOS версии 7.0 занимает 95 Кбайт. Расширение СОМ оставили для совместимости с программами, написанными в более старых версиях DOS.

И что же получится, если мы попробуем заразить тот же `command.com`, который, по сути, является `command.exe`?

Мы просто его испортим. То есть он перестанет работать вообще. Вот, собственно, для этого и следует проверить размер файла, прежде чем заражать его.

Обратите внимание, каким образом мы это делаем. После того как функция `4Fh` или `4Eh` нашла файл, в DTA по смещению `1Ah` заносится его размер. Так как два байта могут хранить число до 65 535, то для определения размера файла используются два слова. Первое (смещение `1Ah`) — младшая половинка, второе (смещение — `1Ch`) — старшая. Вспоминаем, что данные в памяти хранятся в обратном порядке.

Что мы еще не сделали в вирусе, так это не проверили первые два символа заражаемого файла ("mz" или "zm"). Для чего? Может случиться так, что EXE-файл имеет длину менее 64 Кбайт, но расширение СОМ. В этом случае зараженный файл также утратит свою работоспособность. Реализовать же проверку на ассемблере труда не составит, поэтому в целях экономии места мы ее опустим.

Следующие шаги:

1. Открываем найденный файл на чтение/запись (функция `3D02h` прерывания `21h`).
2. Читаем его первые шесть байт в память (функция `3Fh` прерывания `21h`).
3. Проверяем, заражен ли уже этот файл нашим вирусом или еще нет.

Рассмотрим эти шаги подробнее. В нашем примере мы используем сигнатуру 1122h для определения того, заражен ли найденный файл нашим вирусом или нет. Эта сигнатура должна быть расположена по смещению +4 от начала файла. Обратите внимание, что мы проверяем

```
cmp word ptr [bx+4],1122h
```

а в файле сигнтура 1122h будет располагаться наоборот: 2211h. Вот вам еще одно подтверждение того, что данные в компьютере хранятся в обратном порядке.

А что, если в каком-то незараженном файле по смещению +4 от начала уже есть такие байты? Получается, что мы посчитаем, что файл уже заражен? Да, это так. Но как вы думаете, какова вероятность того, что эта сигнтура будет находиться в произвольном файле по такому смещению? Возможно, даже еще и файла такого не существует в природе.

Однако, при желании, программист может всегда увеличить сигнтуру до 10—20 байт, чтобы быть уверенным в том, что найденный файл действительно уже заражен нашим вирусом.

Итак, проверку на соответствие файла необходимым требованиям произвели. Теперь устанавливаем указатель файла на его конец. Будем дописывать в "хвост" "файла-жертвы" собственно код нашего вируса.

Для перемещения указателя открытого файла на его конец следует воспользоваться функцией 4202h прерывания 21h (табл. 20.2).

Таблица 20.2. Функция 4202h прерывания 21h: установка указателя на конец файла

Вход	Выход
ах = 4202h bx = номер файла cx, dx = количество байтов, которые необходимо отсчитать от конца файла и установить на них указатель	jc — ошибка

Если в регистры cx и dx мы загрузим число 0, то указатель будет установлен сразу за последним байтом файла. Пока нас только это и интересует.

Указатель служит для уведомления операционной системы, из какого места следует читать или в какое место следует писать байт (байты). Если указатель не переместить на конец файла, то, записав тело вируса, мы затрем файл, начиная со смещения 7.

Запомните, что:

- при открытии файла указатель устанавливается на начало файла автоматически. Если указатель установлен на середину файла, то, после закрытия и повторного открытия файла, указатель снова установится на начало файла;
- при чтении файла указатель перемещается на количество прочитанных байтов. То есть если файл имеет длину 3000 байт, то, после прочтения 1500 байт, указатель переместится на середину файла.

В нашем примере мы устанавливаем на конец файла указатель, чтобы записать вирус в "хвост" "файла-жертвы" (листинг 20.2).

Листинг 20.2. Установка указателя на конец файла

```
...
mov ax, 4202h      ;Установим указатель чтения/записи на конец файла.
mov bx, Handle
xor cx, cx        ;Отсчитывать 0 байт от конца файла.
xor dx, dx
int 21h
jc Error_infect
...
...
```

После того как указатель чтения/записи был установлен на конец файла, можно дописывать в заражаемый файл код нашего вируса, используя известную уже вам функцию 40h. Обратите особое внимание на количество записываемых байтов (листинг 20.3).

Листинг 20.3. Запись тела вируса

```
...
mov ah, 40h          ;В bx уже есть номер файла.
;Пишем в "хвост" "файла-жертвы"
mov cx, offset Finish-100h-F_bytes
mov dx, 100h          ;Пишем тело вируса.
int 21h
jc Error_infect
...
...
```

После того как мы записали тело нашего вируса в "хвост" "файла-жертвы", можно сказать, что этот файл уже заражен. Его размер увеличился на длину нашего файла.

Код вируса в "файл-жертву" мы записали, но наш вирус не получит управление. Нужно сделать так, чтобы вирус запустился первым, выполнил необходимые действия и запустил зараженную программу на выполнение, не нарушив ход ее работы.

Для этого необходимо первые байты "файла-жертвы" (начинающиеся с адреса 100h) заменить переходом на тело нашего вируса, а затем, когда вирус отработает, восстановить эти байты в памяти и "прыгнуть" по адресу 100h.

Следующий шаг: записываем после нашего вируса первые 6 байт, которые мы меняем (листинг 20.4).

Листинг 20.4. Замена байт "файла-жертвы"

```
...
;После тела вируса дописываем первые настоящие 6 байт "файла-жертвы"...
mov ah,40h
mov cx,F_bytes
mov dx,offset Finish
int 21h
...

```

Теперь нам нужно сделать так, чтобы наш вирус, прежде чем получит управление "файл-жертва", выполнил определенные действия, чтобы затем "файл-жертва" заработал, не подозревая о том, что до этого некая сторонняя программа успешно завершила работу.

Для этого необходимо вместо первых 6 байт, которые мы записали в "хвост" файла после тела вируса, переместить по адресу, по которому находится метка `Init`. При запуске файла наш вирус первым получит управление, отработает, восстановит в памяти первые 6 байт и передаст управление "файлу-жертве".

Как заменить первые 6 байт на диске? Вспомните, каким образом мы уже вычислили размер файла и сохранили его в регистре `bp` (см. примечания в файле-приложении). Передаем управление "файлу-жертве", используя нестандартный способ (листинг 20.5).

Листинг 20.5. Передача управления "файлу-жертве"

```
...
push адрес перехода на метку Init
ret
...

```

Если "файл-жертва" занимает, к примеру, 3000 байт, то после `push` будет находиться один адрес, а если, скажем, 5000 байт, то переход будет осуществлен, соответственно, на другой адрес.

Все! Теперь файл заражен корректно. При его запуске первым получит управление наш вирус. Осталось восстановить в памяти первые 6 байт "файла-жертвы" (мы-то их сохранили после тела вируса!) и передать управление по адресу `100h`.

20.1.3. Общая схема работы вируса

Итак, чтобы корректно заразить файл, мы выполняем следующие шаги:

1. Ищем СОМ-файл.
2. Читаем первые 6 байт найденного файла.
3. Если найденный файл больше 64 Кбайт, то начинаем поиск заново (переходим к шагу 1). Если меньше — работаем дальше.

4. Если в найденном файле по смещению +4 от начала находится `1122h`, то начинаем поиск заново (переходим к шагу 1). Если нет — работаем дальше.
5. Пишем в "хвост" найденного файла тело нашего вируса.
6. Пишем после тела вируса первые 6 настоящих байт "файла-жертвы".
7. Закрываем файл.
8. Создаем искусственный переход на метку `Init` путем вычисления размера файла.
9. Открываем файл (указатель в этот момент находится на начале файла).
10. Пишем первые 6 байт перехода на тело вируса `+1122h` для опознавания того, что файл уже заражен нашим вирусом.
11. Закрываем файл.
12. Восстановим первые 6 реальных байт "файла-жертвы" в памяти, которые сохранены в "хвосте" "файла-жертвы" (см. шаг 6).
13. Передаем управление "файлу-жертве".

Вам осталось только внимательно изучить файл-приложение и комментарии к нему.

Еще раз подчеркиваем: будьте предельно осторожны в экспериментах. Лучше всего создать отдельный каталог, куда переписать СОМ-файлы, над которыми можно будет производить эксперименты. Обязательно используйте отладчик в работе и тестировании работы вируса. Ваши старания обязательно увенчиваются успехом!

20.2. Резюме

Уважаемые читатели! Поздравляем вас! Вы только что написали первый вирус. Что мы можем сказать в завершение?

Прежде всего, хотелось бы, чтобы информация, полученная вами из настоящей главы, пошла во благо, а не во вред.

Мне часто задавали вопрос: "Вы действительно уверены в том, что хотите обучать людей написанию вирусов?" Да, уверен. Уверен просто потому, что знаю, изучи вы ассемблер, первое, что придет большинству из вас в голову, — так это написать вирус. Я также уверен, что, изобретая вирус в три часа ночи, вы вдруг поймете, как работает та или иная программа. Вы поймете многое. Через это проходит большинство программистов.

Вернемся немного в прошлое, когда тех, кто писал вирусы, считали "суперпрофессиональными" программистами. Когда вирусов боялись больше всего на свете.

Вирус DIR в свое время наводил такой ужас в компьютерных центрах, что многие готовы были отказаться от всего, что у них было (дискеты, компьютеры, игры...). Но, познакомившись с ассемблером поближе, начинающие программисты понимали, что вирус — это прежде всего программа, которую написал человек.

Почти каждый программист на начальном этапе писал вирусы, при этом в процессе экспериментов портил свои файлы, жесткие диски. Но тот, кто упорно шел вперед, обязательно узнавал что-то новое. Причем не только об одних вирусах!

Поверьте: гораздо интереснее экспериментировать с резидентными программами и файловой системой. Безусловно, в своей жизни вы все равно напишете хоть

один вирус. Нельзя сказать, что это плохо. Это нормально, даже хорошо, интересно. Но хорошо в том случае, если ваш вирус не выходит во "внешний мир". Более того, я вынужден повторно предупредить вас, что за распространение вирусоподобных программ вы попадаете под статью Уголовного кодекса Российской Федерации. Так что, прежде чем нести ваш вирус кому-то, подумайте хорошо: а зачем это вообще нужно? Сколько раз вы в своей жизни страдали от того, что вирус, написанный каким-то новичком-программистом, форматировал ваш винчестер? Сколько драгоценного времени вы тратили впустую, восстанавливая поврежденные программы? Сколько полезного вы могли бы сделать за это время?

"Зачем же тогда в этой книге мы подробно изучали работу вирусов и даже написали один, пусть даже самый простой и безобидный?" — спросите вы.

На это можно ответить так: изучив ассемблер, вы все равно рано или поздно его бы написали. Дело только во времени. Целью данной главы было не научить вас писать вирусы, а дать понять, что вирус — это не так страшно, показать на конкретном примере принцип работы вирусов, а также научить вас бороться с ними. Очень хочется верить, что у многих страх перед "ужасными, злыми и беспощадными" вирусами развеялся.

Мы написали элементарнейший нерезидентный вирус, заражающий СОМ-файлы в текущем каталоге. Но принцип работы у всех вирусов одинаковый. Запомните: любой серьезный вирус, заражающий файлы типа СОМ и EXE, обязательно хранит в своем теле первые байты зараженной СОМ-программы или изначальную структуру EXE-программы. Первые байты могут быть как в закодированном виде, так и храниться в открытом виде (как в нашем примере). Вирусы могут быть резидентными и нерезидентными. Но в теле любого вируса можно всегда найти те байты, которые нужны для восстановления зараженного файла, если, конечно, вирус не разрушает напрочь программу, делая ее невосстановимой и, соответственно, неработоспособной.

Именно исследовав принцип работы вируса, вирусологи пишут антивирус, который восстанавливает изначальную структуру файла, вырезая из него тело вируса.

Вот, собственно, и все! В главе 21 мы продолжим писать оболочку.



Глава 21

Работа с блоками основной памяти

21.1. Оболочка SuperShell

21.1.1. Теория

Если вы уже запустили нашу "оболочку", то ничего особенно интересного вы, скорее всего, не увидели. Она просто читает содержимое текущего каталога в память и выводит первые двадцать найденных файлов на экран. Однако, изучив файл-приложение, вы поймете, что наша программа проделывает уйму работы. А именно:

1. Ищет первый файл.
2. Если это '!', то переходит к поиску следующего файла (выполняет шаг 4).
3. Заносит в память имя найденного файла.
4. Ищет следующий файл.
5. Заносит имя найденного файла в память.
6. Проверяет, не закончились ли файлы.
7. Если нет, то продолжает поиск файлов (переход к шагу 4).
8. Если файлы закончились, то начинает их вывод на экран.
9. Если еще не выведено 20 файлов, то продолжает вывод (шаг 8). Если 20 файлов уже выведено, то переходит к выполнению шага 10.
10. Ждет от пользователя нажатия клавиши.
11. Если пользователь нажал клавишу <Esc>, то переходит к шагу 12. Если нет, то — к шагу 10.
12. Запрашивает пользователя: уверен ли он, что хочет выйти из программы.
13. Если пользователь нажал все, что угодно, кроме <Y> или <y>, то переходит к шагу 10.
14. Если пользователь нажал <Y> или <y>, то выходим.

Вот, собственно, и алгоритм работы нашей оболочки.

¹ В любом каталоге первым всегда расположен символ точки. Это не файл, а обозначение текущего каталога. Если в командной строке набрать 'CD .', то каталог не изменится. Нам не нужно выводить этот символ в оболочке. В отличие от оболочек команда DIR выводит этот каталог на экран.

21.1.2. Практика

Новшество первое

Обратите внимание, как мы вызываем процедуру `Draw_frame` (`Main_proc, main.asm`). На первый взгляд — ничего особенного. Но это только кажется. Раньше мы заносили в стек адреса строк и их атрибуты, которые будут выводиться вверху и внизу окна оболочки. Теперь мы атрибуты в стек не заносим. Но как же тогда задать цвет, которым должны выводиться сообщения? Обратите внимание, что находится перед строками `Mess_head` и `Mess_down`:

```
Mess_head    db 1Eh, ' Super Shell, Версия 1.0 ',0
```

```
Mess_down    db 1Dh, ' Россия, Москва, 2010 ',0
```

Вот это и есть как раз нужные нам атрибуты соответствующих строк. Поступая таким образом, мы экономим байты в программе, более того, мы не нагружаем стек. Отпадает необходимость заносить в стек два слова атрибутов для верхней строки и два слова для нижней. Процедура вывода строк на экран (`Draw_messfr, display.asm`) перед тем, как выводить строку, занесет в `ah` первый символ, который и будет являться атрибутом. Это гораздо проще и нагляднее (листинг 21.1).

Листинг 21.1. Вывод строки на экран

```
...
;SI уже содержит смещение выводимой строки
mov ah,[si]           ;Первый символ в строке – атрибут
inc si                ;Следующий байт – начало строки
call Count_strmid    ;Вычисляем середину строки
call Print_string     ;Выводим строку на экран
...

```

На рис. 21.1 отображено состояние отладчика перед вызовом процедуры вывода строки на экран.

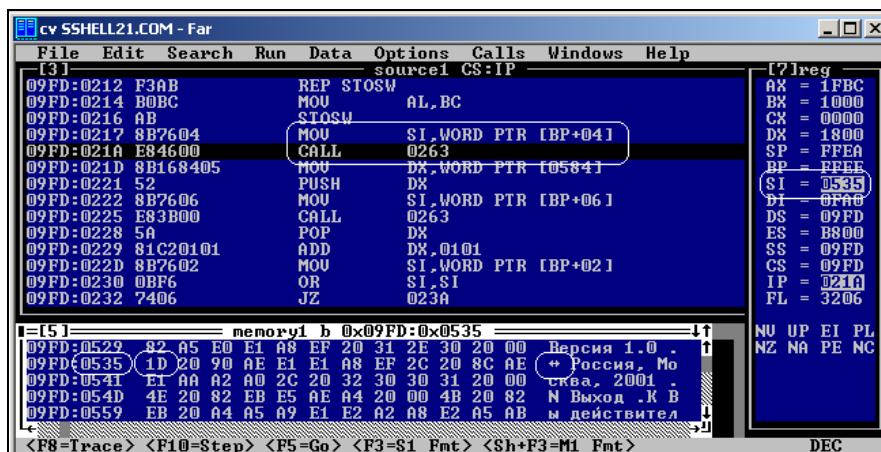


Рис. 21.1. Атрибут в начале строки

Новшество второе

Теперь процедура рисования рамки (`Draw_frame`, `display.asm`) выводит еще и линию вверху окна по нашему требованию. Для этого нам понадобилось просто изменить один бит (листинг 21.2).

Листинг 21.2. Вывод линии вверху окна

```
...
push 10b           ;Экран не копировать, но вывести верхнюю линию.
call Draw_frame    ;Рисуем рамку
...
```

Нулевой бит последнего заносимого слова в стек (если он установлен) указывает на то, следует ли копировать экран в буфер или нет. Второй бит — следует ли рисовать линии вверху окна или нет.

21.1.3. Оператор `test`

Как проверить, включен ли нулевой бит или нет, при этом не затронув остальных битов? Похоже, что команда `cmp` в данном случае нам не поможет.

Да, действительно, для проверки состояния одного бита в регистре или переменной оператор `cmp` не годится. И вот почему.

Представьте, что первый раз мы выводим окно, предварительно скопировав экран, но не рисуем линию вверху. Тогда нулевой бит должен быть равен 1. Перед вызовом процедуры `Draw_frame` в стек занесем просто единицу (листинг 21.3).

Листинг 21.3. Первый вариант вызова подпрограммы рисования окна

```
...
;Рисуем рамку, предварительно сохранив экран, но не выводим линию вверху
push 1           ;Отвечает за конфигурацию окна
call Draw_frame
...
```

В подпрограмме `Draw_frame` заносимый в стек параметр из листинга 21.3 загрузим в переменную `other`. Чтобы проверить, какое именно число занесено в эту переменную, в данном случае воспользуемся оператором `cmp`:

```
cmp Other,1
```

Если после проверки флаг нуля устанавливается, т. е. `Other=1`, то нужно предварительно скопировать экран в память.

Здесь все просто. Теперь представим, что нужно вывести окно с линией вверху, но не копировать экран, т. е. первый бит будет равен единице, а нулевой — 0 (листинг 21.4).

Листинг 21.4. Второй вариант вызова подпрограммы рисования окна

```
...
push 10b      ;или push 2
call Draw_frame
...
cmp Other,2
...
```

И, наконец, представим, что нужно и вывести линию вверху, и сохранить экран (листинг 21.5).

Листинг 21.5. Третий вариант вызова подпрограммы рисования окна

```
...
push 11b      ;или push 3
call Draw_frame
...
cmp Other,3
...
```

Визуально кажется, что все в порядке, и, тем не менее, существует проблема.

Нулевой бит (сообщающий подпрограмме о том, следует ли сохранять содержимое экрана или нет) мы проверяем в начале процедуры `Draw_frame` и до того момента, как вывели окно, а вывод верхней линии — после того, как экран сохранен, и окно нарисовано.

Вопрос: можно ли с помощью команды `cmp` проверить, установлен ли определенный бит, игнорируя оставшиеся биты проверяемого слова (`Other`)?

Произвести такую проверку можно с помощью логического оператора `and` и команды `cmp`. В листинге 21.6 приведен фрагмент кода, который осуществляет проверку отдельного бита переменной.

Листинг 21.6. Проверка бита в переменной с помощью команды `cmp`

```
...
;Перенесем временно в ax значение переменной Other
mov ax,Other
push ax
;Аннулируем все биты, кроме нулевого (вспомним логические команды)
and ax,1
;Проверим, равен ли теперь ax 1?
cmp ax,1
pop ax
mov Other,ax    ;Восстановим переменную Other
je Ravno       ;Перейдем, если равен
...
...
```

Следует отметить, что операторы `mov` и `pop` не изменяют флаги (в том числе и флаг нуля), поэтому проверять результат сравнения можно через неограниченное количество таких команд, что мы и делаем в листинге 21.6.

Как на ваш взгляд: красивый получился код? Вполне красивый, за исключением того, что для проверки всего лишь одного бита мы написали программу, состоящую из 7 строк.

Вероятно, в ассемблере есть другие способы? Безусловно! Ассемблер — такой гибкий язык, что может делать невероятные вещи! Чтобы временно не сохранять регистр или переменную перед проверкой одного, двух, трех... битов, используется оператор `test` (табл. 21.1).

Таблица 21.1. Оператор `test`

Команда	Перевод	Назначение	Процессор
<code>test приемник, источник</code>	Test — тест, проверка	Проверка одного и более битов	8086

Пример использования данного оператора приведен в листинге 21.7.

Листинг 21.7. Пример использования оператора `test`

```
...
mov ax,10100001b
test ax,1          ;Проверим, равен ли нулевой бит единице.
jnz Ravno         ;Переход, если равен
...

```

Обратите внимание, что после команды сравнения `cmp` мы используем операторы условного перехода `je/jz` для перехода на определенную метку в случае, если приемник и источник равны. В команде `test` все совсем наоборот.

Если проверяемый бит установлен, то флаг нуля сбрасывается и переход на метку в этом случае осуществляют команды `jne/jnz`. Если же этот бит сброшен, т. е. равен нулю, то флаг нуля устанавливается, и переход на метку произведут операторы `je/jz` (листинг 21.8).

Очень важно это запомнить и не путать!

Листинг 21.8. Проверка третьего бита с помощью оператора `test`

```
...
mov cl,100101b
test cl,1000b    ;Проверим, равен ли третий бит единице
jz Ne_ravno     ;Переход, если НЕ равен
...

```

Итак, в переменной `Other` может храниться от одного до 8 различных параметров для выводимого окна. Причем каждый из этих параметров на вопрос должен отвечать "ДА" или "НЕТ", т. е. "РАВНО" или "НЕ РАВНО". Как, например, в нашей процедуре `Draw_frame`. Этот способ очень компактный, удобный и не требует заведения отдельной переменной для каждого параметра, как это обычно реализуется в языках высокого уровня.

Напомним, что первый бит переменной `other` отвечает за рисование линии вверху окна, а нулевой — за копирование участка экрана перед выводом окна. Вспомните окно подтверждения выхода из оболочки. Прежде чем будет выведено это окно, система копирует текущее содержимое экрана в определенную память, чтобы впоследствии его восстановить. Использование битов переменной для подобных целей, как уже отмечалось, подходит как нельзя лучше.

В листинге 21.9 приведен фрагмент кода из файла-приложения, выполняющий проверку одного бита переменной.

Листинг 21.9. Проверка бита переменной `Other` в нашей оболочке

```
...
mov ax,Other      ;Получим дополнительную информацию
test al,1         ;Нулевой бит равен 0?
jz No_copyscr   ;Если так, то копировать экран не нужно.
...

```

На рис. 21.2 приведен этот же участок кода в отладчике.

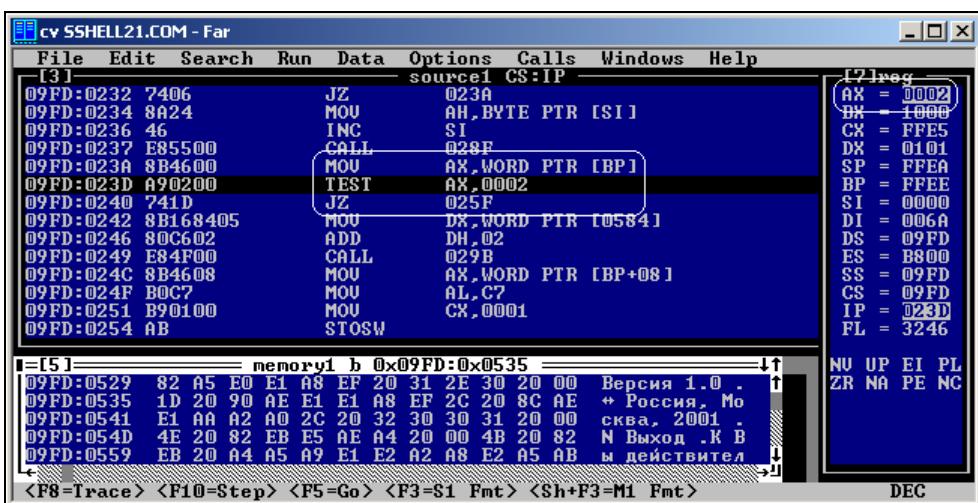


Рис. 21.2. Использование оператора `test`

21.2. Работа с основной памятью DOS

21.2.1. Управление памятью

В этом разделе рассмотрим принципы и методы работы с основной памятью в режиме эмуляции MS-DOS. Основная память имеет объем до 640 Кбайт. Подробно рассмотреть управление памятью в одной главе невозможно. Мы затронем лишь ключевые, основополагающие моменты.

Как только программа загрузилась, DOS автоматически отводит для нее всю свободную основную память. Программист может по своему усмотрению урезать блоки памяти, отводить другие, а также освобождать отведенные участки (блоки) памяти.

Зачем это нужно и где это можно применять? Представьте ситуацию: мы написали некоторую программу, которая, в свою очередь, должна загружать другую. Но так как вся память изначально выделена только нашей программе, то мы не сможем загрузить ничего более. Ведь загружаемой программе (порождаемому процессу) также необходимо некоторое количество памяти не только для работы, но и для загрузки.

Для того чтобы урезать память, используется функция 4Ah прерывания 21h (табл. 21.2).

Таблица 21.2. Функция 4Ah прерывания 21h: изменить объем свободной основной памяти

Вход	Выход
$ah = 4Ah$ $es =$ сегмент распределенного блока $bx =$ размер блока в 16-байтовых параграфах	$\text{ jc } — \text{ ошибка, при этом:}$ $ax =$ код ошибки

Распределенный блок в нашем случае — вся память, отведенная программе, начиная с нулевого смещения сегмента cs и заканчивая последним свободным байтом. Поэтому перед вызовом функции 4Ah следует убедиться в том, что es указывает на сегмент, куда мы загрузились. В табл. 21.3 показано состояние памяти после загрузки программы в память для ее последующего выполнения.

Таблица 21.3. Состояние памяти после загрузки программы

Данные	Описание
Системные файлы	Память занята от 0 до текущего байта
Резидентные программы	Память занята от 0 до текущего байта
Наша программа	Память занята от 0 до текущего байта
Метка Finish	Память занята от 0 до текущего байта
Отведенная память нашей программе	Память занята от 0 до конца

После того как наша программа загрузилась в память, вся основная свободная память отведена только нашей программе.

Задача: урезать занятую программой память до метки Finish. Для чего это нужно — рассмотрим позже. Обратите внимание, каким образом мы ужимаем существующий блок памяти (`Prepare_memory, main.asm`) (листинг 21.10).

Листинг 21.10. Освобождение отведенной памяти

```
...
mov bx,offset Finish      ;bx=последний байт нашей программы
;Так как bx должен содержать не количество байтов, а количество блоков
;по 16 байт, то мы должны сдвинуть биты вправо на 4
shr bx,4
inc bx          ;Увеличим bx на один (на всякий случай)
mov ah,4Ah      ;Функция уменьшения/увеличения существующего блока памяти
;В данном случае урезаем, т. к. наша программа, естественно, меньше
;отведенного блока памяти после загрузки.
int 21h
...
...
```

После выполнения функции 4Ah состояние основной памяти будет следующим (табл. 21.4).

Таблица 21.4. Состояние памяти после освобождения блока

Данные	Описание
Системные файлы	Память занята от 0 до текущего байта
Резидентные программы	Память занята от 0 до текущего байта
Наша программа	Память занята от 0 до текущего байта
Метка Finish	Память занята от 0 до текущего байта
Память за меткой Finish	Память свободна до конца 640 Кбайт, начиная с первого байта, следующего за меткой Finish

То есть размер свободной памяти равен 640 Кбайт минус смещение метки Finish.

После освобождения памяти можно попробовать отвести отдельный блок памяти размером, скажем, 1000h 16-байтовых блоков (параграфов) или 65 536 байт (листинг 21.11).

Листинг 21.11. Отведение блока памяти размером 65 536 байт

```
...
mov ah,48h
mov bx,1000h
```

```

int 21h
...
mov Seg_files,ax      ;Сохраним сегмент отведенного блока
...

```

В этом примере мы использовали новую функцию. Она описана в табл. 21.5.

Таблица 21.5. Функция 48h прерывания 21h: выделить блок основной памяти

Вход	Выход
<code>ah = 48h</code> <code>bx = размер блока в 16-байтовых параграфах</code>	<code>jc</code> — ошибка, при этом: <code>ax = код ошибки</code> Иначе: <code>ax = сегмент выделенного блока</code>

Состояние памяти после отводения блока размером 65 536 байт показано в табл. 21.6.

Таблица 21.6. Состояние памяти после отводения блока памяти

Данные	Описание
Системные файлы	Память занята от 0 до текущего байта
Резидентные программы	Память занята от 0 до текущего байта
Наша программа	Память занята от 0 до текущего байта
Метка Finish	Память занята от 0 до текущего байта
Память за меткой Finish + 65 536 байт	Память занята
Память, начиная с адреса Finish + 65 537 байт	Свободна

На рис. 21.3 показан в отладчике участок кода, который отводит память с помощью функции 4Ah прерывания 21h.

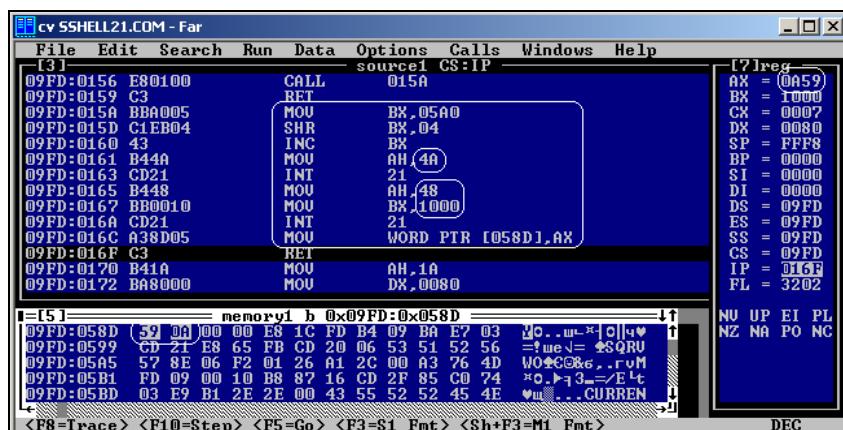


Рис. 21.3. Урезание и отведение памяти после загрузки программы

21.2.2. Считываем файлы в отведенную память

Для каких целей мы выделяем блок памяти размером 64 Кбайт?

