

Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Южно-Уральский государственный университет
Кафедра «Автоматика и управление»

004.43
В858

Вставская Е.В.

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

Конспект лекций

Челябинск
2010

ВВЕДЕНИЕ

Основные определения

Программирование — процесс создания компьютерных программ с помощью языков программирования. В узком смысле слова, программирование рассматривается как кодирование — реализация одного или нескольких взаимосвязанных алгоритмов на некотором языке программирования. Под программированием также может пониматься разработка логической схемы для ПЛИС, а также процесс записи информации в ПЗУ. В более широком смысле программирование — процесс создания программ, то есть разработка программного обеспечения.

Процессор (центральное процессорное устройство, ЦПУ) – устройство, непосредственно предназначенное для выполнения вычислительных операций. Процессор работает под управлением программы, выполняя вычисления или принимая логические решения, необходимые для обработки информации. Он имеет сложную внутреннюю структуру, состоящую из арифметических устройств, схем управления, регулирующих выполнение различных операций, а также регистров, предназначенных для временного хранения данных.

Язык программирования — формальная знаковая система, предназначенная для описания алгоритмов в форме программы, которая удобна для исполнителя (например, компьютера). Компьютерные программы применяются для передачи компьютеру инструкций по выполнению того или иного вычислительного процесса и организации управления отдельными устройствами. Язык программирования определяет набор лексических, синтаксических и семантических правил, используемых при составлении компьютерной программы. Он позволяет программисту точно определить то, на какие события будет реагировать компьютер, как будут храниться и передаваться данные, а также какие именно действия следует выполнять над этими данными при различных обстоятельствах.

Со времени создания первых программируемых машин человечество придумало уже более двух с половиной тысяч языков программирования. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие

становятся известны миллионам людей. Профессиональные программисты иногда применяют в своей работе более десятка разнообразных языков программирования.

Языки программирования обычно делятся на машинно-независимые и машинно-ориентированные. К машинно-независимым относятся Си, Си++, Паскаль и др. Они позволяют быстро писать довольно сложные программы. В тех случаях, когда необходимо построить наиболее компактный машинный код и создать самые быстродействующие программы, используют машинно-ориентированные языки. Они оперируют непосредственно ячейками памяти и программно-доступными элементами микропроцессора.

Различные языки программирования поддерживают различные стили программирования. Отчасти искусство программирования состоит в том, чтобы выбрать один из языков, наиболее полно подходящий для решения имеющейся задачи. Разные языки требуют от программиста различного уровня внимания к деталям при реализации алгоритма, результатом чего часто бывает компромисс между простотой и производительностью (или между временем программиста и временем пользователя).

Единственный язык, напрямую выполняемый процессором — это машинный язык (также называемый машинным кодом). Изначально все программисты прорабатывали каждую мелочь в машинном коде, но сейчас эта трудная работа уже не делается. Вместо этого программисты пишут исходный код, и компьютер (используя компилятор, интерпретатор или ассемблер) транслирует его, в один или несколько этапов, уточняя все детали, в машинный код, готовый к исполнению на целевом процессоре. Даже если требуется полный низкоуровневый контроль над системой, программисты пишут на языке ассемблера, мнемонические инструкции которого преобразуются один к одному в соответствующие инструкции машинного языка целевого процессора.

Язык ассемблера — тип языка программирования низкого уровня, представляющий собой формат записи машинных команд, удобный для восприятия человеком.

На языке ассемблера пишут:

- программы, требующие максимальной скорости выполнения: основные компоненты компьютерных игр, ядра операционных систем реального времени и просто критичные по времени куски программ;
- программы, взаимодействующие с внешними устройствами: драйверы, программы, работающие напрямую с портами, звуковыми и видеоплатами;
- программы, использующие полностью возможности процессора: ядра многозадачных операционных систем, серверы;
- программы, полностью использующие возможности операционной системы: вирусы, антивирусы, защита от несанкционированного доступа, программы обхода защиты от несанкционированного доступа.

Достоинства языка ассемблера

- Максимальная оптимизация программ, как по скорости выполнения, так и по размеру
- Максимальная адаптация под соответствующий процессор

Недостатки

- Трудоемкость написания программы больше, чем языке высокого уровня
- Трудоемкость чтения
- Непереносимость на другие платформы, кроме совместимых
- Ассемблер малопригоден для совместных проектов

Алгоритм — это точный набор инструкций, описывающих порядок действий некоторого исполнителя для достижения результата, решения некоторой задачи за конечное число шагов.

Основные свойства алгоритмов следующие:

1. Понятность для исполнителя — исполнитель алгоритма должен понимать, как его выполнять. Иными словами, имея алгоритм и произвольный вариант исходных данных, исполнитель должен знать, как надо действовать для выполнения этого алгоритма.

2. Дискретность (прерывность, отдельность) — алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов (этапов).

3. Определенность — каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче.

4. Результативность (или конечность) состоит в том, что за конечное число шагов алгоритм либо должен приводить к решению задачи, либо после конечного числа шагов останавливаться из-за невозможности получить решение с выдачей соответствующего сообщения, либо неограниченно продолжаться в течение времени, отведенного для исполнения алгоритма, с выдачей промежуточных результатов.

5. Массовость означает, что алгоритм решения задачи разрабатывается в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма.

СИСТЕМЫ СЧИСЛЕНИЯ

Основные определения

Системой счисления называется совокупность приемов наименования и записи чисел.

Каждое число изображается в виде последовательности цифр, а для изображения каждой цифры используется какой-либо физический элемент, который может находиться в одном из нескольких устойчивых состояний.

Для проведения расчетов в повседневной жизни общепринятой является десятичная система счисления. В этой системе для записи любых чисел используются только десять различных знаков (цифр): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Эти цифры введены для обозначения десяти последовательных целых чисел от 0 до 9. Обозначая число «ДЕСЯТЬ», мы используем уже имеющиеся цифры «10». При этом значение каждой из цифр поставлено в зависимость от того места (позиции), где она стоит в изображении числа. Такая система счисления называется **позиционной**. При этом десять единиц каждого разряда объединяются в одну единицу соседнего, более старшего разряда.

Так, число 252,2 можно записать в виде выражения

$$2 \cdot 10^2 + 5 \cdot 10^1 + 2 \cdot 10^0 + 2 \cdot 10^{-1}.$$

Аналогично десятичная запись произвольного числа x в виде последовательности цифр

$$a_n a_{n-1} \dots a_1 a_0, a_{-1} \dots a_{-m} \dots$$

основана на представлении этого числа в виде полинома

$$x = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0 + a_{-1} \cdot 10^{-1} + \dots + a_{-m} \cdot 10^{-m} + \dots,$$

где $a_i \in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$. При этом запятая, отделяющая целую часть от дробной и является, по существу, началом отсчета.

Число P единиц какого-либо разряда, объединяемых в единицу более старшего разряда, называется **основанием** системы счисления, а сама система счисления называется P -ичной. Так, в десятичной системе счисления основанием системы является число 10. Для записи произвольного числа в P -ичной системе счисления достаточно иметь P различных цифр. Цифры, служащие для обозначения чисел в заданной системе счисления называются **базисными**.

Запись произвольного числа x в позиционной системе счисления с основанием P в виде полинома

$$x = a_n \cdot P^n + a_{n-1} \cdot P^{n-1} + \dots + a_1 \cdot P + a_0 + a_{-1} \cdot P^{-1} + \dots + a_{-m} \cdot P^{-m} + \dots$$

Каждый коэффициент a_i данной записи может быть одним из базисных чисел и изображается одной цифрой. Числа в P -ичной системе счисления записываются в виде перечисления всех коэффициентов полинома с указанием положения запятой:

$$x = a_n a_{n-1} \dots a_1 a_0, a_{-1} \dots a_{-m} \dots$$

В качестве базисных чисел обычно берутся числа от 0 до $P-1$ включительно. Для указания того, в какой системе счисления записано число, основание системы указывается в виде нижнего индекса в десятичной записи, например

$$12,43_8.$$

Двоичная, восьмеричная и шестнадцатеричная системы счисления

Двоичная система счисления

Для представления чисел в микропроцессоре используется двоичная система счисления. Это обусловлено тем, что любой цифровой сигнал может иметь два устойчивых состояния: «высокий уровень» и «низкий уровень». В двоичной системе счисления для изображения любого числа используются две цифры соответственно: 0 и 1. Тогда произвольное число x запишется в виде

$$x = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2 + a_0 + a_{-1} \cdot 2^{-1} + \dots + a_{-m} \cdot 2^{-m} + \dots$$

