Глава 3

Введение в язык ассемблера

В этой главе...

- Представление данных
- Основы языка ассемблера
- Разработка программы на языке ассемблера
- Работа в DOS под Windows NT
- Инструментальные средства
- Пример простой программы
- Ассемблер Microsoft
- Отладчик
- Резюме
- Контрольные вопросы

Язык ассемблера помогает раскрыть все секреты аппаратного и программного обеспечения. С его помощью можно получить представление о том, как аппаратная часть взаимодействует с операционной системой и как прикладные программы обращаются к операционной системе. Большинство программистов работают с языками высокого уровня, где отдельное утверждение преобразовывается во множество процессорных команд. Ассемблер — язык машинного уровня; каждая команда непосредственно интерпретируется в машинный код, что дает основание считать его языком низкого уровня.

Наиболее часто язык ассемблера используется для написания дополнений к операционной системе или для написания программ прямого доступа к аппаратуре. Он необходим также при оптимизации критических блоков в прикладных программах с целью повышения их быстродействия.

Представление данных

Поскольку общение с компьютером происходит на машинном уровне, необходимо иметь представление о том, как сохраняется и обрабатывается информация. Для этого используются электрические элементы, которые могут принимать только два состояния: “включено” и “выключено”. При сохранении данных в устройствах хранения, последовательность электрических или магнитных зарядов также интерпретируется как состояние “включено” или “выключено”, что и составляет содержимое записанной информации.

Двоичные числа

Компьютер сохраняет команды и данные в оперативной памяти как последовательность заряженных или разряженных ячеек. Образно можно представить состояние каждой ячейки как переключатель с двумя состояниями: “включено и выключено” или “истина и ложь”. Такие ячейки идеально подходят для хранения двоичных чисел, которые используют *базовое число* 2, так как отдельные биты могут принимать только два состояния — 0 или 1. Ячейки памяти, соответствующие единице, имеют повышенный заряд, а соответствующие нулю — почти разряжены. На рис. 3.1 условно показано соответствие переключателей и двоичных чисел.

Включено	Выключено	Включено	Включено	Выключено	Выключено	Включено	Выключено
1	0	1	1	0	0	1	0

Рис. 3.1. *Соответствие переключателей и двоичных чисел*

Первые компьютеры имели наборы механических переключателей, управляемых вручную. На смену им пришли электромеханические переключатели, и только позже стали использоваться транзисторы. Ранее тысячи, а сегодня и миллионы электронных переключателей размещаются в микропроцессорном чипе.

Биты, байты, слова, двойные и учетверенные слова

Каждый разряд в двоичном числе называется *битом*. Восемь битов составляют *байт* — отдельно адресуемый элемент памяти в большинстве компьютеров. Байт может содержать простую машинную команду, символ или число. Следующим по размеру элементарным понятием является *слово.* В процессорах Intel слово составляет 16 бит (2 байта).

Размер слова не является жестко определенным. Компьютеры, в которых применяются процессоры Intel, используют 16-, 32- или 64-разрядные операнды, поэтому длина слова определяется в 16, 32 или 64 бит, т.е. слово может состоять из двух, четырех или восьми байт, как показано на рис. 3.2.

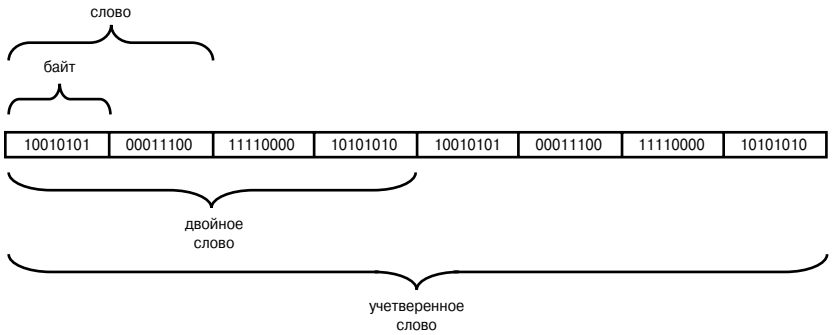

Рис. 3.2. *Размеры слов*

В табл. 3.1 представлены диапазоны значений в зависимости от количества разрядов, составляющих байты, слова, двойные слова и учетверенные слова. Наибольшее значение в диапазоне определяется как 2^b-1, где `b` — число битов.

Таблица 3.1. Размеры в битах и диапазон целых чисел

Тип слова	*Биты*	*Диапазон*
байт без знака	8	0–255
слово без знака	16	0–65 535
двойное слово без знака	32	0–4 294 967 295
учетверенное слово без знака	64	0–18 446 744 073 709 551 615

Команды и данные

В языках высокого уровня команды и данные имеют существенное логическое различие, однако в машине они все представлены одинаково, как наборы нулей и единиц. Например, следующая последовательность двоичных разрядов может включать первые три символа алфавита, сохраненные в строковой переменной, или может быть машинной командой.

```
010000010100001001000011
```

Именно поэтому программисты, использующие язык ассемблера, должны разделять данные и команды, чтобы процессор не "выполнял" переменные и не воспринимал команды как переменные.

Числовые системы

Каждая числовая система имеет *основание системы счисления*, или *базовое число* — максимальное значение, которое может быть присвоено отдельной цифре. В табл. 3.2 приведены разрешенные значения для различных систем счисления. Во всех последующих главах при отображении записей в памяти, значений регистров и адресов будут использоваться шестнадцатеричные числа, для которых основанием системы счисления является число `16`. Для компактного отображения значений больше `9` используются *шестнадцатеричные символы* от A до F, соответствующие десятичным значениям от `10` до `15`.

Когда записывают двоичное, восьмеричное или шестнадцатеричное число, к нему добавляют определенный символ, представленный строчной буквой. Например, шестнадцатеричное число `45` должно быть записано как `45h`, восьмеричное `76` — как `76o`, а двоичное `11010011` необходимо записать как `11010011b`. Таким образом ассемблер распознает числовые константы в исходной программе.

Таблица 3.2. Цифры в различных числовых системах

Система	*Базовое число*	*Разрешенные значения*
Двоичная	2	0 1
Восьмеричная	8	0 1 2 3 4 5 6 7
Десятичная	10	0 1 2 3 4 5 6 7 8 9
Шестнадцатеричная	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Правила представления числовых данных

Очень важно знать и понимать правила представления данных в памяти и отображения их на экране. Для примера воспользуемся десятичным числом `65`. Сохраненное в памяти как один байт, оно будет представлено в двоичном виде как `01000001`. Отладочная программа, вероятнее всего, будет его отображать как `41`, т.е. как шестнадцатеричное значение.

Но если это число послать в память видеоадаптера, то он воспримет его как символ, поэтому на экране увидим букву A. Это происходит потому, что в соответствии с кодировкой ASCII для символа A выбрано значение 01000001. Таким образом, интерпретация данного значения зависит от определенных условий, которые и придают ему смысл.

- ***Двоичное число*** — сохраняется в памяти как последовательность битов, готовых к использованию в расчетах. Целые двоичные числа сохраняются по 8, 16 или 32 разряда.
- ***Символы стандартного набора ASCII*** — могут быть представлены в памяти подобно числовому значению, например как 123 или 65. Для отображения символов может быть использован любой числовой формат, как показано в табл. 3.3.

Таблица 3.3. Представление буквы "A" в различных форматах

Формат	*Значение*
Двоичный символ ASCII	01000001
Восьмеричный символ ASCII	101
Десятичный символ ASCII	65
Шестнадцатеричный символ ASCII	41

Преобразование двоичных чисел в десятичные

Довольно часто приходится переводить двоичные числа в соответствующие десятичные. В табл. 3.4 показано соответствие двоичных и десятичных чисел от 2^0 до 2^{15}. Каждая ячейка представляет степень числа 2.

Чтобы перевести двоичное число в десятичное, просуммируйте десятичные эквиваленты всех позиций двоичного числа, в которых находится единица. Пример преобразования числа 00001001 показан на рис 3.3.

Таблица 3.4. Значения разрядов двоичного числа

2^n	*Десятичное значение*	2^n	*Десятичное значение*
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

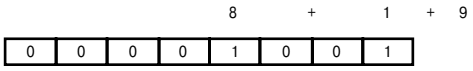

Рис. 3.3. *Преобразование двоичного числа в десятичное*

Шестнадцатеричные числа

Большие двоичные числа почти невозможно прочитать, поэтому используют шестнадцатеричные числа, которые удобно преобразовывать в двоичные числа и которые довольно хорошо воспринимаются при просмотре листингов. Их используют и в языке ассемблера, и в отладчиках для отображения двоичных данных и машинных команд. Каждое шестнадцатеричное число заменяет четыре двоичных бита, а два шестнадцатеричных числа представляют байт.

На рис. 3.4 показано представление двоичного числа `000101100000011110010100` в шестнадцатеричном виде `160794h`.

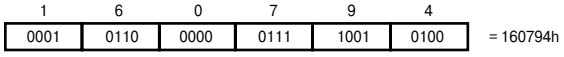

Рис. 3.4. *Соответствие двоичного и шестнадцатеричного чисел*

Одно шестнадцатеричное число может принимать значения от `0` до `15`, поэтому наравне с числами `0–9` для отображения значений от `10` до `15` используют символы от `A` до `F`: `A=10`, `B=11`, `C=12`, `D=13`, `E=14`, `F=15`. В табл. 3.5 показано, как последовательность четырех битов переводится в десятичное и шестнадцатеричное значение.

Таблица 3.5. Двоичные, десятичные и шестнадцатеричные эквиваленты

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

Каждая позиция шестнадцатеричного числа представляет степень числа `16`, что используется при вычислении десятичного значения числа, как показано в табл. 3.6.

Для преобразования шестнадцатеричного значения в десятичное необходимо умножить значение каждого разряда на соответствующий десятичный эквивалент, а потом их просуммировать. На рис. 3.5 приведен пример преобразования числа `3BA4h`. Берется наибольшее значение `3` и умножается на десятичный эквивалент позиции — `4096`. Следующее число `B` умножается на `256`, `A` умножается на `16` и последнее число `4` умножается на `1`. Все просуммировав, получим соответствующее десятичное число `15268`.

Таблица 3.6. Степени числа 16

16^n	Десятичное	16^n	Десятичное
16^0	1	16^4	65 536
16^1	16	16^5	1 048 576
16^2	256	16^6	16 777 216
16^3	4096	16^7	268 435 456

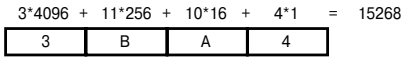

Рис. 3.5. *Преобразование шестнадцатеричного числа 3BA4 в десятичное*

Числа со знаком

Двоичные числа могут быть как со знаком, так и без знака. Числа без знака используют все восемь битов для получения значения (например, `11111111 = 255`). Просуммировав значения всех битов для преобразования в десятичное число, получим максимально возможное значение, которое может хранить байт без знака (`255`). Для слова без знака это значение будет составлять `65535`. Байт со знаком использует только семь битов для получения значения, а старший восьмой бит зарезервирован для знака, при этом `0` соответствует положительному значению, а `1` — отрицательному. На представленном ниже рис. 3.6 показано отображение положительного и отрицательного числа `10`.

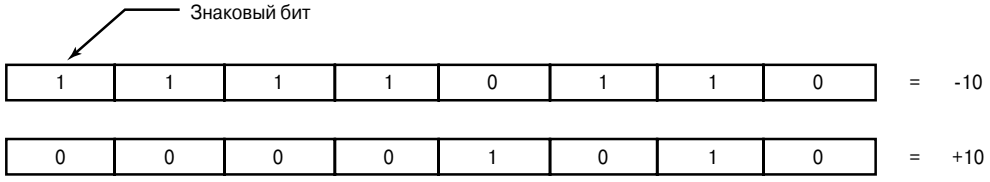

Рис. 3.6. *Отображение положительного и отрицательного числа 10*

Дополнение до двух

Чтобы не усложнять процессор, отдельный блок для реализации операции вычитания не делают; эту операцию выполняет блок суммирования. Перед суммированием отрицательные числа преобразовываются в дополнительные числа. Это такое число, которое в сумме с исходным числом дает `0`. Например, десятичное `-6` будет дополнением к `6`, так как `6+(-6)=0`. Таким образом, вместо операции вычитания `A-B` процессор суммирует с положительным числом `A` дополнительное к `B`: `A+(-B)`. Вместо того чтобы вычесть `4` из `6`, процессор просто складывает `-4` и `6`.

При работе с двоичными числами для дополнительного числа используется термин *дополнение до двух* (встречается также определение *двоичное дополнение*). Например, для двоичного значения `0001` двоичным дополнением до двух будет `1111`. Такое число получается из исходного числа после изменения всех единиц на нули, а нулей на единицы (инверсия) и прибавления к полученному числу единицы, как показано ниже. Инверсия

битов в двоичном числе обозначается NOT(n). Поэтому двоичное дополнение может быть представлено выражением NOT(n)+1.