В эту область памяти будет загружаться информация о найденных файлах в текущем каталоге. Таким же образом поступают и все оболочки: просто считывают файлы в отведенный блок памяти, а затем с ними работают.

Но, например, внутренняя команда `DIR` не загружает в память найденные файлы. Да и нет смысла. Она нашла файл, тут же вывела информацию о нем на экран и ищет следующий. Выделять, копировать и удалять файлы эта команда, естественно, не может.

Каким образом заносим в отведенную память найденные файлы? Алгоритм простейший:

1. Ищем первый файл. Если файлов нет, то переходим к шагу 5.
2. Заносим имя файла в отведенный блок памяти. Дописываем после имени файла ноль.
3. Ищем следующий файл. Если файлов больше нет, переходим к шагу 5.
4. Заносим следующий файл в память сразу же за найденным предыдущим, продолжаем поиск файлов (шаг 3).
5. Заносим еще один ноль после последнего найденного файла для того, чтобы дать понять процедурам нашей оболочки, что это был последний найденный файл.
6. Выводим указанное количество файлов на экран (сколько находится в переменной `Number_files`) (`Out_files, files.asm`).

Труда разобраться с тонкостями процедур не составит, тем более что в файле-приложении вполне достаточно комментариев.



Глава 22

Часто задаваемые вопросы

В процессе изучения нового языка программирования, как и любого другого предмета, всегда возникает множество вопросов. В этой главе мы рассмотрим наиболее типичные вопросы, которые задают начинающие программисты экспертам портала <http://RFpro.ru>.

1. Что такое дизассемблер и для чего он нужен?

Дизассемблер — это программа, которая переводит машинный код на язык ассемблера. Дизассемблер может применяться при получении исходного кода для изучения работы программы или корректировки его кода с последующим ассемблированием.

Если в программе необходимо изменить два-три байта (например, убрать ограничение на ее работу в более поздних версиях MS-DOS), то можно воспользоваться и специальным текстовым редактором Hacker's View, который и может выступать в роли дизассемблера.

2. Перечислите флаги процессора. Где они расположены и за что отвечают?

- **Флаг переноса CF** (Carry Flag). Содержит 1, если произошел перенос единицы при сложении или заем единицы при вычитании. Используется также в циклических операциях и операциях сравнения.
- **Флаг четности PF**. Содержит 1, если в результате операции получено число с четным количеством значащих разрядов, т. е. дополняет результат до нечетного числа — используется в операциях обмена для контроля данных.
- **Флаг внешнего переноса AF**. Контролирует перенос из 3-го бита данных. Полезен при операциях над упакованными десятичными цифрами.
- **Флаг нуля ZF** (Zero Flag). Равен 1, если в результате операции получен 0, и равен 0 — в противном случае.
- **Флаг знака SF** (Sign Flag). Равен 1, если в результате операции получено отрицательное число (с единицей в старшем разряде, т. е. последний бит (биты нумеруются справа налево)).
- **Флаг трассировки TF** (Trass Flag). Равен 1, если программа выполняется по шагам, с передачей управления после каждой выполненной команды прерыванию с вектором 1.

- *Флаг прерывания IF* (Interrupt Flag). Содержит 1, если разрешена обработка прерываний микропроцессором.
- *Флаг управления DF* (Direction Flag). Управляет направлением передачи данных: если он содержит 0, то после каждой индексной операции содержимое индексных регистров увеличивается на 1, в противном случае — уменьшается на 1.
- *Флаг переполнения OF* (Overflow Flag). Равен 1, если в результате операции получено число, выходящее за разрядную сетку процессора, т. е. число, превышающее максимальный размер регистра или переменной.

3. Я только начинаю изучать ассемблер, но у меня уже возникла необходимость иметь документацию по всем прерываниям MS-DOS. Где можно ее найти?

Документацию на русском языке можно скачать на сайте <http://www.Kalashnikoff.ru>. Программа называется helpassm.exe.

Если у вас возникли вопросы по работе с ассемблером, вы можете свободно обращаться к нашим экспертам портала **RFpro.ru**, которые дадут вам исчерпывающие ответы. Желаем вам успехов!



Глава 23

Область PSP и DTA. Системные переменные (окружение DOS)

Настоящая глава информативная и достаточно интересная. В ней мы рассмотрим то, что не затрагивалось в предыдущих главах, а именно:

- что такое системные переменные (в MS-DOS назывались окружением DOS);
- как уменьшить размер резидента в памяти;
- как менять "на лету" параметры резидента;
- как получить переданные нашей программе параметры в командной строке;
- как фиксировать в файле нажимаемые пользователем клавиши (в том числе и ввод паролей в программах командной строки);
- и прочее, по мелочам.

Прежде чем начнем рассматривать подробно работу резидента, мы хотели бы обратить ваше внимание на то, как теперь задается имя файла, в котором хранятся нажатые пользователем клавиши (в дальнейшем этот файл именуется LOG-файлом).

В одной из предыдущих глав мы также сохраняли действия программ (запуск, создание, удаление файла, создание/удаление каталога и пр.) в LOG-файле, имя которого указывалось в ассемблерном листинге. Впоследствии, после ассемблирования, мы не могли менять имя LOG-файла и его местоположение в каталоге (если, конечно, не менять его вручную непосредственно в СОМ-файле, используя какой-нибудь текстовый редактор типа Hacker's View). Теперь же наш резидент стал более гибким. Имя LOG-файла можно задавать при запуске программы, указав его в параметрах командной строки. Например:

```
RESID23.COM c:\assm\log_file.txt
```

После загрузки резидент создаст файл C:\assm\log_file.txt, в который и будет заносить нажатые пользователем клавиши.

В настоящей главе рассмотрим, как можно менять параметры резидента "на лету". Для начала зададим себе несколько вопросов.

Допустим, мы загрузили резидент, указав перед его запуском имя LOG-файла A:\super\file.log. Имеется ли возможность изменить имя LOG-файла, не перезагружая резидента, т. е. не удаляя его из памяти, а затем, не загружая его, опять с новым параметром в командной строке? Можно ли запускать резидент без каких-либо параметров вообще? И всегда ли нужно указывать параметры?

23.1. Структура командной строки

Прежде всего разберемся, что собой представляют параметры командной строки. Мы уже вкратце рассматривали данную тему, а сейчас заполним пробелы.

Параметры командной строки указываются при запуске файла и должны располагаться после имени запускаемого файла. Например:

SYS.COM C: A:

Здесь SYS.COM C: A: — командная строка, а C: A: — параметры командной строки, которые передаются файлу SYS.COM после загрузки. В данном примере SYS.COM скопирует системные файлы с диска C: на диск A:. Как же нашей программе получить эти параметры?

Как уже упоминалось, параметры командной строки, передаваемые файлу, располагаются в PSP по смещению 80h. PSP, в свою очередь, находится в том сегменте, в который загрузился наш COM-файл. Проще говоря, первая инструкция COM-файла начинается со смещения 100h (org 100h), а по смещению 80h находятся параметры командной строки, а также по умолчанию DTA. Вспомните, мы переносили тело вируса в область экрана для того, чтобы не затереть эти параметры при поиске файлов.

Каким образом параметры командной строки записываются по этому адресу и кто или что это делает?

Все эти действия выполняет операционная система, и программисту нет необходимости делать что-либо дополнительного.

Что именно находится по смещению 80h и какова структура размещения параметров?

Первый байт, расположенный по смещению 80h, указывает на длину командной строки. Если файлу никаких параметров не было передано, то этот байт будет равен нулю. Второй байт, как правило, — пробел (20h). За ним идет собственно то, что мы указали после имени запускаемого файла. Заканчиваются параметры командной строки символом 0Dh (код клавиши <Enter>). Для наглядности представим это в табл. 23.1.

Таблица 23.1. Командная строка

Смещение	Значение
80h	Длина строки
81h	<Пробел> (20h = 32)
82h	Параметры
?	Символ 0Dh (13)

Допустим, мы запускаем программу SYS.COM таким образом:

SYS.COM C: D:

Тогда по смещению 80h будет находиться следующее (табл. 23.2).

Таблица 23.2. Данные командной строки

Смещение	Значение
80h	06h
81h	<Пробел> (20h = 32)
82h	43h 3Ah 20h 44h 3Ah
87h	Символ 0Dh (13)

ПРИМЕЧАНИЕ

43h 3Ah 20h 44h 3Ah — ASCII-символы "C: _ D:" (символ "_" — пробел).

Длина 6 байт устанавливается потому, что система считает пробел, расположенный по адресу 81h, частью командной строки.

Параметры командной строки можно наблюдать в отладчике. Для этого загрузите программу под отладчиком следующим образом:

afd resid23.com params

где params — параметры командной строки (рис. 23.1).

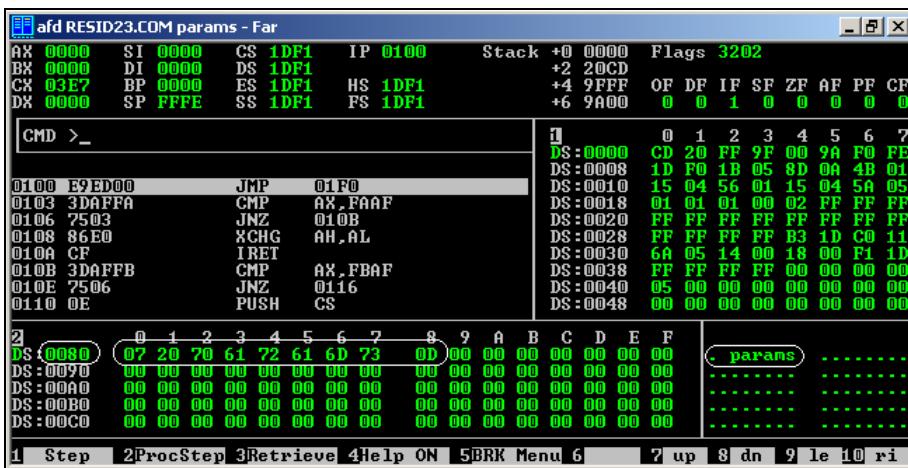


Рис. 23.1. Параметры командной строки в DTA после загрузки программы

23.2. Системные переменные (окружение MS-DOS)

Системные переменные содержат параметры, устанавливаемые автоматически при запуске консоли либо вручную с помощью внутренних команд. Пример системных переменных:

PATH C:\WINDOWS;C:\WINDOWS\COMMAND

COMSPEC=C:\COMMAND.COM

TEMP=C:\TEMP

PROMPT=\$P\$G

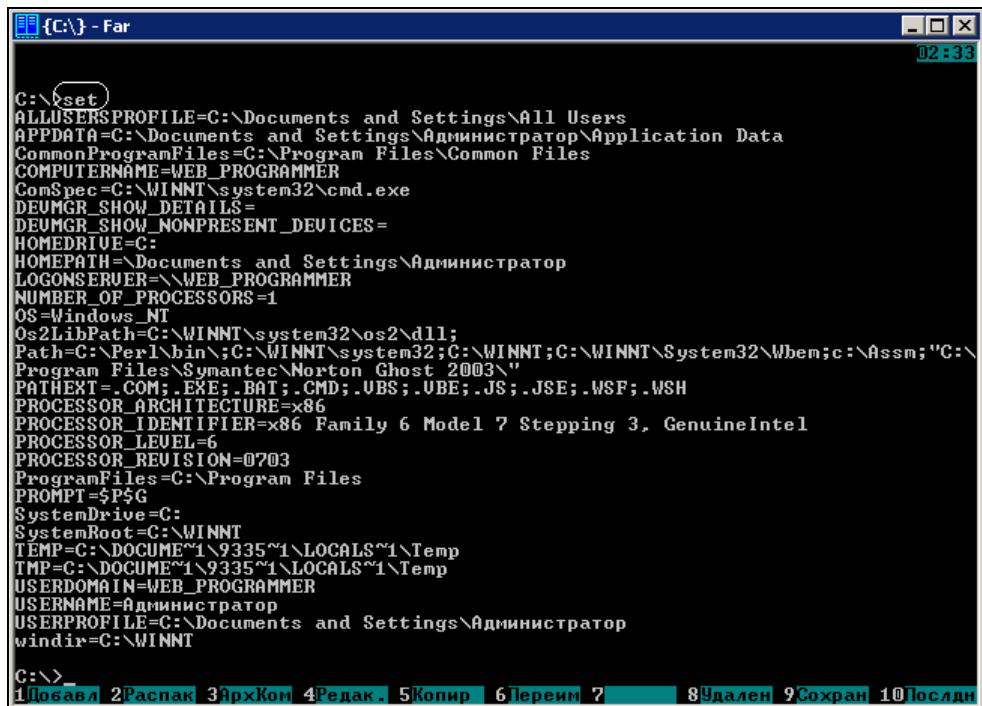
... и прочие.

Системные переменные передаются каждой программе, которая загружается. Причем не важно, резидент это или обычная программа. Читая данные переменные, многие программы находят файлы по пути, который содержит переменная path. Более того, вирусы могут заражать файлы, используя значение этой переменной.

Посмотреть системные переменные можно с помощью внутренней команды SET, набрав ее в командной строке. На рис. 23.2 показаны эти переменные под операционной системой Windows 2000. Изменения системных переменных с помощью команды SET действительны только для текущего сеанса консоли.

Также посмотреть и изменить эти переменные можно следующим способом: правая кнопка мыши на значке **Мой компьютер**, далее **Свойства | Дополнительно | Переменные среды | Системные переменные**. Изменение переменных в данном окне сохраняется и действует при следующем сеансе работы в операционной системе.

Эти параметры создает DOS автоматически после загрузки программы в память, но до передачи ей управления и заносит в сегмент, адрес которого находится в PSP по смещению 2Ch, причем смещение всегда равно нулю. Пример чтения окружения рассмотрим далее. Сейчас разберем, чем оно может помешать нашему резиденту.



```
C:\> set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Application Data
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=WEB_PROGRAMMER
ComSpec=C:\WINNT\system32\cmd.exe
DEUMGR_SHOW_DETAILS=
DEUMGR_SHOW_NONPRESENT_DEVICES=
HOMEDRIVE=C:
HOMEPATH=\Documents and Settings\Administrator
LOGONSERVER=\WEB_PROGRAMMER
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Os2LibPath=C:\WINNT\system32\os2\dll;
Path=C:\Perl\bin\;C:\WINNT\system32\;C:\WINNT;C:\WINNT\System32\Wbem;c:\Assm;"C:\Program Files\Symantec\Norton Ghost 2003\";
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.UVE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 7 Stepping 3, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0703
ProgramFiles=C:\Program Files
PROMPT=$P$G
SystemDrive=C:
SystemRoot=C:\WINNT
TEMP=C:\DOCUMENTS\19335\1\LOCALS\1\Temp
TMP=C:\DOCUMENTS\19335\1\LOCALS\1\Temp
USERDOMAIN=WEB_PROGRAMMER
USERNAME=Administrator
USERPROFILE=C:\Documents and Settings\Administrator
windir=C:\WINNT
C:\>
```

1Добавл 2Распак 3АрхКом 4Редак . 5Копир 6Переим 7 8Удален 9Сохран 10Послдн

Рис. 23.2. Параметры окружения DOS

Дело в том, что резидент, как правило, занимает больше места в оперативной памяти после его установки, чем на диске. Иными словами, СОМ-файл на винчестере, например, может иметь размер 130 байт, а в памяти — 350 и более байт. Почему так происходит? Вспомните, что перед вызовом прерывания 27h, которое оставляет программу в памяти, необходимо указать в регистре dx последний освобождающийся байт. Все, что расположено перед смещением, указанным в dx, остается в памяти. В табл. 23.3 приведено состояние памяти после загрузки на выполнение файлов типа СОМ.

Таблица 23.3. Состояние памяти после загрузки программы

Смещение	Значение	Примечание
0000h	Начало PSP. Команда int 20h	Остается в памяти вместе с резидентом
002Ch	Сегментный адрес окружения DOS	Остается в памяти вместе с резидентом
0080h	Командная строка и по умолчанию DTA	Остается в памяти вместе с резидентом
00FFh	Последний байт PSP	Остается в памяти вместе с резидентом
0100h	Метка начала программы (например: Begin)	Резидентная часть программы (останется в памяти). Вспомним, что метки памяти не занимают
?	Тело резидента. То есть то, что будет постоянно находиться в памяти	Резидентная часть
0134h	Метка, указывающая на то, что после нее можно освобождать память (например: Init)	Часть программы, которая будет удалена из памяти после вызова прерывания 27h
?	Инициализация резидента (вывод сообщений на экран, получение, сохранение и установка векторов прерываний и пр.)	Часть программы, которая будет удалена из памяти после вызова прерывания 27h
0154h	Команда int 27h (оставляет часть программы в памяти и выходит в DOS)	Часть программы, которая будет удалена из памяти после вызова прерывания 27h
0156h	Данные, строки для вывода и пр., которые не требуются резиденту в процессе работы. Необходимы только для того, чтобы вывести сообщения на экран, сохранить в процессе инициализации резидента некоторые переменные и пр. Удаляются из памяти после выполнения команды int 27h	Часть программы, которая будет удалена из памяти после вызова прерывания 27h

На рис. 23.3 показано состояние памяти после загрузки программы в память.

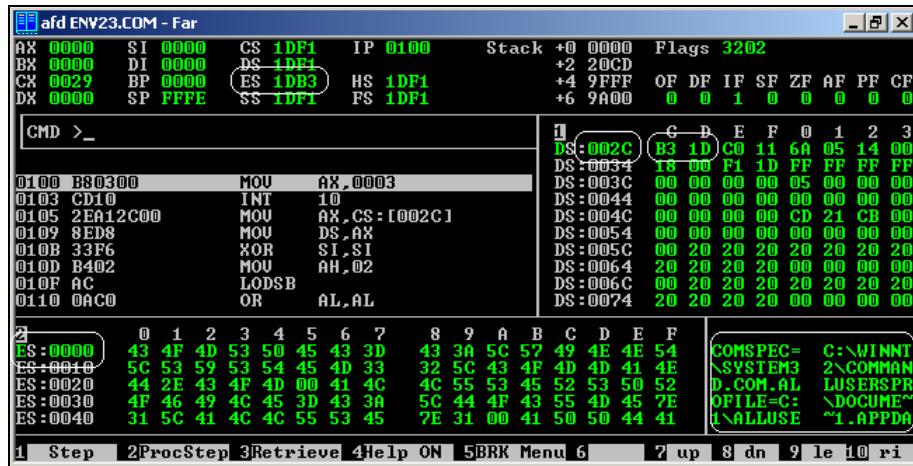


Рис. 23.3. Системные переменные в только что запущенной программе

Как уже упоминалось ранее, оставлять PSP в памяти вместе с резидентной программой обычно не нужно. В прилагаемом примере мы используем префикс (256 байт), как буфер. Для этого высчитаем размер программы и ее данных, остающихся в памяти: резидентная часть + 256 байт PSP.

Резидентная часть — та часть программы, которая остается в памяти. *Резидент* — более широкое понятие, включающее в себя еще и процедуру инициализации. Иногда называют резидентом и резидентную часть, что не совсем верно, зато коротко.

Но и это еще не все, что остается в памяти вместе с резидентной частью нашей программы. Помимо PSP, в памяти находится также и сегмент с окружением DOS, создаваемый операционной системой для каждого процесса (программы). Сегмент может занимать 32 Кбайта и более, но, как правило, не превышает 50—100 байт.

Каким образом можно избавиться от окружения DOS, тем самым, освободив дополнительно некоторое количество памяти для других программ? Подобная процедура была актуальна для программ, работающих под управлением MS-DOS и в режиме реального использования памяти. Максимальный объем ОЗУ в этом режиме составляет 640 Кбайт. Тем не менее, в данной главе мы рассмотрим, каким образом можно освободить память, занятую системными переменными, а также параллельно изучим кое-что новое.

В PSP по смещению 2Ch находится сегмент окружения DOS. Вспомните, как мы в одной из прошлых глав урезали и отводили блоки памяти. То же самое нам надо сделать и с памятью, занимаемой системными переменными. Вот, как это выглядит в файле-приложении (листинг 23.1).

Листинг 23.1. Освобождение памяти

```
...
mov es,word ptr cs:[2Ch]      ;Получим сегмент строки окружения.
mov ah,49h                    ;Функция освобождения памяти.
int 21h                       ;Освобождаем память...
...
```

В листинге нам встречается новая функция. Ее описание приведено в табл. 23.4.

Таблица 23.4. Функция 49h прерывания 21h: освободить блок основной памяти

Вход	Выход
ah = 49h es = сегмент распределенного блока	jc — ошибка, при этом: ax = код ошибки

После выполнения данной функции память освобождается. Это легко проверить, если запустить резидент, освобождая память (т. е. используя функцию 49h), а затем запустить резидент, не освобождая память.

Для того чтобы лучше понять, что же такое окружение DOS, к данной главе прилагается программа env23.asm, которая выводит на экран все системные переменные. Для более детального и наглядного изучения окружения DOS настоятельно рекомендуется вначале посмотреть результат работы этого файла (что именно он выводит на экран), а затем запустить его под отладчиком и тщательно изучить.

23.3. Основной резидент

Теперь остановимся подробней на работе резидента. Как уже упоминалось, наш резидент сохраняет в файле нажатые пользователем клавиши. Как именно он это делает?

В данном случае мы будем использовать PSP в качестве буфера, где будет храниться определенное количество ASCII-кодов нажатых клавиш, которые будут периодически (после выполнения определенных условий) записываться в специальный LOG-файл. Почему периодически? Можно, конечно, сохранять сразу же нажатую клавишу, но это будет не совсем красиво, да и пользователь может заподозрить "что-то неладное", т. к. после каждого нажатия будет производиться запись на диск, при этом замедляя работу и выдавая тем самым присутствие нашей программы в памяти. Эта проблема исчезает, если загружена программа кэширования записи или этот режим включен системой. Однако мы будем рассчитывать на то, что кэширование дисков отключено. В таком случае, гораздо проще сохранять в буфер, скажем, максимум 80 символов, а затем их сбрасывать в файл. Более того, мы сможем производить некоторую коррекцию вводимых пользователем строк. Например, замеченные опечатки, которые будут меняться при нажатии клавиши <Backspace>. Мы в памяти также будем это делать, что избавит LOG-файл от "мусора".

Итак, в качестве буфера для нажатых пользователем клавиш будем использовать PSP. Все равно память, отведенная под PSP, резиденту не потребуется. А это, как уже отмечалось, лишние 256 (100h) байт.

Клавиши будут сохраняться, начиная со смещения 0 сегмента, в который загрузился наш резидент. По смещению 00F6h будем хранить смещение, по которому следует заносить очередной символ (и количество введенных пользователем символов). В принципе, хватило бы и одного байта, но слово удобнее будет использовать, например, при записи символов в файл.

При каких условиях будет осуществляться сброс буфера в LOG-файл?

Запись содержимого буфера будет производиться в случае, если:

- пользователь нажмет клавишу <Enter>;
- пользователь нажмет клавишу, имеющую расширенный код (например: <F1>—<F10>, клавиши со стрелками и пр.);
- пользователь введет 80 символов, ни один из которых не будет равен коду, перечисленному в двух предыдущих пунктах (дабы избежать переполнения буфера). В буфер очередной символ заносит процедура `Store_sym` (листинг 23.2).

Листинг 23.2. Процедура `Store_sym`

```
...
;Занесем в di количество введенных символов:
(1)    mov di,cs:[0F6h]
;Оно больше 79?
(2)    cmp di,79
;Нет, меньше. Тогда дописываем в буфер очередной символ.
(3)    jb OK_store
(4)    push di           ;Иначе сбросим буфер в файл.
(5)    push ax
(6)    call Save_string
(7)    pop ax
(8)    pop di

(9)  OK_store:
(10)   stosb            ;Занесем очередной символ в буфер.
(11)   inc word ptr cs:[0F6h] ;Увеличим счетчик.
(12)  ret                ;Конец процедуры...
...

```

23.3.1. Команды безусловного перехода

Никаких вопросов здесь не должно возникнуть, кроме вопроса по команде из строки (3). Обратите внимание, что в строках (2), (3) мы проверяем, больше ли число 79, чем то, которое находится в регистре `di`. Если меньше, то переходим на метку `OK_store`. Это, как вы уже знаете, — условный переход. Однако данную инструкцию мы пока не рассматривали. В принципе, должно быть все понятно. `je` (от англ. *jump if equal* — переход, если равно) — это переход, если приемник равен источнику, а `jb` (от англ. *jump if below* — переход, если меньше) — это, как уже ясно из перевода, переход, если приемник меньше источника (т. е. в данном случае, если `di` меньше 9). Список команд условного перехода приведен в табл. 23.5, а пример их использования — в листинге 23.3.

Таблица 23.5. Список команд условного перехода

Команда	Описание
JB (Jump if Below)	Переход, если приемник меньше источника
JBE (Jump if Below or Equal)	Переход, если меньше или равно
JNB (Jump if Not Below)	Переход, если не меньше (равносильна JAE)
JA (Jump if Above)	Переход, если больше
JAE (Jump if Above or Equal)	Переход, если больше или равно
JNA (Jump if Not Above)	Переход, если не больше (равносильна JBE)

Листинг 23.3. Пример использования команд условного перехода

```
...
mov ax,34
cmp ax,35
...
```

Переход будет выполнен при использовании следующих команд после сравнения:

- JNE;
- JAE;
- JNA;
- JB;
- JBE.

Переход не будет выполнен при использовании следующих команд после сравнения:

- JE;
- JA;
- JNB.

Ваша задача сейчас — внимательно разобрать и проанализировать приведенные выше команды.

23.3.2. Команды управления флагами

Команд управление флагом переноса (CF) существует две: stc и clc (табл. 23.6 и 23.7).

Таблица 23.6. Команда stc

Команда	Перевод	Назначение	Процессор
stc	Set carry flag — установить флаг переноса	Установка флага переноса	8086

Данная команда устанавливает флаг переноса, а следующая — сбрасывает его.

Таблица 23.7. Команда clc

Команда	Перевод	Назначение	Процессор
clc	Clear carry flag — сбросить флаг переноса	Сброс флага переноса	8086

Для чего нужны данные команды? Где они обычно применяются?

В наших примерах мы ими в основном пользуемся перед выходом из процедуры. Например, возьмем часть такого кода (листинг 23.4).

Листинг 23.4. Использование команд управления флагом переноса

```
...
call Find_symbol
jc Not_found
...
...
```

В данном примере вызывается процедура поиска какого-то символа в памяти. В конце этой процедуры мы устанавливаем флаг переноса, если символ найден, и сбрасываем, если нет. Это проще и быстрее, чем если бы мы использовали для этой цели какой-нибудь регистр (листинг 23.5).

Листинг 23.5. Использование регистра

```
...
call Find_symbol
cmp ax,1
je Not_found
...
...
```

Если $ax=1$, то символ не найден. Как видите, удобнее пользоваться флагом переноса. Однако следует помнить, что некоторые команды могут менять этот флаг. Поэтому нужно придерживаться следующих правил:

- между установкой/сбросом флага переноса и его проверкой не следует вызывать никакие прерывания или пользоваться командами типа add, sub, mul и пр.;
- после установки/сброса флага его нужно проверять как можно быстрее.

Рассмотрим еще две команды управления флагами направления (табл. 23.8, 23.9).

Таблица 23.8. Команда std

Команда	Перевод	Назначение	Процессор
std	Set destination flag — установить флаг направления	Установка флага направления	8086

Таблица 23.9. Команда cld

Команда	Перевод	Назначение	Процессор
cld	Clear destination flag — сбросить флаг направления	Сброс флага направления	8086

Какие действия выполняют эти команды? Для чего служит флаг направления?

Данный флаг служит при указании направления для инструкций работы со строками (`lod$, stos, movs` и пр.). До сих пор мы перемещали байты только вперед, например, от 0 до 4000, а не от 4000 до 0 (вспомните вирус, который перемещал свой код в область видеопамяти). Вот этот флаг отвечает за то, каким образом производить перемещение, поиск и пр. Иначе говоря, в каком направлении это делать (отсюда и название флага — флаг направления). Как правило, применяют направление вперед, и, следовательно, флаг обычно сброшен. Но бывают случаи, когда необходимо установить флаг, тем самым производя работу "назад". В нашем резиденте мы сбрасываем флаг направления для того, чтобы команды `lod$, stos` и пр., используемые в нем, работали "вперед".

Как уже упоминалось, некоторые прерывания (в том числе и 09) могут быть вызваны в любой момент работы какой-нибудь программы (в тот момент, когда пользователь нажмет клавишу). Установлен ли этот флаг или сброшен — нам не известно. Проще всего сбросить его самим в начале резидентной части, что мы, собственно, и делаем:

```
cld      ;Направление — вперед!
```

Обращаем ваше внимание еще раз: как правило, этот флаг сброшен, но лучше не рисковать и самим сбросить или установить его. Практиковаться с ним мы будем позже, при написании оболочки.

23.3.3. Изменение параметров резидента "на лету"

Допустим, наш резидент загружен в память и уже успешно работает (т. е. в LOG-файле фиксирует нажатые пользователем клавиши). А что, если мы хотим поменять имя LOG-файла в процессе работы резидента? Для этой цели будем использовать процедуру обработки прерывания 10h. Впрочем, не только для этой цели. Она нам необходима еще и для того, чтобы проверять резидент на повторную загрузку (листинг 23.6).

Листинг 23.6. Процедура обработки прерывания 10h

```
...
(1) Int_10h_proc proc
(2)     cmp ax,0FAAFh      ;Проверяем на повторную загрузку?
(3)     jne Next_step
(4)     xchg ah,al
```

```

(5)    iret

(6) Next_step:
(7)    cmp ax,0FBAFh      ;Получаем текущий адрес LOG-файла?
(8)    jne Run_int
(9)    push cs            ;Заносим в es сегмент LOG-файла
(10)   pop es
(11)   mov di,offset File_name ;Заносим в di смещение LOG-файла
(12)   iret
(13) Run_int:
(14)   jmp dword ptr cs:[0F8h]
(15) Int_10h_proc endp
...