или
$$x = (a_n a_{n-1} \dots a_1 a_0, a_{-1} \dots a_{-m} \dots)_2.$$

Ниже представлена таблица чисел в двоичной системе счисления

1_{10}	1_2
2_{10}	10_2
3_{10}	11_2
4_{10}	100_2
5_{10}	101_2
6_{10}	110_2
7_{10}	111_2
8_{10}	1000_2

9_{10}	1001_2
10_{10}	1010_2
11_{10}	1011_2
12_{10}	1100_2
13_{10}	1101_2
14_{10}	1110_2
15_{10}	1111_2
16_{10}	10000_2

Шестнадцатеричная система счисления

В шестнадцатеричной системе счисления базисными числами являются числа от нуля до пятнадцати включительно. Поэтому для обозначения базисных чисел одним символом кроме арабских цифр 0...9 в шестнадцатеричной системе счисления используются буквы латинского алфавита:

$$10_{10} = A_{16}$$

$$12_{10} = C_{16}$$

$$14_{10} = E_{16}$$

$$11_{10} = B_{16}$$

$$13_{10} = D_{16}$$

$$15_{10} = F_{16}$$

Например, число 175_{10} в шестнадцатеричной системе счисления запишется как AF_{16} . Действительно,

$$x = 10 \cdot 16^1 + 15 \cdot 16^0 = 160 + 15 = 175.$$

Смешанные системы счисления

Смешанной называется такая система счисления, в которой числа, заданные в некоторой системе счисления с основанием P изображаются с помощью цифр другой системы счисления с основанием Q , где $Q < P$. В такой системе P называется **старшим основанием**, Q — **младшим основанием**, а сама система счисления называется Q - P -ичной. Для того, чтобы запись числа в смешанной системе счисления была однозначной, для представления любой P -ичной цифры отводится одно и то же количество Q -ичных разрядов, достаточное для представления любого базисного числа P -ичной системы.

Так, в двоично-десятичной системе для изображения каждой цифры отводится 4 двоичных разряда. Например, число 925_{10} запишется в двоично-десятичной системе как 1001 0010 0101. Здесь последовательные четверки (тетрады) двоичных разрядов изображают цифры 9, 2 и 5 десятичной записи соответственно.

Следует отметить, что, хотя в двоично-десятичной записи используются только цифры «0» и «1», эта запись отличается от двоичного изображения данного числа. Например, двоичный код 1001 0010 0101 соответствует десятичному числу 2341, а не 925.

Особого внимания заслуживает случай, когда $P=Q^l$ (l – целое положительное число). В этом случае запись какого-либо числа в смешанной системе счисления

тождественно совпадает с изображением этого числа в системе счисления с основанием Q : $A_{2_{16}} = 1010\ 0010_2$.

Перевод чисел из одной системы счисления в другую

Задача перевода заключается в следующем: Пусть известна запись числа x в системе счисления с каким-либо основанием P :

$$x = (p_n p_{n-1} \dots p_0, p_{-1} p_{-2} \dots)_P,$$

где p_i – цифры P -ичной системы ($0 \leq p_i \leq P-1$). Требуется найти запись этого числа x в системе счисления с основанием Q :

$$x = (q_s q_{s-1} \dots q_0, q_{-1} q_{-2} \dots)_Q,$$

где q_i – искомые цифры Q -ичной системы ($0 \leq q_i \leq Q-1$).

Для перевода любого числа достаточно отдельно перевести его целую и дробную части.

Перевод целых чисел.

Представим число x в Q -ичной системе в виде полинома

$$x = q_s q_{s-1} \dots q_1 q_0 = q_s \cdot Q^s + q_{s-1} \cdot Q^{s-1} + \dots + q_1 \cdot Q + q_0 \quad (1)$$

Для определения в поставленной задаче q_0 разделим обе части равенства (1) на Q , причем в левой части произведем фактическое деление, поскольку запись числа x в P -ичной системе нам известна, а в правой части деление выполним аналитически:

$$\left[\frac{x}{Q} \right] = q_s \cdot Q^{s-1} + q_{s-1} \cdot Q^{s-2} + \dots + q_1. \quad (2)$$

Таким образом, младший коэффициент q_0 в разложении (1) является остатком от деления x на Q . Число $x_1 = \left[\frac{x}{Q} \right]$ является целым, и к нему тоже можно применить описанную процедуру:

$$\left[\frac{x_1}{Q} \right] = q_s \cdot Q^{s-2} + q_{s-1} \cdot Q^{s-3} + \dots + q_2, \quad (3)$$

а q_1 – остаток от деления (3).

Этот процесс продолжается до тех пор, пока не получено $x_{s+l}=0$. Для записи числа x в Q -ичной системе счисления запишем каждый из полученных коэффициентов q_i одной Q -ичной цифрой.

Пример 1: Перевести число 47_{10} в двоичную систему счисления ($Q = 2$).

$$\begin{array}{r}
 47 \quad | \quad 2 \\
 \hline
 -46 \quad 23 \quad | \quad 2 \\
 1 \quad -22 \quad 11 \quad | \quad 2 \\
 \quad \quad 1 \quad -10 \quad 5 \quad | \quad 2 \\
 \quad \quad \quad \quad 1 \quad -4 \quad 2 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad 1 \quad -2 \quad 1 \quad | \quad 2 \\
 \quad \quad \quad \quad \quad \quad \quad \quad 0 \quad -0 \quad 0 \\
 \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad 1
 \end{array}$$

Искомое число $47_{10} = 101111_2$.

Пример 2: Перевести число 3060_{10} в шестнадцатеричную систему счисления ($Q = 16$).

$$\begin{array}{r}
 3060 \quad | \quad 16 \\
 \hline
 -16 \quad 191 \quad | \quad 16 \\
 146 \quad -16 \quad 11 \quad | \quad 16 \\
 -144 \quad 31 \quad -0 \quad 0 \\
 20 \quad -16 \quad 11 \\
 -16 \quad 15 \\
 \hline
 4
 \end{array}$$

Таким образом, $q_0=4_{10} = 4_{16}$, $q_1=15_{10} = F_{16}$, $q_2=11_{10} = B_{16}$. Искомое число $3060_{10} = BF4_{16}$.

Перевод дробных чисел

Пусть необходимо перевести в Q -ичную систему правильную дробь x ($0 < x < 1$), заданную в P -ичной системе счисления.

Поскольку $x < 1$, то в Q -ичной системе запись числа x будет иметь вид

$$x = 0, q_{-1}q_{-2} \dots q_{-t} \dots = q_{-1} \cdot Q^{-1} + q_{-2} \cdot Q^{-2} + \dots + q_{-t} \cdot Q^{-t} + \dots \quad (4)$$

Умножив обе части выражения (4) на Q , получим

$$x \cdot Q = q_{-1} + q_{-2} \cdot Q^{-1} + \dots + q_{-t} \cdot Q^{-t+1} + \dots,$$

где q_{-1} является целой частью, а $(q_{-2} \cdot Q^{-1} + \dots + q_{-t} \cdot Q^{-t+1} + \dots)$ – правильная дробь.

Искомые коэффициенты q_i могут быть определены по формуле

$$q_{-(i+1)} = [x_i \cdot Q],$$

где $[]$ – целая часть. Процесс продолжается до тех пор, пока не будет получено $x_{i+1} = 0$, либо не будет достигнута требуемая точность числа.

Пример 3: Перевести число $0,25_{10}$ в двоичную систему счисления

$$\begin{array}{r} 0 \ 25 \\ \times \quad 2 \\ \hline 0 \ 50 \\ \times \quad 2 \\ \hline 1 \ 00 \end{array}$$

Искомое число $x = 0,25_{10} = 0,01_2$.

Арифметические действия в системах счисления с основанием, отличным от 10

Выполнение арифметических действий в любых позиционных системах счисления производится по тем же правилам, которые используются в десятичной системе счисления.

Так же, как и в десятичной системе счисления, для выполнения арифметических действий необходимо знать таблицы сложения (вычитания) и умножения. Ниже представлены данные таблицы для двоичной системы счисления

Таблица 1

Сложение	Вычитание	Умножение
$0 + 0 = 0$	$0 - 0 = 0$	$0 \cdot 0 = 0$
$0 + 1 = 1$	$1 - 0 = 1$	$0 \cdot 1 = 0$
$1 + 0 = 1$	$1 - 1 = 0$	$1 \cdot 0 = 0$
$1 + 1 = 10$	$10 - 1 = 1$	$1 \cdot 1 = 1$

Пользуясь приведенными таблицами, произведем арифметические операции над двоичными числами.

$$\begin{array}{r} \mathbf{s \ s} \\ + 1011 \quad + 11 \\ \hline 1010 \quad 10 \\ \hline 10101_2 \quad 21_{10} \end{array}$$

$$\begin{array}{r} \mathbf{r \ r} \\ - 10110 \quad - 22 \\ \hline 1001 \quad 9 \\ \hline 1101_2 \quad 13_{10} \end{array}$$

$$\begin{array}{r} \mathbf{r \ r \ r \ r} \\ - 10000 \quad - 16 \\ \hline 11 \quad 3 \\ \hline 1101_2 \quad 13_{10} \end{array}$$

В тех случаях, когда занимается единица старшего разряда, она дает две единицы младшего разряда. Если занимается единица через несколько разрядов, то она дает

единицы во всех промежуточных нулевых разрядах и две единицы в младшем нулевом разряде.