число	0001
инверсированное число	1110
добавить 1	1111

Если сложить N и дополнение до двух к N, получим 0: 0001+1111=0000. Операция получения дополнения до двух полностью обратима. Например, для отрицательного числа -10 дополнением до двух будет 10.

```
 11110110                = -10
 00001001                инверсия бит
+00000001                добавить 1
 00001010                = +10
```

Еще несколько примеров преобразования чисел приведено в табл. 3.7 (для дополнения до двух используем аббревиатуру NEG(n)).

Таблица 3.7. Преобразование чисел

Десятичное	*Двоичное*	*NEG(n)*	*Десятичное*
+2	00000010	11111110	–2
+16	00010000	11110000	–16
+127	01111111	10000001	–127

Максимальные и минимальные значения

Число со знаком из n разрядов может использовать только n-1 бит для получения значения. Например, знаковый байт использует только семь битов (от 0 до 127). В табл. 3.8 показаны максимальные и минимальные значения для байт, слов, двойных и учетверенных слов со знаком. Наименьшие значения (-128, -32768, -2147483648) являются недопустимыми. Нетрудно убедиться, что двоичное дополнение до -128 (10000000) будет также 10000000.

Таблица 3.8. Целые числа со знаком и без знака

Тип хранения	*Биты*	*Диапазон*
байт со знаком	7	от –128 до +127
слово со знаком	15	от –32 768 до +32 767
двойное слово со знаком	31	от –2 147 483 648 до +2 147 483 647
учетверенное слово со знаком	63	от –9 223 372 036 854 775 808 до +9 223 372 036 854 775 807

Хотя процессор выполняет вычисления без учета знака числа, в программе знак операнда необходимо обязательно указывать. Сложение операндов со значениями +16 и -23 будет выглядеть в командах ассемблера следующим образом.

```
MOV AX,+16
ADD AX,-23
```

В двоичном выражении число 16 будет выглядеть как 00010000, а -23 — как 11101001. Когда процессор складывает эти числа, он получает 11111001. Это двоичное число соответствует десятичному -7, как показано в примере.

```
 00010000                                                16
+11101001                                               -23
=11111001                                                -7
```

Таким образом, сложение чисел со знаком получается корректным. Но в данном случае двоичное число можно интерпретировать и как десятичное число без знака 249. Именно поэтому программист должен отслеживать эти величины и четко представлять, какой тип имеет полученное значение.

Хранение символов

Компьютеры могут хранить только двоичные значения, но нам необходимо работать не только с численными значениями, но и с символами, такими как "A" или "$". Для этого компьютер использует схему кодирования символов, которая позволяет преобразовывать символы в числа и наоборот. Наиболее известная система кодирования для компьютеров обозначается аббревиатурой ASCII (American Standard Code for Information Interchange). В ASCII каждому символу присваивается уникальный код, включая контрольные символы, используемые при печати и передаче данных между компьютерами. Стандартный ASCII-код использует только 7 разрядов в диапазоне 0-127. Значения от 0 до 31 заняты служебными кодами, используемыми при печати, передаче информации и выводе на экран. В обычном режиме они не отображаются на экране. Остальные значения, допустимые в байте, — дополнительные, их применяют для расширения символьного ряда. В операционной системе MS DOS значения 128-255 используются для получения графических символов и греческих букв. В операционной системе Windows существует множество наборов символов, и в каждом из них дополнительным значениям соответствуют различные символы.

Строка символов представляет в памяти последовательность байт. Например, числовым кодам строки "ABC123" будет соответствовать последовательность значений 41h, 42h, 43h, 31n, 32h и 33h.

Таблица кодов ASCII приведена в справочном разделе. Чтобы найти шестнадцатеричное значение нужного символа, используйте соответствующие значения второй строки и второй колонки, на пересечении которых находится символ. Старший разряд числа находится в строке, младший — в колонке. Например, чтобы найти шестнадцатеричное значение символа "a", посмотрим на соответствующее значение в строке. Это будет 6, соответствующее значение в колонке будет 1. Таким образом, получаем шестнадцатеричное значение 61h.

Хранение чисел

Как наиболее эффективно сохранять числа в памяти? Это зависит от того, как эти числа будут использоваться. Если числа используются для вычислений, должно быть применено двоичное представление числа, и наоборот, лучше хранить коды ASCII, если данные значения будут использоваться для отображения символов на экране. Например, число 123 можно сохранить в памяти двумя способами: как последовательность кодов ASCII для чисел 1, 2 и 3 или как один байт со значением 123, как показано на рис. 3.7.

Двоичное содержимое памяти всегда можно просмотреть, но по одному лишь значению нельзя определить, что оно представляет. Предположим, два байта памяти имеют

значения 01000001 и 01000010. Но что это такое? Данные, код или текст? Это невозможно узнать, пока не идентифицирован определенный участок памяти. Программа должна отслеживать состояние данных и тип представления, чтобы избежать конфликтов. Языки высокого уровня не позволят использовать переменные вместо команд, чего нельзя сказать о языке ассемблера. Ограничения языка помогают избежать серьезных ошибок, но в языке ассемблера почти нет ограничений, и программист должен учитывать все, даже незначительные детали.

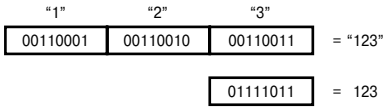

Рис. 3.7. *Хранение строки "123" и числа 123 в памяти*

Основы языка ассемблера

Как уже отмечалось, программы можно писать непосредственно в машинных кодах, используя числовые значения, но на языке ассемблера делать это значительно удобнее. Понятные аббревиатуры команд позволяют легко запомнить их назначение и писать программы, которые затем можно читать и модернизировать. Однако поскольку ассемблер — язык низкого уровня, то для выполнения простейшего математического выражения требуется несколько команд или даже несколько десятков команд. Поэтому обычно почти невозможно анализировать логику программы, используя только мнемонику языка ассемблера.

Команды языка ассемблера

Команды языка ассемблера представляют взаимно однозначное соответствие с машинными кодами. В простейшем варианте они состоят из мнемокода команды с последующими операндами. Все это непосредственно преобразуется в машинные коды. Команды могут либо иметь, либо не иметь операндов, как показано на примере ниже.

```
CLC           ; мнемокод
INC EAX       ; мнемокод с одним операндом
MOV EAX,EBX   ; мнемокод с двумя операндами
```

Любую команду можно сопроводить *комментарием*, отделяя его от команды точкой с запятой "`;`".

Ассемблер является языком низкого уровня, потому что его команды, по сути, машинные, т.е. команды языка ассемблера имеют взаимно однозначное соответствие с машинными кодами. И наоборот, одно утверждение в языке высокого уровня, таком как Delphi или C#, обычно транслируется в несколько машинных кодов.

Операнд может быть регистром, переменной, ячейкой памяти или непосредственным значением, как показано в табл. 3.9.

Таблица 3.9. Представление операндов

Операнд	*Описание*
`10`	(непосредственное значение)
`count`	(переменная)
`EAX`	(регистр)
`[0200]`	(ячейка памяти)

Константы и выражения

Цифровой литерал является комбинацией цифр и дополнительных символов — знаков, десятичных точек и экспонент:

- 5;
- 5,5;
- 26,E+05.

Целочисленные константы могут оканчиваться дополнительным буквенным символом, который является указателем базы системы счисления: h – шестнадцатеричная, q (или o) — восьмеричная, d — десятичная, b — двоичная. Если дополнительного буквенного символа нет, то по умолчанию принимается десятичная система счисления. Буквенный символ может быть строчным или заглавным. В табл. 3.10 приведено несколько примеров целочисленных констант.

Если число не сопровождается дополнительным буквенным символом, то по умолчанию принимается, что для данного числа используется десятичная система счисления.

Таблица 3.10. Представление целочисленных констант

Константа	*Система счисления*
1Ah	шестнадцатеричная
26	десятичная
1101b	двоичная
36q	восьмеричная
2BH	шестнадцатеричная
42Q	восьмеричная
36D	десятичная
47d	десятичная
0F6h	шестнадцатеричная

Хотя дополнительный символ может быть и заглавной буквой, рекомендуется использовать строчные буквы для унификации записи. Если шестнадцатеричная константа начинается с буквы, то перед ней должна ставиться цифра 0.

Константное выражение состоит из комбинации цифровых литералов, операторов и определенных символьных констант. Значение константного выражения определяется во время трансляции программы и не может меняться во время выполнения программы. Ниже приведено несколько примеров константных выражений, включающих только цифровые литералы:

- 5;
- 26,5;
- 4 * 20;
- -3 * 4/6;
- -2,301E+04.

Символические константы являются именами константных выражений.

```
rows = 5
columns = 10
tablePos = rows * columns
```

Обратите внимание на то, что хотя это утверждение и выглядит подобно выражению времени выполнения в языках высокого уровня, но определяется оно во время трансляции.

Символы или *символьные константы* могут быть представлены отдельными символами или строками символов, заключенными в двойные или одинарные кавычки. Внутренние и внешние кавычки должны быть разного типа, как показано в следующих примерах:

- `'ABC'`;
- `'X'`;
- `"This is a test"`;
- `'4096'`;
- `"This isn't a test"`;
- `'Say "hello" to Bill.'`.

При этом необходимо хорошо представлять, что символьная константа "`4096`" занимает в памяти четыре байта, так как каждая цифра определяется в соответствии с кодом ASCII и занимает один байт, о чем уже говорилось ранее. Также не забывайте, что кодировки символов русского алфавита для DOS и Windows в диапазоне 127–255 не совпадают. Поэтому когда вы пишете программу в текстовом редакторе для Windows, а потом выполняете ее с выводом на экран консоли, которая все символы отображает в кодировке для DOS, то буквы русского алфавита будут отображаться неправильно. Поэтому либо используйте английский алфавит, либо делайте перекодировку с помощью специальной процедуры.

Утверждения

В языке ассемблера *утверждение* состоит из имени, мнемокода, операндов и комментариев. Утверждения бывают двух типов: команды и директивы. *Команды* — это утверждения, выполняемые в программе, а *директивы* — утверждения, информирующие ассемблер о том, как создавать выполняемый код. Общая форма утверждения выглядит так:

```
[имя] [мнемокод] [операнды] [; комментарии]
```

Утверждение имеет свободную форму записи. Это означает, что его можно записывать с любой колонки и с произвольным количеством пробелов между операндами. Утверждение должно быть записано на одной строке и не заходить за 128-ю колонку. Можно продолжить запись со следующей строки, но при этом первая строка должна заканчиваться символом "\" (обратная косая черта), как показано в примере ниже.

```
longArrayDefinition DW l000h, 1020h, 1030h \
1040h, 1050h, 1060h, 1070h, 1080h
```

Команда — это утверждение, которое выполняется процессором во время работы программы. Команды могут быть нескольких типов: передачи управления, передачи данных, арифметические, логические и ввода-вывода. Команды транслируются ассемблером непосредственно в машинные коды. Ниже приведен фрагмент листинга со всеми используемыми категориями команд.

```
CALL MySub    ; Передача управления.
MOV EAX,5     ; Передача данных.
ADD EAX,20    ; Арифметическая.
JZ next1      ; Логическая (переход, если установлен флаг нуля).
IN AL,20      ; Ввод-вывод (чтение из аппаратного порта).
```

Директива — это утверждение, которое выполняется ассемблером во время трансляции исходной программы в машинные коды. Например, директива DB заставляет ассемблер выделить память для однобайтовой переменной, названной count, и поместить туда значение 50.

```
count DB 50
```

Следующая директива .STACK заставляет ассемблер зарезервировать пространство памяти для стека.

```
.STACK 4096
```

Имена

Имена определяют метки, переменные, символы или ключевые слова. Они могут состоять из символов, приведенных в табл. 3.11.

Таблица 3.11. Допустимые для имен символы

Символы	*Описание*
A ... Z, a ... z	Буквы
0 ... 9	Цифры
?	Знак вопроса
_	Подчеркивание
@	Знак @
$	Знак доллара

Для имен есть следующие ограничения.

- Максимальное количество символов — 247 (в MASM).
- Заглавные и строчные буквы не различаются.
- Первым символом могут быть @, _ или $. Последующими могут быть эти же символы, буквы или цифры. Избегайте использования в начале имени символа "@", так как многие предопределенные имена начинаются именно с него.
- Выбранные программистом имена не должны совпадать со словами, зарезервированными в языке ассемблера.