```

В строках (2)—(5) мы проверяем, вызывается ли прерывание 10h с числом 0FAAFh. Если вызывается, то меняем местами ah/al, что сигнализирует нашему резиденту, загружаемому повторно, о том, что он уже в памяти. Подобные вещи рассматривались в предыдущих главах, поэтому подробно останавливаться на этом не будем.

Нас сейчас больше интересуют строки (6)—(12). Если прерывание 10h вызывается с числом 0FBAFh в ah, то это значит, что наш резидент, загружаемый повторно, просит получить сегмент и смещение имени текущего LOG-файла. Для чего? Как уже упоминалось, резидент может менять в процессе работы имя своего LOG-файла. Для этого необходимо запустить его повторно, с указанием имени в командной строке. Например, так:

```
Resid23.com c:\newfile.txt
```

Резидент, при попытке повторно загрузиться в память, прежде всего проверит параметры в командной строке. Но как повторно загружаемому резиденту узнать, в каком сегменте находится его резидентная копия? Для этого мы воспользуемся созданной нами функцией 0FBAFh прерывания 10h, которое резидент перехватывает. В результате, прерывание 10h (а точнее, наша процедура обработки прерывания 10h) вернет в es сегмент, а в di — смещение имени LOG-файла (листинг 23.7).

Листинг 23.7. Получение сегмента

```

...
mov ax,0FBAFh      ;Получим сегмент и смещение имени LOG-файла
int 10h             ;Теперь es — сегмент, а di — смещение LOG-файла в памяти.
...

```

А теперь посмотрите первые команды нашего обработчика прерывания 10h, который, в случае получения в ah числа 0FBAFh, загружает в es:di сегмент и смещение LOG-файла и немедленно выходит из этого прерывания. Обратите внимание,

что при инициализации резидента мы сперва проверяем, загружен ли он в память или нет, и если загружен, то получаем адрес LOG-файла. Теперь осталось только перенести параметры командной строки в полученные сегмент и смещение!

В листинге 23.8 приведен фрагмент нашего обработчика прерывания 10h (`Int_10h_proc`).

Листинг 23.8. Обработчик прерывания 10h

```
...
cmp ax,0FBADh      ;Получаем текущий адрес LOG-файла
jne Run_int
push cs            ;Заносим в es сегмент LOG-файла
pop es
mov di,offset File_name    ;Заносим в di смещение LOG-файла
iret
...
```

23.4. Задание для закрепления сведений из данной главы

Обязательно разберитесь с прилагаемым файлом, т. к. информация, приведенная в нем, очень важна для понимания работы резидентных программ. Описаний в файле-приложении более чем достаточно.



Глава 24

Резидентный антивирус

ВНИМАНИЕ!

Наш антивирус корректно обезвреживает только вирус, код которого приведен в главе 20. Если вы добавили или убрали хоть один байт в коде вируса, то антивирус не сможет корректно вылечить зараженный файл!

24.1. Регистры микропроцессоров 80386/80486. Хранение чисел в памяти

Прежде чем приступить к рассмотрению работы нашего антивируса, ознакомимся вкратце с некоторыми регистрами процессоров 80386/80486. В табл. 24.1 приведен пример на основе аккумулятора.

Таблица 24.1. Регистры процессоров 386+

Количество разрядов	Регистр			
32	eax			
16	ax		ax	
8	ah	al	ah	al

Как видно из таблицы, eax — 32-разрядный регистр. Он может хранить число $65\,535 \times 65\,535$ (т. е. 65 535 в квадрате). До сих пор мы пользовались только 16-разрядными регистрами (ax, bx, cx и пр.). С данной главы начнем использовать и 32-разрядные. По большому счету, у нас нет острой необходимости пользоваться ими. Единственная цель — показать вам возможности 32-разрядных регистров, а также попробовать использовать их на практике.

В приведенной выше таблице мы рассмотрели только регистр eax. По аналогии с ним, можно добавить и регистры ebx, ecx, edx, edi, esi, ebp.

При использовании 32-разрядных регистров необходимо в ассемблерный листинг включить одну из следующих директив:

но никак не

.8086

.286

Это связано с тем, что 32-разрядные регистры появились только в процессоре 80386. Соответственно, нужно использовать и ассемблер-программу (MASM/TASM), которая поддерживает инструкции процессоров 386+. Если в процессе ассемблирования программа выдает ошибку на первой строке (.386), то вам необходимо искать ассемблер-программу более новой версии. Например, TASM 5.0 или MASM 6.13 (последнюю можно загрузить с сайта <http://www.Kalashnikoff.ru>).

Далее приведены простые примеры использования 32-разрядных регистров:

```
mov eax,0          ;eax=0, ax=0
mov eax,15h        ;eax=15h, ax=15h
mov ax,0FF00h      ;ax=0FF00h, eax=0FF00h
mov eax,12345678h  ;eax=12345678h, ax=5678h
```

Не стоит забывать, что данные в компьютере хранятся в обратном порядке. Примеры:

```
...
(1)    mov Variable,12345678h
(2)    mov eax,Variable
(3)    mov ax,word ptr Variable
...
(4) Variable dd ?
...
```

В строке (1) мы заносим в переменную `Variable` 32-разрядное число `12345678h`. Обратите внимание, что сама переменная должна иметь тип `dd` (от англ. *double word* — двойное слово) (строка (4)). В строке (2) мы загружаем в `eax` это число, при этом 32-разрядный аккумулятор становится равным `12345678h`. Ничего не меняется, поскольку мы обращаемся к данной переменной как к двойному слову и читаем ее как двойное слово. Однако в памяти число `12345678h` будет располагаться в обратном порядке, т. е. `78563412h`. При загрузке его в 32-разрядный регистр оно снова "перевернется".

Другое дело, когда программист обращается к подобным 32-разрядным переменным как к слову, выделяя из него два байта (строка (3)). Учитывая, что число хранится в памяти в обратном порядке, то после выполнения строки (3) `ax` будет равен `5678h`. Поэкспериментируйте с этим, т. к. только на экспериментах можно полностью понять принцип хранения чисел в памяти.

Обратите также внимание, как мы загружаем в `ax` число (строка (3)). Перед `Variable` находится `word ptr`. Это указывает процессору на то, что в `ax` нужно занести два байта (слово). `Word ptr` опускается, если переменная имеет тип `dw`, и загружаем мы в 16-разрядный регистр. Однако если программисту необходимо загрузить в 16-разрядный регистр число из 32-разрядной переменной, имеющей тип `dd`, то указывать `word ptr` нужно обязательно.

24.1.1. 16- и 32-разрядные отладчики

До сих пор мы часто использовали в своей практике 16-разрядный отладчик AFD Pro версии 1.0, распознающий инструкции процессоров 8086—80286. Если с его помощью мы попробуем отладить программу, использующую регистры и команды процессоров 80386+, то отладчик будет отображать их некорректно по вполне понятным причинам. Данная проблема может быть разрешена путем использования отладчиков, поддерживающих и распознающих машинные коды процессоров 80386+. Таковыми являются, например, CodeView, который входит в комплект MASM 6.13, TurboDebugger от компании Borland из комплекта Turbo Assembler либо SoftIce, предназначенный для опытных программистов.

На рис. 24.1 и 24.2 показано, как отладчики AFD и CodeView отображают 32-разрядные регистры. На рис. 24.3 показано, как включить режим просмотра 32-разрядных регистров в отладчике CodeView.

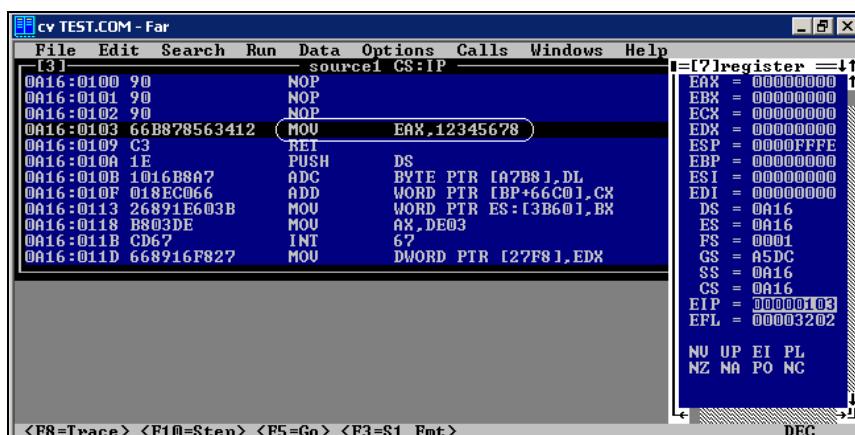


Рис. 24.1. Корректное отображение 32-разрядных регистров в CodeView

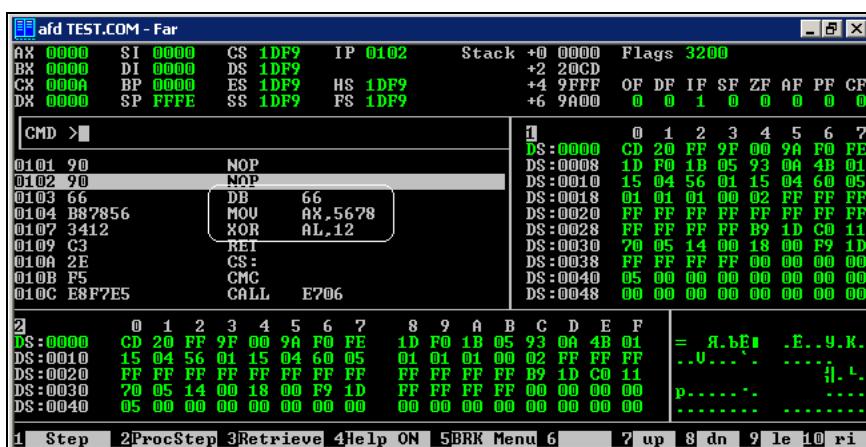


Рис. 24.2. Некорректное отображение 32-разрядных регистров в AFD

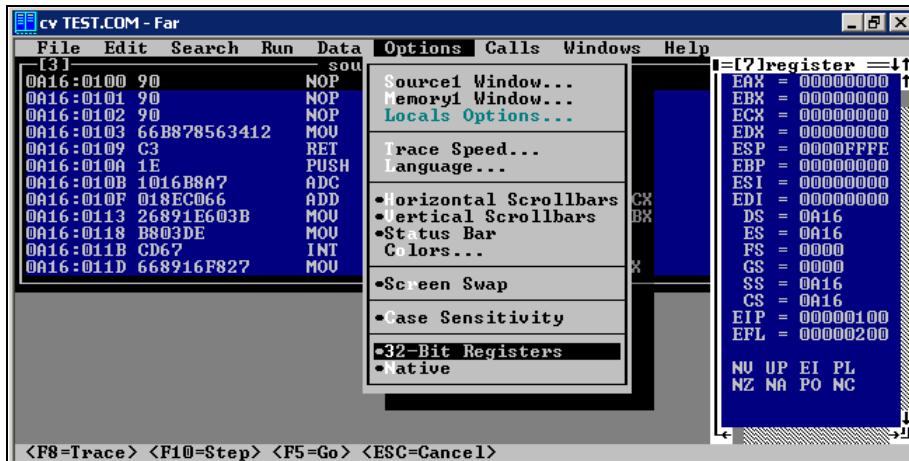


Рис. 24.3. Включение режима просмотра 32-разрядных регистров в CodeView

24.1.2. Директива *use16/use32*

При исследовании файла-приложения к данной главе обратите внимание на новую директиву: *use16*.

```
cseg segment use16      ;По умолчанию 16-разрядные данные
```

use16 сообщает ассемблеру о том, что по умолчанию будут использоваться 16-разрядные регистры, данные и в целом адресация.

Дело в том, что если в листинге программы стоит директива *.386* и выше, то ассемблер воспринимает всю адресацию как 32-разрядную по умолчанию (*use32*), которая в основном используется при написании полноценных 32-разрядных приложений под операционную систему Windows. Мы же пока пишем 16-разрядные программы, которые только в некоторых случаях будут использовать 32-разрядную адресацию. Поэтому, с точки зрения программирования, будет удобней явно указать *use16*. Здесь мы затронули эту директиву лишь по той причине, что она тесно связана с директивой *.386*. В настоящий момент вам необходимо уяснить следующее.

Убирая данную директиву *use16*, ассемблер-программа будет использовать *use32* (т. е. 32-разрядную адресацию) и выдавать ошибку на строках вида:

```
mov dx,offset Message
```

В данном случае (если мы уберем *use16*) нам следует записывать:

```
mov edx,offset Message
```

24.1.3. Сопоставление ассемблера и языков высокого уровня

Почему программы, написанные на языках высокого уровня, имеют больший объем, чем аналогичные на ассемблере?

В нашем примере мы постарались это показать. Обратите внимание, что файлу *antivr24.asm* сопутствует уже знакомый нам файл *display.asm* из оболочки, в кото-

ром собраны процедуры для работы с видеокартой. Для того чтобы заново не писать процедуру вывода окна на экран (а наш антивирус выводит окна прямым отображением в видеобуфер), мы просто воспользуемся уже готовыми и написанными нами процедурами.

Однако в файле `display.asm` существуют также подпрограммы, которые, например, прячут и восстанавливают курсор. Наш антивирус не будет управлять курсором, следовательно, подобные процедуры можно убрать, тем самым сократив объем программы. Да, на ассемблере это возможно. Но на языках высокого уровня — нет.

Например, в С используется директива `#include stdio.h`, которая подобна ассемблерной `include display.asm`. Однако программист не может посмотреть или изменить то, что находится внутри `stdio.h`. А в ней есть множество процедур, которые, скорее всего, программист и вызывать-то не будет. Отсюда и рост размера программы.

Обратите внимание, что мы специально не убирали подпрограмму `Hide_cursor` и подобную ей, имитируя, насколько это возможно, языки высокого уровня.

Вы можете добавлять файлы из оболочки для применения в своих программах, как мы это делаем в нашем антивирусе. Только не стоит забывать о переменных, используемых во вставляемых процедурах! Их можно и нужно размещать в той процедуре, которая их использует, тем самым упростив перенос в другие программы.

24.2. Резидентный антивирус. Практика

Все, что расположено после метки `Init`, вам уже известно. Обращаем только ваше внимание на то, что вызывать прерывание `21h` в резидентной части мы будем как `int 99h`, для чего скопируем оригинальный вектор `21h`-го в `99h`-й. Для чего это нужно — мы уже рассматривали в прошлых главах.

Наш резидентный антивирус будет находиться в памяти и контролировать открытие или запуск программы. "На лету" проверит на зараженность, и если программа заражена, то вылечит ее. Искать будем только наш вирус, который писали в *главе 20*. Для экспериментов возьмите его из этой главы. Еще раз предупреждаем: будьте предельно осторожны! Используйте только тот вирус, код которого приведен в *главе 20*, не изменяя в нем ни одного байта.

Наш резидент будет интенсивно использовать стек, поэтому лучше мы воспользуемся свободной памятью, предназначенней для PSP, на которую установим регистры `ss:sp`, во избежание переполнения стека программы, вызывающей прерывание `21h` (листинг 24.1).

Листинг 24.1. Перенос стека в область PSP

```
...
(1)    cli           ;Запретим прерывания
(2)    mov cs:[0],ss ;Сохраним сегментные регистры
(3)    mov cs:[2],sp
(4)    ;Установим стек на область PSP нашего резидента
```

```
(5)    push cs
(6)    pop ss
(7)    mov sp,0FEh
...
...
```

Обратите внимание, что перед сменой стековых регистров необходимо запретить прерывания командой `cli` (строка (1)).

Как видно из приведенного выше кода, мы сохраняем `ss:sp` в нашем сегменте по смещению 0 (т. е. в PSP нашего антивируса) (строки (2)—(3)). Отдельные переменные заводить не будем, т. к. это увеличит размер программы.

После того как мы изменили стек, необходимо настроить сегментные регистры (листинг 24.2).

Листинг 24.2. Настройка сегментных регистров

```
...
push ds
pop es
push cs
pop ds
...
...
```

Как мы уже рассматривали, перед вызовом функций `4Bh` (запуск программы) и `3Dh` (открытие файла), `ds` должен указывать на сегмент, а `dx` на смещение имени файла. Для того чтобы не затереть эти регистры, мы `ds` перенесем в `es`, а сам же `ds` сделаем равным `cs`. Теперь адрес запускаемого или открываемого другой программой файла находится в `es:dx`. Нужно внимательно следить за тем, чтобы корректно сохранить `dx`. Можно, конечно, занести его в переменную и не волноваться, но так, как мы делаем в нашем примере, будет проще, да и меньше байтов потребуется. Хотя, если подобных данных нужно хранить много, то лучше завести отдельную переменную.

Теперь вызываем процедуру `Check_prog`, которая проверяет, файл какого типа запускается/открывается — СОМ-файл или иной (листинг 24.3).

Листинг 24.3. Часть подпрограммы `Check_prog`

```
...
(1)    cld          ;Направление – вперед!
(2)    mov di,dx    ;Ищем в имени файла точку
(3)    mov al,'.'
(4)    mov cx,65    ;Всего будем просматривать 65 символов

(5) Next_sym:
(6)    repne scasb ;Ищем в es:di, пока НЕ найдем точку.
(7)    jne No_com   ;Не нашли точку вообще? Тогда на выход
...
...
```

Оператор `scasb` ищет в строке, адрес которой должен быть в `es:di`, символ, находящийся в `al`. Максимальная длина строки для поиска задается в регистре `cx`. Как мы помним, в `dx` осталось смещение имени запускаемого/открываемого файла, в `es` — сегмент. Следовательно, нам нужно в `di` загрузить содержимое регистра `dx` (`es` у нас уже готов) (смотрите строку (2)). В `al` заносим символ точки (.), а в `cx` — число 65, т. е. длина строки с именем файла не должна превышать 65 байт. Почему именно 65? Дело в том, что в DOS имя файла с полным путем к нему (т. е. имя диска + каталоги + имя файла + расширение) не должны превышать 65 байт.

Обратите внимание, что, как только инструкция `scasb` находит точку, `di` будет содержать смещение не на нее, а на следующий за ней байт.

Здесь стоит рассмотреть один недочет, который часто допускают новички. Представим, что открывается файл `readme` (без расширения). Тогда оператор `scasb` точку, естественно, не найдет. Может получиться, что в пределах 65 байт, в которых будет происходить поиск разделителя, идут машинные коды и один из них — `com+ASCII0`. Тогда наша программа посчитает, что это и есть файл с расширением `com`, а это не верно.

В данном случае лучший способ — искать сперва ASCII 0, а затем проверить, не являются ли три байта перед найденным нулем "com" или "COM". Тем не менее в файле-приложении используется описанный выше способ. Вы же можете без труда усовершенствовать наш антивирус, если будет в этом необходимость.

Итак, нашли точку в имени файла. Это может быть точка как каталога, так и файла. Чтобы удостовериться, что это имя файла, нам нужно проверить четыре байта, идущие за точкой + ASCII 0:

```
это com+ASCII 0 ?  
это COM+ASCII 0 ?
```

Как видите, в ассемблере очень важно следить за регистром (т. е. строчные и прописные символы различаются), точно так же, как и при нажатии клавиши.

Из приведенных выше проверок видно, что мы проверяем расширение файла плюс следующий за ним байт, который должен быть "нулем" (т. е. ASCII 0). Однако, если пользователь запустит файл примерно таким образом:

```
prog.com  
или  
prog.Com
```

то наша программа "не поймет", что это СОМ-файл. В данном примере мы не будем проверять все варианты. Можно сделать все очень просто: все символы представить ПРОПИСНЫМИ либо строчными и далее проверять только один вариант. Для этого изучите таблицу кодов и символов ASCII, приведенную в *приложении 3*, а затем, используя логические команды, напишите процедуру, которая будет преобразовывать любую строку в верхний или нижний регистр, игнорируя цифры, знаки препинания и пр. Подобное упражнение заставит вас немного поломать голову и научит не только пользоваться логическими командами процессора, но и составлять довольно интересные алгоритмы.

В листинге 24.4 приведен фрагмент кода, который проверяет расширение файла.

Листинг 24.4. Проверка расширения имени файла

```

...
mov ebx,es:[di]      ;Занесем в ebx четыре байта расширения файла+0
cmp ebx,006D6F63h    ;Это 'com'0 ?
je Got_file          ;Да!
cmp ebx,004D4F43h    ;Может, тогда 'COM'0 ?
jne Next_sym          ;Нет! Это было не расширение файла
...

```

Как видите, здесь мы в качестве демонстрации 32-битных регистров загрузили четыре байта в ebx, а затем произвели проверку. Посмотрите внимательно, как мы это делаем, но не забывайте про хранение данных в памяти и в регистрах в обратном порядке. Следовательно, и приемник в команде cmp должен быть расположен соответствующим образом, хотя в памяти храниться будет как "com". Еще раз подчеркиваем: очень сложно объяснить это на словах, но просто понять принцип на практике. Путаница никогда не возникает. Главное — привыкнуть и понять принцип.

Следующий шаг. Если оказывается, что запускаемый/открываемый файл — COM, то нужно проверить его на зараженность. Это делает процедура Check_file. В первых ее строках мы переносим имя запускаемого/открываемого файла в сегмент нашего антивируса по смещению 20 (листинг 24.5).

Листинг 24.5. Перенос имени найденного файла

```

...
push es                ;Настроим сегментные регистры...
pop ds
push cs
pop es
mov cx,65              ;Перенос 65 символов файла...
mov si,dx
mov di,20              ;... в PSP, начиная с 20-го байта.
rep movsb              ;Так для программиста гораздо удобнее.
push cs                ;Восстановим измененные регистры
pop ds
...

```

По смещению же 19 заносим атрибут для вывода имени файла в окно (так требует процедура Draw_frame). Здесь все понятно.

Дальше читаем первые 6 байт файла. Помните, наш вирус сохранял 2 байта — 1122h в начале файла. Мы их (а также первый байт 68h (т. е. команду push)) и будем проверять (листинг 24.6).

Листинг 24.6. Проверка на зараженность нашим вирусом

```

...
mov bx,ax          ; В bx - номер уже открытого файла
mov Handle,ax
mov ah,3Fh
mov cx,6
mov dx,10
int 99h          ; Читаем первые шесть байтов.
jc Not_infected ; Ошибка чтения!

mov ah,3Eh          ; Закроем файл.
mov bx,Handle
int 99h

cmp byte ptr cs:[10],68h    ; Первый байт - 68h (команда push)?
jne Not_infected          ; Нет - тогда файл не заражен!

mov eax,dword ptr cs:[12]   ; Берем следующие байты...
and eax,0FFFFFF00h          ; Обнулим один байт

; Проверяем: это метка нашего вируса (в обратном порядке, причем
; первый байт аннулирован)?
cmp eax,1122C300h
jne Not_infected           ; Нет! Файл чистый!

call Cure_file            ; Файл заражен нашим вирусом. Лечим его...
...

```

Перед лечением файла следует обратить внимание на использование команды `and`, которая обнуляет один "лишний" байт (см. листинг 24.6). Лечит файл процедура `Cure_file`, которая проста и не требует дополнительных пояснений. Алгоритм работы следующий:

1. Отводим блок памяти размером 64 Кбайт. Если отвести не удалось (может быть так, что вся память уже отведена какой-то программой), то будем использовать память текстовых режимов видеостраниц, в которой всего-то 28 Кбайт.
2. Как только отвели память, занесем ее сегмент в `es`, а количество возможных байтов для чтения — в переменную `Bytes_read`. Вызываем процедуру `Kill_zarazu`, которая вылечивает файл.

Как это практически происходит, думаю, вы без труда разберетесь, т. к. в файле-приложении достаточно описаний.

24.3. Резюме

В принципе, из данной главы вы почерпнули не так много нового для себя. Тем не менее, нам для полноты картины нужно разобрать работу антивируса. Вначале планировалось написать обычный нерезидентный антивирус, но это было бы слишком просто и неинтересно. Более того, мы не ставили перед собой задачу написать оптимальный алгоритм, а также красиво вывести информацию на экран. Все это вы без особого труда можете сделать самостоятельно либо в рассмотренном файле-приложении, либо в любой другой своей программе, используя материал из данной главы и из всех предыдущих.



Глава 25

Работа с сопроцессором

25.1. Ответы на некоторые вопросы

1. Что такое сопроцессор?

Сопроцессор (FPU, Floating Point Unit — устройство для работы с плавающей точкой) — это специальное устройство, устанавливаемое либо на материнскую плату, либо встраиваемое внутрь основного процессора (располагающееся на одном кристалле с ним).

2. Для чего нужен сопроцессор?

Сопроцессор служит для выполнения арифметических операций (сложение, вычитание, умножение, деление, вычисление квадратного корня и пр.) с плавающей точкой. Обычно для выполнения простых операций с целыми числами используется основной процессор, однако иногда можно прибегать и к помощи FPU. Сразу подчеркнем, что арифметические операции с *целыми числами* с помощью сопроцессора выполняются медленнее, чем основным процессором.

3. Как определить наличие сопроцессора в компьютере?

В старых машинах сопроцессор устанавливался на материнскую плату как отдельное устройство. Начиная с 80486DX, сопроцессор встраивается внутрь основного процессора. Можно с уверенностью сказать, что если на вашем столе стоит как минимум Pentium, то сопроцессор в нем присутствует.

4. А если нужно произвести сложные математические расчеты, можно ли не использовать сопроцессор вообще, а обойтись инструкциями центрального процессора?

В принципе, да. Но это будет очень сложно и довольно-таки медленно.

5. Сложно ли программировать сопроцессор на ассемблере?

В общем-то, нет. Главное — навык.

6. А почему мы начали рассматривать сопроцессор в данной главе? Что, наша оболочка будет работать только с ним?

Да.

7. А зачем? Можно ведь обойтись без него.

Во-первых, мы должны изучить не поверхности программирования на ассемблере, а затронуть по возможности все области. В том числе и сопроцессор.

Во-вторых, наши программы носят сугубо обучающий характер. Пройдя полный курс программирования на ассемблере, вы сможете написать собственную программу так, как вам этого хочется.

В-третьих, то, что мы делаем с помощью сопроцессора, сложно реализовать без него.

8. Как мы используем сопроцессор?

Мы будем выводить целые десятичные числа на экран. Например, размеры файлов. Это будет гораздо проще, чем использовать только процессор.

9. Сложно ли на ассемблере вывести десятичное число на экран? В Бейсике, например, достаточно набрать PRINT 25*4, и мы увидим результат умножения.

На ассемблере очень просто выводить на экран шестнадцатеричные и двоичные числа, но никак не десятичные. Поэтому только в *этой главе* мы затрагиваем данную тему, т. к. вы должны уже неплохо разбираться в командах ассемблера и понимать принцип программирования на этом мощном и очень простом языке.

10. Сильно ли отличаются ассемблерные команды сопроцессора и принцип его работы от программирования процессора?

Да, довольно-таки сильно. Сегодня вы сами лично "нащупаете" эту маленькую микросхему. По крайней мере увидите, как она работает...

11. А если мне ничего не будет понятно?

Не отчаивайтесь! Все станет предельно ясно со временем. Вспомните, у вас, вероятно, были трудности с тем или иным оператором, процедурой. Но в процессе экспериментов все стало на свои места. Так и здесь. Если вы сразу поймете все, что написано в данной главе, то можно сказать, что мы достигли той цели, которую ставили перед собой. Если нет — экспериментируйте, спрашивайте у наших экспертов на <http://RFpro.ru>. Этую тему желательно усвоить всем!

25.2. Введение в работу с сопроцессором

Прежде всего стоит отметить, что для того, чтобы использовать инструкции сопроцессора, необходимо включить в начало программы директиву .8087 (.287, .387). То есть указать, команды какого сопроцессора мы будем использовать. Принцип такой же, как и при указании процессора: либо только 8086 (.8086), либо 8086 и 80286 (.286) и т. д. В листинге 25.1 приведен пример включения директивы, которая указывает программе-ассемблеру (MASM/TASM), инструкции какого сопроцессора будут использоваться.

Листинг 25.1. Включение директивы сопроцессора

```
;Не забывайте указывать процессор, если вы собираетесь использовать  
;команды не только 8086!
```

```
; Будем использовать команды не только 386 процессора, но и 8087,  
; а также 80287 сопроцессоров.
```

```
.287
```

```
...
```

В некоторых случаях TASM может выдавать сообщение при попытке ассемблирования программы, начинающейся с указанных выше строк. Что-то типа: "Внимание: Вы используете команды процессора 386 и сопроцессора 287!" Зачем нужно было это делать — не совсем понятно! Программист может использовать, например, команды процессора 486, а также сопроцессора 8087. Ничего страшного в этом нет и быть не может! В любом случае просто игнорируйте это сообщение.

В распоряжении программиста имеется набор нескольких инструкций и 8 регистров (или 8 ячеек памяти) сопроцессора. Принцип загрузки числа в тот или иной регистр похож на принцип работы стека. Далее будем много тренироваться. Пока только теория.

Регистры сопроцессора имеют следующие названия: `st(0)`, `st(1)`, `st(2)`, ..., `st(7)`. Иногда для краткости `st(0)` называют просто `st`. Загрузить число в любой из указанных регистров командой `mov` не получится. Для этого существуют специальные команды сопроцессора.

Хорошим тоном перед использованием сопроцессора является его инициализация командой `fini` (от англ. *FPU initialization* — инициализация сопроцессора). Вообще, все команды сопроцессора начинаются с символа `f`, что является их отличительной чертой. Инструкция `fini` инициализирует сопроцессор. При этом сбрасываются все регистры, биты, флаги, а также происходит очистка 8 регистров `st`. Таким образом, после инициализации сопроцессора программист может быть уверен в том, что никакой регистр не занят и не получится путаницы или переполнения стека при попытке выполнить ту или иную операцию.

Сопроцессор может работать с 16—80-разрядными числами. Следовательно, в регистры `st(0)`—`st(7)` можно загружать число в указанных выше пределах. Сопроцессор сам определит, 16-разрядное ли это число или 64-разрядное.