Рассмотрим операции умножения и деления двоичных чисел.

$$\begin{array}{r}
 \begin{array}{r}
 1101 \\
 \times 101 \\
 \hline
 1101 \\
 + 0000 \\
 \hline
 1101 \\
 \hline
 1000001_2
 \end{array}
 \quad
 \begin{array}{r}
 \times 13 \\
 \times 5 \\
 \hline
 65_{10}
 \end{array}
 \end{array}
 \qquad
 \begin{array}{r}
 \begin{array}{r}
 1000110 \quad |111 \\
 - \quad 111 \quad 1010_2 \\
 \hline
 00111 \\
 - \quad 111 \\
 \hline
 000
 \end{array}
 \quad
 \begin{array}{r}
 70 \quad |7 \\
 - 70 \quad 10 \\
 \hline
 0
 \end{array}
 \end{array}$$

Зная операции двоичной арифметики, можно переводить числа из двоичной системы счисления в любую другую.

Пример: Перевести число 101111011_2 в десятичную систему счисления.

Поскольку $10_{10} = 1010_2$, запишем

$$\begin{array}{r}
 \begin{array}{r}
 101111011 \quad |1010 \\
 -1010 \\
 \hline
 1110 \\
 -1010 \\
 \hline
 10011 \\
 -1010 \\
 \hline
 1001
 \end{array}
 \quad
 \begin{array}{r}
 100101 \\
 -1010 \\
 \hline
 10001 \\
 -1010 \\
 \hline
 111
 \end{array}
 \quad
 \begin{array}{r}
 |1010 \\
 11 \\
 -0 \\
 \hline
 11
 \end{array}
 \quad
 \begin{array}{r}
 |1010 \\
 0
 \end{array}
 \end{array}$$

Полученные остатки, $q_0=1001_2 = 9_{10}$, $q_1=111_2 = 7_{10}$, $q_2=11_2 = 3_{10}$. Искомое число $101111011_2 = 379_{10}$.

В случае перевода чисел из одной недесятичной системы в другую возникает сложность выполнения действий в недесятичной системе счисления. В этом случае удобнее может быть делать перевод в два этапа $m \rightarrow 10 \rightarrow q$, где m и q – основания систем счисления соответственно.

Двоично-восьмеричные и двоично-шестнадцатеричные преобразования

Двоичная система счисления удобна для выполнения аппаратными средствами микропроцессора арифметических действий, но неудобна для восприятия человеком, поскольку требует много разрядов. Поэтому в вычислительной технике помимо

двоичной системы счисления широкое применение нашли восьмеричная и шестнадцатеричная системы счисления.

Рассмотрим перевод чисел из двоичной системы счисления в восьмеричную.

$$x_2 = a_n a_{n-1} \dots a_1 a_0$$

Запишем число x в полиномиальной форме. Получим

$$x_2 = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0$$

Разделим обе части полученного выражения на 8. Учитывая, что $8 = 2^3$, получим

$$\frac{x}{8} = a_n \cdot 2^{n-3} + a_{n-1} \cdot 2^{n-4} + \dots + (a_2 \cdot 4 + a_1 \cdot 2 + a_0).$$

Таким образом, остаток от деления $q_0 = a_2 \cdot 2^2 + a_1 \cdot 2 + a_0 \cdot 2^0$, что является двоичным разложением десятичного числа, лежащего в диапазоне $[0; 7]$ (для изображения данной десятичной цифры в двоичной системе счисления требуется 3 разряда).

Таким образом, чтобы преобразовать двоичное число в восьмеричное, нужно объединить двоичные цифры в группы по 3 разряда справа налево. При необходимости в начале исходного числа нужно добавить незначащие нули. Затем каждая триада заменяется восьмеричной цифрой.

Пример: Преобразовать число 1101110_2 в восьмеричную систему счисления.

Объединяем двоичные цифры триады справа налево. Получаем

$$001\ 101\ 110_2 = 156_8.$$

Аналогичным образом производятся преобразования из двоичной системы счисления в шестнадцатеричную, только двоичные цифры объединяются в группы по 4 разряда (тетрады).

Пример: Преобразовать число 1101110_2 в шестнадцатеричную систему счисления.

Объединяем двоичные цифры триады справа налево. Получаем

$$0110\ 1110_2 = 6E_{16}.$$

Обратный и дополнительный коды и их применение в операциях с отрицательными числами

Приведенные выше примеры арифметических операций рассмотрены для случая, когда все операнды положительны. Однако очень часто в вычислениях должны использоваться не только положительные, но и отрицательные числа.

Число со знаком в вычислительной технике представляется путем добавления еще одного разряда к величине самого числа. Принято считать, что «0» в знаковом разряде означает знак «плюс» для данного числа, а «1» – знак «минус».

Выполнение арифметических операций над числами с разными знаками представляется для аппаратной части довольно сложной процедурой. В этом случае нужно определить большее по модулю число, произвести вычитание и присвоить разности знак большего по модулю числа.

Применение обратного и дополнительного кодов позволяет выполнить операцию алгебраического суммирования и вычитания на обычном сумматоре. При этом не требуется определения модуля и знака числа.

При представлении чисел в прямом коде значащая часть положительных и отрицательных чисел совпадает. Отличие состоит лишь в знаковом разряде. В прямом коде число «0» имеет два представления «+0» и «-0».

Обратный код для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется из прямого кода положительного числа путем инвертирования всех значащих разрядов прямого кода.

Дополнительный код для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется путем прибавления «1» к обратному коду. Добавление «1» к обратному коду числа «0»: 1111 теперь дает 0000. Таким образом, дополнительный код числа «0» имеет одно значение. Однако это приводит к асимметрии диапазонов представления чисел относительно нуля. Так, в шестнадцатиразрядном представлении диапазон изменения чисел с учетом знака

$$-32768 \leq x \leq 32767.$$

Ниже приведена таблица прямого, обратного и дополнительного кода некоторых чисел.

Число	Прямой код	Обратный код	Дополнительный код
-7	1111	1000	1001
-1	1001	1110	1111
0	1000 0000	1111 0000	0000
1	0001	0001	0001
7	0111	0111	0111

Сложение и вычитание чисел со знаком в дополнительном коде

Если оба числа имеют n -разрядное представление, то алгебраическая сумма будет получена по правилам двоичного сложения (включая знаковый разряд), если отбросить возможный перенос из старшего разряда. Если числа принадлежат диапазону представимых данных и имеют разные знаки, то сумма всегда будет лежать в этом диапазоне. Переполнение может иметь место, если оба слагаемых имеют одинаковые знаки.

Пример 1: $6 - 4 = ?$

6 – положительное число с кодом 0110

-4 – отрицательное число с дополнительным кодом $\overline{0100} + 1 = 1011 + 1 = 1100$

$$\begin{array}{r} 0110 \\ + \\ \underline{1100} \end{array} \quad (\text{перенос игнорируется}): 6 - 4 = 2.$$

$1|0010_2$

Пример 2: $-5 + 2 = ?$

2 – положительное число с кодом 0010

-5 – отрицательное число с дополнительным кодом $\overline{0101} + 1 = 1010 + 1 = 1011$

$$\begin{array}{r} 1011 \\ + \\ \underline{0010} \end{array}$$

1101_2

Число с кодом 1101 является отрицательным, модуль этого числа имеет код $0011_2 = 3_{10}$.

АРХИТЕКТУРА ПЕРСОНАЛЬНОГО КОМПЬЮТЕРА

История развития вычислительной техники

Историю развития вычислительной техники условно делят на 5 поколений.

1-е поколение (1945-1954 гг.) — время становления машин с фон-неймановской архитектурой (Джон фон Нейман), основанной на записывании программы и ее данных в память вычислительной машины. В этот период формируется типовой набор структурных элементов, входящих в состав ЭВМ. Типичная ЭВМ должна состоять из следующих узлов: центральный процессор (ЦП), оперативная память (или оперативное запоминающее устройство — ОЗУ) и устройства ввода-вывода (УВВ). ЦП, в свою очередь, должен состоять из арифметико-логического устройства (АЛУ) и управляющего устройства (УУ). Машины этого поколения работали на ламповой элементной базе, из-за чего поглощали огромное количество энергии и были очень ненадежны. С их помощью, в основном, решались научные задачи. Программы для этих машин уже можно было составлять не на машинном языке, а на языке ассемблера.