Переменные — это данные какой-либо программы, которым присвоены имена.

```
count1 DB 50        ; Переменная (место в памяти размером 1 байт).
```

Метка является именем, которое размещается в пространстве кодов. Метки отмечают те строки программы, на которые необходимо делать переход из других мест. Метка может стоять в начале пустой строки или за ней могут находиться команды. В приведенном ниже фрагменте листинга метки указывают на определенные строки в программе.

```
Label1:  MOV EAX,0
         MOV EBX,0
Label2:
         JMP Label1  ; Переход на Label1.
```

Ключевые слова всегда имеют предопределенный смысл в языке ассемблера. Это могут быть команды или директивы, например MOV, PROC, TITLE, ADD, AX или END. Ключевые

слова не могут использоваться программистом для каких-либо других целей, например как имена. Если использовать ключевое слово ADD как метку, это будет синтаксической ошибкой.

```
ADD: MOV EAX,10  ; Синтаксическая ошибка!
```

Разработка программы на языке ассемблера

Хотя внешне программы, написанные на языке ассемблера, сильно отличаются от программ, созданных на языке высокого уровня, тем не менее технология их разработки одинакова. Однако следует учесть, что разработка программ на языке ассемблера требует большего внимания и аккуратности. При этом необходимо последовательно выполнить следующие этапы.

- Поставить задачу и составить проект программы. На этом этапе нередко составляются блок-схемы — эскиз выполняемых программой действий.
- Ввести команды программы в компьютер с помощью редактора. При сложной логике программы удобно предварительно написать комментарии на обычном языке с описанием предполагаемых действий, а затем вставлять между ними соответствующие команды языка ассемблера. В качестве редактора можно использовать любой текстовый редактор, который создает файлы с расширением `.txt`. Подойдет даже простейший Notepad.
- Откомпилировать программу с помощью ассемблера. Если ассемблер обнаружит ошибки, исправить их в редакторе и откомпилировать программу заново.
- Преобразовать результат работы ассемблера в исполняемый модуль с помощью компоновщика.
- Выполнить программу.
- Проверить результаты. Если они не соответствуют ожидаемым, найти ошибки с помощью отладчика. Данный этап называется отладкой и обычно занимает большую часть времени, затрачиваемого на разработку программы.

Если программа простая и короткая, то ее разработка не займет много времени. Однако сложные программы требуют значительного времени на каждом этапе, поэтому необходимо тщательно планировать процесс разработки уже на этапе проектирования, иначе этап отладки может никогда не закончиться.

Программа, написанная в командах ассемблера, называется *исходной программой*, а ее преобразованный в коды процессора вид именуется *объектной программой*. Таким образом, функцией ассемблера является преобразование исходной программы, доступной восприятию человеком, в объектную программу, понятную процессору.

Операционная система при выполнении может разместить программу в любом подходящем месте памяти и освобождает разработчика от необходимости думать, куда ее поместить. Но чтобы этим воспользоваться, надо преобразовать оттранслированную программу в вид, позволяющий ее перемещение. Такие программы называются *перемещаемыми*. Они создаются с помощью *компоновщика* — программы LINK, которая обязательно входит в комплект поставки ассемблера.

Обычно *объектным модулем* называется файл, содержащий результат трансляции программы ассемблером. А файл, содержащий перемещаемую версию оттранслированной программы, называется *исполняемым модулем*. Таким образом, функцией компоновщика LINK является создание исполняемого модуля из объектного модуля.

Компоновщик необходим также при написании большой программы. Невозможно написать сложную программу как единое целое, поэтому такие программы пишут по частям,

которые потом можно собрать вместе с помощью компоновщика. При этом можно использовать модули, написанные другими программистами, или ранее написанные и отлаженные модули. Если есть набор подходящих модулей, то разработка сложной программы может занять не так уж и много времени. Надо только объединить уже существующие и вновь написанные модули и получить один исполняемый модуль — что и сделает компоновщик.

Компоновщик должен вызываться для любой написанной программы, даже если она состоит только из одного объектного модуля. Одномодульные программы компоновщик сразу преобразует в перемещаемый модуль. Если программа состоит из двух или большего количества модулей, то компоновщик сначала объединяет их, а затем преобразовывает результат в перемещаемый модуль.

Завершенную программу можно вызвать для выполнения двумя способами:

- набрать ее имя в качестве команды или щелкнуть на имени программы мышкой;
- выполнить ее под управлением программы DEBUG.

Обычно программу следует выполнять только в том случае, если есть уверенность в ее безошибочной работе. Пока она не будет полностью отлажена, необходимо вызывать программу только под управлением отладчика DEBUG. Это связано с тем, что непроверенная программа может нанести системе непоправимые повреждения. При программировании под Win32 это маловероятно, но если вы программируете под DOS, то это правило необходимо соблюдать.

С помощью отладчика DEBUG можно управлять процессом выполнения программы. Наряду с другими функциями, DEBUG позволяет отображать и изменять значения переменных, останавливать выполнение программы в заданной точке или выполнять программу по шагам. Таким образом, DEBUG является основным инструментом для поиска и исправления ошибок в программе.

На рис. 3.8 показаны этапы разработки программ с помощью ассемблера. В скобках для каждого модуля указаны расширения файлов, в которых модули сохраняются на диске.

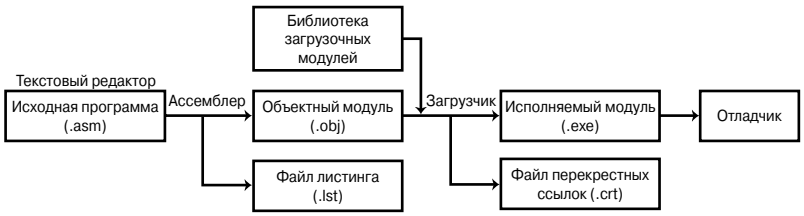

Рис. 3.8. *Этапы разработки программ на ассемблере*

Разработка программы методом “сверху вниз”

При создании программы естественным желанием является начать последовательно вводить команды, реализуя логику программы по мере их ввода. Такой метод может сработать в случае простой или короткой программы, но обычно это приводит к ошибкам или созданию программ, которые трудно понять, а впоследствии еще труднее модифицировать. Благодаря удобным возможностям редактирования текстов, предоставляемым программами обработки текстов и редакторами, можно воспользоваться более легким, удобным и эффективным методом разработки программ. Этот метод разработки называется “сверху вниз”.

Разработка методом “сверху вниз” означает, что вначале создается набросок программы в виде текста на обычном языке, а затем этот набросок постепенно дополняется

деталями. Набросок должен представлять собой ряд строк, в которых описаны действия программы. Например, при разработке программы, которая выполняет одну из нескольких функций по выбору пользователя, набросок может выглядеть следующим образом.

```
;  Изобразить меню возможных функций
;  Запросить у пользователя выбор из меню
;  Прочитать ответ пользователя
;  Проверить допустимость ответа
;  Если ответ допустим, выполнить требуемую функцию
```

Точки с запятой означают, что эти строки представляют собой не команды, а комментарии, которые необходимы только разработчику. Ассемблер пропускает все до конца строки после точки с запятой.

Затем, с учетом этого текста, производится вставка необходимых команд между строками. Так как каждая строка описывает относительно небольшую задачу, проще всего выполнить каждую задачу в отдельности, проверяя решение перед тем, как двинуться дальше. Иначе говоря, начните со вставки первой группы команд (в представленном примере с команд для изображения меню), затем запишите полученную программу на диск и завершите все последующие этапы (трансляцию, компоновку и выполнение). Реализация такой частичной программы покажет, правильно ли она работает. Если она работает неправильно, отладьте ее и попробуйте еще раз. После того как первая часть отлажена, перейдите ко второй части, затем к третьей и т.д.

Может показаться, что это очень медленный метод разработки программы, но только так можно разрабатывать программы, которые содержат мало ошибок и которые впоследствии можно модифицировать. При этом достигаются следующие цели:

- четко просматривается логика программы;
- производится документирование программы с помощью комментариев, впоследствии программу будет легко модифицировать;
- обеспечивается правильность работы каждой части до того, как произойдет переход к разработке следующей части программы.

Работа в DOS под Windows NT

Хотя термины “Windows 3x” и “Windows 9x” связывают с операционными системами, эти графические интерфейсы пользователя являются только оболочкой, сооруженной над DOS. Довольно часто эти системы останавливаются, и связано это с тем, что если одна программа для Windows (или программа DOS, работающая под Windows) дает сбой, то зависает и вся система. Приходится перезагружать компьютер, и, вероятнее всего, часть готовой работы будет потеряна. И все же, благодаря внутренним конструктивным доработкам, в Windows 98 отказов стало меньше, хотя пользователи, работающие с этими системами, ограничены архитектурой DOS, которая не позволяет создать оболочку, устойчивую к неисправностям.

Однако все еще работают сотни миллионов копий DOS и десятки тысяч приложений, созданных для этих систем. Довольно часто программа состоит только из версии DOS, что относится как к большим приложениям, так и к незаметным утилитам. Поэтому технология DOS сохранена и в новейших операционных системах, в которых она используется совсем по-другому. Операционные системы Windows NT (Windows 2000 и Windows XP) совсем не используют DOS. Программа DOS является только отдельной исполняемой программой, как и многие другие. При инициализации Windows система DOS не загружается вообще. Технология NT имеет операционную систему с ядром, которое отвечает за большую часть того, что “умеет” DOS (например, ввод-вывод с помощью клавиатуры

и экрана, работа с загружаемыми драйверами устройств, обработка запросов на ввод-вывод с диска). Windows NT может эмулировать определенные операционные системы с использованием модулей, которые называются подсистемами среды, и DOS — одна из таких операционных систем, работающая как программа под управлением Windows NT. Подсистема среды DOS предоставляет весь системный сервис, как это обычно делает DOS, но эти функции интегрированы в Windows NT, а не расположены отдельно от нее.

Windows NT имеет специальное окно для DOS, которое называется *командная консоль*, и работа в этом окне очень похожа на сеанс MS DOS в Windows 3x или Windows 9x. Интерпретатор команд содержит богатый набор команд, размеры окна можно изменять, окно имеет полосу прокрутки и не ограничено 24 строками, как в DOS.

Эмуляцию DOS обеспечивает 32-разрядное приложение с именем `cmd.exe`, которое является расширенной версией MS DOS и не только обеспечивает совместимость с MS DOS, но и позволяет запускать приложения Windows, OS/2 и POSIX в режиме командной строки.

Сеанс DOS, который создается при запуске приложения DOS из командного окна, можно сконфигурировать с помощью загружаемых драйверов устройств, TSR и т.д.

При запуске программы DOS из Windows NT создается виртуальная машина DOS, благодаря которой программа DOS работает как бы на отдельном компьютере. Windows NT создает отдельную виртуальную машину для каждого запускаемого приложения DOS. Каждая такая машина имеет весь сервис, необходимый для работы как с 16-разрядными, так и с 32-разрядными вызовами DOS в соответствии с требованиями DOS 6. Этой виртуальной машине выделяется 16 Мбайт памяти. При необходимости могут поддерживаться диспетчеры памяти. Есть и ограничения. В целях безопасности и защиты ядра приложения DOS должны быть изолированы. Для этого подсистема среды DOS перехватывает все процессы ввода-вывода, проверяет их, а затем направляет данные по назначению. Этот процесс обслуживается перехватчиками ввода-вывода, которые, в свою очередь, передают данные программе NT Executive для доставки по назначению.

Любые традиционно написанные программы DOS, которые выполняют ввод-вывод с помощью стандартных системных вызовов DOS, будут работать под Windows NT без проблем. А программы, выполняющие запись непосредственно на устройство, для которого драйверы не разрешают прямого доступа (например, драйверы жестких дисков), будут прерваны программой контроля безопасности, что приведет к сообщению об ошибке и завершению опасной программы.

Если программа пытается записывать и читать из портов COM и LPT, с экрана или клавиатуры, то Windows NT выполнит это безупречно, однако другие виды прямого управления памятью, диском или системным устройством разрешены не будут.

Большинство программ DOS, которые используют драйвер мыши или клавиатуры, будут работать, поскольку строки ввода драйвера мыши и клавиатуры эмулируются (их используют очень многие программы DOS).

Операционная система DOS использует командную строку для ввода отдельных команд. Перечень всех доступных команд можно получить, если ввести команду `help`. Список всех команд приведен в табл. 3.13.

Инструментальные средства

Термины “трансляция”, “компоновка”, “отладка” и другие, связанные с этапами работы ассемблера, уже упоминались в этой книге, но о самом ассемблере пока ничего не говорилось. В этом разделе будут даны начальные сведения о наиболее популярных ассемблерах TASM и MASM.

Ассемблер фирмы Borland (TASM)

Ассемблер фирмы Borland, как и любой другой, поставляется с полным набором программ, необходимых для компиляции исходного файла, получения выполняемого файла, компоновки и редактирования. Последняя версия ассемблера Borland Turbo Assembler 5.0 реализует следующие функциональные особенности:

- объектно-ориентированную технику программирования;
- поддержку 32-разрядной модели и кадра стека;
- полную поддержку процессоров Intel 386, Intel 486 и Pentium;
- использование директив упрощенной сегментации;
- поддержку таблиц;
- гибкую систему макросов.