Все числа из переменных загружаются в регистр сопроцессора `st(0)` и выгружаются из него в обычные переменные. Регистр `st(0)` является, скажем так, главным, основным, первичным. Только в него можно загружать числа из переменных и выгружать результат в переменные.

Например, возьмем команду `fld` (от англ. *FPU integer load* — загрузка целого числа в регистр сопроцессора `st(0)`), которая загружает в регистр `st(0)` число из переменной в памяти (листинг 25.2).

Листинг 25.2. Загрузка числа в регистр сопроцессора

```
...  
FILD Number1  
...  
Number1 dw 10  
...
```

Как видите, здесь мы указываем только имя переменной, значение которой будет загружено в регистр `st(0)` и только в него (как уже упоминалось). Допустим, нам нужно сложить два целых числа, используя сопроцессор. Для этого мы загружаем в `st(0)` число 10. Затем загружаем второе число — 3 (для выполнения операции сложения нужно как минимум два числа) (листинг 25.3).

Листинг 25.3. Загрузка двух чисел в регистры сопроцессора

```
...
(1) FILD Number1
(2) FILD Number2
...
Number1 dw 10
Number2 dw 3
...
```

Теперь внимание! После выполнения строки (1) `st(0)` будет содержать число 10. После выполнения строки (2) число из `st(0)` переместится в регистр `st(1)`, а его место в `st(0)` займет число 3. То есть работа системы похожа на работу стека! Далее нужно сложить числа, которые находятся в `st` и `st(1)`, используя команду сложения `FADD`.

`FADD`

Команда `fadd` (от англ. *FPU addition* — сложение с использованием регистров сопроцессора) без параметров складывает два числа, которые находятся в регистрах `st(0)` и `st(1)`, при этом очищая `st(1)` и помещая результат сложения в `st(0)`.

Рассмотрим, что происходит в регистрах в процессе выполнения данных команд (табл. 25.1). Итак, пока никаких действий мы не выполняли.

Таблица 25.1. Изначальное состояние регистров сопроцессора

Регистр	Значение
<code>st(0)</code>	Пусто
<code>st(1)</code>	Пусто
<code>st(2)</code>	Пусто
...	
<code>st(7)</code>	Пусто

Заносим первое число в регистр `st` командой `fld Number1` (табл. 25.2).

Таблица 25.2. Загружено одно число в регистры сопроцессора

Регистр	Значение
<code>st(0)</code>	10
<code>st(1)</code>	Пусто

Таблица 25.2 (окончание)

Регистр	Значение
st (2)	Пусто
...	
st (7)	Пусто

Теперь заносим второе число командой `fild Number2` (табл. 25.3).

Таблица 25.3. Заагружено второе число в регистры сопроцессора

Регистр	Значение
st (0)	3
st (1)	10
st (2)	Пусто
...	
st (7)	Пусто

Обратите внимание, что числа как бы вталкиваются внутрь. Как уже отмечалось, заносить числа в сопроцессор можно только в регистр `st (0)`. Поэтому-то второй параметр в команде `fild` отсутствует — и так все понятно...

Теперь произведем сложение двух чисел с помощью команды `fadd` (табл. 25.4).

Таблица 25.4. Результат выполнения команды FADD

Регистр	Значение
st (0)	13
st (1)	Пусто
st (2)	Пусто
...	
st (7)	Пусто

Как нам теперь получить результат сложения? Существует также команда, противоположная `fild`:

`fist Result`

Команда `fist` (от англ. *FPU integer store*) — сохранение целого числа) сохранит число, которое находится в регистре `st (0)`, в переменную `Result`.

Теперь ассемблируйте, запускайте отладчик и смотрите, что происходит.

Смеем предположить, что многие из вас столкнутся с проблемой: AFD выводит на экран совсем не то, что мы набрали, а именно: `wait`, `esc` и пр. Здесь ошибки нет.

Просто одна команда сопроцессора на самом деле преобразуется в выражения вида:

WAIT

ESC

Именно это и будет отображать на экране отладчик AFD, как показано на рис. 25.1.

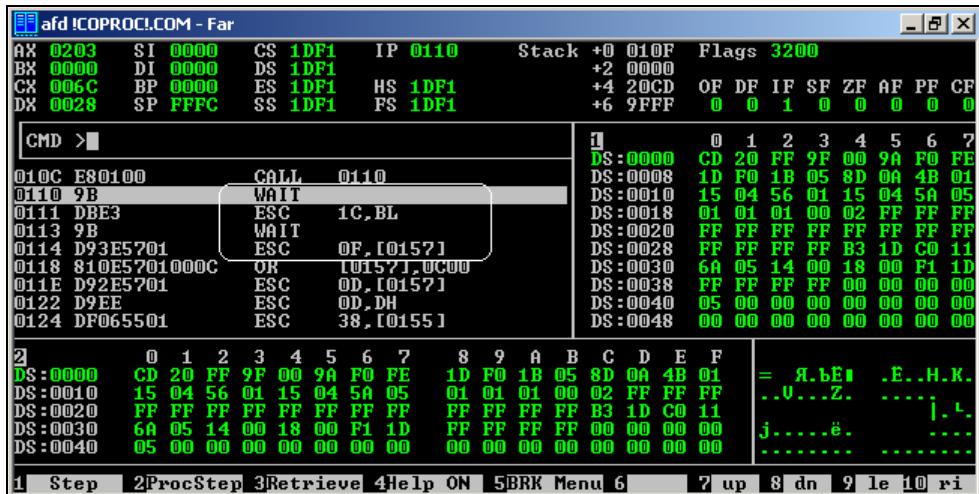


Рис. 25.1. Отладка программы с командами сопроцессора

Однако лучше всего отлаживать программу, которая использует инструкции сопроцессора, в отладчике TurboDebugger. Мы настоятельно рекомендуем производить отладку и смотреть результаты выполнения операций именно в этом отладчике, т. к. он очень удобен и наглядно отображает все, что вам нужно. Более того, TurboDebugger позволяет настроить рабочий стол таким образом, чтобы пользователь видел все необходимое одновременно: регистры основного процессора и сопроцессора, результаты вычислений, код программы, дамп памяти и пр. Если вы посмотрели работу приведенной выше программы в TurboDebugger, то читайте дальше...

Вопрос: сложили два числа, получили результат, но ведь в `st(0)` осталось число 13! Получается, что нам нужно перед каждой арифметической операцией инициализировать сопроцессор командой `finit`, чтобы очистить регистры от результатов вычислений? Можно ли как-то убрать это число иным способом? Например, за один раз (одной командой) перенести его в переменную `Result` и освободить `st(0)`?

Ответ: конечно, можно. Существует ряд команд, облегчающих работу с сопроцессором. Описать все в одной книге невозможно. В данной главе мы рассмотрим определенное количество команд, используя которые вы уже сможете выполнять многие операции.

Как правило, мнемоника команд сопроцессора имеет определенную систему, подобную мнемоникам команд процессора: команды представляют собой сокращения английских слов. Например, известная вам инструкция `xchg` (от англ. *exchange* — обменять). Далее приведен список некоторых команд сопроцессора:

- `fild` (от англ. *integer load*) — загрузка целого числа (в регистр сопроцессора);
- `fadd` (от англ. *addition*) — сложение;
- `fist` (от англ. *integer store*) — сохранение целого числа (в переменную в памяти);
- `fistp` (от англ. *integer store and pop*) — сохранение целого числа и выталкивание его из `st(0)`.

Если в нашем примере заменить `fist Result` на `fistp Result` из приведенной выше программы, то в `st(0)` ничего не останется. Безусловно, это достаточно удобно. Советуем поэкспериментировать с этими операторами.

Прежде чем продолжим изучение, следует добавить еще несколько слов о работе сопроцессора.

Сопроцессор имеет специальный регистр управления. Он необходим, в частности, для указания FPU-метода округления действительного числа (вещественного числа, числа с плавающей точкой). Например, число 23,8 может быть округлено до 24, до 23 (в зависимости от установленных битов в регистре управления) либо не округлено вообще. Или, например, если установлен тот или иной бит определенного регистра, то сопроцессор производит вычисления либо с более высокой точностью, либо с более низкой — так в некоторых калькуляторах можно настроить разрядность (количество знаков после запятой).

Для чего мы это затронули? Дело в том, что мы будем устанавливать биты регистра управления сопроцессором `RC`, которые отвечают за округление числа. На практике все увидите...

25.3. Первая программа с использованием сопроцессора

Теперь практикуемся на более сложном примере. В листинге 25.4 приведена заключенная короткая программа, которая просто складывает два целых числа с помощью сопроцессора и помещает результат в переменную `Result`. Все очень просто. Самое главное, что нужно сделать, — посмотреть эту программу под отладчиком TurboDebugger, да так, чтобы были видны регистры сопроцессора при отладке. Дополнительная и более подробная информация об использовании сопроцессора находится в комментариях файлов-приложений.

Листинг 25.4. Сложение двух чисел с помощью сопроцессора

```
.8087      ;Использовать будем инструкции процессора 8086 и сопроцессора 8087
CSEG SEGMENT
ASSUME CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
ORG 100h
```

```

Begin:
    finit      ;Инициализируем сопроцессор
    fild Number1 ;st(0)=Первое число
    fild Number2 ;st(1)=st(0); st(0)=Второе число

    fadd       ;Складываем

    fist Result ;Результат сложения — в переменную Result

ret

Number1 dw 10
Number2 dw 3
Result dw ?

CSEG ENDS
END Begin

```

25.4. Вывод десятичного числа с помощью сопроцессора

Возьмем простое число 1234. Естественно, в десятичном формате. На первый взгляд кажется, что достаточно отделить четверку и вывести ее на экран, затем 3, дальше — 2 и т. д. Например, используя известную нам команду and:

```

...
mov AX,1234
and AX,0004 ;или AND AX,0Fh
...

```

Либо каким-то иным способом. Но проблема в том, что число 1234 будет представлено в шестнадцатеричном формате или — что вернее — в двоичном. Вот что мы увидим в отладчике:

```

...
mov ax,4D2h
and ax,4
...

```

И что же мы получаем в ax? В первом случае (and ax,0004) — 0, а во втором (and ax,0Fh) — 2. Почему? Вспомните информацию из прошлых глав, где мы рассматривали двоичную и шестнадцатеричную системы, а также логические команды.

Так или иначе, но подобные способы не годятся. Тогда как поступить? Нужно делить число на десять, записывая остаток от деления до тех пор, пока не останется ноль. Вот пример:

```

1234/10=123 ;остаток 4
123/10=12   ;остаток 3
12/10=1     ;остаток 2
1/10=0      ;остаток 1

```

Получили 4321. То есть 1234 в обратном порядке. Следовательно, и выводить на экран будем в обратном порядке.

А можно сразу в "нормальном" порядке выводить? Можно, конечно, разделить на 1000, затем на 100, 10, ... Но ведь числа бывают разные! И начинать деление в одном случае приходится с 1000, а в другом — с 10.

Есть другой вариант. Можно завести массив из 20—30 байт, в каждый байт массива заносить одну цифру. Однако проще всего, как уже отмечалось, делить на десять до тех пор, пока делимое (т. е. выводимое число, в нашем примере — 1234) не будет равно нулю, хоть и придется для этого выводить в обратном порядке.

Как нам разделить число на 10 и вывести остаток от деления на экран? Вот это мы и будем делать с помощью сопроцессора. Подобный алгоритм используется в нашей оболочке. Мы специально вынесли его в отдельный файл, чтобы вам было удобнее разобраться, как именно работает программа.

Следует сразу отметить, что более неуклюжих алгоритмов уже не придумаешь. Наш алгоритм будет слишком сложный и немного запутанный. Зачем же нужно было так делать? Дело в том, что хоть этот алгоритм и далек от совершенства, но зато он даст вам возможность понять принцип работы сопроцессора. Именно для этого следует разобрать приведенную далее программу в отладчике TurboDebugger или подобном ему.

Видите, сколько разных программ нужно иметь под рукой для того, чтобы писать на ассемблере! И это далеко не предел. При написании программ под операционную систему Windows необходимо будет воспользоваться известным отладчиком SoftIce, работающим в реальном режиме, а также дополнительными дизассемблерами и прочими программами. После того как вы изучите ассемблер, вы сможете без особого труда разобраться с любой программой, написанной на языке высокого уровня. Так или иначе, можно будет ее скомпилировать, а затем дизассемблировать и посмотреть код.

В завершение хотелось бы обратить ваше внимание, что в файле-приложении к этой главе вы найдете программу с именем !Соргос!.asm, которая выводит на экран число в десятичной форме с использованием инструкций сопроцессора. В данном файле вполне достаточно описаний, чтобы понять работу программы. Ассемблируйте и запускайте ее под отладчиком.

25.5. Оболочка

Что теперь делает наша оболочка? Не зря мы рассматривали работу сопроцессора и вывод с его помощью десятичных чисел. Наша оболочка теперь может выводить не только размер файлов, но и их длинные имена.

25.5.1. Получение и вывод длинного имени файла

Для получения и вывода длинного имени файла следует воспользоваться функциями 714Eh и 714Fh (а не 4Eh и 4Fh) прерывания 21h (листинг 25.5, рис. 25.2).

Листинг 25.5. Получение длинного имени файла

```

...
mov ax,714Eh      ;Функция поиска первого файла
;di должен указывать на буфер, куда будут записываться данные
;о найденном файле (типа DTA).
xor di,di
;на si пока не обращаем внимания.
xor si,si
;Ищем все возможные файлы. Это что-то вроде атрибутов файла
mov cx,0FFh
mov dx,offset All_files      ;Маска поиска (*.*)
int 21h
;Теперь в es:di находится информация о найденном файле!
mov Handle,ax    ;Номер процесса поиска файлов
...

```

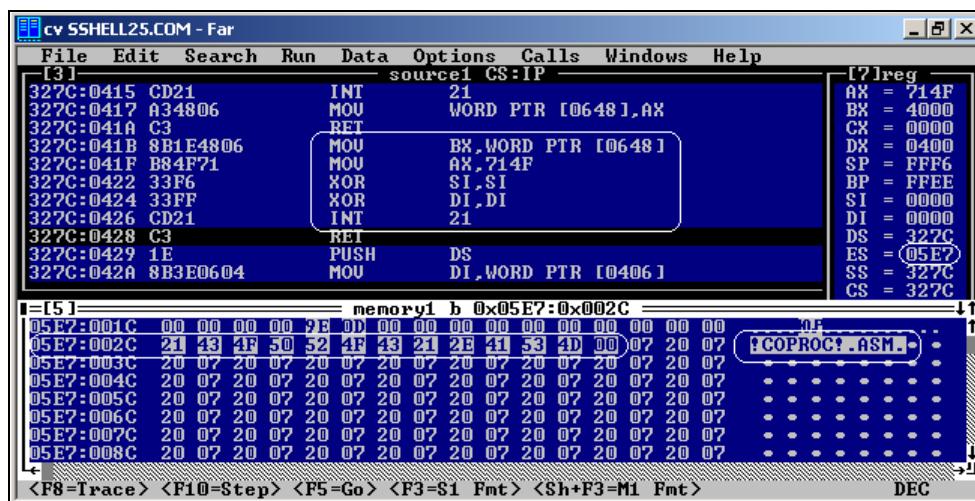


Рис. 25.2. Чтение длинных имен файлов

Отличие функции 714Eh еще в том, что она требует указания номера, который мы сохраним после вызова 714Eh. Все остальное подробно описано в файле-приложении (files.asm). Процедура вывода десятичных чисел находится в файле display.asm. Уверены, что разобраться с программой особого труда не составит.

Как всегда, все файлы-приложения содержат большое количество комментариев. И обычный совет: пользуйтесь отладчиком!



Глава 26

История развития ПК

В данной главе мы рассмотрим историю развития IBM-совместимых компьютеров, а также продолжим писать код нашей оболочки.

26.1. Краткая история развития IBM-совместимых компьютеров

Каждый программист на ассемблере должен знать историю развития ПК. В табл. 26.1 перечислены все модели IBM-совместимых компьютеров в порядке их создания и развития.

Таблица 26.1. История развития ПК IBM PC

Процес-сор	Тип	ОЗУ	Винче-стер	Частота (МГц)	Видео	Отечест-венный аналог
8086/8088	—	256—640 Кбайт	10—20 Мбайт	4,77—8	Mono, CGA	EC-1840, 1841, 1851...
80286	—	До 4 Мбайт	20—60 Мбайт	6—16	CGA-VGA	EC-1849
80386	SX/DX	4—8 Мбайт	40—120 Мбайт	16—40	EGA-VGA	EC-1863
80486	SX/DX	4—12 Мбайт	80—500 Мбайт	25—120	EGA-VGA	—
Pentium I	MMX	16 Мбайт и более	300—1000 Мбайт	60—266	VGA-sVGA	—
Pentium II	MMX	32 Мбайта и более	4 Гбайта и более	300—500	sVGA	—
Pentium III	MMX SSE	64 Мбайта и более	10 Гбайт и более	600—1000	sVGA	—
Pentium 4	MMX SSE	128 Мбайт и более	40 Гбайт и более	От 1000	sVGA	—
Intel Core Duo, Core 2 Duo, Core Quad	MMX SSE I, II, III	512 Мбайт и более	80 Гбайт и более	От 1000	sVGA	—

ПРИМЕЧАНИЕ

Приведенную в таблице информацию не следует считать обязательной для конфигурации того или иного компьютера. Например, при желании можно укомплектовать такую машину: IBM 486 SX-25, 10 Мбайт — винчестер, 32 Мбайт — ОЗУ, MDA — монитор.

26.2. С чего все начиналось

В 1981 году на свет появляется первый персональный компьютер *IBM PC* (IBM Personal Computer — персональный компьютер фирмы IBM). Затем, как уже отмечалось, — PCjr (Personal Computer junior), XT (Extended Technology), AT и т. д.

Процессор 80286 (PC/AT) мало чем отличался от своих старших "братьев". Он имел несколько новых команд (например, можно было помещать напрямую в стек не только значение регистров, но и само число: *push* *число*), а также позволял переводить компьютер в так называемый защищенный режим (*protected mode*). Сомневаемся, что многие программы использовали этот режим, т. к. он сильно отличался от режима, который начали применять в процессорах 80386. Немного увеличилась скорость работы, да и объем оперативной памяти слегка нарастили. Вот, собственно, и все.

Процессоры 80386 существенно отличались от всех предшествующих поколений компьютеров. В них было добавлено много новых функций. Прежде всего, это полноценный защищенный режим, много дополнительных команд для работы с ним, возможность управлять до 4 Гбайт ОЗУ и многое другое. По сути дела, современный Pentium — это не что иное, как обычная "тройка", только работающая в несколько раз быстрее и имеющая несколько дополнительных операторов (MMX, SSE).

"Четверки" (80486) не ушли далеко от 386. Единственное, что их отличало от "трешек", так это возможность размещения сопроцессора на одном кристалле с основным процессором. Существенно увеличилась скорость работы, в некоторых моделях шина стала 32-разрядной (PCI), что позволяло передавать за один раз 32-разрядное число, добавилось несколько новых команд и пр. Все остальное было так же, как у 386-х машин.

Pentium (сюда отнесем модели Pentium I, II, III) уже имел 64-разрядную шину. Более того, появились модификации MMX и SSE.

Расширение MMX имело дополнительный набор команд процессора для работы с мультимедиа. Эти команды, в частности, позволяли передавать данные буквально потоками, что существенно увеличивало пропускную способность, например, при прослушивании MP3-файлов либо при просмотре DVD-дисков и т. п.

Расширение SSE — это дополнительные команды Pentium III. Ничего принципиально нового они не внесли, хотя и ускоряли обмен данными.

Однако если программа (операционная система) не поддерживала инструкции MMX или SSE, то для нее Pentium MMX или Pentium III становился обычным Pentium.

Естественно, Pentium II и более современные модели обязательно включают в себя поддержку расширений MMX.

26.3. Оболочка

ПРИМЕЧАНИЕ

Прилагаемый алгоритм перехода по файлам при нажатии клавиш <↓> и <↑> очень сложный и в этой главе не завершен до конца! Это связано с тем, что написать полноценную процедуру управления клавишами со стрелками не такая простая задача, как может показаться на первый взгляд. Поэтому данную тему мы опишем в двух отдельных главах.

Как изменилась наша оболочка? Теперь она сразу после загрузки переходит в корневой каталог текущего диска, читает файлы, выводит их длинные имена на экран, размеры, текущий каталог вверху экрана, а также...

... А также теперь мы можем самостоятельно менять каталоги, т. е. "лазать" по диску и перечитывать содержимое каталога по нажатию комбинации клавиш <Ctrl>+<F3>! В общем, почти как в Norton Commander! Только существующий алгоритм, как уже отмечалось, не доведен до конца. Например, программа пытается перейти в каталог, который на самом деле является файлом! Или не делает вертикальную прокрутку на экране. Если в каталоге содержится много файлов, и они не помещаются на экран, то курсор просто исчезает из поля видения, когда мы переходим нижнюю границу. Проще всего самим посмотреть, что делает оболочка, и прямо сейчас (рис. 26.1).



Рис. 26.1. Общий вид оболочки

Далее рассмотрим следующие алгоритмы:

- чтение каталога в память;
- поиск отмеченного файла и его вывод на экран должным образом.

Рассмотрим алгоритм считывания файлов в память и перемещение по ним с помощью клавиш со стрелками.

26.3.1. Чтение файлов из каталога и размещение их в отведенной памяти

Есть три способа размещения файлов в отведенной памяти программы.

1. Самый примитивный и простой способ. Предположим, что работаем мы *исключительно* с короткими именами файлов, как это было в предыдущих версиях DOS, когда никто и не догадывался, что имена файлов в недалеком будущем будут иметь длину более 200 символов. Раньше все файлы имели длину не больше чем 12 байт: 8 символов имя файла + "точка" + 3 символа его расширение. В этом случае нам нужно отвести по 13 байт памяти для каждого имени файла. Почему 13? Первый байт будет сигнализировать о статусе файла, т. е. либо отмеченный, либо текущий, на который будет указывать курсор, либо текущий + отмеченный одновременно. Статус файла необходимо получать при проведении некоторых операций с файлами: копирования, удаления, запуска и пр. Таким образом, в 64 Кбайт отведенной памяти можно поместить более 5000 файлов!

Чем хорош данный способ?

По файлам просто "бегать" и искать отмеченный или текущий. Перемещаем указатель на 13 и... получаем смещение следующего файла. Вы улавливаете мысль? Если нет, то попробуйте "проработать" наш файл-приложение под отладчиком. Будет достаточно сложно, но довольно интересно!

2. Второй способ гораздо сложнее, зато экономит память. Файлы с длинными именами могут содержать чуть более 255 символов. Теперь посчитаем, сколько файлов мы сможем поместить в 64 Кбайт: $65\,536/255 = 256$. В один максимальный отводимый сегмент (64 Кбайт) можно поместить всего 256 имен файлов. Но некоторые каталоги содержат 400 и более файлов... Для этого нужно отводить еще один, два, три сегмента или использовать расширенную память (но этим будем заниматься позже), что усложняет задачу.
3. Поскольку длина имени файла не превышает 300 символов, то вместо адресов следующего и предыдущего имен можно хранить длины этих имен файлов. Это позволит освободить 2 байта на каждый файл, а для доступа к следующему или предыдущему имени файла просто прибавлять или отнимать длину + 3 байта.

На наш взгляд, второй и третий методы неэффективны в случае, когда необходимо сделать быстрый скачок через несколько файлов. Например, при нажатии клавиш $<\rightarrow>$ и $<\leftarrow>$, а также $<\text{PageUP}>$, $<\text{PageDown}>$. Гораздо эффективней будет использоваться память, если отдельно от списка файлов создавать массив с адресами их имен. В самом же списке файлов вместо двух адресов соседних имен файлов хранить индекс текущего имени размером в слово. Дополнительно к этому вместе с адресами имен имеет смысл хранить и статус файла для более быстрого доступа к нему. А текущий индекс будет храниться в переменной типа WORD.

26.3.2. Размещение файлов в памяти нашей оболочки

Допустим, в некотором каталоге существуют следующие файлы/каталоги:

- Assm.txt;
- Мои документы;
- Суперновый файл!.asm.

Теперь внимательно следите за тем, как размещаем их в памяти (табл. 26.2 и рис. 26.2). Для этого нужно будет вооружиться листком и ручкой. Допустим, мы размещаем файлы в сегменте 1234h, начиная со смещения 0000h (обратите внимание, что в оболочке загружаем файлы в память по смещению 500!).

Таблица 26.2. Принцип размещения файлов в памяти в нашей оболочке

Адрес	Значение
1234:0000h	Файл Assm.txt
1234:0000h	Статус файла(1 байт)
1234:0001h	Адрес следующего файла (2 байта)
1234:0003h	Адрес предыдущего файла (2 байта)
1234:0005h	Имя файла + ASCII 0 (assm.txt (8 + ASCII 0 = 9 байт))
1234:000Dh	Файл Мои документы
1234:000Dh	Статус файла Мои документы (1 байт)
1234:000Eh	Адрес следующего файла (2 байта)
1234:0010h	Адрес предыдущего файла (2 байта)
1234:0012h	Имя файла + ASCII 0 (Мои документы (13 + ASCII 0 = 14 байт))
1234:001Fh	Файл Суперновый файл!.asm
1234:001Fh	Статус файла Суперновый файл!.asm (1 байт)
1234:0020h	Адрес следующего файла (2 байта)
1234:0022h	Адрес предыдущего файла (2 байта)
1234:0024h	Имя файла + ASCII 0 (Мои документы (13 + ASCII 0 = 14 байт))

ПРИМЕЧАНИЕ

Внимательно изучите таблицу. Посмотрите, насколько поля, приведенные в данной таблице, отличаются от нашей оболочки.

Перед первым файлом мы заносим в поле "Адрес предыдущего файла" число 0000h. Когда пользователь нажмет клавишу <↑> на первом файле, наша программа,

прочитав смещение предыдущего файла, "поймет", что "выше" уже файлов нет, и не выполнит никаких действий.

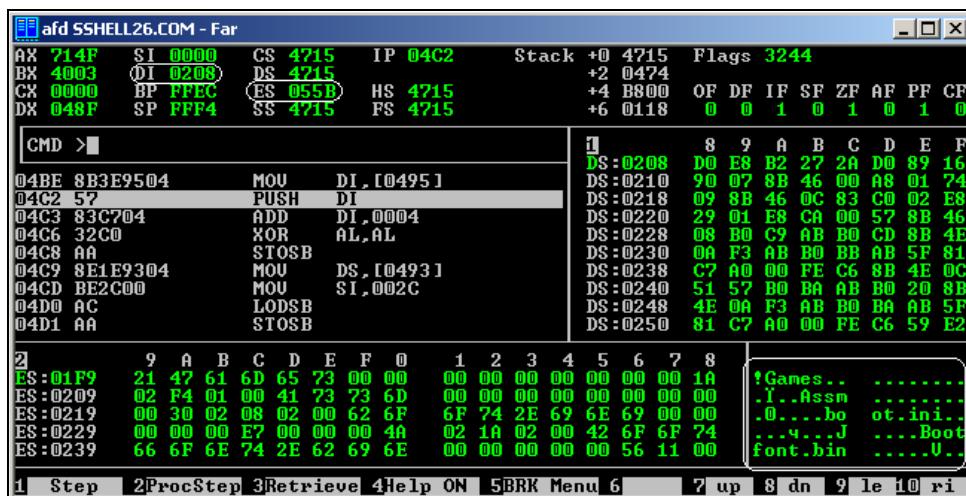


Рис. 26.2. Загруженные в память файлы из текущего каталога

Перед последним файлом также занесем в поле "Адрес следующего файла" число 0000h. Когда пользователь нажмет клавишу \downarrow , то наша программа "поймет", что "ниже" уже файлов нет, и ничего не произойдет.

На самом деле, практически все реализуется достаточно просто, но теоретически описать все это достаточно проблематично. Поэтому настоятельно рекомендую вам посмотреть принцип работы оболочки в отладчике. Особое внимание следует обращать на выделенный оболочкой блок памяти, кудачитываются имена файлов, а также на принцип перехода по файлам при нажатии клавиш со стрелками. Вы заметите, что подпрограммы нашей оболочки выполняют ювелирную работу. Занесенный по ошибке один байт "плюс-минус" разрушит всю базу данных по файлам. Безусловно, принцип будет понятен, когда вы проведете несколько часов перед монитором.

ЗАЧЕМ НУЖНО СИДЕТЬ ПЕРЕД МОНИТОРОМ И ИССЛЕДОВАТЬ ЭТО?

Ответим вопросом на вопрос: "Вы хотите научиться "разбирать по полочкам" чужие программы?" Поверьте, простая программа, написанная на Паскале, гораздо сложнее "понимается" под отладчиком, чем аналогичная программа на ассемблере. Наступит момент, когда вы захотите разобрать чужую программу (или уже захотели!), да вряд ли это получится, если вы не научитесь это делать, а тем более разобрать простейший файл-приложение к данной главе. Это будет для вас хорошей тренировкой. Опыт приходит со временем, и знания увеличиваются пропорционально часам, проведенным за отладчиком!



Глава 27

Удаление резидента из памяти

27.1. Обзор последнего резидента

27.1.1. Перехват прерывания 21h

В этой главе приведен последний пример резидентной программы под DOS, в которой мы рассмотрим принцип удаления резидента из памяти.

Казалось бы: неужели так сложно удалить резидентную программу из памяти? С одной стороны, ничего сложного в этом нет, с другой — существуют некоторые сложности:

- во-первых, не всегда есть возможность удалить из памяти резидентную программу;
- во-вторых, необходима некоторая подготовка и проверка определенных параметров перед удалением резидента;
- и наконец, резидентная часть программы, которая будет удалена, должна передавать по определенному запросу нужную информацию программе, которая ее удалит.

Все это мы рассмотрим в данной главе. Как обычно, освоить тему вам помогут файлы-приложения.

Рекомендую вам внимательно ознакомиться с приведенной в данной главе информацией, т. к. она поможет полностью понять механизм прерываний и принцип работы резидентных программ. Мы уже немало говорили и о прерываниях, и о резидентных программах, но, тем не менее, рассмотрим некоторые вещи повторно, с другого ракурса.