2-е поколение (1955-1964 гг.). Смену поколений определило появление новой элементной базы: вместо громоздкой лампы в ЭВМ стали применяться миниатюрные транзисторы, линии задержки как элементы оперативной памяти сменила память на магнитных сердечниках. Это в конечном итоге привело к уменьшению габаритов, повышению надежности и производительности ЭВМ. В архитектуре ЭВМ появились индексные регистры и аппаратные средства для выполнения операций с плавающей точкой. Были разработаны команды для вызова подпрограмм. Появились языки высокого уровня — Algol, FORTRAN, COBOL, — создавшие предпосылки для появления переносимого программного обеспечения, не зависящего от типа ЭВМ. С появлением языков высокого уровня возникли компиляторы для них; библиотеки стандартных подпрограмм и другие хорошо знакомые нам сейчас вещи: Важное новшество — это появление процессоров ввода-вывода. Эти специализированные процессоры позволили освободить ЦП от управления вводом-выводом и осуществлять ввод-вывод с помощью специализированного устройства одновременно с процессом вычислений. Для

эффективного управления ресурсами машины стали использоваться операционные системы (ОС).

3-е поколение (1965-1970 гг.). Смена поколений вновь была обусловлена обновлением элементной базы: вместо транзисторов в различных узлах ЭВМ стали использоваться интегральные микросхемы различной степени интеграции. Микросхемы позволили разместить десятки элементов на пластине размером в несколько сантиметров. Это, в свою очередь, не только повысило производительность ЭВМ, но и снизило их габариты и стоимость. Увеличение мощности ЭВМ сделало возможным одновременное выполнение нескольких программ на одной ЭВМ. Для этого нужно было научиться координировать между собой одновременно выполняемые действия, для чего были расширены функции операционной системы. Одновременно с активными разработками в области аппаратных и архитектурных решений растет удельный вес разработок в области технологий программирования. В это время активно разрабатываются теоретические основы методов программирования, компиляции, баз данных, операционных систем и т. д. Создаются пакеты прикладных программ для самых различных областей жизнедеятельности человека. Наблюдается тенденция к созданию семейств ЭВМ, то есть машины становятся совместимы снизу вверх на программно-аппаратном уровне. Примерами таких семейств была серия IBM System 360 и наш отечественный аналог - ЕС ЭВМ.

4-е поколение (1970-1984 гг.). Очередная смена элементной базы привела к смене поколений. В 70-е годы активно ведутся работы по созданию больших и сверхбольших интегральных схем (БИС и СБИС), которые позволили разместить на одном кристалле десятки тысяч элементов. Это повлекло дальнейшее существенное снижение размеров и стоимости ЭВМ. В начале 70-х годов фирмой Intel был выпущен микропроцессор (МП) i4004. И если до этого в мире вычислительной техники были только три направления (суперЭВМ, большие Э.ВМ (мэйнфреймы) и мини-ЭВМ), то теперь к ним прибавилось еще одно — микропроцессорное.

Процессором называется функциональный блок ЭВМ, предназначенный для логической и арифметической обработки информации на основе принципа микропрограммного управления. По аппаратной реализации процессоры можно

разделить на микропроцессоры (полностью интегрирующие все функции процессора) и процессоры с малой и средней интеграцией. Конструктивно это выражается в том, что микропроцессоры реализуют все функции процессора на одном кристалле, а процессоры других типов реализуют их путем соединения большого количества микросхем.

5-е поколение можно назвать микропроцессорным. В 1976 году фирма Intel закончила разработку 16-разрядного микропроцессора i8086. Он имел достаточно большую разрядность регистров (16 бит) и системной шины адреса (20 бит), за счет чего мог адресовать до 1 Мбайт оперативной памяти. В 1982 году был создан i80286. Этот микропроцессор представлял собой улучшенный вариант i8086. Он поддерживал уже несколько режимов работы: реальный, когда формирование адреса производилось по правилам i8086, и защищенный, который аппаратно реализовывал многозадачность и управление виртуальной памятью, i80286 имел также большую разрядность шины адреса — 24 разряда против, 20 у i8086, и поэтому он мог адресовать до 16 Мбайт оперативной памяти. Первые компьютеры на базе этого микропроцессора появились в 1984 году. В 1985 году фирма Intel представила первый 32-разрядный микропроцессор i80386, аппаратно совместимый снизу вверх со всеми предыдущими микропроцессорами этой фирмы. Он был гораздо мощнее своих предшественников, имел 32-разрядную архитектуру и мог прямо адресовать до 4 Гбайт оперативной памяти. Микропроцессор i386 стал поддерживать новый режим работы — режим виртуального i8086, который обеспечил не только большую эффективность работу программ, разработанных для i8086, но и позволил осуществлять параллельную работу нескольких таких программ.

Основные термины и определения

Архитектура – совокупность подходов и технических решений, используемых при создании устройства. Общая архитектура процессора определяет комплекс средств, предоставляемых пользователю для решения различных задач. Эта архитектура задаёт базовую систему команд процессора и реализуемых способов адресации, набор программно-доступных регистров (регистровая модель),

возможные режимы работы процессора и обращения к памяти и внешним устройствам (организация памяти и реализация обмена по системной шине), средства обработки прерываний и исключений.

Операция – элементарное действие по обработке единицы информации.

Операнд – объект, над которым выполняется машинная команда или оператор языка программирования.

Регистр – устройство сверхбыстродействующей памяти в процессоре, служащее для временного хранения управляющей информации, операндов и/или результатов выполняемых данной микросхемой операций. Совокупность регистров процессора называется **набором регистров**.

Кэш-память – высокоскоростное устройство буферизации данных.

Арифметико-логическое устройство, АЛУ – часть процессора, выполняющая набор его арифметических и логических команд.

Системная шина – шина (набор проводников, по которым передаются сигналы), соединяющая процессор с такими компонентами на системной плате, как ОЗУ, контроллеры дисков и т. п. Системная шина состоит из шины адреса, шины управления и шины данных.

Шина данных – служит для пересылки данных между процессором и ОЗУ. Сейчас, как правило, используются 32- и 64-разрядные шины данных

Шина адреса, – набор линий в системной шине, используемый для передачи сигналов, с помощью которых определяется местоположение ячейки памяти для выполняемых процессором операций чтения/записи и ввода-вывода. Шина адреса обычно однонаправленная, но может быть и двунаправленной

Шина управления – служит для пересылки управляющих сигналов. Каждая линия этой шины имеет своё особое назначение, поэтому они могут быть как однонаправленными, так и двунаправленными.

Устройство управления, УУ – блок процессора, управляющий организацией исполнения команд.

Управляющий сигнал – набор сигналов, используемый для управления контроллерами периферийных устройств.

Очередь команд – буфер процессора, в который команда попадает после операции выборки и откуда она направляется на соответствующее исполнительное устройство.

Оперативное запоминающее устройство, ОЗУ (RAM) – память, предназначенная для временного хранения программ и данных; место, куда программа загружается для исполнения. Содержимое ячейки ОЗУ можно изменять любое число раз и обращаться к данным в любой последовательности. Разделяется на динамическую и статическую, энергозависимую и энергонезависимую.

Постоянное запоминающее устройство, ПЗУ (ROM) – вид постоянного ЗУ, содержимое которого однократно записывается в микросхемы и может только читаться.

Функциональная структура компьютера

Подавляющее большинство современных вычислительных машин построено по принципу архитектуры фон Неймана. В функциональном устройстве компьютера можно выделить следующие основные блоки (см. рис. 2): устройства ввода-вывода (УВВ), память и процессор. Все они взаимодействуют между собой через системную шину.