К преимуществам данного компилятора следует также отнести высокую скорость компиляции. Используя данный ассемблер, можно программировать для Windows. Однако поскольку фирма Borland закрыла свое направление для языков C/C++, она отказалась и от TASM как от отдельного продукта. И теперь новые версии TASM приходят только в составе таких продуктов, как Delphi и C++ Builder.

Фирма Borland разработала отличный ассемблер, располагающий возможностями, недоступными в других компиляторах, например объектно-ориентированная техника программирования. Так как TASM удобно использовать при написании программ для операционной системы Windows, которая состоит из сообщений, исключений и классов, то эта возможность оказалась как нельзя кстати.

Ассемблер фирмы Microsoft (MASM)

Последняя версия ассемблера Microsoft Macro Assembler 6.15 реализует следующие возможности:

- поддержку 32-разрядной модели памяти и кадра стека;
- полную поддержку всех процессоров вплоть до Pentium II;
- директивы упрощенной сегментации;
- поддержку таблиц;
- директивы языков верхнего уровня.

В отличие от фирмы Borland, Microsoft поддерживала свой продукт и выпускала его как отдельный пакет, так и в составе таких программных пакетов, как Microsoft Quick C, Microsoft Visual Studio и др.

Использование этого ассемблера при написании программ для Windows будет более привычным для тех, кто начал использовать MASM еще с операционной системой DOS и продолжает его применять с Windows. В этом компиляторе предусмотрено все: от разработки простых оконных приложений для Windows до создания виртуальных драйверов устройств VxD (все подробно описано в документации и файлах помощи). В MASM нельзя применять объектно-ориентированный стиль программирования, но можно использовать дополнительные макроопределения, такие как `.IF`, `.WHILE`, `.REPEAT`, `.CONTINUE`, `.BREAK`.

Еще одно преимущество данного ассемблера — новый пакет MASM32, в котором собрано все, что нужно программисту при написании программ для Win32.

Пример простой программы

В представленном листинге приведена простая программа для Win32, которая отображает на экране традиционное приветствие "`Hello, world!`". В этой программе показаны основные особенности приложений на языке ассемблера. В первой строке использована директива `Title`, остальные символы строки трактуются как комментарий, подобно всем символам во второй строке. Исходный код этой программы написан на языке ассемблера и должен быть оттранслирован в машинные коды перед запуском программы.

Сегменты являются строительными блоками программы. *Сегмент кодов* определяет место, где хранятся коды программы, *сегмент данных* включает все переменные, а *сегмент стека* включает исполнительный стек. Стек — это специальное пространство в памяти, которое обычно используется программой при вызове и возврате подпрограмм.

Программа "Hello World" для Win32

```
TITLE      (.asm)
.386
.model flat, stdcall

ExitProcess PROTO,
  x:dword
WaitMsg PROTO
WriteString PROTO
Crlf PROTO

.data
  strHello BYTE "Hello Word!",0

.code
main PROC
  mov EDX, OFFSET strHello  ; Аргументы для WriteString.
  invoke WriteString        ; Вывод строки на консоль.
  invoke Crlf               ; Перевод каретки.
  invoke WaitMsg            ; Запрос для нажатия клавиши.
  invoke ExitProcess,0      ; Корректное окончание программы.
main ENDP

END main
```

Обратите внимание на структуру программы. Она начинается с директивы `Title`, которая не является обязательной, но всегда лучше ее поставить и кратко описать назначение программы (дополнительная информация в данном случае не помешает). Затем идет директива для задания минимального типа процессора, команды которого можно использовать, и директива модели памяти. При создании программ для Win32 всегда должна выбираться директива

```
.model flat, stdcall
```

Далее следуют объявления используемых в программе процедур. В данном случае это процедуры окончания программы (`ExitProcess`), ожидания нажатия клавиши (`WaitMsg`), вывода строки на консоль (`WriteString`) и перевода каретки (`Crlf`).

Далее в сегменте данных (`.data`) объявляется переменная `strHello`, которая представляет собой строку, выводимую на консоль. Наконец, в сегменте кодов (`.code`) описывается логика программы, которая в данном случае реализует вывод строки на консоль и представляет собой последовательный вызов процедур. Эти процедуры могут быть как

процедурами операционной системы Windows (`ExitProcess`), так и процедурами, написанными самим пользователем и помещенными в библиотеки. Разницы между ними нет никакой (обо всем этом речь пойдет далее). Обратите внимание на обязательный порядок расположения директив и команд. Обязательно должна присутствовать процедура `main`, которая является точкой входа в программу, и заключительные директивы

```
main ENDP
END main
```

Иначе компилятор не сможет найти начало и конец программы и не будет ее обрабатывать. Для компиляции программы необходимо в окне консоли ввести команды, как показано на рис. 3.9.

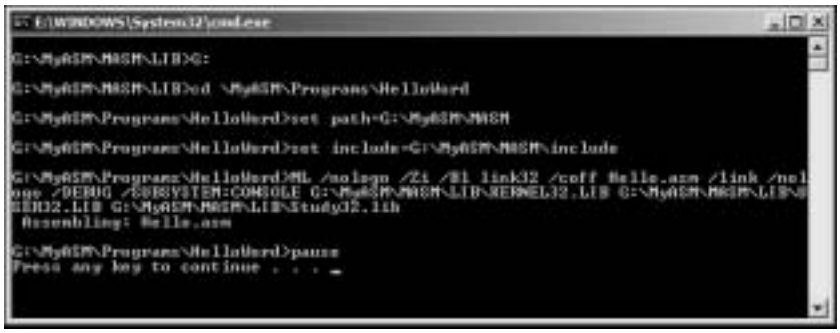

Рис. 3.9. *Компиляция программы*

Это не единственный способ оформления программы, вы можете встретить другие директивы, которые будут обрабатываться компилятором. Но в дальнейшем будем использовать только такой формат — он довольно наглядный и обладает расширенными возможностями. Для сравнения ниже приведена программа для операционной системы DOS.

Программа "Hello World" для DOS

```
TITLE Hello World Program (hello.asm)
; Эта программа отображает слова "Hello, world!"
.MODEL small
.STACK 100h
.DATA
 strHello DB "Hello, world!",0dh,0ah,'$'
.CODE
main PROC
  MOV AX,@data
  MOV DS,AX
  MOV AH,9
  MOV DX,offset message
  INT 21h
  MOV AX,4C00h
  INT 21h
main ENDP
END main
```

Кратко рассмотрим основные строки программы.

- Директива `.MODEL small` сообщает ассемблеру, что в данной программе необходимо использовать не более 64 Кбайт памяти для кодов и не более 64 Кбайт для данных.

Директива .STACK устанавливает размер пространства для стека емкостью 100h (256) байт. Директива .DATA отмечает начало сегмента данных, где сохраняются переменные. Здесь под именем strHello сохраняется строка "Hello, world!", за которой следуют два служебных символа перехода на новую строку (0dh,0ah). Символ "$" используется как символ конца строки для подпрограмм, которые будут считывать эту строку.

- Директива .CODE отмечает начало сегмента кодов, где должны находиться выполняемые команды. Директива PROC объявляет начало процедуры. В этой программе объявлена процедура с именем main.
- Первые две команды процедуры main копируют адрес сегмента данных (@data) в регистр DS. Команда MOV всегда сопровождается двумя операндами: первый указывает, куда поместить данные, а второй — откуда эти данные взять.
- Затем в процедуре main на экран выводится строка символов. При этом вызывается функция DOS, которая непосредственно выводит на экран строку символов, начиная с адреса, который указан в регистре DX. Номер этой функции предварительно должен быть помещен в регистр AH.
- Последние две команды процедуры main (MOV AX,4C00h и INT 21h) заканчивают программу и передают управление операционной системе.
- Утверждение main ENDP использует директиву ENDP, которая отмечает конец главной процедуры.
- В самом конце находится директива END, заканчивающая программу, которая должна быть оттранслирована. Следующая за ней метка main говорит об окончании главной процедуры, или программы. Эта директива необходима для компилятора.

Как видите, отличие есть, и оно заключается в том, что вместо процедур Windows здесь используются прерывания DOS и номера функций, которые выполняют аналогичные действия, но менее удобны в использовании, так как приходится запоминать большое число номеров функций и прерываний и знать, как передавать и получать из них рассчитанные значения.

В табл. 3.12 приведен список наиболее часто используемых директив ассемблера.

Таблица 3.12. Стандартные директивы ассемблера

Директива	*Описание*
END	Окончание трансляции программы
ENDP	Конец процедуры
PAGE	Устанавливает формат листинга
PROC	Начало процедуры
TITLE	Название листинга
.CODE	Отмечает начало сегмента кодов
.DATA	Отмечает начало сегмента данных
.MODEL	Устанавливает режим памяти
.STACK	Устанавливает размер стека

После того как программа "Hello World" написана в текстовом редакторе, ее необходимо сохранить на диске под именем hello.asm. После этого ее можно компилировать.

Анализ программы Hello для Win32

На первый взгляд, программа довольно простая и понятная. Но все это потому, что мы воспользовались библиотекой Study32, которая не входит в стандартную поставку ассемблера и которую нужно написать самому. В библиотеке, которую вы будете писать для себя, необходимо использовать только стандартные процедуры, которые входят в Windows и выглядят несколько сложнее, чем использованные процедуры в программе `Hello` для вывода данных на экран или запроса для нажатия клавиши. Чтобы понять, почему это так, кратко рассмотрим работу операционной системы Windows.

Операционная система Windows

Об этой операционной системе уже говорилось в предыдущих главах, однако следует дополнить общие понятия необходимыми для начинающего программиста сведениями. Система Windows одновременно выполняет множество задач, расчлененных на процессы и потоки, для которых выделяется память и которые независимы друг от друга. Отдельные функциональные программные фрагменты в процессах можно рассматривать как объекты, к которым операционная система обращается с помощью специальных сообщений. Некоторые объекты могут отображаться на экране (часто из называют *элементами управления*), другие такой возможности не имеют, но и те и другие могут получать сообщения Windows и сами отправлять сообщения. Поэтому для того, чтобы, например, вывести в консоли текст, необходимо отправить сообщение нужному элементу управления. Но как определяется каждый объект Windows? Делается это с помощью специальных номеров, или дескрипторов. При создании отдельного объекта операционная система присваивает ему некоторый номер, по которому к этому объекту и можно обратиться.

Поэтому для того чтобы обратиться к консоли, необходимо знать ее дескриптор. Узнать дескриптор консоли можно с помощью процедуры Windows `GetStdHandle`, прототип которой выглядит так:

```
GetStdHandle PROTO,    ; Получить стандартный дескриптор.
  nStdHandle:DWORD     ; Тип устройства (ввод, вывод, ошибки).
```

После того как дескриптор необходимого устройства будет известен, можно использовать также процедуру Windows `WriteConsole`, прототип которой выглядит следующим образом

```
WriteConsole PROTO,                  ; Вывести данные из буфера на консоль.
  handle:DWORD,                      ; Дескриптор выходного устройства.
  lpBuffer:PTR BYTE,                 ; Указатель на буфер.
  nNumberOfCharsToWrite:DWORD,       ; Размер буфера.
  lpNumberOfCharsWritten:PTR DWORD,  ; Число выведенных символов.
  lpReserved:PTR DWORD               ; 0 (зарезервировано)
```

Этот процесс несколько сложнее, чем процедура `WriteString`, которая была использована ранее в программе `Hello`. При вызове процедуры `WriteString` был просто указан адрес буфера (`OFFSET strHello`), т.е. адрес места в памяти, где находится нужная строка, и не указывалось ни длины строки, ни дескриптора выходного устройства; число выведенных символов также не контролировалось.

Понятно, что процедура `WriteString` написана для того, чтобы получать более наглядные и удобные программы, в которых скрыта вся сложность процедур Windows. Можно прийти к выводу, что процедуры Windows не могут быть более простыми, поскольку они должны быть универсальными и рассчитанными для применения в различных ситуациях.

Обратите внимание на тот момент, что в процедуре `WriteString` не указывается длина строки, которая должна быть выведена. Длина строки также рассчитывается в самой

процедуре и признаком окончания строки служит нулевой символ. Именно поэтому при объявлении строки в конце поставлено значение нуль.

```
strHello BYTE "Hello Word!",0
```

Следовательно, для использования процедуры `WriteConsole` предварительно необходимо рассчитать и длину строки.

Полностью программа `Hello`, которая использует только процедуры Windows и не обращается к сторонним библиотекам, будет выглядеть так, как показано в листинге 3.1.