Итак, представим, что никакая резидентная программа не перехватывает прерывание 21h. То есть его адрес (вектор) указывает непосредственно на ядро DOS, как это показано в табл. 27.1.

Таблица 27.1. Вектор 21h указывает на ядро DOS

Адрес	Что перехватывает	Куда отправляет после обработки
1234:0000h	Ядро DOS	Никуда (возврат из прерывания)

Как вам уже известно, MS-DOS имеет свои обработчики некоторых прерываний (например, 20h, 21h и пр.). Использовать прерывания DOS без загрузки самой ОС невозможно. При попытке это сделать, компьютер просто зависнет. Например, в программе загрузочного сектора диска (которая загружается первая, т. е. до загрузки DOS) нельзя пользоваться функциями DOS, т. к. они еще недоступны. Но программе ничего не мешает использовать прерывания BIOS, т. к. они находятся в области ПЗУ и могут вызываться из любой ОС и в любое время.

Теперь внимательно посмотрите на табл. 27.1. Допустим, что процедура обработки 21h-го прерывания MS-DOS загрузилась вместе с ОС по адресу 1234:0000h. Никакая иная программа данное прерывание еще не перехватила. Таким образом, выполняя команду int 21h, процессор передаст управление напрямую процедуре обработки 21h-го прерывания MS-DOS, т. е. на адрес 1234:0000h. Данная процедура, отработав и, например, выведя на экран строку, передаст управление нашей программе. Это простейшая схема.

Теперь допустим, что пользователь загружает некий резидент с именем progA.com, который перехватывает прерывание 21h. В результате мы получим именно то, что изображено в табл. 27.2.

Таблица 27.2. Наша программа перехватила прерывание 21h

Адрес	Что перехватывает	Куда отправляет после обработки
1234:0000h	Ядро DOS	Никуда (возврат из прерывания)
2345:0000h	progA.com	1234:0000h (ядро DOS)

Из табл. 27.2 видно, что программа progA.com перехватила прерывание 21h. Прерывание как бы стало фильтром на пути к оригинальному обработчику (обработчику MS-DOS (1234:0000h)).

После вызова прерывания int 21h первым получит управление обработчик прерывания 21h программы progA.com. Процедура обработки прерывания 21h программы progA.com (в нашем случае она находится по адресу 2345:0000h), получив управление, сделает все необходимое, а затем передаст управление ядру DOS на адрес 1234:0000h. Ядро DOS выполнит определенные действия, а затем выйдет из прерывания командой iret, передав управление программе, которая вызывала прерывание 21h.

Отметим важный момент. Допустим, пользователь загружает обычную нерезидентную программу, которая работает с прерыванием 21h. Вызов данного прерывания этой программы (int 21h) находится по адресу 3456:0100h. Таким образом, при выполнении команды int 21h в стек будет занесен адрес 3456:0102h, т. е. адрес следующей команды, а процессор получит адрес прерывания 21h из таблицы векторов прерываний и передаст управление по этому адресу, т. е. на 2345:0000h, программе progA.com. В свою очередь, progA.com отработает (может, например, изменить значения каких-либо регистров, как наш резидент) и передаст управление ядру DOS.

Вот только вопрос: а куда именно передать управление, на какой адрес? Для этой цели наши резиденты сперва получают текущий адрес обработчика того или иного прерывания, а затем, когда обработчик отработал, передают управление на сохраненный адрес, т. е. дальше по цепочке.

Программа progA.com поступит точно так же. Перед установкой обработчика на свою процедуру она получает адрес прежнего (оригинального) обработчика (в данном случае — ядра DOS, т. к. никакие резидентные программы не загружены), вызвав функцию 35h прерывания 21h либо прочитав адрес напрямую из таблицы векторов прерываний.

ProgA.com отработает и передаст управление по предварительно сохраненному адресу на предыдущую процедуру обработки прерывания 21h.

Управление получит ядро DOS. Выполнит свои действия и вернется. Спрашивается: куда вернется и как? Естественно, с помощью команды iret, которая вытаскивает из стека предварительно занесенный в него адрес. Какой? Адрес progA.com или адрес программы, которая была запущена пользователем и инициировала вызов int 21h?

Все зависит от того, каким образом программа progA.com, выполнив необходимые команды в обработчике 21h, передала управление ядру DOS. Как уже не раз отмечалось в предыдущих главах, существуют два способа передачи управления прежнему (оригинальному) обработчику:

- jmp dword ptr cs:[Int_21h_vect]
- pushf
call dword ptr cs:[int_21h_vect]

В первом случае команда iret ядра DOS передаст управление непосредственно программе пользователя, которая вызывала прерывание 21h командой int 21h. Это и понятно: инструкция jmp в стек ничего не кладет. Получается, что перед выполнением команды jmp в стеке находится адрес возврата на программу пользователя (если, конечно, progA.com не нарушает работу стека). Команда iret обработчика 21h ядра DOS вытащит из стека адрес возврата на программу пользователя, которая будет работать дальше, даже не "подозревая" о том, что некая progA.com "сидит" в памяти и контролирует вызов прерывания 21h!

Во втором случае после выполнения iret управление получит progA.com. Здесь также все понятно: команда call заносит в стек адрес возврата. Команда iret и вытащит его из стека! Зачем мы перед call используем pushf — вам уже известно...

Итак, после завершения работы оригинального обработчика 21h (ядра DOS) программа progA.com снова получила управление. Что теперь будет делать progA.com? Она может посмотреть, что вернула подпрограмма обработки 21h и выполнить определенные действия, если необходимо. После этого командой iret продолжить работу пользовательской программы. Иными словами, iret вытащит из стека адрес возврата на программу, которая и вызывала изначально прерывание командой int 21h.

Для чего нужно программе progA.com перехватывать int 21h? Да по разным причинам. Например, чтобы контролировать открытие/закрытие файла, чтение/запись, да и все, что выполняет прерывание 21h! Некоторые варианты мы с вами рассматривали в предыдущих главах.

Итак, progA.com "повисла" на прерывании 21h, предварительно сохранив в своем теле адрес оригинального обработчика, т. е. обработчика ядра DOS, который в нашем примере расположен по адресу 1234:0000h. Теперь при выполнении команды int 21h управление получает сперва progA.com, а затем уже ядро DOS.

Теперь мы загружаем еще одну программу progB.com, которая также перехватит прерывание 21h. В табл. 27.3 показано, что у нас получится.

Таблица 27.3. Загружена вторая резидентная программа

Адрес обработчика	Кто перехватывает	Куда отправляет после обработки
1234:0000h	Ядро DOS	Никуда (возврат из прерывания)
2345:0000h	progA.com	1234:0000h (ядро DOS)
3456:0000h	progB.com	2345:0000h (progA.com)

Теперь некая загруженная пользователем программа (например, NC.EXE) вызывает прерывание командой int 21h. Что происходит?

Инструкция int 21h кладет в стек адрес возврата, достает из таблицы векторов прерываний адрес прерывания 21h (3456:0000h) и передает ему управление. Программа progB.com выполняет то, что программист ей "приказал", и передает управление на адрес прежнего (оригинального) обработчика прерывания 21h, который ею был предварительно сохранен (на 2345:0000h, т. е. progA.com). ProgA.com, в свою очередь, выполняет необходимые действия и так далее, по цепочке, пока управление в итоге не получит ядро DOS. Таким образом, на одно прерывание можно "насаживать" неограниченное число резидентов.

27.1.2. Как удалять загруженный резидент из памяти?

Загрузим в память Resid27.com — написанную нами программу для практики. Состояние памяти отображено в табл. 27.4.

Таблица 27.4. Состояние памяти после загрузки нашего резидента

Адрес	Что перехватывает	Куда отправляет после обработки
1234:0000h	Ядро DOS	Никуда (возврат из прерывания)
2345:0000h	Resid27.com	1234:0000h (ядро DOS)

Сразу возникает несколько вопросов.

- Как узнать, по какому адресу в памяти загрузилась программа Resid27.com?
- Как узнать адрес обработчика прерывания ядра DOS?
- Как удалить программу из памяти?

Все ответы достаточно просты.

Помните, как мы передавали "позвывной" нашему резиденту? Если резидент откликался, это означало, что он уже загружен.

Таким же способом можно получить сегмент и смещение процедуры обработки прерывания 21h. Сперва проверим на повторную загрузку (`mov ax, 9988h/int 21h`), а затем отправим в ответ какое-нибудь число (в нашем примере — 9999h).

Наш резидент, получив 9999h в ax, моментально выходит из процедуры, передавая в определенных регистрах необходимую информацию, а именно: сегмент и смещение резидентной части, сегмент и смещение оригинального обработчика прерывания 21h (в данном случае — ядра DOS). Этого будет вполне достаточно для удаления резидента.

Затем, убедившись, что резидент загружен, и получив необходимую информацию для удаления, можно приступать к освобождению памяти.

Делаем буквально следующее:

1. Запрещаем все прерывания командой `cli`.
2. Восстанавливаем адрес оригинального обработчика (т. е. ядра DOS).
3. Освобождаем память функцией `49h`.
4. Разрешаем прерывания командой `sti`.

Все! Резидент удален из памяти!

27.1.3. Случай, когда резидент удалить невозможно

Представим такую ситуацию. Вслед за Resid27.com загружается еще одна резидентная программа, которая также перехватывает прерывание 21h (в нашем примере это будет progA.com). Тогда получаем то, что показано в табл. 27.5.

Таблица 27.5. Загружена сторонняя программа после нашего резидента

Адрес	Что перехватывает	Куда отправляет после обработки
1234:0000h	Ядро DOS	Никуда (возврат из прерывания)
2345:0000h	Resid27.com	1234:0000h (ядро DOS)
3456:0000h	progA.com	2345:0000h (Resid27.com)

Возникает вопрос: что произойдет, если мы удалим Resid27.com из памяти?

Процессор, выполнив команду `int 21h` любой программы, передаст управление на адрес 3456:0000h, т. е. программе progA.com. Последняя, отработав, передаст управление на адрес 2345:0000h. И что же будет находиться по этому адресу, если мы удалили Resid27.com? Обычный "мусор". Компьютер, скорее всего, просто зависнет.

Возникает еще один вопрос: а можно ли удалить Resid27.com, если после него загрузился еще один резидент, который также перехватывает прерывание 21h? К сожалению, в таком случае наш резидент уже удалить не получится. Чтобы это сделать, нам нужно вначале удалить progA.com, а затем уже удалять Resid27.com.

То есть удалять резиденты следует в соответствии с их загрузкой, в порядке работы стека. Мы абсолютно никак не сможем "вырезать" резидентную программу из середины.

В заключение остается добавить, что удалить чужой резидент возможно только в том случае, если мы знаем, по какому адресу в цепочке прерывания 21h он дальше передает управление.

Например, Volcov Commander удаляет резиденты, начиная с загруженных в последнюю очередь. Причем могут быть удалены только те резидентные программы, которые были загружены после загрузки самого Volcov Commander. Как уже упоминалось, это происходит согласно принципу работы стека. Резидентные программы загружаются в память в следующем порядке:

```
push ax (первый загруженный резидент)  
push bx (второй резидент)  
push cx (третий резидент)
```

Выгрузка резидентных программ из памяти происходит в таком порядке:

```
pop cx (третий резидент)  
pop bx (второй резидент)  
pop ax (первый резидент)
```

Удаление резидентных программ в Volcov Commander возможно в случае, когда он сохраняет в своих внутренних переменных адреса последующих обработчиков того или иного прерывания в цепочке. Иными словами, перед загрузкой любой программы Volcov Commander сохраняет в специально отведенной области памяти текущую таблицу векторов прерываний. Если после выполнения запущенной программы таблица векторов изменилась, то это значит, что загруженная программа была резидентной и перехватила одно или несколько прерываний. Volcov Commander также запоминает измененную таблицу векторов и в случае необходимости удаления резидента восстанавливает предыдущую сохраненную таблицу, освобождает отведенную загруженному резиденту память, что приводит к корректному удалению резидентной программы.

27.2. Практика

Как экспериментировать с прилагаемыми к данной главе файлами?

Загружаем test27.com, который выведет на экран 50 символов "A". Убедитесь, что это так.

1. Запустите Resid27.com без параметров в командной строке (рис. 27.1).
2. Запустите снова test27.com.
3. Запустите Resid27.com с параметром /u (т. е. resid27.com /u).
4. Запустите снова test27.com.
5. Запустите Resid27.com.
6. Запустите Resid27_.com.
7. Запустите Resid27.com с параметром /u (т. е. resid27.com /u) (рис. 27.2).

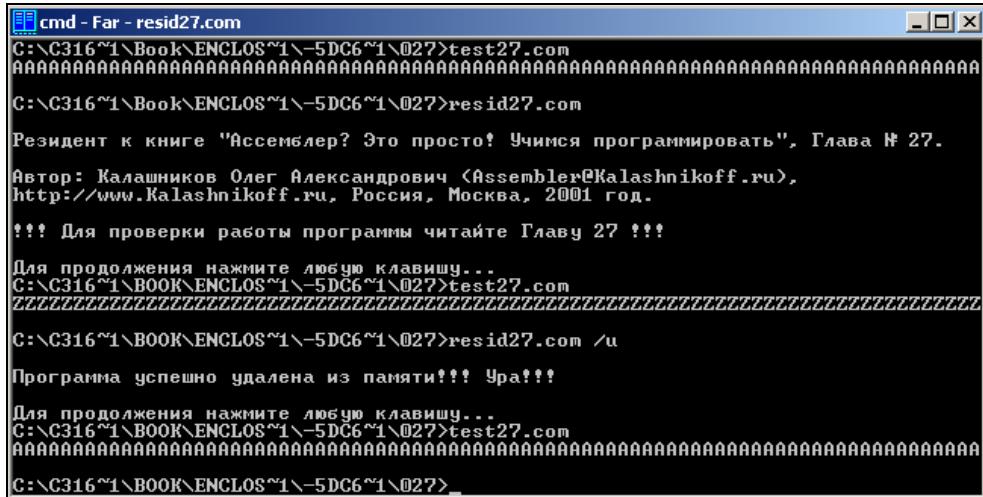


Рис. 27.1. Результат работы резидента

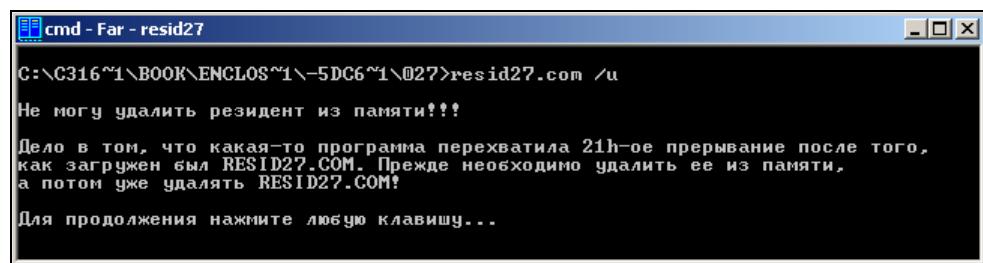


Рис. 27.2. Невозможность удаления резидента из памяти

Подробные описания, как обычно, находятся в файлах-приложениях. Увлекательного вам исследования!



Глава 28

Алгоритм считывания имен файлов в память

Итак, теперь оболочка работает без сбоев, т. е. мы можем читать корневой каталог, заходить в подкаталоги и возвращаться обратно в корневой. Работают также клавиши <PageUp>, <PageDown>, <Home>, <End>, <Insert> (отметка файлов), <Ctrl>+<F3>.

28.1. Новый алгоритм считывания файлов в память

Алгоритм считывания файлов и вывод их на экран полностью изменился. В данном разделе мы рассмотрим его подробно. Считывание каталога происходит следующим образом (листинг 28.1).

Листинг 28.1. Подготовка сегментных регистров

```
...
(01) mov fs,Seg_offset      ;fs – сегмент смещений файлов
(02) xor di,di
(03) mov Current_offset,di
;Получаем смещение файла в банке данных смещений
(04) mov fs:[di],di
(05) mov bp,2                ;Следующий файл будет помещаться по адресу 2
...
...
```

Обратите внимание на новый сегментный регистр `fs` (01), который начал использоваться в процессоре 286. Ничего особенного в нем нет. Поэтому считайте его дополнительным сегментным регистром типа `es`.

Чтобы разместить файлы в памяти, отведем память под смещения для файлов (листинг 28.2 и рис. 28.1).

Листинг 28.2. Отводим память под смещения для файлов

```
...
;Здесь будут смещения для файлов. Отводим 4000 байт для ссылок.
(01) mov ah,48h
```

```
(02) mov bx,250           ;250 * 16 = 4000 байт = 2000 ссылок на файлы
(03) int 21h
(04) mov Seg_offset,ax    ;Сохраним сегмент
```

;Отводим память для хранения файлов:

;Здесь будут сами файлы. Для имен файлов отводим 64 Кбайт

```
(05) mov ah,48h
(06) mov bx,4096          ;4096 * 16 = 65536 байт
(07) int 21h
(08) mov Seg_files,ax    ;Сохраним сегмент
```

...

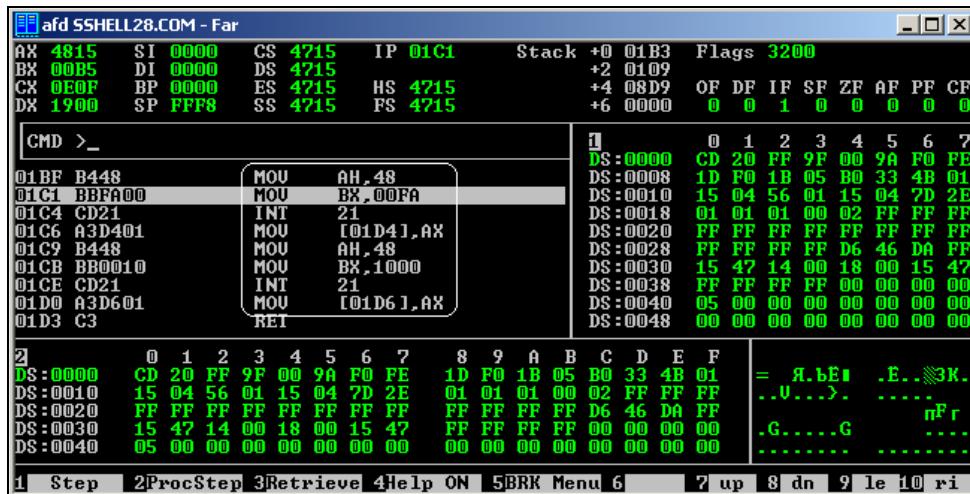


Рис. 28.1. Выделяем память для указателей и названий файлов

Предположим, в каталоге, файлы которого наша оболочка считывает в память, есть два файла: main.asm и other.asm. Таким образом, файлы в памяти будут размещены так, как показано в табл. 28.1.

В табл. 28.1 вы видите два сегмента, которые мы отвели: Seg_offset (04) и Seg_files (08). В первом сегменте будут сохраняться смещения на файлы, которые находятся во втором сегменте. Сигналом окончания файлов служит число 0FFFFh вместо очередного смещения в первом сегменте.

Таблица 28.1. Размещение файлов в памяти оболочки

Сегмент	Файл		
	main.asm	other.asm	Конец файлов
Seg_offset	0000h	0009h	0FFFFh
Seg_files	main.asm+ASCII	other.asm+ASCII	

28.2. Процедура вывода имен файлов на экран

Получаем число (2 байта), расположеннное в сегменте `Seg_offset` по смещению `0000h`. В первом случае это будет `0000h`. Затем читаем и выводим файл, который расположен по полученному смещению из сегмента `Seg_offset` (рис. 28.2).

Увеличиваем указатель смещения в первом сегменте на 2 (т. е. перейдем на следующий файл). Получаем число `0009h`. По этому смещению находится в памяти следующий файл (`other.asm`). Его и выводим.

Увеличиваем указатель на 2. Получаем очередное смещение. Это `0FFFh`? Значит, предыдущий файл был последним. Все! Вывод закончен (рис. 28.3).

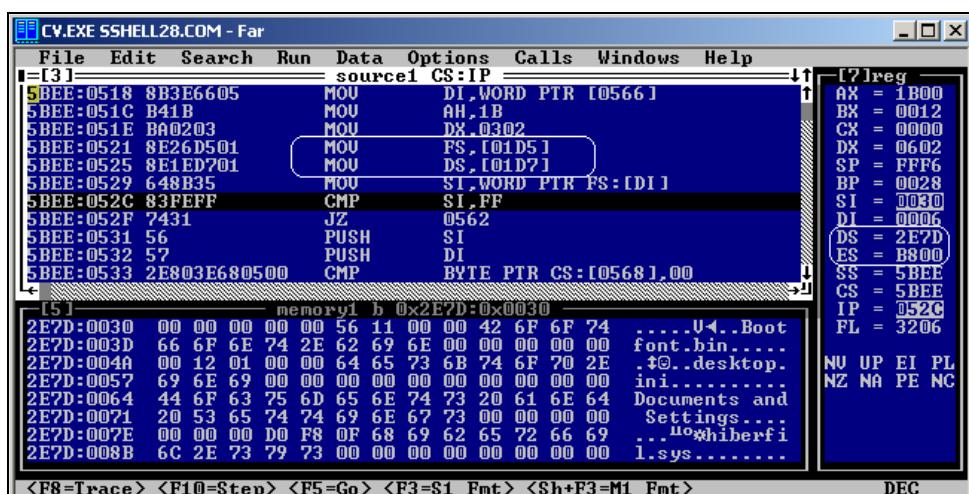


Рис. 28.2. Подготовка к выводу файлов на экран

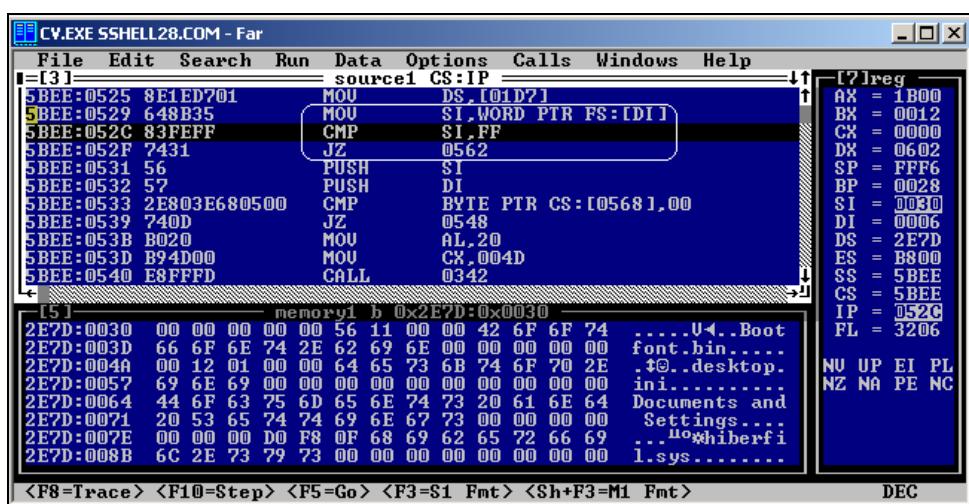


Рис. 28.3. Вывод файлов закончен

Все просто! Более того, обратите внимание, что и перемещаться по каталогу в памяти достаточно удобно. Хотим получить имя пятого файла в цепочке? Производим нехитрые вычисления: $5 \times 2 = 10$. Умножаем на 2 потому, что смещение занимает 2 байта. Таким образом, получаем смещение пятого файла в сегменте Seg_files. Остается только вывести его на экран.

Реально наша оболочка заносит не только имя файла + ASCII 0, но и его статус (текущий, отмеченный) и размер. Сюда же можно добавить еще атрибуты и дату создания/изменения файла.

Вот, собственно, и весь алгоритм. Теоретически все просто, но на практике могут возникнуть некоторые трудности. Безусловно, мы понимаем, что чем больше программа, тем сложнее в ней разобраться. Помните: за терпение, упорство и желание изучить ассемблер вам воздастся полностью.

28.3. Новые переменные в оболочке

Рассмотрим еще несколько новых переменных, которые задействованы в оболочке:

```
Current_file
Start_file
Out_fstat
```

Первая переменная Current_file хранит в себе смещение текущего файла, т. е. файла, на котором установлен курсор. Так удобней и быстрее будет искать текущий файл.

Вторая переменная Start_file содержит смещение файла, с которого следует выводить файлы на экран. Наша оболочка может отображать на экране 21 файл. Если же в каталоге больше файлов, то, дойдя до нижнего или верхнего файла на экране, необходимо "прокрутить" список файлов. Верхний/нижний файл на экране убирается, а на его месте появляется следующий/предыдущий. Переменная Start_file как раз и будет хранить смещение файла, с которого нужно выводить на экран все последующие файлы.

Обратите внимание, что под словом "прокрутка" понимается не так называемый скроллинг, а запись файлов поверх себя. Дело в том, что скроллинг осуществляется с помощью прерывания 10h, что существенно тормозит работу. Мы будем пользоваться нашим способом, который не требует вызова прерывания. Назовем этот способ "псевдоскроллинг".

Третья переменная Out_fstat хранит состояние вывода одного файла, а именно данные о том, чистить ли строку перед выводом или нет. Зачем чистить строку перед выводом файла? Предположим, что список файлов прокручивается на экране. Названия файлов будут размещаться на экране так, как показано в табл. 28.2.

Таблица 28.2. Размещение файлов на экране

M	A	I	N	.	A	S	M	
O	T	H	E	R	.	A	S	M

Таким образом, после прокрутки списка файлов вниз, название файла main.asm "ляжет" поверх названия other.asm, а на экране мы увидим буквально следующее:

```
main.asm
```

От названия файла, которое располагалось на этом месте до прокрутки, останется символ "m". Чтобы избежать этого, мы перед выводом очередного файла чистим (забиваем пробелами) строку на экране.

Переменная `Out_fstat` сообщает процедуре о том, следует ли чистить строку перед тем, как вывести очередной файл или нет. Почему в одних случаях мы будем чистить строку на экране, а в других — нет?

Если чистить строку каждый раз, то создастся эффект мерцания файлов на экране, что не очень удобно для просмотра, особенно на медленных компьютерах. Иными словами, весь экран будет забиваться пробелами, а затем все текущие файлы будут выводиться заново. Безусловно, это происходит практически мгновенно, однако в некоторых случаях "мерцание" экрана может быть заметно.

Например, пользователь нажимает клавишу `<Insert>` на названии верхнего файла, что приводит к отметке файла, т. е. на экране название будет выделено светло-желтым цветом (рис. 28.4).

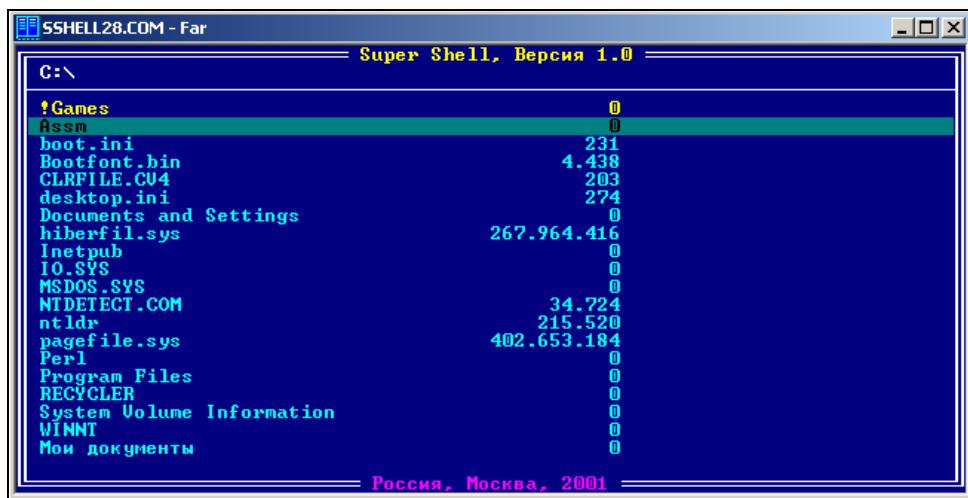


Рис. 28.4. Выделение файла

Делается это следующим образом. У нас существует процедура вывода файлов (`Out_files`, `files.asm`), которая выводит список файлов, начиная с позиции `Start_file` и до конца (либо всего 21 файл). Как уже отмечалось, в буфер заносится не только имя найденного файла, но и его статус, а все файлы изначально имеют нулевой статус, т. е. не являются ни отмеченными, ни текущими. После перечитывания каталога мы автоматически делаем первый файл текущим (статус 1). *Текущий файл* — тот, на названии которого стоит курсор.

Допустим, пользователь нажал клавишу `<Insert>`. Статус текущего файла меняется на 2, файл становится отмеченным. Первый байт в буфере для файла (его ста-

тус) отмечается как 2. Следующему файлу присваиваем статус 1 (текущий) и вызываем процедуру `Out_files`. Процедура `Out_files` проверяет статус первого файла. Если это 1, то файл выводится с обычными атрибутами (как будто на него указывает курсор), если 2 — как отмеченный (светло-желтым цветом), если статус иной, то файл выводится и как отмеченный, и как текущий. И так с каждым файлом.

Таким образом, присвоив нужный статус тому или иному файлу, мы вызываем процедуру `Out_files`, которая обновит файлы на экране.

Если не было "прокрутки" файлов, то мы вызываем `Out_files`, при этом заносим в переменную `Out_fstat` число 0, указывая процедуре `Out_files` на то, что чистить строки не надо, т. к. все символы будут ложиться поверх уже существующих, только в некоторых случаях с разными атрибутами: отмеченный, обычный и т. п.

Если же произошла прокрутка файлов, то вызываем процедуру `Out_files`, при этом заносим в `Out_fstat` число 1, указывая процедуре чистить строку перед выводом каждого файла.

28.4. Обработка клавиш <PageUp> и <PageDown>

При нажатии одной из этих клавиш программа имитирует нажатие клавиши $\leftarrow\uparrow$ или $\leftarrow\downarrow$ столько раз, сколько файлов отображено на экране.

28.5. Обработка клавиш <Home> и <End>

Процедуры `Up_pressed` и `Down_pressed` (нажатие клавиш $\leftarrow\uparrow$ и $\leftarrow\downarrow$) устанавливают флаг переноса, если достигнуты конец или начало списка файлов. Работа же самих процедур показана в листинге 28.3.