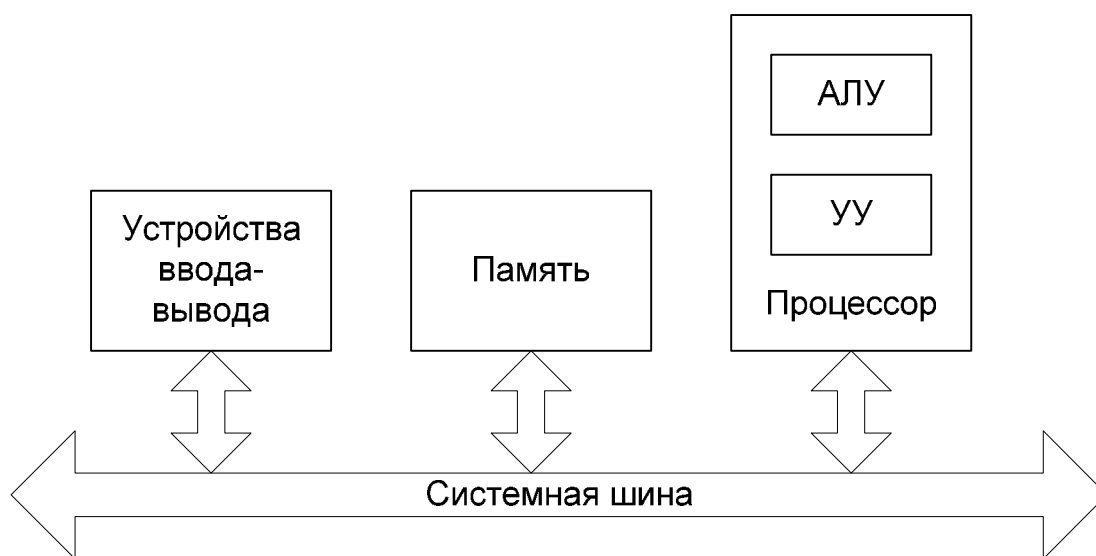


Рисунок 1

Устройства ввода принимают закодированную информацию от операторов, электромеханических устройств (клавиатура, мышь) или от других компьютеров сети. Полученная информация либо сохраняется в памяти компьютера для дальнейшего использования, либо немедленно используется АЛУ для выполнения

необходимых операций. Последовательность шагов обработки определяется хранящейся в памяти программой. Полученные результаты обратно отправляются во внешний мир посредством устройств вывода. Все эти действия координируются устройством управления.

Архитектура микропроцессора

Все 32-разрядные процессоры Intel, начиная с i386, реализуются с использованием архитектуры IA-32 (Intel Architecture 32).

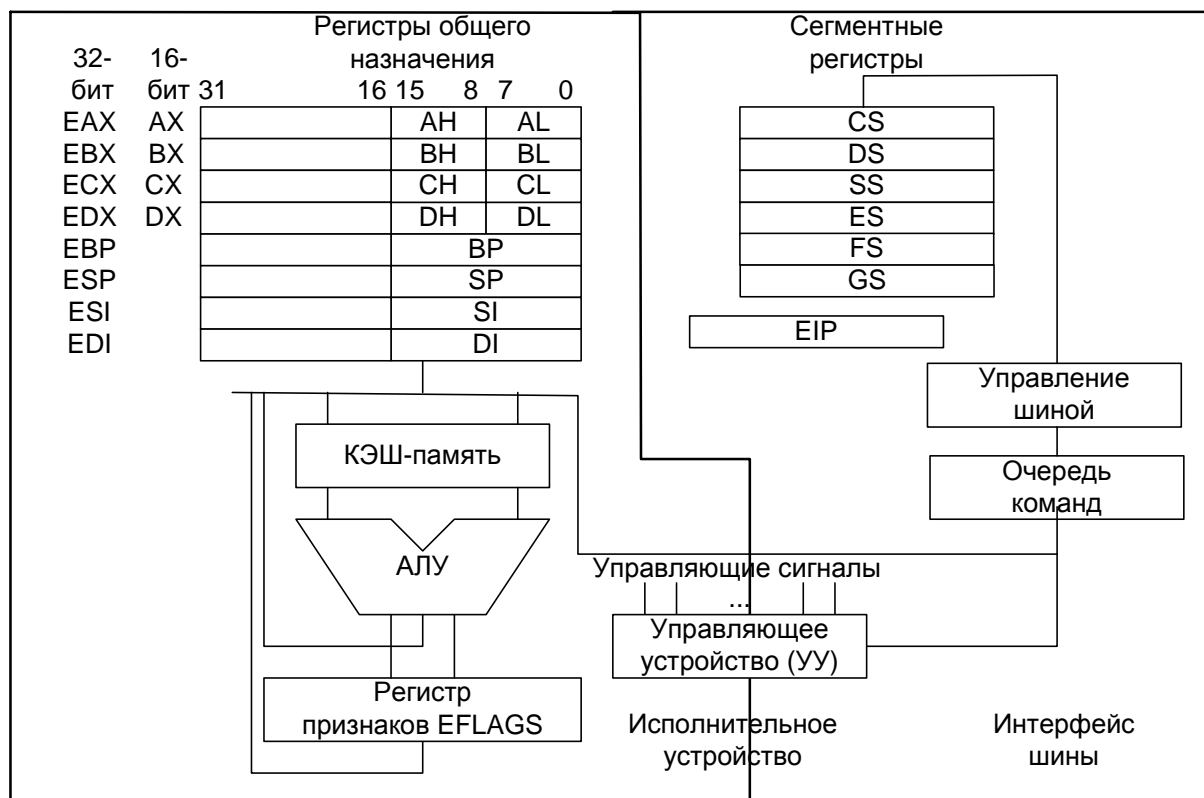


Рисунок 2

Микропроцессор состоит из двух основных частей: **операционного узла**, выполняющего команды, и **шинного интерфейса**, который производит выборку команд, формирует очередь команд, производит считывание операндов и запись результата. Более полно структура микропроцессора показана на рис. 1.

Регистры общего назначения и сегментные регистры

На рис. 2 показаны основные регистры микропроцессора. Ниже приводится их краткая характеристика.

Регистр AX (EAX) (аккумулятор) – автоматически применяется при операциях *умножения, деления* и при работе с портами *ввода-вывода*. Его использование в арифметических, логических и некоторых других операциях позволяет увеличить скорость их выполнения.

Регистр BX (EBX) (регистр базы) – может содержать адреса элементов оперативной памяти. По умолчанию эти адреса будут представлять собой *смещение в сегменте данных*.

Регистр CX (ECX) (счетчик) – используется в различных *операциях повторения*, например в циклах, в строковых командах и т. д.

Регистр DX (EDX) (регистр данных) – является единственным элементом, который может хранить *адреса портов ввода-вывода* в командах типа *in* (получить из порта) и *out* (вывести в порт). Без его помощи невозможно обратиться к портам с адресами от 256 до 65535. Этот регистр применяется также в операциях *умножения и деления*.

Регистры AX, BX, CX, DX позволяют независимо обращаться к их старшей (H) и младшей (L) половине. Соответствующие подрегистры являются 8-разрядными и имеют адреса AH, AL, BH, BL, CH, CL, DH, DL.

Регистр SI (ESI) (регистр индекса источника), как и регистр BX, может содержать адреса элементов в оперативной памяти. По умолчанию эти адреса будут представлять собой смещение в *сегменте данных*. При выполнении операций со строками в этом регистре содержится смещение *строки источника* в сегменте данных.

Регистр DI (EDI) (регистр индекса приемника), как и SI, может содержать адреса элементов в оперативной памяти. По умолчанию эти адреса будут представлять собой смещение в сегменте данных. При выполнении операций со строками в этом регистре содержится смещение *строки приемника* в сегменте данных.

Регистр BP (EBP) (указатель базы) может содержать адреса элементов в оперативной памяти. Эти адреса будут представлять собой *смещение в сегменте стека*.

Регистр SP (ESP) (указатель стека) используется для записи данных в стек и чтения их из стека. Фактически он содержит *смещение в сегменте стека*, которое определяет нужное слово памяти. Значения этого регистра автоматически меняются командами для работы со стеком типов *push, pop, pushf, popf, call, ret*.

Регистр IP (EIP) (указатель команд) всегда содержит смещение в сегменте кода следующей выполняемой команды. Как только некоторая команда начинает выполняться, значение IP увеличивается на ее длину так, что будет адресовать следующую команду. Обычно команды выполняются в той последовательности, в которой они расположены в программе. Нарушают эту последовательность только команды переходов (они начинаются с буквы *j*: *jxx*), команды вызова подпрограммы (*call*), обработчиков прерываний (*int*) и возврата (*ret, iret*). Непосредственно содержимое IP нельзя изменить или прочитать. Косвенно загрузить в регистр IP новое значение могут только команды *jxx, call, int, ret, iret*.

Регистр CS (регистр сегмента кода) определяет стартовый адрес сегмента, в который помещается код выполняемой программы. Это единственный сегментный регистр, который нельзя загрузить непосредственно. Косвенно загрузить в регистр CS новое значение могут команды вида *jxx, call, int, ret, iret*. Физический адрес команды в памяти выполняемой программы определяет пара регистров CS:IP. Аналогичные формы записи используются для указания физического адреса в других сегментах.

Регистр DS (регистр сегмента данных) определяет стартовый адрес сегмента, в который помещаются данные для программы. По умолчанию смещения в сегменте данных задаются в регистрах BX, SI и DI.