Листинг 3.1. Программа "Hello World" для Win32 с использованием только процедур Windows

```
TITLE      (.asm)
.386
.model flat, stdcall

STD_OUTPUT_HANDLE = -11    ; Предопределенная для Windows константа
STD_INPUT_HANDLE  = -10    ; Предопределенная для Windows константа

Initialize PROTO           ; Объявление процедуры.
WaitMsg PROTO              ; Объявление процедуры.
WriteString PROTO          ; Объявление процедуры.

ExitProcess PROTO,         ; Окончание процесса (Windows).
  x:dword

GetStdHandle PROTO,        ; Получить стандартный дескриптор (Windows).
  STD_OUTPUT_HANDLE:DWORD

WriteConsoleA PROTO,                ; Вывести данные на консоль (Windows).
  handle:DWORD,                     ; Дескриптор выходного устройства.
  lpBuffer:PTR BYTE,                ; Указатель на буфер с данными.
  nNumberOfCharsToWrite:DWORD,      ; Размер буфера.
  lpNumberOfCharsWritten:PTR DWORD, ; Число выведенных символов.
  lpReserved:PTR DWORD              ; 0 (зарезервировано).

ReadConsoleA PROTO,                 ; Считать данные с консолм (Windows).
  handle:DWORD,                     ; Дескриптор входного устройства.
  lpBuffer:PTR BYTE,                ; Указатель на буфер для записи.
  nNumberOfCharsToRead:DWORD,       ; Число символов для ввода.
  lpNumberOfCharsRead:PTR DWORD,    ; Число введенных символов.
  lpReserved:PTR DWORD              ; 0 (зарезервировано).

FlushConsoleInputBuffer PROTO,      ; Очистить буфер консоли (Windows).
  nConsoleHandle:DWORD              ; Дескриптор устройства ввода.

.data
  strHello BYTE "Hello Word!",13,10,0

.code
main PROC
  invoke Initialize
  mov EDX, OFFSET strHello  ; Аргументы для WriteString.
  invoke WriteString        ; Вывод строки на консоль.
  invoke WaitMsg            ; Запрос для нажатия клавиши.
  invoke ExitProcess,0      ; Корректное окончание программы.
main ENDP
```

```
;-----------------------------------------------------
Initialize PROC private
; Получить стандартные дескрипторы консоли для входа и выхода
;-----------------------------------------------------
.data
  consoleOutHandle DWORD ?
  consoleInHandle DWORD ?
.code
  pushad

  INVOKE GetStdHandle, STD_INPUT_HANDLE
  mov [consoleInHandle],eax

  INVOKE GetStdHandle, STD_OUTPUT_HANDLE
  mov [consoleOutHandle],eax

  popad
  ret
Initialize ENDP

;-----------------------------------------------------
Str_length PROC USES edi,
  pString:PTR BYTE       ; Указатель на строку
; Возвращает длину строки с нулевым окончанием.
; Принимает: pString - указатель на строку
; Возвращает: EAX = длина строки
;-----------------------------------------------------
mov edi,pString
mov eax,0       ; Счетчик символов.
L1:
  cmp BYTE PTR [edi],0  ; Конец строки? (сравниваем с нулем)
  je  L2                ; Да: выход. (переход на метку L2)
  inc edi               ; Нет: выбираем следующий символ.
  inc eax               ; Добавляет в счетчик 1.
  jmp L1                ; Переход на метку L1.
L2: ret                 ; Возврат из процедуры.
Str_length ENDP

;-----------------------------------------------------
WriteString PROC
; Записывает строку с нулевым окончанием в стандартный выход.
; Принимает: EDX указывает на строку.
;-----------------------------------------------------
pushad                  ; Сохраняем регистры.
INVOKE Str_length,edx   ; Возвращает длину строки в EAX.
cld                     ; Необходимо выполнить до WriteConsole.
INVOKE WriteConsoleA,
    consoleOutHandle,   ; Дескриптор выхода.
    edx,                ; Указывает на строку.
    eax,                ; Длина строки.
    OFFSET strHello,    ; Число записанных байт.
    0
popad                   ; Восстанавливаем регистры.
ret
WriteString ENDP

;-----------------------------------------------------
```

```
WaitMsg PROC
; Отображает запрос и ожидает нажатия клавиши <Enter>.
;-----------------------------------------------------
.data
waitmsgstr DB "Press [Enter] to continue...",0
localBuf  BYTE 5 DUP(?)
bytesRead DWORD ?
.code
  pushad              ; Сохраняем регистры.
  mov  edx,OFFSET waitmsgstr
  call WriteString
w1:  INVOKE FlushConsoleInputBuffer,consoleInHandle
  INVOKE ReadConsoleA,
    consoleInHandle,  ; Дескриптор устройства ввода.
    OFFSET localBuf,  ; Указатель на буфер.
    5,                ; Размер буфера.
    OFFSET bytesRead,
    0
  cmp bytesRead,2     ; Сравнение с 2.
  jnz w1              ; Повторение, пока не считано 2 байта

  popad               ; Восстанавливаем регистры.
  ret
WaitMsg ENDP

END main
```

Итак, основная часть программы, которая находится в сегменте данных и кодов, почти не изменилась. В сегменте данных несколько изменилось объявление строки `strHello`, где в конце добавлены два байта со значениями 13 и 10. Эти байты будут восприниматься устройством вывода на консоль как перевод каретки, что аналогично использованию в первом случае процедуры `Crlf`. Это не принципиально и сделано для показа различных возможностей ассемблера.

Также не принципиально и появление новой процедуры `Initialize`, которая получает и присваивает значения дескрипторов соответствующим переменным, используемым при инициализации устройств ввода-вывода. Эта процедура использовалась и ранее, только это было не так явно.

Теперь для компиляции этой программы необходимо использовать команды, как показано на рис. 3.10

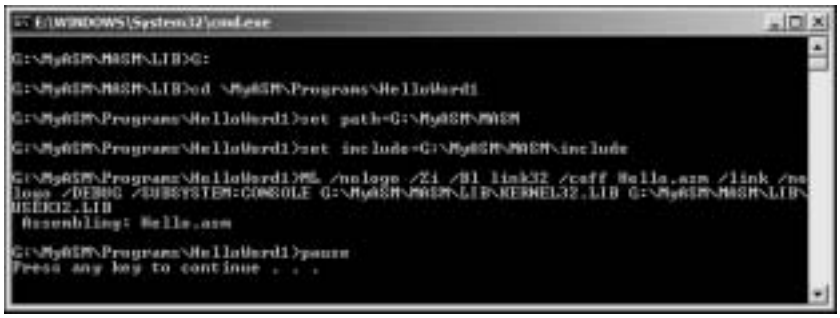

Рис. 3.10. *Компиляция программы, в которой использованы только процедуры Windows*

Давайте последовательно пройдем шаги, которые необходимы для того, чтобы вернуть эту программу в исходное состояние, т.е. представим ее в удобном для программиста и тех, кто ее анализирует, виде. Для этого существуют такие возможности, как создание библиотек и заголовочных файлов, в которые и будет вынесено все, что не используется при описании логики программы, но что необходимо для того, чтобы компилятор мог реализовать эту логику.

Сначала создадим собственную библиотеку с именем `MyLib` и вынесем в нее все, что не составляет суть программы. Это будут все объявления и процедуры, которые описаны после конца процедуры `main`: `Initialize`, `WriteString`, `WaitMsg` и `Str_length`.

Создадим отдельный файл и перенесем в него все эти объявления и процедуры. Этот файл может выглядеть так

```
TITLE Библиотека необходимых процедур  (MyLib.asm)
.386
.model flat, stdcall

STD_OUTPUT_HANDLE = -11    ; Константа Windows для устройства вывода.
STD_INPUT_HANDLE  = -10    ; Константа Windows для устройства ввода.

; Ниже идут объявления процедур Windows.

ExitProcess PROTO,                  ; Окончание процесса (Windows).
  x:dword

GetStdHandle PROTO,                 ; Получить стандартный дескриптор
                                      (Windows).
  STD_OUTPUT_HANDLE:DWORD

WriteConsoleA PROTO,                ; Вывести данные на консоль (Windows).
  handle:DWORD,                     ; Дескриптор выходного устройства.
  lpBuffer:PTR BYTE,                ; Указатель на буфер с данными.
  nNumberOfCharsToWrite:DWORD,      ; Размер буфера.
  lpNumberOfCharsWritten:PTR DWORD, ; Число выведенных символов.
  lpReserved:PTR DWORD              ; 0 (зарезервировано).

ReadConsoleA PROTO,                 ; Считать данные с консоли (Windows).
  handle:DWORD,                     ; Дескриптор входного устройства.
  lpBuffer:PTR BYTE,                ; Указатель на буфер для записи.
  nNumberOfCharsToRead:DWORD,       ; Число символов для ввода.
  lpNumberOfCharsRead:PTR DWORD,    ; Число введенных символов.
  lpReserved:PTR DWORD              ; 0 (зарезервировано).

FlushConsoleInputBuffer PROTO,      ; Очистить буфер консоли (Windows).
  nConsoleHandle:DWORD              ; Дескриптор устройства ввода.

; Конец объявлений процедур Windows.

.code
;-----------------------------------------------------
Initialize PROC
; Получить стандартные дескрипторы консоли для входа и выхода.
;-----------------------------------------------------
.data
  consoleOutHandle DWORD ?  ; Дескриптор устройства вывода.
  consoleInHandle DWORD ?   ; Дескриптор устройства ввода.
.code
```

```
  pushad

  INVOKE GetStdHandle, STD_INPUT_HANDLE
  mov [consoleInHandle],eax

  INVOKE GetStdHandle, STD_OUTPUT_HANDLE
  mov [consoleOutHandle],eax

  popad
  ret
Initialize ENDP

;-----------------------------------------------------
Str_length PROC USES edi,
  pString:PTR BYTE       ; Указатель на строку
; Возвращает длину строки с нулевым окончанием.
; Принимает: pString - указатель на строку
; Возвращает: EAX = длина строки
;-----------------------------------------------------
mov edi,pString
mov eax,0       ; Счетчик символов.
L1:
  cmp BYTE PTR [edi],0  ; Конец строки? (сравниваем с нулем)
  je  L2                ; Да: выход. (переход на метку L2)
  inc edi               ; Нет: выбираем следующий символ.
  inc eax               ; Добавляет в счетчик 1.
  jmp L1                ; Переход на метку L1.
L2: ret                 ; Возврат из процедуры.
Str_length ENDP

;-----------------------------------------------------
WriteString PROC
; Записывает строку с нулевым окончанием в стандартный выход.
; Принимает: EDX указывает на строку.
;-----------------------------------------------------
.data
wsBuf DWORD ?
.code
pushad                  ; Сохраняем регистры.
INVOKE Str_length,edx   ; Возвращает длину строки в EAX.
cld                     ; Выполнить до WriteConsole.
INVOKE WriteConsoleA,
    consoleOutHandle,   ; Дескриптор выхода.
    edx,                ; Указывает на строку.
    eax,                ; Длина строки.
    OFFSET wsBuf,       ; Число записанных байт.
    0
popad                   ; Восстанавливаем регистры.
ret                     ; Возврат из процедуры.
WriteString ENDP

;-----------------------------------------------------
WaitMsg PROC
; Отображает запрос и ожидает нажатия клавиши <Enter>.
;-----------------------------------------------------
.data
waitmsgstr DB "Press <Enter> to continue...",0
localBuf  BYTE 5 DUP(?) ; Буфер.
```

```
bytesRead DWORD ?       ; Число считанных символов.
.code
  pushad              ; Сохраняем регистры.
  mov  edx,OFFSET waitmsgstr
  call WriteString
w1:  INVOKE FlushConsoleInputBuffer,consoleInHandle
  INVOKE ReadConsoleA,
    consoleInHandle,  ; Дескриптор устройства ввода.
    OFFSET localBuf,  ; Указатель на буфер.
    5,                ; Размер буфера.
    OFFSET bytesRead, ; Указатель на переменную.
    0
  cmp bytesRead,2     ; Сравниваем число введенных байт с 2.
  jnz w1              ; Повторение, пока не считано 2 байта
  popad               ; Восстанавливаем регистры.
  ret                 ; Возврат из процедуры.
WaitMsg ENDP
END
```

Откомпилируем его с помощью команд, как показано на рис. 3.11.