Листинг 28.3. Процедуры обработки клавиш <Home> и <End>

```

...
; === Клавиша Home ASCII 49h ===
(01) K_Home proc
; Выводить будем до тех пор, пока процедура Up_pressed не вернет
; установленный флаг переноса (Carry Flag)
(02) Next_khome:
(03)     call Up_pressed
(04)     jnc Next_khome
(05)     ret
(06) K_Home endp

; === Клавиша End ASCII 49h ===
(07) K_end proc
(08) Next_kend:
```

```
(09)    call Down_pressed  
(10)   jnc Next_kend  
(11)   ret  
(12) K_End endp  
...
```

Таким образом, нажатие на клавишу <Home> или <End> — не что иное, как имитация нажатия клавиш < \uparrow > или < \downarrow > до тех пор, пока процедуры не достигнут начала или конца списка файлов. При достижении первого или последнего файла устанавливается флаг переноса, который сигнализирует о том, что больше файлов нет. Именно поэтому мы и вызываем процедуру нажатия клавиш < \uparrow > или < \downarrow > до тех пор, пока не будет установлен флаг переноса.



Глава 29

Загрузка и запуск программ

Данная глава полностью посвящена функции 4Bh прерывания 21h. Эта функция выполняет загрузку и запуск программ. Запустить программу на ассемблере — задача не такая простая, как может показаться с первого взгляда, и нередко по данной теме возникает много вопросов. Поэтому рассмотрим запуск программ подробно.

Обратите внимание, что в приложении находятся три небольших файла: 4bh-1.asm, 4bh-2.asm и test.asm. Но не спешите их сразу ассемблировать и запускать! Желательно прочитать эту главу до конца.

29.1. Подготовка к запуску программы и ее загрузка

Прежде чем запустить программу, необходимо тщательно подготовиться, а именно:

1. Выделить память для загружаемой программы.
2. "Ужать" стек, если это СОМ-файл.
3. Сохранить необходимые регистры.
4. Подготовить EPB.
5. Подготовить строку с именем файла, который будем загружать.
6. Подготовить командную строку.
7. Сохранить сегментные регистры, если необходимо.
8. Сохранить стековые регистры ss и sp в переменных.
9. Запустить программу, вызвав прерывание 21h.

С момента вызова прерывания 21h наша программа (родительская) находится в памяти до тех пор, пока запущенная (порожденная) программа не отработает. Как только последняя завершила работу, управление получает наша (родительская) программа. Дальше необходимо сделать следующее:

1. Восстановить стековые регистры ss и sp.
2. Восстановить сохраненные в стеке регистры (сегментные и пр., если их сохранили).
3. Произвести другие необходимые действия.

Вот, собственно, и все.

29.1.1. Выделяем память для загружаемой программы

Теперь рассмотрим, как это все происходит на практике. В листинге 29.1 и на рис. 29.1 показан наш первый шаг.

Листинг 29.1. Урезаем основную память

```
...
(1)    mov bx,offset Finish
(2)    shr bx,4
(3)    inc bx
(4)    mov ah,4Ah
(5)    int 21h
...
...
```

Помните, что традиционно метка `Finish` у нас является последней в коде.

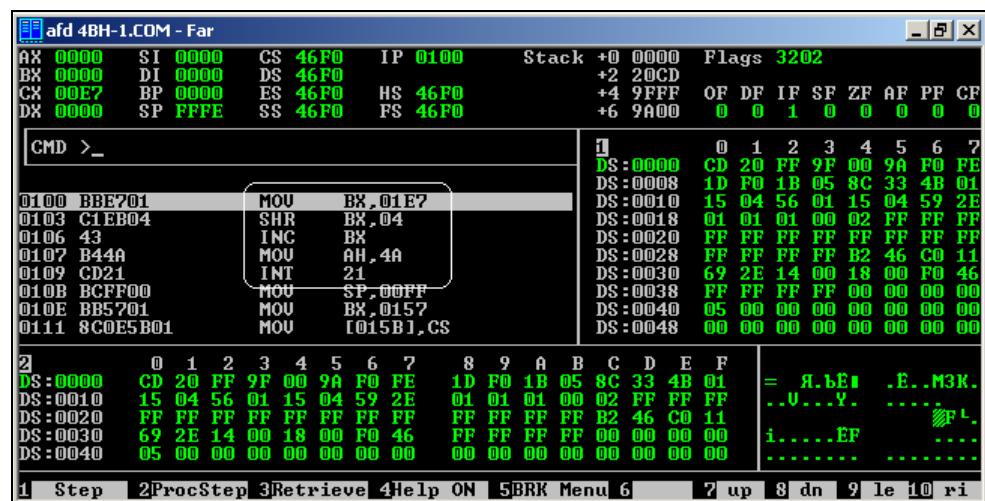


Рис. 29.1. Урезаем память

Зачем необходимо урезать память перед загрузкой?

Как мы уже отмечали в предыдущих главах, после загрузки программы вся память отводится только этой программе. Следовательно, если мы попытаемся загрузить другую программу, то получим ошибку из-за недостатка свободной памяти.

В первой строке загружаем в регистр `bx` адрес (смещение) последнего байта нашей программы.

Зачем во второй строке мы сдвигаем на 4 бита вправо это смещение?

Дело в том, что функция 4Ah (изменение размера отведенной памяти) требует указания в регистре bx блока памяти размером 16 байт. Загрузив в bx единицу, мы отведем не 1 байт, а сразу 16, т. е. один параграф. Загружая в bx адрес последнего байта нашей программы, мы получаем не количество параграфов, а количество байтов. Таким образом, не разделив bx на 16, мы отведем памяти в 16 раз больше. Делим же мы так: просто сдвинем число в bx на 4 бита вправо, что эквивалентно делению bx на 16.

А для чего увеличиваем bx на единицу (3)?

Лучше отведем на 16 байт больше, чем на один меньше.

Теперь у нас bx содержит количество параграфов (блоков памяти по 16 байт), которые заняла наша программа в памяти.

29.1.2. Переносим стек в область PSP

В листинге 29.2 и на рис. 29.2 показано выполнение второго шага.

Листинг 29.2. Переносим стек в область PSP (0FFh)

```
...
(1)    mov sp,0FFh
...
```

Рис. 29.2. Переносим стек в область PSP

Вспомните, что при загрузке COM-программы в память смещение стека находится в самом низу сегмента, в который загрузилась программа, т. е. 0FFEh. Но мы

ведь ужимаем память до метки `Finish!` Получается, что программа будет загружаться в первый байт после этой метки. В таком случае мы затираем стек.

Чтобы этого не произошло, перенесем стек в область PSP. Область PSP не используется в нашем примере, а это лишние 256 байт, куда можно переместить указатель стека.

29.1.3. Подготовка EPB

EPB (EXEC Parameter Block) — блок параметров функции загрузки файла. Информация, которая должна содержаться в этом блоке, представлена в табл. 29.1.

Таблица 29.1. Структура EPB

Смещение	Длина (байт)	Значение
00h	2	Сегмент окружения DOS для порождаемого процесса
02h	4	Смещение и сегмент адреса командной строки
06h	4	Первый адрес блока FCB
0Ah	4	Второй адрес блока FCB
0Eh	2	Длина EPB

Подготовка EPB проводится так, как показано в листинге 29.3.

Листинг 29.3. Готовим EPB

```
...
(1)    mov bx,offset EPB
(2)    mov C_F,cs
...
```

В ассемблерном коде EPB мы обозначим следующим образом (листинг 29.4).

Листинг 29.4. Структура EPB в ассемблере

```
...
; === Exec Parameter Block (EPB) для функции 4Bh ===
EPB:
Env dw 0          ;Сегмент среды (окружения DOS) для загружаемой программы
C_O dw offset Comm_line ;Смещение командной строки + ...
C_F dw 0          ;... + сегмент командной строки
                  ;FCB 1
                  ;FCB 2
Len dw $-EPB      ;Длина EPB
...
```

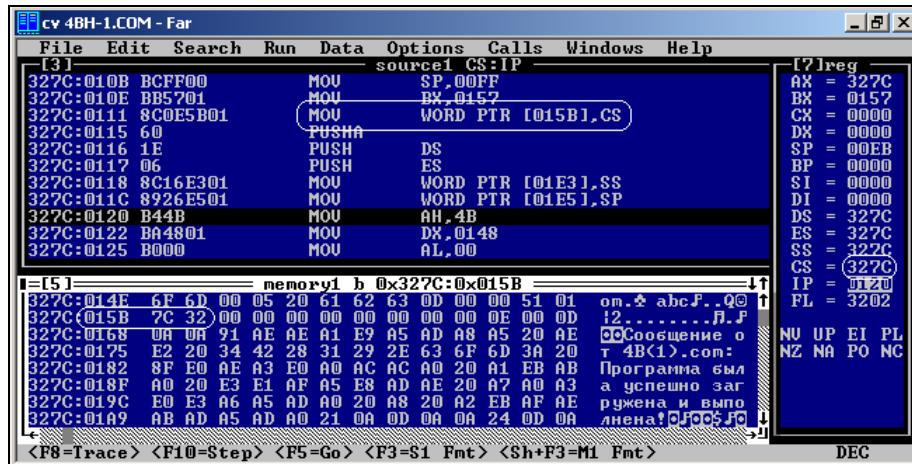


Рис. 29.3. Готовим EPB

Сравните код в отладчике из рис. 29.3 с тем, который приведен в табл. 29.1.

Еще несколько слов о системных переменных (сегменте окружения DOS)

Мы уже знаем, что такое системные переменные (окружение DOS). Рассматривали подробно данную тему в предыдущих главах. Но повторим еще раз, т. к. в данный момент это очень важно.

Системные переменные хранят в себе настройки (текущие параметры), которые устанавливаются при загрузке ОС (раньше они передавались в файле autoexec.bat). Например:

```
C:\>set
CLIENTNAME=Console
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=MOSHOSTER
ComSpec=C:\WINDOWS\system32\cmd.exe
FARHOME=C:\Program Files\Far Manager
HOMEDRIVE=C:
LOGONSERVER=\MOSHOSTER
NUMBER_OF_PROCESSORS=2
OS=Windows_NT
ProgramFiles=C:\Program Files
PROMPT=$P$G
SESSIONNAME=Console
windir=C:\WINDOWS
```

Посмотреть текущие параметры можно, выполнив внутреннюю команду SET в консоли.

Итак, если в поле "Сегмент окружения DOS..." находится 0, то порожденной программе будут переданы те параметры, которые получила наша программа, т. е. значения передаются по умолчанию.

Если же для порождаемого процесса (для загружаемой программы) мы хотим передать собственные параметры, то нам нужно выделить блок памяти в отдельном сегменте, внести туда необходимые значения и в ЕРВ по смещению 0 занести сегмент отведенного блока (как это делать — рассмотрим далее). Но, как правило, в это поле практически все программы заносят ноль.

Для чего нужно создавать свое окружение DOS?

В разных целях... Например, некоторая антивирусная программа, запуская чужую программу, может не передать текущее окружение, поскольку некоторые вирусы размножаются по пути (PATH), который они сами находят в памяти (вспомним из прошлых глав, что найти сегмент окружения DOS можно по смещению 2Ch сегмента, в который загрузилась программа; смещение всегда будет равно нулю).

В другом случае программа может дополнить окружение DOS своими параметрами (например, добавить путь к своим оверлеям в переменную PATH).

Вроде все понятно. Однако помните, что если вы решили создать свое окружение, то нужно придерживаться определенных правил. Как именно устроены системные переменные — см. в главе 23. Там все подробнейшим образом описано. По такому образцу вам необходимо будет создавать свое окружение для порождаемого процесса.

Обратите также внимание, что нам необходимо заносить только 1 байт (смещение), а не 2 (сегмент + смещение). Как уже отмечалось ранее, смещение окружения DOS всегда будет равно нулю.

Сегмент и смещение командной строки

Здесь все просто. Создаем некий массив и заносим в это поле вначале смещение, а затем сегмент этого массива.

Например, так (командная строка):

```
Comm_line db 5, 'abc', 0Dh
```

А теперь обратите внимание, что по умолчанию в ЕРВ заносится смещение командной строки (в процессе ассемблирования программы), а сегмент придется заносить самим (см. разд. 29.1.4):

```
...
;Заносим сегмент командной строки в изначально неопределенную переменную C_F
mov C_F,cs
...
;Область переменных
;Сегмент массива командной строки (по умолчанию равен нулю)
C_F dw 0
;Смещение массива командной строки (заносится автоматически в процессе
```

;ассемблирования программы)

C_O dw offset Comm_line

...

Теперь еще раз остановимся на структуре командной строки, адрес которой у нас будет загружен в переменную Comm_line:

- первый байт содержит длину строки (включая символ 0Dh);
- остальные байты — саму строку;
- последний байт — обязательно содержит 0Dh.

Первый и второй адрес блоков FCB

Блоки FCB, как правило, уже давно не используются, поэтому не будем их вообще касаться в данной книге. В эти переменные просто занесем нули, тем самым сообщив ОС, что FCB задействовано не будет.

29.1.4. Сохранение регистров

Сохранение регистров процессора показано в листинге 29.5 и на рис. 29.4.

Листинг 29.5. Сохраняем регистры

...

```
(1)    pusha
(2)    push ds
(3)    push es
(4)    mov ss_Seg,ss
(5)    mov sp_Seg,sp
...

```

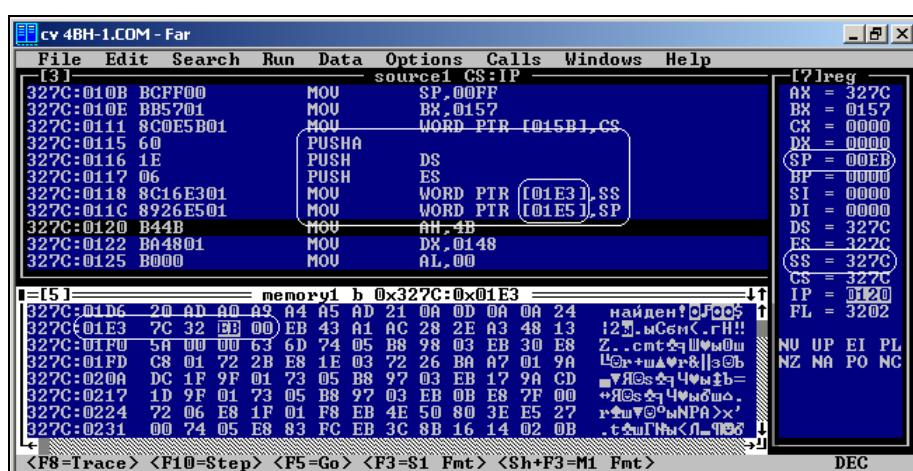


Рис. 29.4. Сохранение регистров процессора

Заносить в стек или сохранять иным образом такие регистры, как ax, bx, cx, dx, si, di и т. п., не обязательно, если вы, конечно, не оставляете в них информацию перед запуском программы, которая понадобится после завершения порожденного процесса. Однако в нашем примере мы это делаем для наглядности и полноты (строка (1)).

Сегментные регистры ds и es сохранять не обязательно, если они перед запуском новой программы были равны cs. В этом случае мы можем после завершения порожденного процесса выполнить следующие действия (листинг 29.6).

Листинг 29.6. Восстановление сегментных регистров после завершения порожденного процесса

```
...
push cs
push cs
pop ds
pop es
...
...
```

Тем не менее, для полноты картины мы сохраним эти регистры (строки (2), (3) листинга 29.5).

Затем идут регистры указателя стека ss:sp. Сохранить в стеке мы их не можем, и это понятно почему. Для такой цели выделяем две переменные, в которые их и занесем: ss_Seg, sp_Seg. Иного способа не существует...

29.1.5. Запуск программы

Теперь можно перейти к запуску программы (листинг 29.7, рис. 29.5).

Листинг 29.7. Запуск программы

```
...
(1)    mov ah, 4Bh
(2)    mov dx, offset File
(3)    mov al, 0
(4)    int 21h
...
...
```

Итак, мы подошли вплотную к вызову функции 4Bh (1). В регистр dx заносим смещение строки с именем файла (2), который следует загрузить и выполнить.

Подфункция 0 функции 4Bh прерывания 21h — загрузка и запуск программы (3). Это значит, что программа будет загружена в память и запущена.

Существуют также подфункция 1, которая только загружает программу, но не передает ей управление, и подфункция 3, загружающая оверлей. Подфункции 1 и 3

используются редко, и мы их пока рассматривать не будем. Наша цель — заставить программу загрузиться и выполниться.

После выполнения строки (4) начнется загрузка программы. Следующая за ней строка получит управление только тогда, когда завершит работу загруженная программа.

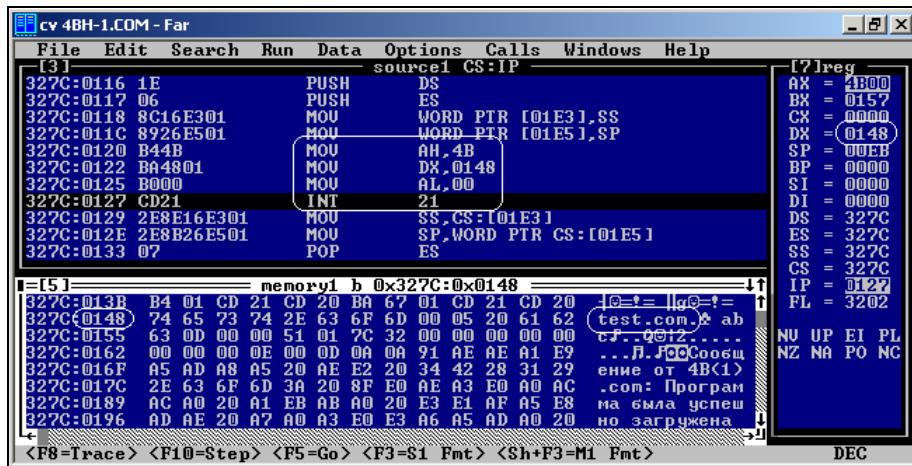


Рис. 29.5. Запуск программы test.com

29.2. "Восстановительные работы"

После того как запущенная программа отработала, необходимо произвести "восстановительные работы". Первое, что мы должны сделать, — восстановить регистры стека. Запущенный процесс вполне мог поменять эти регистры, а без стека мы не сможем работать дальше (листинг 29.8).

Листинг 29.8. Восстановление регистров стека

```
...
mov ss,cs:ss_Seg
mov sp,cs:sp_Seg
...
```

Обратите внимание: т. к. мы не уверены, что отработанная программа не поменяла регистры `ds`, то обращаемся к переменным `ss_Seg` и `sp_Seg`, используя регистр `cs`, который, как мы уже знаем, всегда равен тому сегменту, в котором выполняется текущая команда.

Мы сохраняли в стеке не только сегментные, но и все остальные регистры, которые помещает в стек инструкция `pusha`. Следовательно, нам нужно восстановить их, дабы не нарушить работу стека (листинг 29.9, рис. 29.6).

Листинг 29.9. Восстановление иных сохраненных регистров

```
...
pop es
pop ds
popa
...
```

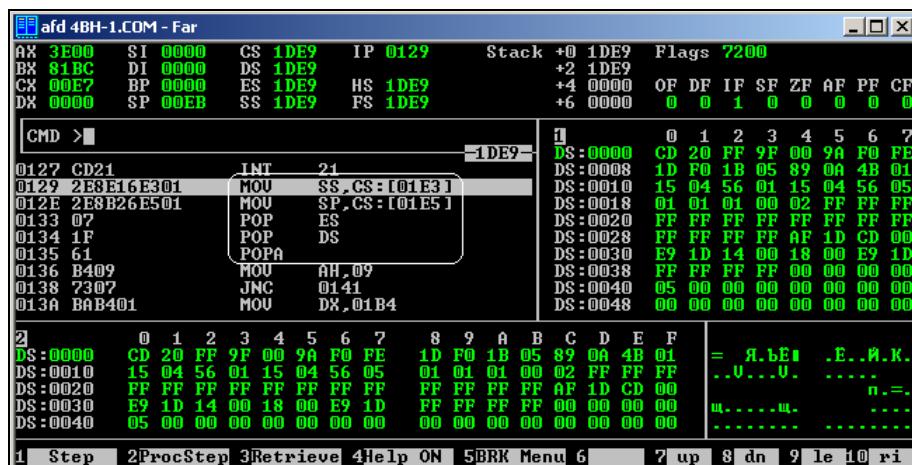


Рис. 29.6. Восстановление регистров

Вот и все! Можно, конечно, еще и память "вернуть на место", но это не столь принципиально в данном случае.

Дополнительную информацию вы, как всегда, найдете в файлах-приложениях. Внимательно ознакомьтесь с тремя прилагаемыми файлами. Это позволит вам лучше понять принцип запуска программ на ассемблере!



Глава 30

Работа с расширенной памятью

30.1. Расширенная (XMS) память. Общие принципы

В этой главе рассмотрим работу с XMS-памятью. А именно:

- определим объем доступной XMS-памяти (XMSmem.asm);
- скопируем файл autoexec.bat в XMS-память, а затем выведем его содержимое на экран из расширенной памяти (XMSblock.asm);
- научимся копировать файлы, используя XMS-память (XMScopy.asm).

В приложении вы найдете 3 файла, каждый из которых выполняет свою отдельную функцию. Как обычно, будем изучать от простого к сложному (XMSmem, XMSblock, XMScopy).

Прежде чем приступать к рассмотрению материала, обращаем ваше внимание на следующее:

- для получения доступа к XMS-памяти необходимо загрузить драйвер himem.sys либо подобный ему, который откроет линию A20, а также загрузит процедуры для работы с расширенной памятью, входящие в состав ядра операционной системы;
- для использования расширенной (XMS) памяти в DOS необходимо прибегать к помощи прерываний и процедур. Иного способа не существует. Пересылка данных при помощи команд `movs` не годится. Также невозможно загружать данные непосредственно в расширенную память;
- работа с XMS-памятью — задача не такая простая, как может показаться на первый взгляд. Более того, отладчики типа AFD, CodeView, Turbo Debugger и т. п. не позволяют просматривать содержимое расширенной памяти. Это означает, что при возникновении каких-либо ошибок в программе программисту придется самому исследовать код и искать ошибку, что, безусловно, усложняет процесс отладки.

Однако подобных трудностей не возникает у программ, которые написаны специально для Windows. В этой операционной системе используется совсем другой принцип обращения к расширенной памяти, подобный принципу обращения к основной памяти в MS-DOS.

30.2. Программа XMSmem.asm.

Получение объема XMS-памяти

30.2.1. Подготовка к использованию расширенной памяти и вывод объема XMS-памяти

Первое, что мы сделаем, — определим объем доступной XMS-памяти и выведем количество килобайт на экран при помощи известной нам уже процедуры с использованием сопроцессора (ее мы уже рассматривали в предыдущих главах).

Как уже отмечалось ранее, чтобы использовать расширенную память, необходимо, чтобы в памяти был загружен специальный драйвер, который бы открыл доступ к ней. В DOS/Windows такой файл называется `himem.sys`. Если загрузить "чистую" DOS без этого драйвера, то программы смогут обращаться только к основной памяти (640 Кбайт). Следовательно, все файлы-приложения будут работать только в том случае, если загружен указанный выше файл (или его аналог).

Первое, что нужно сделать, — проверить наличие драйвера `himem.sys` в памяти. Это позволяет сделать функция `4300h` прерывания `2Fh` (табл. 30.1).

Таблица 30.1. Функция `4300h` прерывания `2Fh`: проверка на присутствие в памяти драйвера `himem.sys`

Вход	Выход
<code>ax = 4300h</code>	<code>al = 80h</code> — драйвер загружен

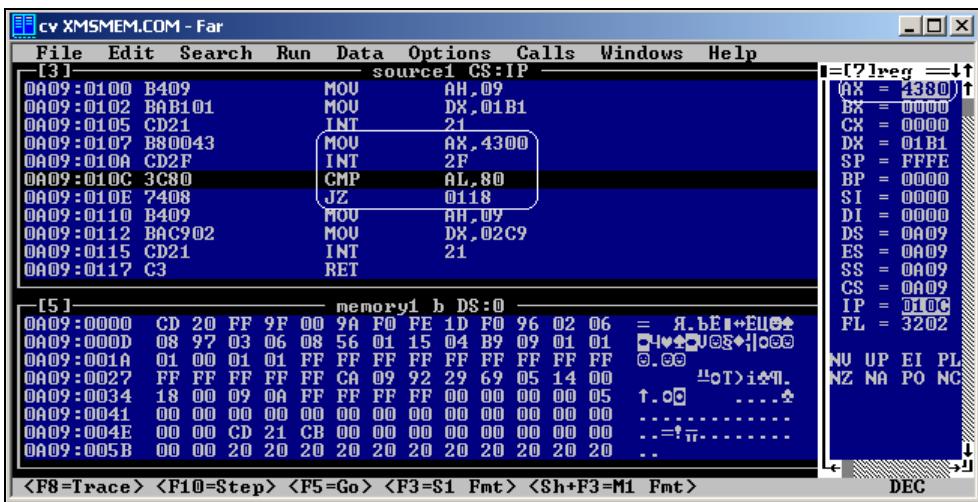


Рис. 30.1. Проверка драйвера `himem.sys`

Как провести проверку, показано в листинге 30.1 (во избежание недоразумений проверка будет осуществляться во всех трех файлах-приложениях).

Листинг 30.1. Проверяем наличие файла himem.sys

```

...
;Проверим на наличие himem.sys в памяти...
mov ax,4300h
int 2Fh

cmp al,80h
je Himem_loaded ;Если al = 80h, то himem.sys загружен.

;Иначе выводим сообщение о том, что himem в памяти не обнаружен...
...

```

Если драйвер присутствует, то можно попробовать отвести блок расширенной памяти. Для этого вызываем собственную процедуру `Prepare_XMS`, которая подготовится к работе с расширенной памятью, а также отведет блок памяти.

Все процедуры по управлению XMS-памятью вызываются *не* с помощью прерываний (как функции DOS — `int 21h`), а с использованием команды " дальний call" и указанием после нее сегмента и смещения самой процедуры. Получить точку входа (адрес) процедур по управлению XMS-памятью позволяет функция `4310h` прерывания `2Fh` (табл. 30.2).

Подобный принцип вызова процедур очень похож на вызов WinAPI.

Таблица 30.2. Функция 4310h прерывания 2Fh: получить точку входа процедур управления XMS-памятью

Вход	Выход
<code>ax = 4310h</code>	<code>es = сегмент, bx = смещение</code>

Рассмотрим это на примере.

После того как получили точку входа, все остальные обращения к процедурам работы с XMS-памятью будут осуществляться следующим образом (естественно, нужно предварительно загрузить необходимые данные в регистры или подготовить массивы данных):

```
call dword ptr XMS_Addr
```

В листинге 30.2 показано, как получить адрес обработчика XMS-функций и количество килобайт расширенной памяти.

Листинг 30.2. Получаем адрес обработчика и объем свободной XMS-памяти

```

...
mov ax,4310h
int 2Fh
mov word ptr XMS_Addr,bx
mov word ptr XMS_Addr+2,es ;Сохраним обработчик XMS-функций

```

```

mov ah,88h ;Получить количество килобайт XMS-памяти
call dword ptr XMS_Addr
mov dword ptr Number_dec,edx ;Сохраним полученное число
...

```

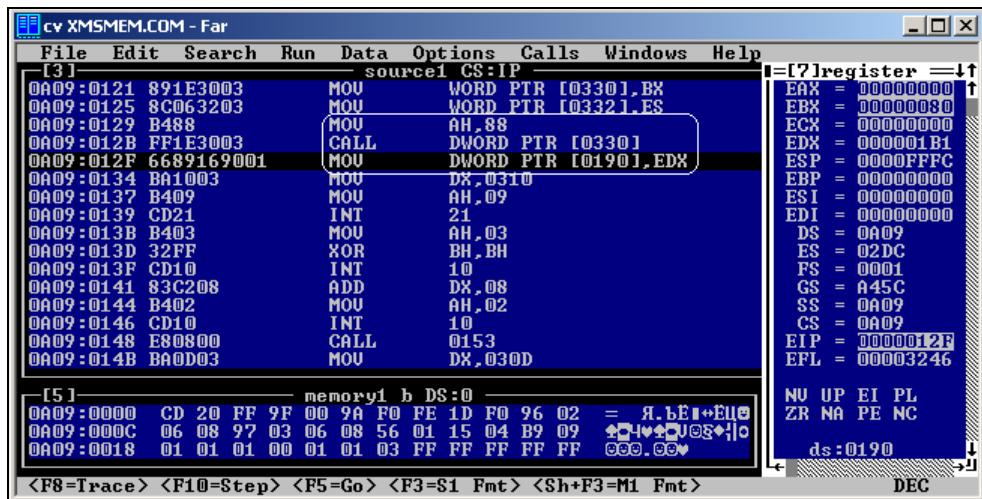


Рис. 30.2. Получить объем XMS-памяти

Как видно из примера, функция возвращает количество килобайт (не байт!) в 32-разрядный регистр edx.

Обращаем ваше внимание, что данная функция может работать некорректно в Windows NT/2000/XP. В этом случае просто загрузите "чистую" MS-DOS 5.00 и выше и проверьте работоспособность этой функции. Более того, количество килобайт свободной XMS-памяти возвращает непосредственно операционная система, которая берет информацию не из CMOS, а ведет учет самостоятельно. Это значит, что информация, которую вернет функция, может не соответствовать действительности. DOS может указать не только реальный объем XMS-памяти, но и добавить к нему размер SWAP-файла Windows либо наоборот показать меньший размер, чем есть на самом деле, вычтя из реального объема размер памяти, отведенной под кэш. Все зависит от версии DOS/Windows.

Теперь осталось только вывести полученный результат, используя написанную и изученную нами процедуру вывода десятичного числа Out_dec.

30.3. Программа XMSblock.asm. Чтение файла в расширенную память и вывод его на экран

Рассматривать подготовку и отведение блока расширенной памяти мы уже не будем. Для данного файла эти операции проводятся аналогично вышеописанным. Единственное исключение в том, что мы еще и отводим блок расширенной памяти

при помощи функции 09h процедуры управления XMS-памятью (рис. 30.3). При этом регистр dx должен содержать размер отводимого блока памяти в килобайтах (листинг 30.3).