Регистры ES, FS, GS (регистры сегментов дополнительных данных) определяет стартовый адрес сегмента, в который помещаются дополнительные данные для программы. Например, в случае строковых команд, DS определяет сегмент для строки-источника, а ES – сегмент для строки-приемника. За исключением строковых команд, доступ к данным в сегменте ES обычно менее эффективен, чем в сегменте DS.

Регистр SS (регистр сегмента стека) определяет стартовый адрес сегмента, в который помещается стек для программы. По умолчанию смещения для сегмента стека задаются в регистрах SP и BP.

Сегментные регистры 16-битны.

Регистр признаков FLAGS (EFLAGS) включает биты, каждый из которых устанавливается в единичное или в нулевое состояние при определенных условиях.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0							0										0		0		1	
										ID	VIP	VIF	AC	VM	RF		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF	CF		
										x	x	x	x	x	x	x	x	x	s	c	x	x	s	s		s		s	s		

s - флаг состояния (STATUS)
c - флаг управления (CONTROL)
x - системный флаг (SYSTEM)

CF – бит переноса: устанавливается в 1, когда арифметическая операция генерирует перенос или выход за разрядную сетку результата. сбрасывается в 0 в противном случае. Этот флаг показывает состояние переполнения для беззнаковых целочисленных арифметических действий. Он также используется в арифметических действиях с повышенной точностью. Может быть установлен командой STC или сброшен командой CLC.

PF – бит четности: устанавливается в 1, если результат последней операции имеет четное число единиц.

AF – бит вспомогательного переноса: устанавливается в 1, если арифметическая операция генерирует перенос из младшей тетрады битов (из 3 бита в 4), сбрасывается в 0 в противном случае. Этот флаг используется в двоично-десятичной арифметике.

ZF – бит нулевого значения: устанавливается в 1, если результат нулевой, сбрасывается в 0 в противном случае.

SF – знаковый бит: устанавливается равным старшему биту результата, который определяет знак в знаковых целочисленных операциях (0 – положительное число, 1 – отрицательное число).

OF – бит переполнения: устанавливается в 1, если целочисленный результат выходит за пределы разрядной сетки. Тем самым данный бит указывает на потерю старшего бита результата.

DF – бит направления: управляет строковыми командами (MOVS, CMPS, SCAS, LODS, STOS). Если $DF = 1$ (команда STD), то содержимое индексных регистров SI, DI увеличивается, если $DF = 0$ (команда CLD), то содержимое индексных регистров SI, DI уменьшается.

IF – бит прерываний: при значении 1 микропроцессор реагирует на внешние аппаратные прерывания по входу INTR. При значении 0 микропроцессор игнорирует внешние прерывания.

TF – бит пошаговой отладки: устанавливается в 1 для включения режима пошаговой отладки программы, сбрасывается в 0 в противном случае.

IOPL – уровень приоритета ввода-вывода отображает уровень приоритета ввода-вывода для выполняемой в данное время программы или задачи. Действительный приоритет задачи может быть меньше или равен IOPL.

NT – флаг вложенной задачи: управляет последовательностью вызванных и прерванных задач. Установлен в 1, если текущая задача связана с предыдущей, сброшен в 0, если текущая задача не связана с другими задачами.

RF — флаг возобновления: используется при обработке прерываний от регистров отладки.

VM — флаг виртуального 8086: признак работы процессора в режиме виртуального 8086: 1 – процессор работает в режиме виртуального 8086, 0 – процессор работает в реальном или защищенном режиме.

AC — флаг контроля выравнивания: предназначен для разрешения контроля выравнивания при обращениях к памяти. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут вызывать исключительную ситуацию.

VIF — флаг виртуального прерывания: при определенных условиях (одно из которых – работа микропроцессора в V-режиме) является аналогом флага IF. Флаг VIF используется совместно с флагом VIP.

VIP — флаг отложенного виртуального прерывания: устанавливается в 1 для индикации отложенного прерывания. Используется совместно с VIF в V-режиме.

ID — **флаг поддержки идентификации процессора:** используется для отображения поддержки микропроцессором инструкции CPUID.

Управляющие регистры

Регистр CR0.

PA	31	CD	30	NW	29	-	28	-	27	-	26	-	25	-	24	-	23	-	22	-	21	-	20	-	19	-	18	AM	-	17	WP	16	-	15	-	14	-	13	-	12	-	11	-	10	-	9	-	8	-	7	-	6	-	5	NE	4	ET	3	TS	2	EM	1	MP	0	PE
----	----	----	----	----	----	---	----	---	----	---	----	---	----	---	----	---	----	---	----	---	----	---	----	---	----	---	----	----	---	----	----	----	---	----	---	----	---	----	---	----	---	----	---	----	---	---	---	---	---	---	---	---	---	---	----	---	----	---	----	---	----	---	----	---	----

0-й бит, разрешение защиты (PE). Переводит процессор в защищенный режим.

1-й бит, мониторинг сопроцессора (MP). Вызывает исключение 7 по каждой команде WAIT.

2-й бит, эмуляция сопроцессора (EM). Вызывает исключение 7 по каждой команде сопроцессора.

3-й бит, бит переключения задач (TS). Позволяет определить, относится данный контекст сопроцессора к текущей задаче или нет. Вызывает исключение 7 при выполнении следующей команды сопроцессора.

4-й бит, индикатор поддержки инструкций сопроцессора (ET).

5-й бит, разрешение стандартного механизма сообщений об ошибке сопроцессора (NE).

5-15-й бит, не используются.

16-й бит, разрешение защиты от записи на уровне привилегий супервизора (WP).

17-й бит, не используется.

18-й бит, разрешение контроля выравнивания (AM).

19-28-й бит, не используются.

29-й бит, запрет сквозной записи кэша и циклов аннулирования (NW).

30-й бит, запрет заполнения кэша (CD).

31-й бит, включение механизма страничной переадресации.

Регистр CR1 пока не используется.

Регистр CR2 хранит 32-битный линейный адрес, по которому был получен последний отказ страницы памяти.

Регистр CR3 - в старших 20 битах хранится физический базовый адрес таблицы каталога страниц.

Остальные биты.

3-й бит, кэширование страниц со сквозной записью (PWT).

4-й бит, запрет кэширование страницы (PCD).

Регистр CR4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	FSR	PMC	PGE	MCE	PAE	PSE	DE	TSD	PVI	VME

0-й бит, разрешение использования виртуального флага прерываний в режиме V8086 (VME).

1-й бит, разрешение использования виртуального флага прерываний в защищенном режиме (PVI).

2-й бит, превращение инструкции RDTSC в привилегированную (TSD).

3-й бит, разрешение точек останова по обращению к портам ввода-вывода (DE).

4-й бит, включает режим адресации с 4-мегабайтными страницами (PSE).

5-й бит, включает 36-битное физическое адресное пространство (PAE).

6-й бит, разрешение исключения MC (MCE).

7-й бит, разрешение глобальной страницы (PGE).

8-й бит, разрешает выполнение команды RDPMC (PMC).

9-й бит, разрешает команды быстрого сохранения/восстановления состояния сопроцессора (FSR).

Системные адресные регистры

GDTR - 6-байтный регистр, в котором содержится линейный адрес глобальной дескрипторной таблицы.

IDTR - 6-байтный регистр, содержащий 32-битный линейный адрес таблицы дескрипторов обработчиков прерываний.

LDTR - 10-байтный регистр, содержащий 16-битный селектор (индекс) для GDT и 8-байтный дескриптор.

TR - 10-байтный регистр, содержащий 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи.

Регистры отладки

DR0...DR3 - хранят 32-битные линейные адреса точек останова.

DR6 (равносильно DR4) - отражает состояние контрольных точек.

DR7 (равносильно DR5) - управляет установкой контрольных точек.

Режимы работы микропроцессора

Реальный режим (Real Mode). После инициализации (системного сброса) МП находится в реальном режиме. В реальном режиме МП работает как очень быстрый 8086 с возможностью использования 32-битных расширений. Механизм адресации, размеры памяти и обработка прерываний (с их последовательными ограничениями) МП 8086 полностью совпадают с аналогичными функциями других МП IA-32 в реальном режиме.

Режим системного управления (System Management Mode). В новых поколениях МП Intel появился еще один режим работы - режим системного управления. Он предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже от операционной системы. МП переходит в этот режим только аппаратно: по низкому уровню на контакте SMI# или по команде с шины APIC (Pentium+). Никакой программный способ не предусмотрен для перехода в этот режим. МП возвращается из режима системного управления в тот режим, при работе в котором был получен сигнал SMI#. Возврат происходит по команде RSM. Эта команда работает только в режиме системного управления и в других режимах не распознается, генерируя исключение #6 (недействительный код операции).