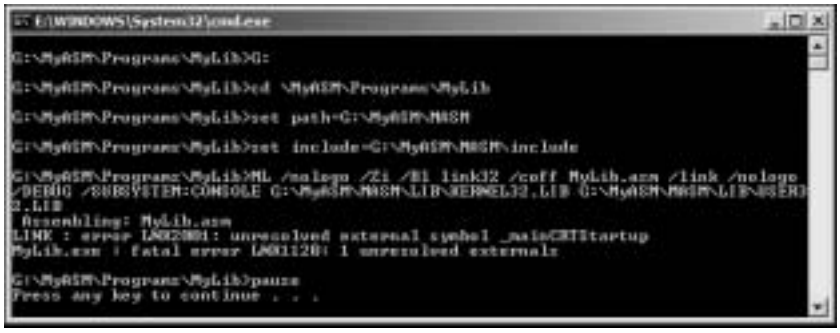

Рис. 3.11. *Компиляция библиотеки*

Как видно из рисунка, после строки

```
Assembling: MyLib.asm
```

появилось сообщение об ошибке

```
LINK : error LNK2001: unresolved external symbol _mainCRTStartup
MyLib.exe : fatal error LNK1120: 1 unresolved externals
```

Не обращайте на него внимания — это сообщение говорит о том, что компоновщик не может обработать объектный модуль по той причине, что не нашел начала программы, т.е. процедуры `main`. Но ее и не должно быть в библиотеке, библиотека только компилируется, а не компонуется. Поэтому можно вызвать программу `ML` с опцией `/c` (компилировать без компоновки), тогда сообщение об ошибке не появится.

Итак, после успешной компиляции получен объектный модуль `Mylib.obj`. Это и будет необходимой библиотекой. Используйте ее при компиляции программы, состоящей только из необходимых строк

```
TITLE      (.asm)
.386
.model flat, stdcall

Initialize PROTO
```

```
ExitProcess PROTO,
  x:DWORD
WaitMsg PROTO
WriteString PROTO

.data
strHello BYTE "Hello Word!",13,10,0

.code
main PROC
  invoke Initialize
  mov EDX, OFFSET strHello  ; Аргументы для WriteString.
  invoke WriteString        ; Вывод строки на консоль.
  invoke WaitMsg            ; Запрос для нажатия клавиши.
  invoke ExitProcess,0      ; Корректное окончание программы.
main ENDP
END main
```

При компиляции будем использовать вновь созданную библиотеку `MyLib.obj`, т.е. будем использовать команды, как показано на рис. 3.12.

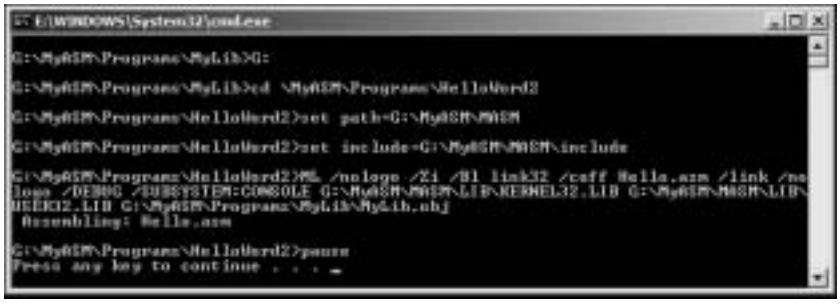

Рис. 3.12. *Компиляция с использованием собственной библиотеки*

Здесь также можно наблюдать некоторое отличие от первой программы `Hello`, что наглядно демонстрирует гибкость ассемблера — вы можете так настроить программу, что она будет полностью отвечать вашим запросам.

Например, в библиотеке необходимо четко разделить процедуры и объявления. Все объявления следует вынести в отдельный заголовочный файл, который постоянно будет использоваться с различными библиотеками, а в самой библиотеке оставить только описания процедур, для чего она и предназначена. Обо всем этом и пойдет речь в дальнейшем.

Ассемблер Microsoft

Как уже не раз отмечалось, операционная система DOS сохранена для работы с множеством все еще существующих программ, написанных под DOS. Ассемблер Microsoft — одна из таких программ. Для этой программы нет соответствующего оконного интерфейса под Windows, и приходится работать в менее удобном консольном окне. Поэтому для начала работы с ассемблером необходимо запустить интерпретатор команд DOS, для чего выбирается команда Start⇨All Programs⇨Accessories⇨Command Prompt в системном меню. Затем необходимо установить пути для удобного запуска программ ассемблера. Для этого существует команда PATH из набора команд DOS. Рассмотрим подробнее работу в окне DOS.

Работа с DOS

Для работы с DOS необходимо запустить интерпретатор команд DOS. Для операционной системы Windows XP это делается так, как описано в предыдущем разделе. Получим окно, показанное на рис. 3.13.

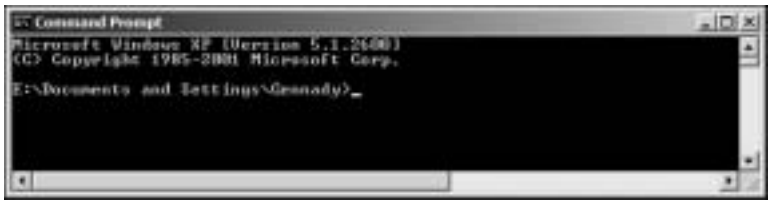

Рис. 3.13. *Окно интерпретатора команд DOS*

Теперь необходимо перейти в каталог, где находится программа интерпретатора команд DOS — `cmd.exe`. Это каталог `..\WINDOWS\system32`. Здесь не указан том, в котором находится этот каталог, так как он жестко не регламентируется и может отличаться. Для того чтобы сделать этот каталог активным, необходимо использовать команду: `cd путь_к_каталогу`.

После этого можно ввести команду `help` — появится перечень всех доступных команд, который приведен в табл. 3.13.

Таблица 3.13. Перечень команд DOS для Windows XP

Команда	*Описание*
ASSOC	Отображает или модифицирует связанные с программами расширения
AT	Вызывает таблицу команд и программ для запуска на компьютере
ATTRIB	Отображает или изменяет атрибуты файла
BREAK	Устанавливает или сбрасывает проверку комбинации клавиш
CACLS	Отображает или модифицирует список контроля доступа файлов (ACL)
CALL	Вызывает одну командную программу из другой
CD	Отображает или изменяет текущий каталог
CHCP	Отображает или устанавливает номер активной кодовой страницы
CHDIR	Отображает или изменяет текущий каталог
CHKDSK	Проверяет диск и отображает отчет
CHKNTFS	Отображает или модифицирует проверку диска во время компоновки
CLS	Очищает экран
CMD	Запускает новый экземпляр интерпретатора команд Windows
COLOR	Устанавливает цвета символов и фона
COMP	Сравнивает содержимое двух файлов
COMPACT	Отображает или изменяет сжатие файлов на разделах NTFS
CONVERT	Конвертирует тома FAT в NTFS. Нельзя конвертировать текущий том
COPY	Копирует один или несколько файлов
DATE	Отображает или устанавливает дату
DEL	Стирает один или несколько файлов

Продолжение табл. 3.13

Команда	Описание
DIR	Отображает список файлов и подкаталогов
DISKCOMP	Сравнивает содержимое двух флоппи-дисков
DISKCOPY	Копирует содержимое флоппи-диска на другой диск
DOSKEY	Редактирует командную строку, повторяет команды и создает макросы
ECHO	Отображает сообщения или выключает и включает отображение вводимых символов на экране (эхо)
ENDLOCAL	Завершает изменения локализации Windows в командных файлах
ERASE	Удаляет один или несколько файлов
EXIT	Закрывает интерпретатор команд
FC	Сравнивает содержимое нескольких файлов и отображает различие между ними
FIND	Выполняет поиск текстовой строки в одном или нескольких файлах
FINDSTR	Осуществляет поиск строк в файлах
FOR	Запускает указанную команду для каждого файла из набора нескольких файлов
FORMAT	Выполняет форматирование диска для работы с Windows
FTYPE	Осуществляет отображение или модификацию типов файлов, используемых в перечне соответствий типов и расширений
GOTO	Выполняет переход к помеченной строке в командных файлах
GRAFTABL	Разрешает системе Windows отображать расширенный набор символов в графическом режиме
HELP	Используется для получения справочной информации
IF	Используется для создания условного утверждения в командных файлах
LABEL	Создает, изменяет или удаляет метки томов на диске
MD	Создает каталог
MKDIR	Создает каталог
MODE	Конфигурирует системные устройства
MORE	Отображает часть выходных данных на экран за один раз (чтобы получить продолжение, необходимо еще раз выполнить эту команду)
MOVE	Перемещает один или несколько файлов из одного каталога в другой
PATH	Отображает или устанавливает путь поиска файлов для операционной системы
PAUSE	Приостанавливает обработку команд в командных файлах и отображает сообщение
POPD	Восстанавливает предыдущее значение текущего каталога, сохраненного командой PUSHD
PRINT	Распечатывает текстовый файл
PROMPT	Изменяет вид запроса
PUSHD	Сохраняет текущий каталог перед его изменением
RD	Удаляет каталог
RECOVER	Восстанавливает доступную информацию с запорченных дисков
REM	Отмечает комментарии в командных файлах или файле CONFIG.SYS
REN	Переименовывает один или несколько файлов
RENAME	Переименовывает один или несколько файлов

Окончание табл. 3.13

Команда	Описание
REPLACE	Заменяет файлы
RMDIR	Удаляет каталог
SET	Отображает, устанавливает или удаляет переменные окружения Windows
SETLOCAL	Начинает локализацию окружения Windows в командных файлах
SHIFT	Сдвигает позицию заменяемых параметров в командных файлах
SORT	Выполняет сортировку
START	Запускает указанную команду или программу в отдельном окне
SUBST	Создает виртуальный диск
TIME	Отображает или устанавливает системное время
TITLE	Устанавливает имя окна для интерпретатора команд
TREE	Графически отображает структуру каталогов
TYPE	Отображает содержимое текстового файла
VER	Отображает версию Windows
VERIFY	Устанавливает режим проверки файлов на корректность записи на диск
VOL	Отображает метку тома и серийный номер
XCOPY	Копирует файлы и каталоги

Как видите, команд операционной системы DOS не так уж и много и запомнить их не составляет особого труда. Но даже этого делать не надо (те несколько команд, которые необходимы для запуска и работы с ассемблером, будут рассмотрены подробно далее, а аналоги всех команд есть в системе Windows, так как режим DOS просто эмулируется системой Windows и в конечном итоге производится обращение к процедурам Windows). В Windows 98 и в более ранних версиях можно работать и непосредственно в DOS, так как эти системы являются только надстройками над DOS, но начиная с версии Windows NT все обстоит по-другому. Здесь системы DOS в чистом виде уже нет и вся работа производится только через процедуры Windows.

Чтобы получить более подробное описание каждой команды, необходимо набрать в командной строке имя команды и дополнить ее символами "`/?`". Например, если набрать `cd /?`, то получим следующую справку.

```
Отображает или изменяет имя текущего каталога.
CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..]
```

Две точки (`..`) указывают, что необходимо перейти к каталогу, включающему данный каталог.

Если набрать CD без параметров, то отобразится обозначение текущего тома и каталог.

Если набрать CD с обозначением тома, то отобразится текущий каталог.

Параметр `/D` используется для переключения текущего тома на указанный том с необходимым каталогом.

Команда CHDIR используется в тех случаях, когда имена файлов или каталогов содержат пробелы.

А вот командные файлы, которые существовали еще в первых системах DOS, могут принести пользу даже при работе в системе Windows. Рассмотрим подробнее командные файлы.

Командный файл

Как уже отмечалось, в среде DOS работать довольно неудобно, так как приходится вручную набирать довольно длинные команды и пути файлов, что часто приводит к ошибкам. Командные файлы созданы для облегчения работы в среде DOS, но и в среде Windows они тоже приносят довольно ощутимую пользу.

Командный файл — это обычный текстовый файл с последовательностью команд, которые должна выполнить система. Командный файл можно написать в обычном текстовом редакторе и сохранить с расширением `.bat`. Файлы с таким расширением операционная система трактует как командные и вызывает для их выполнения интерпретатор команд, а не текстовый редактор.

В командный файл можно включать все команды DOS, а также команды условного перехода `for`, `goto` и `if`, которые позволяют реализовывать различную последовательность выполнения команд в зависимости от наличия определенных условий. Еще несколько команд позволяют контролировать ввод-вывод и вызывать другие командные файлы.

Контролировать процесс выполнения можно по возвращаемым приложениями кодам ошибок, которые могут быть равны `0` (ошибок нет) или `1` (большие значения при наличии ошибок).

Командный файл может иметь параметры, которые дописываются в командный файл при его запуске на выполнение. Для подстановки параметров в команды используются переменные от `%0` до `%9`. Если используется переменная `%0`, то вместо ее при запуске подставляется имя командного файла, а переменные от `%1` до `%9` заменяются соответствующими аргументами. Для доступа к аргументу за пределами `%9` используется команда `shift`. Переменная `%*` ссылается на все аргументы, за исключением `%0`.