Листинг 30.3. Отведение блока XMS-памяти

```
...
mov ah,9
mov dx,1024           ;Отводим 1024 Кбайт XMS-памяти
call dword ptr XMS_Addr
or ax,ax              ;Ошибка?
jnz XMS_OK
...

XMS_OK:
mov XMS_id,dx         ;Сохраним id отведенного блока
...

```

В случае если произошла ошибка (запрашиваемый блок больше имеющегося в распоряжении и т. п.), то значение ax будет равно не нулю, а коду ошибки.

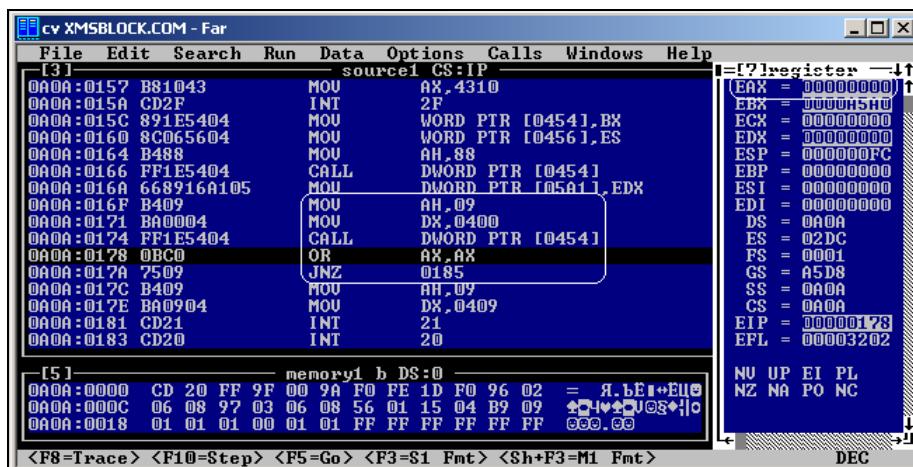


Рис. 30.3. Отводим блок расширенной памяти

Если ошибки не произошло, то dx будет содержать идентификационный номер отведенного блока. В дальнейшем мы будем обращаться к отведенному блоку, используя этот номер. Принцип обращения к отведенному блоку аналогичен принципу работы с файлами: при открытии файла функции 3Dh прерывания 21h возвращает номер открытого файла.

30.3.1. Работа с расширенной памятью

Как уже ранее отмечалось, загружать данные напрямую в XMS-память в DOS невозможно. Для этого используется основная память (640 Кбайт). Сперва данные загружаются в отведенный массив в основной памяти, затем при помощи специальной процедуры пересыпаются в расширенную. Пересылка осуществляется блоками, но не более чем по 64 Кбайт.

В программе XMSblock.asm мы вначале читаем файл C:\autoexec.bat в отведенный массив основной памяти и выводим его на экран. Затем перебрасываем прочитанные байты в расширенную память и обратно в основную, но уже по другому адресу. И снова выводим то, что перебрасывали. Идентичность выводимой информации подтверждает то, что программа не имеет ошибок: данные пересыпаются в XMS-память и обратно.

30.3.2. Структура массива при работе с XMS-памятью

Для работы с XMS-памятью (пересылка данных) заведем специальный массив (листинг 30.4). Для удобства зададим каждой переменной название. В дальнейшем адрес данного массива будет загружаться в регистры ds:si перед вызовом специальных подфункций функции управления XMS-памятью.

Листинг 30.4. Переменные для работы с расширенной памятью

```
...
;Количество байтов для пересылки
XMS_str dd 0

;Источник
XMS_src dw 0

;Смещение в блоке-источнике или адрес в основной памяти
XMS_offsrc dd 0

;Идентификатор приемника
XMS_rec dw 0

;Смещение в блоке-приемнике или адрес в основной памяти
XMS_offrec dd 0
...
```

Рассмотрим отдельно каждую переменную.

- xms_str** (два слова) — должна содержать количество байтов для пересылки из основной памяти в XMS либо наоборот.

- `xms_src` (слово) — указывает идентификатор источника. Если эта переменная равна нулю, то источником (откуда пересылаются данные) является основная память. В противном случае, переменная должна содержать идентификатор отведенного блока XMS-памяти, который выдает функция управления XMS-памятью при отведении блока (см. в листинге 30.3 `xms_id`).
- `xms_offsrc` (два слова) — содержит смещение в блоке-источнике (если копирование идет из расширенной памяти в основную) или сегмент и смещение в основной памяти (если копирование идет из основной памяти в расширенную).
- `xms_rec` (слово) — указывает идентификатор приемника. Если эта переменная равна нулю, то приемником (куда пересылаются данные) является основная память. В противном случае, переменная должна содержать идентификатор отведенного блока XMS-памяти, который выдает функция управления XMS-памятью при отведении блока (см. в листинге 30.3 `xms_id`).
- `xms_offrec` (два слова) — содержит смещение в блоке-приемнике (если копирование идет в расширенную память из основной) или сегмент и смещение в основной памяти (если копирование идет из расширенной памяти в основную).

Теоретически мы рассмотрели работу с XMS-памятью. Вам осталось лишь внимательно изучить прилагаемый файл `XMSblock.asm` и описания к нему.

30.4. Программа `XMScopy.asm`. Копирование файла с использованием расширенной памяти

Обратите внимание:

- программа корректно копирует файл, размер которого не превышает объем XMS-памяти, установленной в вашем компьютере! Если в вашей машине стоит 512 Мбайт ОЗУ, то максимальный размер файла, который можно скопировать с помощью этой программы, будет 512 Мбайт минус 640 Кбайт;
- программа копирует файл `C:\file.txt` в `C:\filenew.txt`. Прежде чем запускать программу, необходимо создать файл `file.txt` в корневом каталоге диска С: (либо продублировать любой другой, задав имя дубликату `C:\file.txt`). Файл `C:\filenew.txt` создается автоматически нашей программой. После того как программа отработает, оба файла должны быть идентичны, что свидетельствует о корректной работе программы.

Все эти недоработки вы сможете сами легко устраниТЬ. Здесь же был приведен простейший алгоритм копирования файла.

Программа работает в такой последовательности:

1. Читает файл `C:\file.txt` в память:

- читает 64 Кбайт файла в основную память;
- перебрасывает прочитанный блок в расширенную память;
- сдвигает указатель, задающий, куда помещать следующую порцию данных в расширенную память;

- если не достигнут конец файла, то читает следующий блок. И так до тех пор, пока не будет достигнут конец файла.
2. После того как файл целиком загружен в память, программа начинает записывать прочитанные данные в файл C:\filenew.txt:
- перебрасывает 64 Кбайт из расширенной памяти в основную;
 - записывает данные в предварительно созданный файл C:\filenew.txt;
 - сдвигает указатель расширенной памяти;
 - перебрасывает следующий блок, если есть еще данные...
- В принципе, это все, что нужно знать о работе с расширенной памятью в DOS. Материал в этой главе простой и понятный. Дополнительную информацию вы, как обычно, найдете в файлах-приложениях.
- Также не забывайте, что вам всегда готовы помочь наши эксперты на портале профессионалов **RFpro.ru**.



Глава 31

Обзор дополнительных возможностей оболочки

31.1. Оболочка Super Shell

Вот мы потихоньку и добрались до конца курса базового программирования на ассемблере. Простейшие функции нашей оболочки уже написаны и многократно изучены. Осталось только написать процедуры копирования файлов с использованием XMS-памяти и запуска программ. Эти процедуры мы подробно рассматривали в главе 30. Изучить их и вставить в свой код труда не составит.

Процедуры удаления файлов, перемещения, переименования и пр. не нуждаются в подробных объяснениях. Вы уже имеете неплохой опыт программирования на ассемблере и можете самостоятельно разобрать перечисленные выше функции. Все они работают очень просто, с использованием соответствующих функций прерывания 21h (удаление, переименование) или комбинирования (последовательного вызова) двух и более готовых подпрограмм (копирование + удаление = перемещение). Однако все процедуры работы с файлами в нашей оболочке связаны с одной подпрограммой — поиском первого и/или последующих отмеченных пользователем файлов.

Общий принцип работы следующий:

1. Вызываем процедуру поиска первого отмеченного пользователем файла, которая возвращает его имя.
2. Если ни один файл не отмечен, то функция отмечает текущий файл, на который указывает курсор, и возвращает имя этого файла только в случае, если ее вызов происходит первый раз. В противном случае см. шаг 6.
3. После того как получено имя первого найденного файла, вызывается соответствующая процедура (копирования, удаления, переименования и т. п.), в зависимости от того, какую операцию собирается выполнить пользователь с выбранным файлом.
4. После того как операция завершена (файл скопирован, удален и т. п.), вызывается функция снятия отметки этого файла.
5. Далее происходит поиск нового отмеченного файла (см. шаг 1).
6. Если отмеченных файлов больше нет, то функция выдает сообщение о том, что ее работа закончена, текущий каталог перечитывается заново, и программа ждет дальнейших указаний от пользователя.

Наша файловая оболочка будет обрабатывать следующие функции:

- копирование файла с использованием XMS-памяти;
- перемещение файла;
- удаление файла;
- запуск файлов СОМ, EXE и BAT через интерпретатор;
- создание каталога;
- удаление пустого каталога.

Опираясь на материал данной книги, вы без труда сможете самостоятельно реализовать следующие функции:

- копирование каталогов с вложенными подкаталогами;
- выполнение внутренних команд консоли (DIR, COPY и т. п.);
- загрузку файла с любым расширением с предварительным запуском соответствующей программы-обработчика этого расширения и передачей ему в командной строке имени загружаемого файла (например: prog.asm — ml.exe prog.asm /AT; readme.txt — notepad.exe readme.txt и т. п.);
- и многие другие по вашему усмотрению.

Наша же задача состояла в том, чтобы дать вам основы программирования, подготовить к написанию программ под Windows, научить использовать операторы ассемблера, открыть возможности операционной системы MS-DOS, показать алгоритмы и принципы их работы. Вам достаточно будет иногда заглядывать в готовый ассемблерный файл оболочки (впрочем, как и любых других программ, рассматриваемых в этой книге), чтобы "вытащить" из него необходимые подпрограммы, прочитать описания к ним, использовать в своих программах.

Как вы уже поняли, написание полноценной файловой оболочки для DOS требует дополнительных усилий. Тем более, что широко известный Norton Commander не сразу начинался с версии 5.0.

31.1.1. Вызов внешних вспомогательных программ

Что же касается процедуры редактирования файлов, то можно сказать следующее. Данная процедура довольно сложна и требует много времени для ее написания на чистом ассемблере. В настоящий момент существует множество мощных внешних многофункциональных редакторов (например, тот же Hacker's View), которые вполне можно привязать к оболочке. Необходимо предоставить пользователю возможность указать полный путь (или просто имя) к текстовому редактору, который будет вызываться при нажатии, например, комбинации клавиш <F3>/<F4>, с передачей в командной строке текущего файла. Один из алгоритмов редактирования файла см. далее.

Аналогичным образом можно поступить и с архивами. Если, например, пользователь пытается посмотреть или запустить файл с расширением ZIP, RAR и т. п., то нужно будет загрузить соответствующий архиватор и передать ему имя открываемого или запускаемого файла. Пользователю оболочки необходимо будет указать путь к архиватору и его имя, которые должны сохраняться в INI-файле (конфигурационном файле) оболочки и при ее запуске автоматически считываться в память.

Многие архиваторы позволяют упаковывать файлы, список которых находится в некотором временном файле, передаваемом им в командной строке с определенными опциями. Например, RAR. Это можно использовать и в своих целях: если пользователь собирается заархивировать отмеченные файлы, то прежде чем вызвать архиватор, оболочка должна сохранить список отмеченных пользователем файлов в каком-нибудь своем, временном файле и передать эти параметры архиватору в командной строке. Поддерживает ли тот или иной архиватор необходимые опции, можно посмотреть в его справочной информации.

31.1.2. Редактирование файла

Редактирование файла можно реализовать следующим образом. Отводятся два сегмента, скажем, 64 Кбайта и 128 Кбайт (один должен быть в два раза больше второго), в меньший из которых загружается файл, а в больший заносятся ссылки (смещения) размером в слово на очередной символ (табл. 31.1).

Таблица 31.1. Редактирование файла. Вид изнутри

Сегмент с файлом (64 Кбайт)	На экране	Сегмент смещений (128 Кбайт)
Текст	Текст	0001 0002 0003 0004 0005 FFFF

В левой колонке таблицы (**Сегмент с файлом**) показано, что загружен файл размером 5 байт ("Текст"), в средней (**На экране**) — то, как файл будет отображаться на экране, а в правой (**Сегмент смещений**) расположены 16-разрядные смещения на каждый символ в сегменте с файлом. Символ "T" следует записывать на диск и выводить на экран первым, "e" — вторым, "к" — третьим и т. д. Последнее число (FFFFh) сигнализирует о конце файла. Так как файл только что загружен и еще не редактировался, то символы располагаются в памяти последовательно, друг за другом.

Теперь допустим, что пользователь вставил между символами "к" и "с" символ "ц". Содержимое памяти у нас должно измениться (табл. 31.2).

Таблица 31.2. Добавление одного символа в редактируемый файл

Сегмент с файлом (64 Кбайт)	На экране	Сегмент смещений (128 Кбайт)
Текстц	Текцст	0001 0002 0003 0006 0004 0005 FFFF

При сохранении же файла следует производить запись побайтно, получая необходимое смещение из колонки **Сегмент смещений**: первым записываем символ "T", вторым — "е", третьим — "к", четвертым — "ц" и т. д. Следует отметить, что запись будет происходить довольно быстро, т. к. BIOS пишет данные не по байтам, а сразу блоками.

Это один из самых простых и далеко не оптимальных алгоритмов. Вместо вставки смещения можно было просто вставлять сам символ. Попробуйте сами

реализовать какой-нибудь алгоритм редактирования файлов. Это будет для вас очень хорошей практикой применения ассемблера.

Для чего мы привели именно этот алгоритм? Дело в том, что в главе 32 мы рассмотрим таблицу размещения файлов. Принцип хранения очередной порции (очередного сектора) практически полностью совпадает с данным алгоритмом.

31.2. Антивирусные возможности оболочки

31.2.1. Как защитить компьютер от заражения его резидентными вирусами

Любой резидентный вирус в определенный момент должен перехватывать некоторые прерывания для своей активизации. Как правило, перехватываются следующие прерывания: 21h — сервис DOS, 1Ch — таймер, вызываемый автоматически приблизительно 18,2 раза в секунду, 13h — работа с дисками средствами BIOS, 25h и 26h — работа с дисками средствами DOS. Если вирус не перехватит ни одно прерывание, то он так и останется в памяти, но активизироваться не будет, а следовательно, и ничего сделать не сможет. Вспомните все резидентные программы из книги, которые мы рассматривали. Каждый резидент в обязательном порядке перехватывал то или иное прерывание.

Возникает вопрос: можно ли сделать так, чтобы вирус не смог перехватить ни одно прерывание?

Один из самых надежных способов защиты от резидентных вирусов заключается в следующем. Пишется простой резидент, который сразу после загрузки считывает в свой буфер текущее состояние таблицы векторов прерываний при помощи команды `movs` или подобной ей. Как известно, таблица располагается по адресу 0000:0000h и занимает 256 (количество векторов прерываний) × 4 (сегмент + смещение вектора) = 1024 байта. Резидент также следит за тем, вызывает ли какая-нибудь программа функцию 4Bh прерывания 21h — загрузка и запуск программы. Ее может вызвать, например, Norton Commander, который запускает программу по требованию пользователя. Если так, то после того, как запущенная программа отработает, резидент копирует сохраненную таблицу из своего буфера в таблицу векторов прерываний. Таким образом, если какая-либо программа, в том числе и резидентный вирус, перехватила некоторое прерывание, то наш резидент "вырубит" его вызов. Получится, что вирус и находится в памяти, но не может активизироваться.

Этот способ имеет два недостатка:

- пока работает запущенная зараженная программа, вирус находится в памяти и уже может заражать другие файлы;
- если запускается обычный резидент, то наша программа также "отсечет" его, вследствие чего загруженный резидент также не сможет активизироваться.

Эти недостатки можно исключить, усовершенствовав наш резидент. Например, его можно включить в собственную оболочку, а в ней самой организовать возможность переключения опции сохранения/восстановления таблицы векторов преры-

ваний после каждого запуска программы. Ведь перед запуском программ часть любой файловой оболочки так или иначе остается в памяти. Если пользователь не уверен, что запускаемый файл "чистый", он может установить опцию сохранения и восстановления таблицы векторов прерываний путем простого нажатия комбинации некоторых клавиш. И наоборот.

Более того, программа-резидент или оболочка могут получить количество свободной памяти перед запуском программы, и, как только она отработала, проверить, изменился ли объем основной памяти или нет. Если уменьшился, значит, в памяти осталась какая-то программа, о чем следует немедленно сообщить пользователю.

Здесь стоит отметить, что не все резидентные вирусы уменьшают объем свободной памяти. Некоторые из них располагаются в специальных областях либо "обманывают" ОС, не уменьшая память. Но ни один резидентный вирус не сможет запретить нашей программе-резиденту восстановить предварительно сохраненную таблицу векторов прерываний. Кроме случаев, когда разработчик вируса знал о существовании нашей программы и специально разработал процедуру против нашего резидентного модуля. Но для этого ему придется сперва разобрать нашу программу, а уж затем написать обработчик, отключающий наш резидент.

Если вы пишете свою оболочку, то рекомендую написать такую подпрограмму, а затем попробовать ее действие на любой резидентной программе.

31.2.2. Как защитить компьютер от программ-разрушителей дисковой информации

Как правило, программы для работы с файлами используют прерывание MS-DOS 21h, которое обращается к прерыванию BIOS 13h. Прерывание 13h позволяет читать, писать и форматировать секторы жесткого или гибкого диска.

В ПЗУ есть мощные, но довольно сложные процедуры для выполнения указанных выше действий. В очень редких случаях программы работают с диском напрямую, т. е. используют свои процедуры чтения и записи секторов. К таким программам можно отнести DiskEditor из утилит Norton. Вирусы также очень редко имеют подобные собственные подпрограммы, т. к. процедура чтения одного сектора довольно-таки громоздка и не универсальна. Тем не менее, такие экзотические вирусы существуют. К сожалению, следить за их действиями практически невозможно.

Однако программы, которые используют прерывание BIOS 13h или DOS 25h (чтение сектора) / 26h (запись сектора) для работы с дисками, легко контролировать. Достаточно перехватить прерывание 13h и следить за вызовом соответствующей функции. Если вызывается функция записи или форматирования, то просто вернуть вызывающей это прерывание программе код ошибки. Программа, пытающаяся что-либо записать на диск или его отформатировать, получит сообщение об ошибке и посчитает, что либо диск защищен от записи, либо сектор не найден, либо что-нибудь другое (все зависит от того, какой код ошибки вернет наша процедура).

Можно поступить еще хитрее. Фактически запись/форматирование не производить, но передавать программе сообщение об успешном выполнении того или

иного действия, при этом зафиксировав попытку записи/форматирования в свой файл отчета, который может посмотреть пользователь.

Этот способ можно также включить в свою оболочку, причем в разных вариантах. Например:

- производить запись/форматирование всегда;
- производить запись/форматирование только в оболочке;
- не производить запись/форматирование вообще.

При этом пользователь должен выбрать тот вариант, который ему больше всего подходит в определенное время. Таким образом, можно исключить на 99% вероятность заражения других файлов вирусом, который находится на незнакомой диске и запускается пользователем.

Обращаем ваше внимание, что все функции работы с дисками и файлами MS-DOS работают с помощью прерывания 13h. Поэтому достаточно перехватить только прерывание BIOS.

Комбинируя первый и второй способы в вашей оболочке или программе, можно довольно надежно защитить компьютер от заражения неизвестным вирусом и спокойно запускать любые новые программы с чужих дисков.



Глава 32

Все о диске и файловой системе

32.1. Что находится на диске?

В этой главе мы рассмотрим, что располагается в определенных участках жесткого и гибкого дисков. Для более детального и практического изучения следует воспользоваться специальными программами. Оптимальной является DiskEdit из пакета известных утилит Norton Commander (Norton Utilities). Любому ассемблерщику рано или поздно придется прибегнуть к помощи этой программы или подобной ей, особенно если он пишет код для работы с дисками.

32.1.1. Таблица разделов жесткого диска

После включения или перезагрузки компьютера, первое, что считывается в память, — таблица разделов жесткого диска, представляющая собой специально написанную программу с информацией о жестком диске. Она выполняет определенные действия, а затем загружает и передает управление загрузочному сектору (*см. далее*). Таблица разделов жесткого диска считывается в память всегда, если в компьютере присутствует винчестер, который должным образом разбит на разделы. Даже в случае, если пользователь загружает операционную систему с гибкого диска. Дискеты не содержат эту таблицу, и первое, что на них находится, — это загрузочный сектор.

Таблица разделов хранит в себе информацию о разделах винчестера: название раздела, стартовый сектор, количество секторов и пр. Эту информацию изначально заносит в нулевой цилиндр, нулевую дорожку, первый сектор программа FDISK (или подобная ей) перед тем, как винчестер будет отформатирован. Пользователь может создать всего один раздел (тогда буква С будет относиться к жесткому диску, D — к CD-ROM и к прочим устройствам), а может разбить винчестер на несколько разделов (C, D, E — жесткие диски, F — CD-ROM и т. п.).

Если вручную или программно обнулить этот сектор, то вся информация о разделах будет уничтожена, вследствие чего операционная система не сможет получить доступ к диску. ОС вообще не распознает, что это жесткий диск, т. к. программа, расположенная в этом секторе, не загрузится в память. Писать в эту область диска можно только с помощью прерывания 13h или прямого программирования контроллера жесткого диска.

Однако следует иметь в виду, что в случае уничтожения информации из таблицы разделов пользователь сможет все-таки получить доступ и даже восстановить ее, если:

- он предварительно сохранил таблицу разделов на дискету. Это позволяют делать многие специальные программы. Например: Norton Utilities, Nuts&Bolts и пр.;
- жесткий диск имеет всего один раздел. В таком случае опытный пользователь может вручную записать начальный и конечный секторы напрямую в таблицу разделов (которые известны, т. к. весь диск отведен для одного раздела);
- получить доступ к файлам с помощью специальных программ. Например, с помощью DiskEdit из пакета утилит Norton Commander или с помощью DOS Navigator.

В любом случае, на восстановление таблицы разделов жесткого диска может понадобиться много времени и нервов, поэтому эксперименты с ней требуют от программиста двойной бдительности и осторожности.

32.1.2. Загрузочный сектор

После таблицы разделов следует загрузочный сектор (BOOT-сектор), к которому можно получить доступ не только с помощью прерывания 13h, но 25h (чтение сектора) / 26h (запись сектора) DOS.

Как уже упоминалось ранее, и таблица разделов, и загрузочный сектор — это программы, которые загружаются перед всеми остальными файлами операционной системы. Чтобы в этом убедиться, а также разобрать программы (например, для изучения их работы, если вы пишете собственную ОС), достаточно прочитать эти секторы в память, сохранить в файле и посмотреть в отладчике или в Hacker's View.

Загрузочный сектор хранит следующую информацию (табл. 32.1).

Таблица 32.1. Поля загрузочного сектора

Смещение	Описание
00h	"Прыжок" на метку инициализации
03h	Идентификатор — содержит сведения о программе, создавшей этот сектор
0Bh	Количество байт на сектор — как правило — 512, хотя есть и 128, 256, 1024
0Dh	Количество секторов на кластер
0Eh	Количество зарезервированных секторов — после них идет таблица размещения файлов (FAT)
10h	Количество копий таблицы размещения файлов (FAT) — как правило, две
11h	Количество файлов и подкаталогов, которые могут храниться в корневом каталоге — их количество зависит от объема диска
13h	Общее количество секторов на диске

Таблица 32.1 (окончание)

Смещение	Описание
15h	Дескриптор — для жесткого диска — 0F8h, для дискет 3,5 дюйма и CD-ROM — 0F0h
16h	Количество секторов в FAT
18h	Количество секторов на дорожку (необходимо для прерывания 13h)
1Ah	Количество сторон (необходимо для прерывания 13h)
1Ch	Количество специальных спрятанных секторов (как правило, 0)
24h	Физический номер диска (начиная от 0)
26h	Расширенная сигнатура загрузочного диска
27h	Серийный номер тома
2Bh	Метка тома (может не соответствовать реальной)
36h	Идентификатор файловой системы диска (FAT, FAT32, VFAT и т. д.)

Boot-сектор создается в процессе форматирования дискеты, причем его код различается в зависимости от того, какой программой производится форматирование.

Например, утилита FORMAT.COM записывает программу, которая выдает на экран сообщение "Non-system disk or disk error" (несистемный диск или диск с ошибкой), если на диске не найдены файлы операционной системы или они находятся в области плохих секторов (не смогли быть полностью прочитаны). После того как системные файлы загрузились, программа, расположенная в Boot-секторе, передает им управление и на этом ее работа заканчивается.

Загрузочный сектор легко восстанавливается программами типа ScanDisk, Norton DiskDoctor и пр.

32.1.3. Таблица размещения файлов (FAT)

Куда сложнее восстановить таблицу размещения файлов (File Allocation Table, FAT). Можно сказать, что 100-процентному восстановлению она вообще не подлежит.

Таблица размещения файлов содержит ссылки на секторы, в которых последовательно находятся очередные порции файлов. Принцип идентичен тому, который описывался в разд. 31.1.2.

Начальный сектор файла находится в том каталоге, в котором расположен файл. По сути дела, любой подкаталог, кроме корневого, располагается на диске как обычный файл. В нем содержится следующая информация:

- имена и расширения файлов и подкаталогов, находящихся в этом каталоге;
- их размеры, дата и время создания, атрибуты;
- стартовый кластер файла.

Когда необходимо загрузить некоторый файл, неважно, текстовый, программный или иной, операционная система получает в каталоге, в котором находится

этот файл, его стартовый кластер и считывает в память. Если файл имеет больший размер, чем кластер, то ОС находит в таблице размещения файлов следующий кластер, затем следующий и так по цепочке до тех пор, пока не получит сигнал последнего кластера. Сигналом является число FFF0h для 16-битной FAT (FAT16) или FFFF:FFF0h для 32-битной (FAT32).

Следует отметить, что операционная система для более быстрого доступа к файлам постоянно хранит в памяти заранее считанную таблицу размещения файлов. Любое изменение длины или содержимого файла заносится на диск не только в область данных, но и в FAT, и в каталог, если стартовый кластер или размер файла изменились.

В начальных секторах диска содержится, как правило, две копии FAT — если одна повредится, то ОС будет использовать вторую копию. При проверке диска специальными программами происходит сравнение двух копий. Если одна не соответствует другой, то проверяющая программа делает их идентичными, как и должно быть.

32.2. Удаление и восстановление файла

Вы, вероятно, обратили внимание, что создание и запись файла на диск происходит дольше, чем его удаление. Это связано с тем, что операционная система не очищает заполненные удаляемым файлом сектора, а просто вносит изменения в каталог и FAT. При этом первый символ файла в каталоге меняется на 0E5h. Вспомните программу восстановления удаленных файлов — она запрашивает у пользователя первое имя восстанавливаемого файла. Если в названии файла первый символ 0E5h, то ОС такие файлы на экран не выводит, считая их удаленными, хотя физически названия в каталоге остаются до тех пор, пока создаваемый файл их не перезапишет.

При удалении файла изменяется также и таблица размещения файлов. Те кластеры, которые отмечены за удаляемым файлом, просто обнуляются, давая возможность записывать на их место другую информацию. Обращаем ваше внимание, что содержимое секторов не обнуляется, только FAT! При этом становится возможным восстановление удаленного файла. Но не всегда!

Если некоторая другая программа или ее часть уже записалась в освободившиеся секторы, то восстановить файл полностью, естественно, не удастся. Запомните: если вы случайно удалили важный файл, то немедленно прекратите всякую запись на диск, даже загрузку ОС с этого диска, т. к. операционная система сама периодически производит запись (в частности, изменяет SWAP-файл). Лучший способ — загрузиться с другой дискеты и запустить программу восстановления файлов.

Однако и в этом случае нет 100-процентной гарантии его успешного восстановления. Если диск был сильно фрагментирован, да и файл имел немаленький размер, то программа восстановления файлов вам вряд ли поможет, т. к. файл может быть разбросан по всему диску: часть его может находиться в начале диска, часть в середине и часть в конце. А так как FAT обнулена, то и восстанавливать придется "вручную", используя при этом прямое редактирование диска (например, программой DiskEdit). Но это дело очень кропотливое и долгое.

32.3. Ошибки файловой системы

32.3.1. Потерянные кластеры файловой системы FAT, FAT32

Довольно часто на диске возникают так называемые *потерянные кластеры*. Это кластеры, которые отмечены как используемые (принадлежат какому-то файлу), но на самом деле не задействованы. В таком случае программа проверки дисков, обнаружив потерянные кластеры, преобразует их в файлы в корневом каталоге (просто присваивает им имена) или отмечает как свободные (на выбор пользователя). Потерянные цепочки кластеров возникают в том случае, если в процессе работы внезапно прекратилась подача электропитания или пользователь в момент записи данных на диск выключил питание компьютера.

Как известно, данные не сразу пишутся на диск, а попадают в специально созданную кэш-память (если загружена программа типа SmartDrv). Когда некоторая программа отправила порцию данных на диск, операционная система помещает их в кэш оперативной памяти, при этом может изменить FAT, корневой каталог и пр., но данные не записываются сразу. Возможен также вариант, когда часть данных записалась, а часть не успела. И если в этот момент пользователь выключит компьютер или нажмет кнопку Reset, то на диске появляются эти самые потерянные кластеры.

Напоследок еще раз хотелось бы порекомендовать вам поработать с программой DiskEdit. Уверяем вас, что от этого вы ничего не потеряете, даже наоборот. Возьмите чистую дискету, запишите на нее несколько файлов и просматривайте в DiskEditor. Вам откроются все секреты файловых систем FAT и FAT32.