Защищенный режим (Protected Mode) является основным режимом работы МП. Ключевые особенности защищенного режима: виртуальное адресное пространство, защита и многозадачность. МП может быть переведен в защищенный режим установкой бита 0 (Protect Enable) в регистре CR0. Вернуться в режим реального адреса МП может по сигналу RESET или сбросом бита PE.

В защищенном режиме программа оперирует с адресами, которые могут относиться к физически отсутствующим ячейкам памяти, поэтому такое адресное пространство называется виртуальным. Размер виртуального адресного пространства программы может превышать емкость физической памяти и достигать 64Тбайт.

Виртуальный режим i8086 (V86). В режим V86 процессор может перейти из защищённого режима, если установить в регистре флагов EFLAGS бит виртуального режима (VM-бит). Номер бита VM в регистре EFLAGS - 17. Когда процессор

находится в виртуальном режиме, его поведение во многом напоминает поведение процессора i8086. В частности, для адресации памяти используется схема <сегмент:смещение>, размер сегмента составляет 64 килобайта, а размер адресуемой в этом режиме памяти - 1 мегабайт. Виртуальный режим предназначен для работы программ, ориентированных на процессор i8086 (или i8088). Но виртуальный режим – это не реальный режим процессора i8086, имеются существенные отличия. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.

В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое размером в 1 мегабайт.

Все задачи виртуального режима обычно выполняются в третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме.

Обработчики прерываний защищённого режима могут моделировать функции соответствующих прерываний реального режима, что необходимо для правильной работы программ, ориентированных на реальный режим операционной системы MS-DOS.

Память компьютера

Память – способность объекта обеспечивать хранение данных.

Числовые и символьные операнды, равно как и команды, хранятся в памяти компьютера. Память состоит из ячеек, в каждой из которых содержится 1 **бит** информации, принимающий значения «0» или «1». Как правило, биты обрабатывают группами фиксированного размера. Для этого память организуется таким образом, что группы по n бит могут записываться и считываться за одну базовую операцию. Группа из n бит называется **словом** информации, а значение n – **длиной слова**. Слова последовательно располагаются в памяти компьютера.

Длина слова современных компьютеров составляет 16-64 бит. Восемь последовательных битов называются **байтом**.

1 килобайт (Кбайт) = 2^{10} = 1 024 байт

1 мегабайт (Мбайт) = 2^{10} Кбайт = 2^{20} байт = 1 048 576 байт

1 гигабайт (Гбайт) = 2^{10} Мбайт = 2^{30} байт = 1 073 741 824 байт

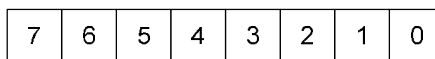
Для доступа к памяти с целью записи или чтения отдельных элементов информации, будь то слова или байты, необходимы имена, или **адреса**, определяющие их расположение в памяти. В качестве адресов используются числа из диапазона от 0 до $2^k - 1$ со значением k , достаточным для адресации всей памяти компьютера. Все 2^k адресов составляют **адресное пространство компьютера**.

Диапазон значений физических адресов зависит от разрядности шины адреса микропроцессора. Адресное пространство процессора i8086 составляет 1 Мбайт, для процессоров i486 и Pentium – 4 Гбайт (2^{32} байт), для микропроцессоров семейства P6 (Pentium Pro/II/III) – 64 Гбайт (2^{36}).

Чаще всего адреса назначаются байтам памяти. Память, в которой каждый байт имеет свой адрес, называется **памятью с байтовой адресацией**. При этом последовательные байты имеют адреса 0, 1, 2 и т. д.

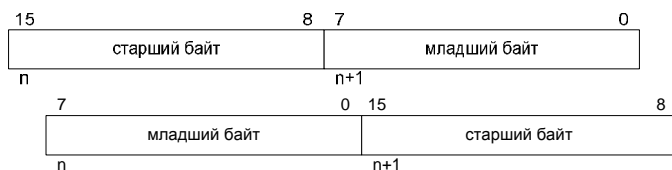
Существует прямой и обратный способы адресации байтов. При **обратном** способе адресации байты адресуются слева направо, так что самый старший (левый) байт слова имеет наименьший адрес. **Прямым** способом называется противоположная система адресации. Наряду с порядком следования байтов в слове

необходимо определить порядок следования битов в байте. Типичный способ расположения битов в байте:



обратная адресация

прямая адресация

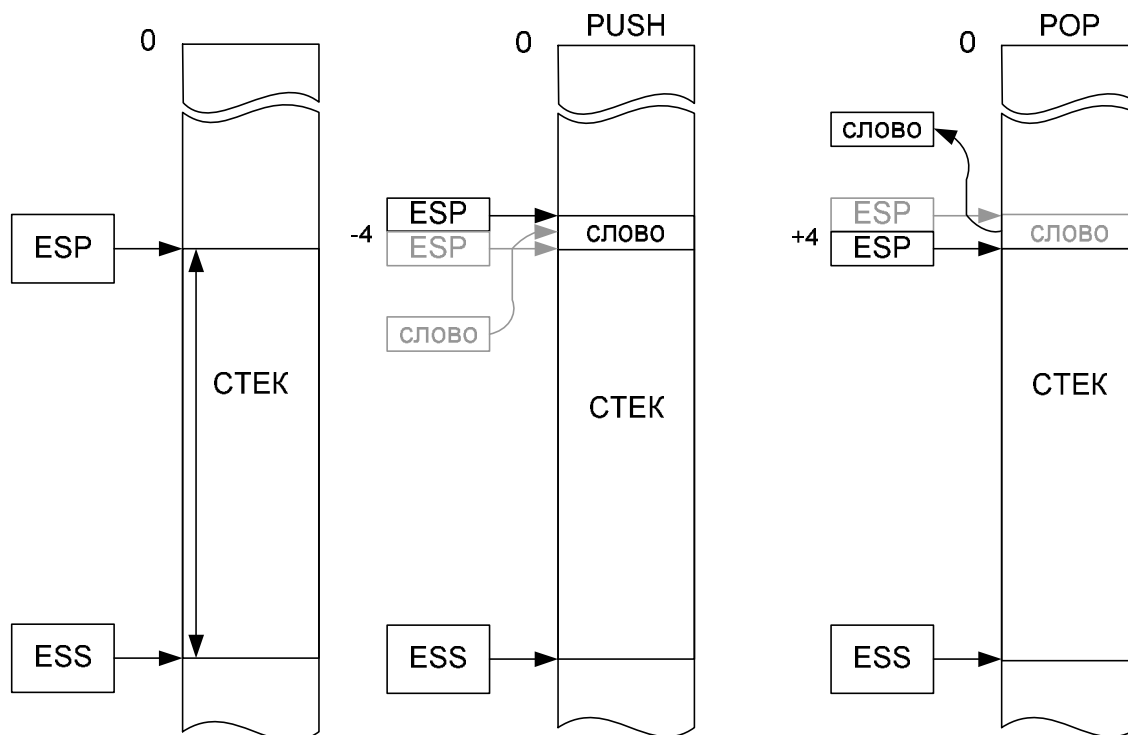


Обычно число занимает целое слово. Поэтому для того, чтобы обратиться к нему в памяти, нужно указать адрес слова, по которому это число хранится. Компиляторы высокоуровневых языков поддерживают прямой способ адресации.

Микропроцессор аппаратно поддерживает две модели памяти: сегментированную и страничную.

Организация стека

Стек – это такая структура данных в памяти, которая используется для временного хранения информации. Программа может поместить слово в стек (команда PUSH) или извлечь его из стека (команда POP). Данные в стеке упорядочиваются специальным образом. Извлекаемый из стека элемент данных – это всегда тот элемент, который был записан туда последним. Такая организация хранения данных сокращенно обозначается LIFO (Last In, First Out – последний поступивший удаляется первым). Если мы поместим в стек два элемента: сначала А, а затем В, то при первом обращении к стеку извлекается элемент В, а при следующем – А. Информация выбирается из стека в обратном по отношению к записи порядке.



В ЭВМ за стеком резервируется блок памяти, адресуемый регистром SS и указатель, называемый указателем стека SP (Stack Pointer). Указатель стека используется программой для того, чтобы фиксировать самый последний записанный в стек элемент данных. При выполнении команды POP или PUSH значение указателя стека соответственно увеличивается или уменьшается на 4.

Организация памяти

Физическая память, к которой микропроцессор имеет доступ по шине адреса, называется **оперативной памятью ОП** (или оперативным запоминающим устройством - ОЗУ).

Механизм управления памятью полностью аппаратный, т.е. программа сама не может сформировать физический адрес памяти на адресной шине.