Например, для копирования содержимого каталога `Folder1` в каталог `Folder2` можно создать командный файл `Mybatch.bat`, содержимое которого будет представлять одну строку

```
xcopy %1\*.* %2
```

и при вызове этого файла как

```
Mybatch.bat C:\folder1 D:\folder2
```

вместо переменной `%1` будет подставлен каталог `Folder1`, а вместо переменной `%2` — каталог `Folder2`.

Тот же самый результат можно получить, если в среде DOS выполнить команду

```
xcopy C:\folder1\*.* D:\folder2
```

Дополнительно с параметрами командного файла можно применять модификаторы. Модификаторы используют информацию о текущем диске и каталоге для расширения параметров. При использовании модификаторов сначала поставьте символ процента (%), затем тильду (~), а за ней — требуемый модификатор.

Все возможные модификаторы перечислены в табл. 3.14.

Таблица 3.14. Модификаторы для командных файлов

Модификатор	*Описание*
%~1	Удаляет все фрагменты, ограниченные кавычками ("")
%~f1	Дополняет аргумент полным путем
%~d1	Дополняет аргумент буквенным символом текущего диска
%~p1	Дополняет аргумент путем
%~n1	Дополняет аргумент именем файла
%~x1	Дополняет аргумент расширением файла
%~s1	Использует только короткие имена
%~a1	Дополняет аргумент атрибутами файла
%~t1	Дополняет аргумент датой и временем создания файла
%~z1	Дополняет аргумент размером файла
%~$PATH:1	Ищет каталоги, перечисленные в переменной окружения PATH и дополняет аргумент полным путем первого найденного каталога

Подготовка к запуску ассемблера

Теперь, когда вы познакомились с командами DOS и командными файлами, сделаем так, чтобы было удобно запускать файлы на компиляцию, компоновку и отладку.

Для начала нужно обязательно указать пути, по которым операционная система будет находить каталог с установленными модулями ассемблера. Например, если ассемблер находится в каталоге `E:\MASM615\`, то необходимо выполнить следующую команду:

```
path E:\MASM615;%path%
```

После выполнения этой команды операционная система уже будет знать, в каких каталогах искать исполняемый файл, если не указан полный путь, а введено только имя файла. Здесь используется модификатор `%path%` для сохранения всех ранее введенных каталогов.

А для того чтобы сделать активным каталог с вашими программами, необходимо выполнить команду `cd`. Например:

```
cd /d H:\Gennady\Work\Assembl\Book1\Programs
```

Чтобы не набирать текст каждый раз, создайте командный файл и включите в него эти строки, после чего нужно будет только запустить командный файл, и все выполнится автоматически.

Теперь можно откомпилировать разработанную ранее программу `Hello`, которая должна находиться в активном каталоге. Для этого необходимо вызвать компилятор. Командная строка для вызова компилятора имеет следующий синтаксис:

```
ML [ /options ] filelist [ /link linkoptions ]
```

Здесь `options` — это опции, или дополнительные элементы настройки; `filelist` — имя файла; `linkoptions` — опции компоновщика. Так как имена заключены в квадратные скобки, то их указывать не обязательно. Если не установить соответствующие опции, эти файлы все равно будут созданы, но с именем исходного файла. Все допустимые опции перечислены в табл. 3.15.

Таблица 3.15. Опции компилятора MASM

Опция	*Описание*
/AT	Разрешена тонкая модель памяти (.COM)
/Bl<linker>	Использовать альтернативный компоновщик
/c	Компилировать без компоновки
/Cp	Сохранять регистры пользовательских идентификаторов
/Cu	Все идентификаторы в верхнем регистре
/Cx	Сохранять регистры для открытых и внешних идентификаторов
/coff	Генерировать объектный файл в формате COFF
/D<name>[=text]	Задать макроопределение
/EP	Вывести листинг препроцессора
/F <hex>	Задать размер стека в байтах
/Fe<file>	Ввести имя исполняемого файла
/Fl[file]	Генерировать листинг
/Fm[file]	Генерировать карту распределений
/Fo<file>	Ввести имя объектного файла
/Fpi	Эмулировать код 80×87
/Fr[file]	Генерировать ограниченную обзорную информацию
/FR[file]	Генерировать полную обзорную информацию
/G<c|d|z>	Использовать вызовы Pascal, C или Stdcall (по умолчанию Stdcall)
/H<number>	Установить максимальную длину внешних имен
/I<name>	Добавить пути для подключаемых файлов
/link	Задать <Опции компоновщика и библиотеки>
/nologo	Не выводить авторские права
/omf	Генерировать объектный файл в формате OMF
/Sa	Задать максимальный размер листинга
/Sc	Генерировать временные метки
/Sf	Генерировать листинг первого прохода
/Sl<width>	Задать длину строки
/Sn	Не выводить таблицу перекрестных ссылок
/Sp<length>	Задать длину страницы
/Ss<string>	Задать подзаголовок
/St<string>	Задать заголовок
/Sx	Выводить ошибочные условия
/Ta<file>	Компилировать файл с не .ASM-расширением
/w	То же самое, что и /W0 /WX
/WX	Трактовать предупреждения как ошибки
/W<number>	Задать уровень предупреждений
/X	Игнорировать путь к INCLUDE
/Zd	Добавить номера строк в отладочную информацию

Окончание табл. 3.15

Опция	Описание
/Zf	Сделать все идентификаторы открытыми
/Zi	Добавить символическую отладочную информацию
/Zm	Обеспечить совместимость с MASM 5.10
/Zp[n]	Выравнивать структуры
/Zs	Проверить синтаксис без создания выходного файла

Теперь можно давать команду на трансляцию рассмотренного выше файла Hello, который находится в каталоге \Work\HelloWord, в следующем виде:

```
G:\Work\HelloWord>ML /nologo /B link32 /coff Hello.asm /link /nologo /
SUBSYSTEM:CONSOLE F:\MASM615\LIB\KERNEL32.LIB F:\MASM615\LIB\USER32.LIB F:\
MASM615\LIB\Study32.lib
```

А это значит, что вы указываете транслятору не выводить информацию об авторских правах (/nologo), использовать 32-разрядный компоновщик (/Bl link32), создать объектный модуль формата coff (/coff) и использовать консольное окно (/SUBSYSTEM:CONSOLE). При компоновке необходимо использовать следующие библиотеки: F:\MASM615\LIB\KERNEL32.LIB, F:\MASM615\LIB\USER32.LIB, F:\MASM615\LIB\Study32.lib

При трансляции программ для Windows следует использовать стандартные библиотеки Windows: KERNEL32.LIB и USER32.LIB, в которых находятся все необходимые процедуры и которые входят в поставку ассемблера. В данном случае использована также библиотека Study32.lib, в которой находятся процедуры, написанные пользователем. (Об этом уже говорилось ранее и еще будет говориться в дальнейшем, когда вы будете разрабатывать собственную библиотеку.)

При успешной трансляции на экране появится следующее сообщение:

```
Assembling: Hello.asm
G:\Work\HelloWord>
```

После трансляции в указанном каталоге появятся файлы с расширением .obj и .lst (при заданной опции). Полностью файл листинга (.lst) приведен ниже.

```
Microsoft (R) Macro Assembler Version 6.15.8803   08/15/06 16:28:02
  (.asm)     Page 1 - 1

TITLE      (.asm)
.386
.model flat, stdcall

ExitProcess PROTO,
  x:dword
WaitMsg PROTO
WriteString PROTO
Crlf PROTO

00000000                .data
00000000 48 65 6C 6C 6F  strHello BYTE "Hello Word!",0
         20 57 6F 72 64
         21 00

00000000                .code
```

```
00000000                main PROC

00000000 BA 00000000 R
     mov EDX, OFFSET strHello ; Аргументы для WriteString.
     invoke WriteString       ; Вывод строки на консоль.
     invoke Crlf              ; Перевод каретки.
     invoke WaitMsg           ; Запрос для нажатия клавиши.
     invoke ExitProcess,0     ; Корректное окончание программы.
0000001B  main ENDP

          END main
Microsoft (R) Macro Assembler Version 6.15.8803  08/15/06 16:28:02
  (.asm)     Symbols 2 - 1
Segments and Groups:
N a m e     Size     Length   Align   Combine  Class FLAT  GROUP
_DATA  . . 32 Bit   0000000C  DWord   Public              'DATA'
_TEXT  . . 32 Bit   0000001B  DWord   Public              'CODE'

Procedures,  parameters and locals:
N a m e      Type    Value      Attr
Crlf . . . .P Near  00000000 FLAT Length= 00000000 External STDCALL
ExitProcess P Near  00000000 FLAT Length= 00000000 External STDCALL
WaitMsg . . P Near  00000000 FLAT Length= 00000000 External STDCALL
WriteString P Near  00000000 FLAT Length= 00000000 External STDCALL
main . . .  P Near  00000000 _TEXT Length= 0000001B Public STDCALL

Symbols:

N a m e                Type    Value      Attr

@CodeSize  . . . . . ..Number  00000000h
@DataSize  . . . . . . Number  00000000h
@Interface . . . . . . Number  00000003h
@Model . . . . . . . . Number  00000007h
@code  . . . . . . . . Text             _TEXT
@data  . . . . . . . . Text              FLAT
@fardata?  . . . . . . Text              FLAT
@fardata . . . . . . . Text              FLAT
@stack . . . . . . . . Text              FLAT
strHello . . . . . . . Byte    00000000 _DATA

   0 Warnings
   0 Errors
```

В этом листинге приведена вся информация о созданной программе, начиная с исходных команд и их машинных эквивалентов и заканчивая распределением имен, памяти и сегментов. При профессиональном программировании приходится довольно часто обращаться к файлам листингов, но на первом этапе изучения языка в этом большой необходимости нет.

Синтаксические ошибки

Очень немного найдется программистов, которые сразу смогут написать даже небольшую программу без ошибок. Поэтому ассемблер проверяет написанные команды и выводит на экран все строки, в которых есть ошибки, причем с их объяснением. Например, если в программе `Hello` первую команду `MOV` ошибочно набрать как `MIV`, то появится следующее сообщение:

```
Assembling: Hello.asm
Hello.asm(17) : error A2008: syntax error : edx
```

Иными словами, компилятор зафиксирует синтаксическую ошибку перед операндом `edx` в строке номер 17.

Компоновка программы

После компиляции будет получен объектный файл с расширением `.obj`, который необходимо преобразовать в исполняемый модуль. На данном этапе используется программа-компоновщик, которая использует объектный файл (в данном случае `hello.obj`) в качестве входного и создает выполняемый файл, называя его `hello.exe`. Командная строка для компоновщика имеет следующий формат:

```
LINK [options] [files] [@commandfile]
```

Здесь `link` — имя программы компоновщика, `files` — файлы библиотек.

Не будем рассматривать все опции компоновщика, так как на первом этапе они не пригодятся. Необходимо знать только опцию `/DEBUG` (она заставляет компоновщик вставлять в создаваемую программу нужные для отладчика коды) и опцию `/SUBSYSTEM:CONSOLE` (приводит к созданию формата файла, который "понимает" Windows и рассчитан на работу в консольном режиме). Чтобы просмотреть все опции 32-разрядного отладчика, просто наберите в командном окне имя отладчика `link32`. Также можно использовать опцию `/MAP`, которая приводит к созданию файла распределения памяти с расширением `.map`. Попробуйте создать этот файл и проанализируйте его. От его использования тоже может быть ощутимая польза.

После работы компоновщика будет создан исполняемый файл с расширением `.exe`, который уже можно запускать на выполнение как любую программу Windows.

Использование программы `ML` позволяет выполнять компиляцию и компоновку одной командой. Но можно использовать отдельно компилятор (`MASM`) и 32-разрядный компоновщик (`LINK32`). Также необходимо отметить, что версии 32-разрядного компоновщика, которые Microsoft использует в более поздних версиях, чем MASM615, называются просто `LINK`, без цифр 32 (в версии MASM615 так называется 16-разрядный компоновщик). Начиная с этой версии, Microsoft уже не разрабатывает 16-разрядные компоновщики.

Запуск программы

Для запуска программы наберите в командной строке имя выполняемой программы `hello` или просто щелкните на ней мышкой в среде Windows.

Отладчик

Основной инструмент, с которым приходится работать при создании программ на ассемблере, — отладчик. В дальнейшем будут рассматриваться небольшие примеры программ на языке ассемблера, и лучшим способом для их изучения является использование отладчика. Отладчик — это программа, позволяющая отображать на экране значения необходимых переменных, получать состояние всех регистров и ячеек памяти при пошаговом выполнении программы, вносить изменения в программу, указывать точки останова и многое другое. Это необходимо при проверке написанных на языке ассемблера программ.

Существует несколько хороших отладчиков, но мы будем использовать универсальный и довольно удобный отладчик Microsoft, который распространяется бесплатно и называется `dbg_x86_6.5.3.8.exe`. Этот отладчик имеет оконный интерфейс и привычные для работающих в среде Windows функции.