ПРИЛОЖЕНИЯ



Приложение 1

Ассемблирование программ (получение машинного кода из ассемблерного листинга)

П1.1. Загрузка MASM 6.10—6.13

Для перевода ассемблерного файла в машинный код необходимо воспользоваться специальной программой-ассемблером. Наиболее популярным можно назвать мощный Microsoft Macro Assembler версий 6.10—6.13, который позволяет создавать машинный код как для операционной системы MS-DOS, так и для Windows.¹ Если у вас уже установлено необходимое программное обеспечение, то просто переходите ко второму шагу.

- Загрузите с сайта <http://www.Kalashnikoff.ru> самораспаковывающийся архив самой программы MASM.EXE (около 6 Мбайт).
- Распакуйте полученный архив. Для этого запустите загруженный файл MASM.EXE в ОС Windows и укажите каталог, в который необходимо сохранить файлы из архива.
- Зайдите в каталог, в который был распакован архив с MASM (то, что вы указали в п. 2 выше).
- В подкаталоге BIN находятся основные программы. В нем вы можете создавать ASM-файлы и ассемблировать их так, как описывается далее.

П1.2. Ассемблирование

Программа-ассемблер (MASM, TASM, WASM, NASM и пр.) создает объектный файл с расширением OBJ. Данный файл является переходным между ассемблерным файлом (ASM) и программой (COM/EXE).

В случае если ассемблерный листинг слишком большой, то программу разбивают на несколько частей. В большинстве случаев обходятся директивой `include` (таким образом мы ассемблировали нашу оболочку).

¹ На сайте компании Microsoft в Центре загрузки (<http://www.microsoft.com/downloads/en/default.aspx>) свободно можно скачать Microsoft Macro Assembler 8.0 (MASM) Package (x86). — Ред.

Однако если файлы, присоединяемые указанной ранее директивой, большие и в основном неизменяемые (т. е. готовые процедуры, не требующие редактирования), то постоянное асsembлирование этих процедур может занять много времени. В таком случае, каждая отдельно взятая часть программы (ассемблерный код) асsembлируется по отдельности, при этом создается один или несколько объектных файлов (с расширением OBJ), которые не требуют постоянного асsembлирования, только компоновки (линковки) (см. разд. П1.3).

П1.3. Компоновка

Если в процессе асsembлирования не было выявлено ошибок в асSEMBлерном листинге, то программа-ассемблер создаст объектный файл (с расширением OBJ).

Затем необходимо воспользоваться компоновщиком (линковщиком), который входит в комплект программы-ассемблера. Данная процедура выполняется гораздо быстрее асsembлирования.

Именно компоновщик создает готовый к запуску файл (программу) с расширением COM или EXE из объектного файла (OBJ). Оба типа имеют отличия в структуре асSEMBлерной программы. Первый тип (COM) не может превышать 64 Кбайт и используется только в MS-DOS (и для совместимости поддерживается в Windows), однако он очень компактный и удобный для написания небольших программ и резидентов. В большинстве случаев, если программа написана на чистом асSEMBлере под MS-DOS, нет необходимости создавать EXE-файлы. В этой книге в части I рассматриваются именно программы типа COM.

В отличие от создания программ типа COM, при создании стандартных EXE-программ под MS-DOS нет необходимости указывать какие-либо параметры линковщику при компоновке. Дело в том, что компоновщик не может автоматически определить, какой тип подвергается компоновке.

Линковщик также проверяет, нет ли каких-либо ошибок в объектном файле, но не грамматических, а логических. Например, отсутствие необходимой объектной библиотеки, указанной в самом файле либо в командной строке (программа-ассемблер этого не делает).

Если ошибки не были обнаружены, компоновщик создает машинный код (программу типа COM или EXE), которую можно запускать на выполнение.

ПРИМЕЧАНИЕ

Исходя из всего вышеизложенного, делаем вывод, что для создания машинного кода необходимо воспользоваться как минимум двумя программами: *программой-ассемблером и компоновщиком*. Однако для MASM версий 6.00—6.13 достаточно запустить файл ml.exe, указав в командной строке параметр /AT в процессе асsembлирования. В таком случае MASM (если не было ошибок в асSEMBлерном листинге) автоматически запустит компоновщик (LINK.EXE), который создаст файл типа COM.

П1.3.1. Ассемблирование и компоновка программ пакетами Microsoft (MASM)

Допустим, вы создали в текстовом редакторе файл с именем PROG.ASM.

Если вы используете MASM 6.11—6.13, то в командной строке необходимо указать следующее:

```
> ML.EXE PROG.ASM /AT
```

В результате будут созданы два файла: PROG.OBJ и PROG.COM. Файл PROG.OBJ, скорее всего, вам больше не понадобится, и его можно удалить, а PROG.COM можно запускать на выполнение. Это и будет машинный код ассемблерной программы. Параметр /AT указывает программе-ассемблеру (MASM), что после ассемблирования, в случае, если ошибок не будет обнаружено, следует запустить компоновщик (LINK.EXE) и передать ему параметры для создания файла типа COM.

ВНИМАНИЕ!

Параметр /AT должен быть набран ПРОПИСНЫМИ символами!



Приложение 2

Типичные ошибки при ассемблировании программы

Tlink32.exe не компилирует файл, выдает ошибку:

```
Fatal: 16 bit segments not supported in module prog.asm
```

TASM32.EXE и TLINK32.EXE — ассемблер и компоновщик только для программ, написанных под ОС Windows!

Для наших примеров на данном этапе необходимы TASM.EXE и TLINK.EXE (мы рекомендуем MASM 6.11—6.13).

LINK при компиляции выдает:

```
LINK : warning L4021: no stack segment
```

Однако файл с расширением EXE создается.

Данное сообщение свидетельствует о том, что вы забыли указать стек в EXE-файле. Если вы написали программу типа COM, а ассемблируете ее как EXE, опуская необходимые параметры для COM-файла, то данная COM-программа будет работать некорректно.

Если вы создаете EXE-файл, то просто игнорируйте это сообщение либо создайте сегмент стека.

Обратите внимание, что в этой книге рассматриваются в большинстве своем программы типа COM. Как получить COM-файл, сказано в *приложении 1*.

Ассемблер (TASM) выдает ошибку:

```
**Error** prog4.asm(15) Near jump or call to different CS
```

Поместите в вашу программу после строки CSEG segment следующее:

```
ASSUME CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
```

Сассемблированный файл не работает: компьютер виснет (программа работает не так, как надо: вместо выводимой строки — какие-то непонятные символы и пр.), хотя программа набрана верно (точь-в-точь, как в примере из книги).

Проблема, вероятно, в том, что вы написали COM-файл, а ассемблируете его, как EXE. Как правильно сассемблировать COM-файл, сказано в *приложении 1*.



Приложение 3

Таблицы и коды символов

П3.1. Основные символы ASCII

В табл. П3.1 приведены ASCII-символы от 00h до 7Fh, их коды в десятичной (колонка **DEC**), шестнадцатеричной (колонка **HEX**) и двоичной (колонка **BIN**) системах счисления.

В колонке **Скан-код** приводятся скан-коды *нажатия* соответствующих клавиш, расположенных на основной клавиатуре. Скан-код *отпускания* клавиши соответствует скан-коду нажатия клавиши с установленным старшим седьмым битом (OR 10000000b).

Полный список скан-кодов клавиатуры см. в табл. П3.4.

В колонке **Описание** содержится дополнительная информация о соответствующем символе и/или его коде.

Условные сокращения:

- **DOS** — отображение символа на экране при помощи вывода с использованием функций 09h и 02h прерывания 21h MS-DOS;
- **ПОВ** — отображение символа на экране при помощи вывода с использованием метода прямого отображения в видеобуфер.

Таблица также позволяет легко и быстро переводить числа в разных системах счисления от 00h до 7Fh (от 0 до 127).

Таблица П3.1. ASCII-символы от 00h до 7Fh, их коды в десятичной, шестнадцатеричной и двоичной системах счисления

Символ	DEC	HEX	BIN	Скан-код	Описание
	0	00	00000000	—	Null DOS, ПОВ: отображает как "пробел" (ASCII 20h)
☺	1	01	00000001	—	—
☻	2	02	00000010	—	—
♥	3	03	00000011	—	—
♦	4	04	00000100	—	Конец передачи (<Ctrl>+<D>)

Таблица П3.1 (продолжение)

Символ	DEC	HEX	BIN	Скан-код	Описание
♣	5	05	00000101	—	—
♠	6	06	00000110	—	—
•	7	07	00000111	—	Звонок DOS: выдает звуковой сигнал в динамике ПОВ: выводит сам символ
•	8	08	00001000	0Eh	<Backspace> (удаление символа слева от курсора) DOS: передвигает курсор на одну позицию влево ПОВ: выводит сам символ
○	9	09	00001001	0Fh	<Tab> DOS: вставляет табуляцию (перемещает курсор максимум на 8 позиций вправо) ПОВ: выводит сам символ
▣	10	0A	00001010	—	Перевод строки DOS: переводит курсор на следующую строку текущего столбца ПОВ: выводит сам символ
♂	11	0B	00001011	—	—
♀	12	0C	00001100	—	—
♪	13	0D	00001101	—	Возврат каретки DOS: переводит курсор в начало текущей строки ПОВ: выводит сам символ
♫	14	0E	00001110	—	—
☼	15	0F	00001111	—	—
▶	16	10	00010000	—	—
◀	17	11	00010001	—	—
↑	18	12	00010010	—	—
!!	19	13	00010011	—	—
¶	20	14	00010100	—	—
§	21	15	00010101	—	—
	22	16	00010110	—	—

Таблица П3.1 (продолжение)

Символ	DEC	HEX	BIN	Скан-код	Описание
↓	23	17	00010111	—	—
↑	24	18	00011000	—	—
↓	25	19	00011001	—	—
→	26	1A	00011010	—	—
←	27	1B	00011011	01h	<ESC> DOS, ПОВ: выводит символ (стрелка влево)
„	28	1C	00011100	—	—
↔	29	1D	00011101	—	—
▲	30	1E	00011110	—	—
▼	31	1F	00011111	—	—
	32	20	00100000	4Bh	<Пробел>
!	33	21	00100001	02h	—
”	34	22	00100010	28h	—
#	35	23	00100011	04h	—
\$	36	24	00100100	05h	—
%	37	25	00100101	06h	—
&	38	26	00100110	08h	—
'	39	27	00100111	28h	—
(40	28	00101000	0Ah	—
)	41	29	00101001	0Bh	—
*	42	2A	00101010	09h	—
+	43	2B	00101011	0Dh	—
,	44	2C	00101100	33h	—
-	45	2D	00101101	0Ch	—
.	46	2E	00101110	34h	—
/	47	2F	00101111	35h	—
0	48	30	00110000	0Bh	—
1	49	31	00110001	02h	—
2	50	32	00110010	03h	—
3	51	33	00110011	04h	—
4	52	34	00110100	05h	—

Таблица П3.1 (продолжение)

Символ	DEC	HEX	BIN	Скан-код	Описание
5	53	35	00110101	06h	—
6	54	36	00110110	07h	—
7	55	37	00110111	08h	—
8	56	38	00111000	09h	—
9	57	39	00111001	0Ah	—
:	58	3A	00111010	27h	—
;	59	3B	00111011	27h	—
<	60	3C	00111100	33h	—
=	61	3D	00111101	0Dh	—
>	62	3E	00111110	34h	—
?	63	3F	00111111	35h	—
@	64	40	01000000	03h	—
A	65	41	01000001	1Eh	—
B	66	42	01000010	30h	—
C	67	43	01000011	2Eh	—
D	68	44	01000100	20h	—
E	69	45	01000101	12h	—
F	70	46	01000110	21h	—
G	71	47	01000111	22h	—
H	72	48	01001000	23h	—
I	73	49	01001001	17h	—
J	74	4A	01001010	24h	—
K	75	4B	01001011	25h	—
L	76	4C	01001100	26h	—
M	77	4D	01001101	32h	—
N	78	4E	01001110	31h	—
O	79	4F	01001111	18h	—
P	80	50	01010000	19h	—
Q	81	51	01010001	10h	—
R	82	52	01010010	13h	—
S	83	53	01010011	1Fh	—
T	84	54	01010100	14h	—

Таблица П3.1 (продолжение)

Символ	DEC	HEX	BIN	Скан-код	Описание
U	85	55	01010101	16h	—
V	86	56	01010110	2Fh	—
W	87	57	01010111	11h	—
X	88	58	01011000	2Dh	—
Y	89	59	01011001	15h	—
Z	90	5A	01011010	2Ch	—
[91	5B	01011011	1Ah	—
\	92	5C	01011100	2Bh	—
]	93	5D	01011101	1Bh	—
^	94	5E	01011110	07h	—
_	95	5F	01011111	0Ch	—
`	96	60	01100000	29h	—
a	97	61	01100001	1Eh	—
b	98	62	01100010	30h	—
c	99	63	01100011	3Eh	—
d	100	64	01100100	20h	—
e	101	65	01100101	12h	—
f	102	66	01100110	21h	—
g	103	67	01100111	22h	—
h	104	68	01101000	23h	—
i	105	69	01101001	17h	—
j	106	6A	01101010	24h	—
k	107	6B	01101011	25h	—
l	108	6C	01101100	26h	—
m	109	6D	01101101	32h	—
n	110	6E	01101110	31h	—
o	111	6F	01101111	18h	—
p	112	70	01110000	19h	—
q	113	71	01110001	10h	—
r	114	72	01110010	13h	—
s	115	73	01110011	1Fh	—
t	116	74	01110100	14h	—

Таблица П3.1 (окончание)

Символ	DEC	HEX	BIN	Скан-код	Описание
u	117	75	01110101	16h	—
v	118	76	01110110	2Fh	—
w	119	77	01110111	11h	—
x	120	78	01111000	2Dh	—
y	121	79	01111001	15h	—
z	122	7A	01111010	2Ch	—
{	123	7B	01111011	1Ah	—
:	124	7C	01111100	2Bh	—
}	125	7D	01111101	1Bh	—
~	126	7E	01111110	29h	—
△	127	7F	01111111	—	DOS, ПОВ: выводит символ как есть

В табл. П3.2 показаны символы ASCII (от 80h до FFh) кодировки IBM cp866 с кодами в десятичной, шестнадцатеричной и двоичной системах счисления. Настоящая кодировка используется в DOS-программах как основная кодировка для отображения русских букв и псевдографики.

Таблица П3.2. Символы ASCII (от 80h до FFh) кодировки IBM cp866 с кодами в десятичной, шестнадцатеричной и двоичной системах счисления

Символ	DEC	HEX	BIN	Символ	DEC	HEX	BIN
А	128	80	10000000	Л	192	C0	11000000
Б	129	81	10000001	Ł	193	C1	11000001
В	130	82	10000010	Т	194	C2	11000010
Г	131	83	10000011	†	195	C3	11000011
Д	132	84	10000100	—	196	C4	11000100
Е	133	85	10000101	+	197	C5	11000101
Ж	134	86	10000110	ƒ	198	C6	11000110
З	135	87	10000111		199	C7	11000111
И	136	88	10001000	ŁŁ	200	C8	11001000
Й	137	89	10001001	ЃЃ	201	C9	11001001
К	138	8A	10001010	҃҃	202	CA	11001010
Л	139	8B	10001011	҄҄	203	CB	11001011

Таблица П3.2 (продолжение)

Символ	DEC	HEX	BIN	Символ	DEC	HEX	BIN
М	140	8C	10001100	¶	204	CC	11001100
Н	141	8D	10001101	=	205	CD	11001101
О	142	8E	10001110	‡	206	CE	11001110
П	143	8F	10001111		207	CF	11001111
Р	144	90	10010000	₩	208	D0	11010000
С	145	91	10010001	₩	209	D1	11010001
Т	146	92	10010010	₩	210	D2	11010010
У	147	93	10010011	₩	211	D3	11010011
Ф	148	94	10010100	£	212	D4	11010100
Х	149	95	10010101	₣	213	D5	11010101
Ц	150	96	10010110	₪	214	D6	11010110
Ч	151	97	10010111	₩	215	D7	11010111
Ш	152	98	10011000	₩	216	D8	11011000
Щ	153	99	10011001	Ј	217	D9	11011001
ъ	154	9A	10011010	Г	218	DA	11011010
ы	155	9B	10011011	█	219	DB	11011011
ь	156	9C	10011100	■	220	DC	11011100
Э	157	9D	10011101	▀	221	DD	11011101
Ю	158	9E	10011110	█	222	DE	11011110
Я	159	9F	10011111	■	223	DF	11011111
а	160	A0	10100000	р	224	E0	11100000
б	161	A1	10100001	с	225	E1	11100001
в	162	A2	10100010	т	226	E2	11100010
г	163	A3	10100011	у	227	E3	11100011
д	164	A4	10100100	ф	228	E4	11100100
е	165	A5	10100101	х	229	E5	11100101
ж	166	A6	10100110	ц	230	E6	11100110
з	167	A7	10100111	ч	231	E7	11100111
и	168	A8	10101000	ш	232	E8	11101000
й	169	A9	10101001	щ	233	E9	11101001
к	170	AA	10101010	ъ	234	EA	11101010

Таблица П3.2 (окончание)

П3.2. Расширенные коды ASCII

Расширенные коды ASCII находятся в табл. П3.3. Расширенный код ASCII возвращается в регистр ah после выполнения функции $00h$ прерывания $16h$, при этом al содержит 0.

ПРИМЕЧАНИЕ

Клавиша, помеченная символом * (звездочка) с двух сторон, находится на цифровой клавиатуре.

Таблица П3.3. Расширенные коды ASCII

Функциональные клавиши							
Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<F1>	3Bh	<Alt>+<F1>	68h	<Ctrl>+<F1>	5Eh	<Shift>+<F1>	54h
<F2>	3Ch	<Alt>+<F2>	69h	<Ctrl>+<F2>	5Fh	<Shift>+<F2>	55h
<F3>	3Dh	<Alt>+<F3>	6Ah	<Ctrl>+<F3>	60h	<Shift>+<F3>	56h
<F4>	3Eh	<Alt>+<F4>	6Bh	<Ctrl>+<F4>	61h	<Shift>+<F4>	57h
<F5>	3Fh	<Alt>+<F5>	6Ch	<Ctrl>+<F5>	62h	<Shift>+<F5>	58h
<F6>	40h	<Alt>+<F6>	6Dh	<Ctrl>+<F6>	63h	<Shift>+<F6>	59h
<F7>	41h	<Alt>+<F7>	6Eh	<Ctrl>+<F7>	64h	<Shift>+<F7>	5Ah
<F8>	42h	<Alt>+<F8>	6Fh	<Ctrl>+<F8>	65h	<Shift>+<F8>	5Bh
<F9>	43h	<Alt>+<F9>	70h	<Ctrl>+<F9>	66h	<Shift>+<F9>	5Ch
<F10>	44h	<Alt>+<F10>	71h	<Ctrl>+<F10>	67h	<Shift>+<F10>	5Dh
<F11>	85h	<Alt>+<F11>	8Bh	<Ctrl>+<F11>	89h	<Shift>+<F11>	87h
<F12>	86h	<Alt>+<F12>	8Ch	<Ctrl>+<F12>	8Ah	<Shift>+<F12>	88h
Клавиши управления курсором							
Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<-->	4Bh	<Alt>+<-->	9Bh	<Ctrl>+<-->	73h	<Ins>	52h
<-->	4Dh	<Alt>+<-->	9Dh	<Ctrl>+<-->	74h	<Alt>+<Ins>	A2h
<↑>	48h	<Alt>+<↑>	98h	<Ctrl>+<↑>	нет		53h
<↓>	50h	<Alt>+<↓>	A0h	<Ctrl>+<↓>	нет	<Alt>+	A3h
<PageUp>	49h	<Alt>+<PgUp>	99h	<Ctrl>+<PgUp>	84h	<Alt>+<BS>	0Eh
<Page-Down>	51h	<Alt>+<PgDn>	A1h	<Ctrl>+<PgDn>	76h	<Alt>+<Tab>	A5h
<Home>	47h	<Alt>+<Home>	97h	<Ctrl>+<Home>	77h	<Ctrl>+<Tab>	94h
<End>	4Fh	<Alt>+<End>	9Fh	<Ctrl>+<End>	75h	<Shift>+<Tab>	0Fh
Символьные клавиши и <Enter>							
Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<Alt>+<A>	1Eh	<Alt>+<H>	23h	<Alt>+<O>	18h	<Alt>+<V>	2Fh
<Alt>+	30h	<Alt>+<I>	17h	<Alt>+<P>	19h	<Alt>+<W>	11h
<Alt>+<C>	2Eh	<Alt>+<J>	24h	<Alt>+<Q>	10h	<Alt>+<X>	2Dh
<Alt>+<D>	20h	<Alt>+<K>	25h	<Alt>+<R>	13h	<Alt>+<Y>	15h

Таблица П3.3 (окончание)

Символьные клавиши и <Enter>							
Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<Alt>+<E>	12h	<Alt>+<L>	26h	<Alt>+<S>	1Fh	<Alt>+<Z>	2Ch
<Alt>+<F>	21h	<Alt>+<M>	32h	<Alt>+<T>	14h	<Alt>+<Enter>	1Ch
<Alt>+<G>	22h	<Alt>+<N>	31h	<Alt>+<U>	16h	*<Alt>+<Enter>*	A6h
Иные комбинации клавиш							
Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<Alt>+<>	2Bh	<Alt>+<[>	1Ah	* <Alt>+</> *	A4h	* <Ctrl>+</> *	95h
<Alt>+<,>	33h	<Alt>+<]>	1Bh	* <Alt>+<*>*	37h	* <Ctrl>+<*>*	96h
<Alt>+<<>	34h	<Alt>+<'>	28h	* <Alt>+<-> *	4Ah	* <Ctrl>+<+> *	90h
<Alt>+<;>	27h	<Alt>+<`>	29h	* <Alt>+<+> *	4Eh	* <Ctrl>+<-> *	8Eh
<Alt>+</>	35h	<Alt>+<=>	8Ch	* <Ctrl>+<.> *	93h	<SysRq>	72h

П3.3. Скан-коды клавиатуры

В табл. П3.4 приведены скан-коды клавиатуры. Скан-код *нажатой* клавиши можно получить путем считывания его с порта 60h при помощи оператора `in`. Скан-код *отпущенной* клавиши соответствует скан-коду нажатой клавиши с установленным старшим седьмым битом (OR 10000000b). Подробнее о скан-кодах см. в главе 15.

Таблица П3.4. Скан-коды клавиатуры

Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
<Esc>	01h	U	16h	<Right Shift>	36h	<F6>	40h
1 !	02h	I	17h	\	2Bh	<F7>	41h
2 @	03h	O	18h	Z	2Ch	<F8>	42h
3 #	04h	P	19h	X	2Dh	<F9>	43h
4 \$	05h	[{	1Ah	C	2Eh	<F10>	44h
5 %	06h] }	1Bh	V	2Fh	<F11>	57h
6 ^	07h	<Enter>	1Ch	B	30h	<F12>	58h
7 &	08h	<Ctrl>	1Dh	N	31h	<Num Lock>	45h
8 *	09h	A	1Eh	M	32h	<Scroll Lock>	46h
9 (0Ah	S	1Fh	, <	33h	<Home>	47h
0)	0Bh	D	20h	. >	34h	-	48h

Таблица П3.4 (окончание)

Клавиша	Код	Клавиша	Код	Клавиша	Код	Клавиша	Код
- _	0Ch	F	21h	/ ?	35h	<PageUp>	49h
= +	0Dh	G	22h	* * *	37h	* - *	4Ah
BS	0Eh	H	23h	<Alt>	38h	*+*	4Eh
Tab	0Fh	J	24h	<Space>	39h	<End>	4Fh
Q	10h	K	25h	<Caps Lock>	3Ah	<PageDown>	51h
W	11h	L	26h	<F1>	3Bh	<Insert>	52h
E	12h	; :	27h	<F2>	3Ch	<Delete>	53h
R	13h	' "	28h	<F3>	3Dh	<Left Win>	5Bh
T	14h	` ~	29h	<F4>	3Eh	<Right Win>	5Ch
Y	15h	<Left Shift>	2Ah	<F5>	3Fh	<Menu>	5Dh



Приложение 4

Содержимое компакт-диска

К книге прилагается компакт-диск, содержащий все необходимое для работы с ассемблером по данной книге (табл. П4.1).

Таблица П4.1. Содержимое компакт-диска

Каталог	Описание
help	Полезный справочник для данного курса
i80386, i80486	Описание 32-разрядных процессоров Intel
Файлы-приложения	Примеры, рассматриваемые в книге

Предметный указатель

A

ASCII-код 66, 71, 163
расширенный 69, 76
ASCII-символы 32, 313, 318, 320

B, C, D

BOOT-сектор 302, 303, 304
COM-файлы 15
DTA 129, 130, 135, 136, 158, 159, 225,
226, 229, 231, 232

E

EPB 278, 281—283
EXE-файлы 15

F

FAT 304—306
FCB 284

H

himem.sys 288—290

M

MASM 47
Masm.exe 4
установка 309

P, T, X

PSP 225, 226, 229, 231, 232
TASM 4, 47, 312
XMS-память 109, 288—295

A

Антивирус 97, 238, 242—247,
299, 300
Архив 297, 298
Ассемблер 4
Ассемблирование 309, 310
программ 15

Б

Байт 23—25, 39
Бит 23—25

В

Вектор прерывания 149—152
таблица векторов
прерываний 170—172
Видеобуфер 142, 183, 184
Видеокарта 25, 112, 113, 132, 139,
140, 144, 193, 194
Видеорежим 113
CGA+ 112
VGA+ 112
Видеостраница 113, 114, 132, 135,
139, 140, 162, 194
адрес 132
Вирус 97, 98, 128, 129, 131, 132,
134, 136, 176—182, 205—211,
299, 300

Д

Дизассемблер 4, 223

3

Загрузка программы 278—287

Загрузочный сектор 302—304

Запуск:

 BAT-файла 297

 COM-файла 297

 EXE-файла 297

 программы 278—287

И

Инициализация 115

К

Каталог:

 создание 297

 чтение в память 260

Командная строка 283

Комментарий 13

Компоновка 310

Курсор, управление 139, 140

М

Модель памяти 137, 138

Монитор, модели 142

О

Оболочка 109, 110, 111, 113

Обработчик прерывания 100—103, 106

Оверлей 61

Окно, вывод на экран 137, 142—145

Окружение:

 DOS 282, 283

 MS-DOS 225—229, 231, 232

Оператор:

 \$ 85, 86, 87

 add 19, 22

 and 160, 161

 assume 47

 call 49, 54, 55, 151

 clc 233, 234

 cld 234, 235

 cli 59, 101

cmp 66

dec 21, 22, 45

div 143, 184

equ 190—192

fadd 251, 252

fild 250—252

fist 252, 253

inc 20, 22

int 31

iret 155—157

JA 233

JAE 233

JB 233

JBE 233

jc 78, 79

je 68

jmp 44, 45, 69

JNA 233

JNB 233

jnc 78, 79

jz 67, 68

lod_s 118, 122, 124, 125

loop 42, 43

mov 13, 17

movs 132—134

mul 146, 184

nop 61, 64, 65

offset 39

or 159, 160

org 27

pop 56, 58

popa 140

popf 104

push 56, 58

pusha 140

pushf 104

rep 118, 122, 124, 125

repe 195

ret 94—96, 152, 153, 157, 185, 186

retf 153—155, 157

scas 194, 195

shl 145

shr 145

stc 233, 234

std 234, 235
 sti 59, 101
 stos 118, 122, 124, 125
 sub 19, 22, 125, 126
 xchg 117
 xor 125, 126, 161, 162
 безусловного перехода 232
 логические 159
 работы со строками 118, 122, 124, 125
 управления флагами 233—235
Отладчик 4, 15, 23, 31, 51, 53, 54, 63, 65,
 106, 107, 165—170, 172, 173, 240, 241

П

Память 279
 кэш 306
 расширенная 288—295
Перевод:
 двоичного числа в десятичное 25
 десятичного числа в двоичное 25
 шестнадцатеричного числа в
 десятичное 26
Переход:
 безусловный 44, 69, 232
 условный 44, 69
Подпрограмма 48, 49, 58, 99
Прерывание 32, 60, 99—104, 107, 108,
 115, 155, 264—267
 MS-DOS 13, 14, 224
 аппаратное 148, 155—158, 162,
 163—166
Процедура См. подпрограмма

Р

Регистр:
 данных 17
 обнуление 125, 126
 процессора 13, 17, 238, 239
 сегментный 18, 27
 сопроцессора 250
 сохранение 284
 флагов 67
Регистры-указатели 18

Резидент 99, 115, 200—204, 230—232,
 235, 236, 299
 повторная загрузка 115—117
 удаление 264—269

С

Сегмент 26, 27, 40, 54, 59
Сегментация памяти 23, 26—29,
 31—33, 39
Сектор загрузочный 302—304
Система счисления:
 двоичная 23, 25, 313, 318
 десятичная 9, 25, 26, 313, 318
 шестнадцатеричная 9, 14, 18, 24, 26,
 313, 318

Скан-код 162, 163, 313, 322
Смещение 26, 28, 39, 40, 59, 174—179
Сопроцессор 248—251, 253—256, 289
Стек 51, 54, 55, 57, 65, 106, 185—187,
 189—192
 восстановление регистров 286, 287

Строка:
 вывод на экран 198, 199
 вычисление длины 192
 командная 226, 227
 подсчет длины нефиксированной
 строки 196, 197

Т

Таблица:
 векторов 300
 прерываний 170—172
 разделов жесткого диска 302, 303
 размещения файлов 304—306
Текстовый редактор 4

Ф

Файл:
 архивный 297, 298
 восстановление 305
 вывод имен на экран 273—277
 вывод на экран 87, 89, 260, 291, 292
(окончание рубрики см. на стр. 328)

Файл (окончание):

длинные имена 256, 257
закрытие 81, 82
заражение 207—210
копирование 296, 297
открытие 76, 77, 79—81, 84—87, 89
переименование 296, 297
перемещение 296, 297
поиск 260
размещение в памяти 261—263, 272
редактирование 297—299
удаление 296, 297, 305

чтение 87, 89

в память 271, 272

в расширенную память 291, 292

Флаг нуля 67, 104, 106

Флаг переноса 78, 79

Флаги процессора 104, 223

Ц, Я

Цикл 43, 46

создание 42

тело цикла 43, 46

Языки высокого уровня 241, 242