Простая программа

Для знакомства с отладчиком напишем на языке ассемблера небольшую программу Sum, которая складывает три числа и сохраняет сумму в памяти. Оттранслируем и выполним компоновку программы с использованием опций отладки, как показано на рис. 3.14, а затем начнем работу с отладчиком.

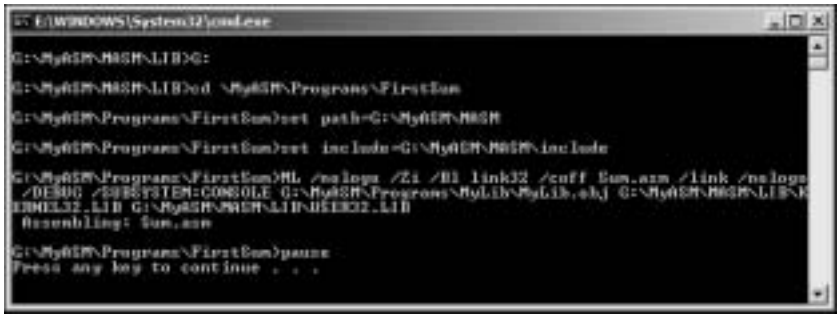

Рис. 3.14. *Компиляция и компоновка программы с опциями отладки*

Программа Sum

```
TITLE Программа суммирования (sum.asm)
.386
.MODEL flat, stdcall  ; Задаем модель памяти.
WaitMsg  PROTO        ; Прототип функции WaitMsg.
ExitProcess PROTO,    ; Прототип функции ExitProcess.
    x: DWORD

.DATA                 ; Задаем сегмент данных.
  sum DW ?            ; Объявляем переменную sum, не присваивая значения.

.CODE                 ; Задаем сегмент кодов.
main PROC             ; Точка входа в программу.
  MOV  EAX,5          ; Поместить в регистр AX значение 5.
  ADD  EAX,10         ; Сложить содержимое регистра EAX с 10.
  ADD  EAX,15         ; Сложить содержимое регистра EAX с 15.
  MOV  sum, AX        ; Сохранить содержимое регистра EAX в sum.
  invoke WaitMsg      ; Запрос для нажатия клавиши.
  invoke ExitProcess,0  ; Корректное окончание программы.
main ENDP             ; Конец процедуры.
END main              ; Конец программы (для компилятора).
```

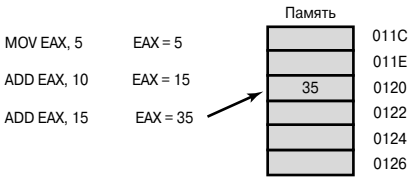

Рис. 3.15. *Диаграмма выполнения простой программы*

Каждая строка в процедуре main этой программы начинается с кода команды (MOV, ADD) с соответствующими операндами или вызова процедуры (макроопределение invoke). Если после этого поставить точку с запятой, то до конца строки все будет восприниматься как комментарий, не влияющий на ход выполнения программы.

Пошаговая диаграмма выполнения показана на рис. 3.15.

Команда MOV заставляет процессор переместить или скопировать значение исходного операнда в принимающий оператор, поэтому в первой строке число 5 перемещается в регистр AX. Во второй строке происходит суммирование числа 10 и содержимого регистра AX, в результате значение регистра становится равным 15. В третьей строке также складываются содержимое регистра AX с числом 20. В регистре AX уже будет находиться число 35. Далее это число копируется в ячейку памяти по адресу, определенному для переменной sum. Команды в пятой и шестой строках необходимы для выхода из программы и передачи управления системе.

Как обычно, программу можно написать в текстовом редакторе и сохранить в файле с именем sum.asm, после чего ее нужно откомпилировать с опцией /Zi, которая заставляет отладчик включить в объектный модуль информацию для отладчика. Затем нужно будет вызвать компоновщик для получения исполняемого файла. Однако поскольку эта программа в дальнейшем будет использоваться с отладчиком, то необходимо запускать компоновщик с опцией /DEBUG, т.е. команда вызова компоновщика будет включать опцию, включающую информацию для отладчика, в исполняемый модуль:

```
/link /DEBUG
```

После получения выполняемого файла можно запустить отладчик и поработать с этой программой, для чего необходимо сначала открыть исходный, а затем исполняемый файл (это команда File⇨Open Sourse File и команда File⇨Open Executable). После чего можно настроить рабочий стол отладчика, как показано на рис. 3.16.

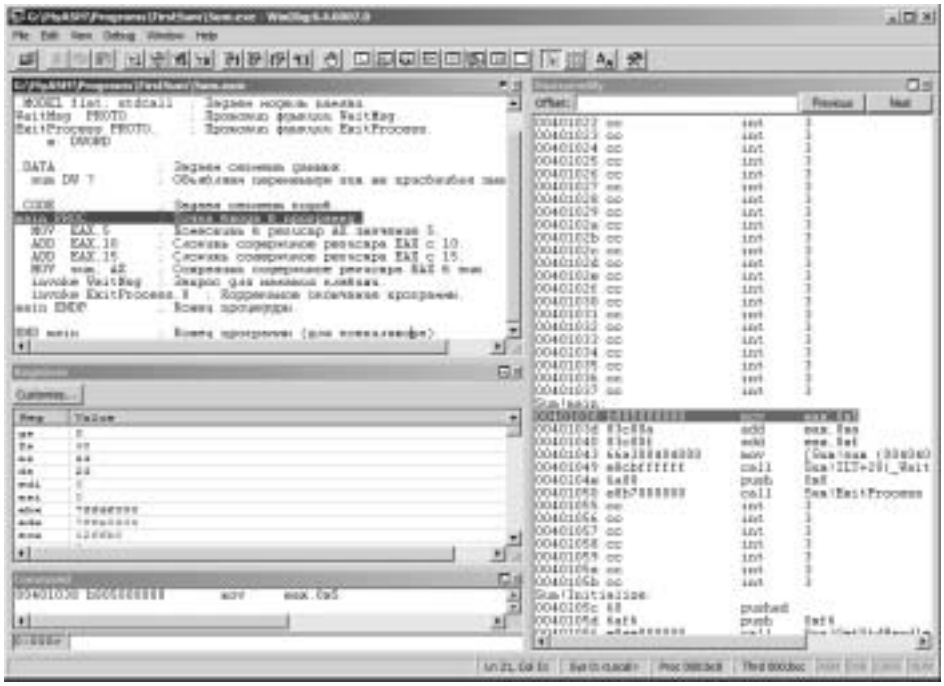

Рис. 3.16. *Рабочий стол отладчика dbg_x86_6.5.3.8*

На рисунке вы видите открытым окно с исходной программой и дополнительные окна Registers (значения всех регистров), Disassembly (восстановленная последовательность команд) и Command (окно ввода команд). Все окна можно закрывать и открывать по желанию

пользователя, создавая удобную для себя среду отладки. Все окна по умолчанию стараются пристыковаться к ближайшей границе (dock) окна, хотя их можно сделать и плавающими (floating). Потребуется некоторая практика, чтобы быстро создавать удобный рабочий стол отладчика. Щелчок правой кнопкой мыши на заголовке окна приводит к появлению контекстного меню, которое позволяет установить необходимый режим работы окна.

Окна можно открывать либо воспользовавшись пунктом меню Window, либо с помощью отдельных кнопок, расположенных на управляющей панели. Названия кнопок, которые появляются в момент задержки курсора на кнопке, перечислены в табл. 3.16.

Таблица 3.16. Названия управляющих кнопок

Название	*Быстрые клавиши*	*Описание*
Open sourse file	Ctrl-O	Открыть файл
Cut	Ctrl-X	Удалить выделенный текст
Copy	Ctrl-C	Копировать выделенный текст
Paste	Ctrl-V	Вставить выделенный текст
Go	F5	Выполнить программу
Restart	Ctrl-Shft-F5	Подготовить отладчик к выполнению программы
Stop debugging	Shft-F5	Закончить отладку
Break	Ctrl-Break	Передача управления отладчику
Step into	F11 (F8)	Заходить в процедуры
Step over	F10	Выполнять программу без захода в процедуры
Step out	Shft-F11	Выйти из процедуры
Run to cursor	Ctrl-F10 (F11)	Выполнить до курсора
Insert or remove breakpoint	F9	Установить или убрать точку останова
Command	Alt-1	Отобразить окно команд
Watch	Alt-2	Показать текущие переменные
Locals	Alt-3	Показать локальные переменные
Registers	Alt-4	Показать регистры
Memory window	Alt-5	Показать фрагмент памяти
Call stack	Alt-6	Показать текущую информацию из стека
Disassembly	Alt-7	Дисассемблировать исходный файл
Scratch pad	Alt-8	Открыть окно буфера обмена
Sourse mode on		Использовать исходный текст
Sourse mode off		Использовать ассемблерный текст
Font		Использовать шрифты
Options		Использовать настройки

Работа с отладчиком не представляет особых трудностей. Необходимо только ввести исходный файл (`.asm`), соответствующий ему исполняемый файл (`.exe`) и выполнить дисассемблирование. Затем установить точку прерывания на процедуре `main` и выполнить команду `Go`. Отладчик остановится на первой команде, после чего можно проводить отладку, просматривая состояние всех регистров, флагов, памяти и текущих команд,

выполняя команды Step into, Step over, Run to cursor и другие. Результаты, получаемые в процессе работы программы, будут отображаться в соответствующем консольном окне.

В окнах отладчика можно видеть как команды ассемблера, так и коды машинных команд. Все это не только значительно помогает в отладке программы, но и удобно при изучении языка ассемблера, а также в процессе исследования операционной системы и аппаратной части компьютера. Фрагмент дисассемблированного текста показан в табл. 3.17.

Таблица 3.17. Фрагмент дисассемблированного текста

Ячейка памяти	*Машинный код*	*Команда*
Sum!main:		
00401038	b805000000	mov eax,0x5
0040103d	83c00a	add eax,0xa
00401040	83c00f	add eax,0xf
00401043	66a300404000	mov [Sum!sum (00404000)],ax
00401049	e8cbffffff	call Sum!ILT+20(_WaitMsg (00401019)
0040104e	6a00	push 0x0
00401050	e8b7000000	call Sum!ExitProcess (0040110c)

В дальнейшем по мере необходимости будут объясняться команды отладчика и описываться приемы работы с ним. Более полное описание команд и директив отладчика приведены в справочном разделе.

Резюме

В этой главе даны общие сведения о языке ассемблера, используемых форматах данных и принципах разработки программ на языке ассемблера. Описана работа с ассемблером и основные этапы разработки программ с использованием ассемблера. Акцент сделан на разработке библиотек, использование которых значительно облегчает работу с программой. Отмечена важность этапа отладки программ и приведены базовые сведения о работе с отладчиком. Для лучшего понимания материала вниманию читателя представлены простейшие примеры разработки и отладки программ.

В данной главе приводились примеры довольно длинных последовательностей команд, которые необходимы для ассемблирования исходного файла. Конечно, команды можно набирать вручную, но лучше использовать командные файлы, которые позволяют значительно облегчить работу. Существенно сэкономить время позволит использование специального интерфейса пользователя (который можно найти на специальных сайтах или разработать самому, если вы знаете язык высоко уровня). Такой интерфейс может создавать необходимые последовательности команд при нажатии определенных кнопок и запускать их на выполнение.

Четкое представление последовательности шагов, выполняемых ассемблером при обработке исходной программы и получении исполняемого файла, значительно облегчит вашу работу в дальнейшем.

Контрольные вопросы

1. Существует ли взаимно однозначное соответствие между командами языка ассемблера и машинными кодами?
2. Можно ли написать программу в машинных кодах?
3. Сколько бит находится в байте, слове, двойном и учетверенном слове?
4. Подсчитайте диапазон значений для слова без знака.
5. Что такое команды и что такое данные?
6. Что такое основание системы счисления?
7. Почему удобно использовать шестнадцатеричную систему счисления для отображения данных?
8. Как представлены в памяти числа без знака и со знаком?
9. Что такое дополнение до двух?
10. Как сохраняется в памяти строка символов? Как подсчитать размер занимаемой памяти для отдельной строки?
11. Как сохраняется в памяти числовое значение? Как подсчитать размер занимаемой памяти для числового значения?
12. Что такое константа? Чем константа отличается от переменной?
13. Какие типы утверждений используются в языке ассемблера?
14. Как разрабатывается программа на языке ассемблера?
15. Назовите основные этапы выполнения программы.
16. Назовите основные опции компилятора.
17. Для чего нужен отладчик?
18. Каковы особенности работы DOS в операционной системе Windows NT?
19. Можно ли использовать все возможности языка ассемблера при работе в DOS под управлением Windows NT?
20. Чем отличаются ассемблеры Microsoft и Borland?