Микропроцессор аппаратно поддерживает несколько моделей использования оперативной памяти:

- сегментированную модель
- страничную модель
- плоскую модель

В сегментированной модели память для программы делится на непрерывные области памяти, называемые сегментами. Сама программа может обращаться только к данным, которые находятся в этих сегментах.

Сегменты - это логические элементы программы. Сегмент представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

Сегментация - механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния.

Программист может либо самостоятельно разбивать программу на фрагменты (сегменты), либо автоматизировать этот процесс и возложить его на систему программирования.

Для микропроцессоров Intel принят особый подход к управлению памятью. Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к 3 основным сегментам: кода, данных и стека и к 3 дополнительным сегментам данных.

1. **Сегмент кодов (.CODE)** – содержит машинные команды, которые будут выполняться. Обычно первая выполняемая команда находится в начале этого сегмента, и операционная система передает управление по адресу данного сегмента для выполнения программы. Регистр сегмента кодов (CS) адресует данный сегмент.

2. **Сегмент данных (.DATA)** – содержит определенные данные, константы и рабочие области, необходимые программе. Регистр сегмента данных (DS) адресует данный сегмент.

3. **Сегмент стека (.STACK)**. Стек содержит адреса возврата как для программы (для возврата в операционную систему), так и для вызовов подпрограмм (для возврата в главную программу). Регистр сегмента стека (SS) адресует данный сегмент. Адрес текущей вершины стека задается регистрами SS:SP.

Регистры дополнительных сегментов (ES, FS, GS), предназначены для специального использования. На рис. 3 графически представлены регистры SS, DS и CS. Последовательность регистров и сегментов на практике может быть иной. Три сегментных регистра содержат начальные адреса соответствующих сегментов, и каждый сегмент начинается на границе параграфа.

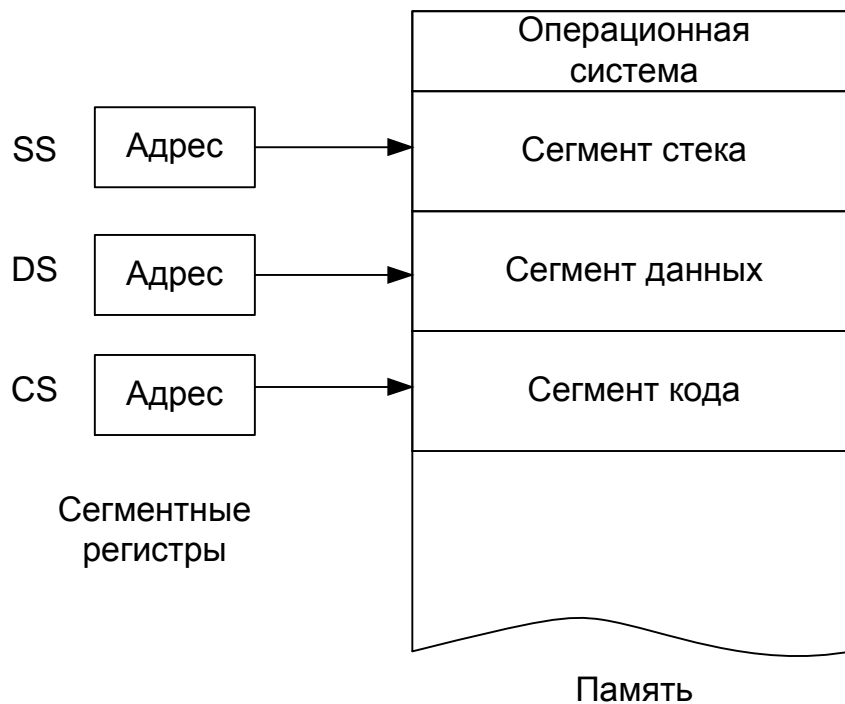
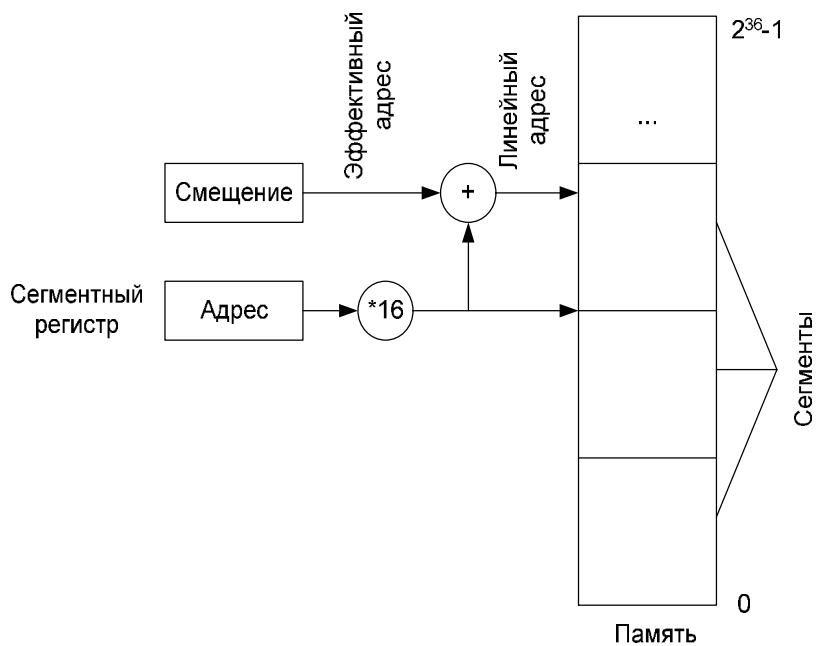


Рисунок 3

Операционная система размещает сегменты программы в ОП по определенным физическим адресам, а значения этих адресов записывает в определенные места, в зависимости от режима работы микропроцессора:

- в реальном режиме адреса помещаются непосредственно в сегментные регистры (cs, ds, ss, es, gs, fs);
- в защищенном режиме - в специальную системную дескрипторную таблицу (Элементом дескрипторной таблицы является дескриптор сегмента. Каждый сегмент имеет дескриптор сегмента -8 байт. Существует три дескрипторные таблицы. Адрес каждой таблицы записывается в специальный системный регистр).

Для доступа к данным внутри сегмента обращение производится относительно начала сегмента линейно, т.е. начиная с 0 и заканчивая адресом, равным размеру сегмента. Этот адрес называется смещением (offset).



Двухбайтовое смещение (16-бит) может быть в пределах от 0000h до FFFFh или от 0 до 65535. Для обращения к любому адресу в программе, компьютер складывает адрес в регистре сегмента и смещение. Например, первый байт в сегменте кодов имеет смещение 0, второй байт – 01 и так далее.

Таким образом, для обращения к конкретному физическому адресу ОП необходимо определить адрес начала сегмента и смещение внутри сегмента.

Физический адрес принято записывать парой этих значений, разделенных двоеточием

segment : offset

Например, 0040:001Ch; 0000:041Ch; 0020:021Ch; 0041:000Ch.

В качестве примера адресации допустим, что регистр сегмента данных содержит 045Fh, и некоторая команда обращается к ячейке памяти внутри сегмента данных со смещением 0032h. Несмотря на то, что регистр сегмента данных содержит 045Fh, он указывает на адрес 045F0, то есть на границу параграфа. Действительный адрес памяти поэтому будет следующий:

Адрес в DS:	045F0
Смещение:	0032
Реальный адрес:	04622

Адрес указывается как DS:0032h

ОС строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в ОП или внешней памяти в дескрипторе отмечает его текущее местоположение (бит присутствия).

Дескриптор содержит поле адреса, с которого сегмент начинается и поле длины сегмента. Благодаря этому можно осуществлять контроль

- 1) размещения сегментов без наложения друг на друга
- 2) обращается ли код исполняющейся задачи за пределы текущего сегмента.

В дескрипторе содержатся также данные о правах доступа к сегменту (запрет на модификацию, можно ли его предоставлять другой задаче) и защита.

Достоинства сегментной организации памяти:

- 1) общий объем виртуальной памяти превосходит объем физической памяти
- 2) возможность размещать в памяти как можно больше задач (до определенного предела) увеличивает загрузку системы и более эффективно используются ресурсы системы

Недостатки:

1) увеличивается время на доступ к искомой ячейке памяти, т.к. должны вначале прочитать дескриптор сегмента, а потом уже, используя его данные, можно вычислить физический адрес (для уменьшения этих потерь используется кэширование - дескрипторы, с которыми работа идет в данный момент размещаются в сверхоперативной памяти - в специальных регистрах процессора);

- 2) фрагментация;
- 3) потери памяти на размещение дескрипторных таблиц
- 4) потери процессорного времени на обработку дескрипторных таблиц.

Страничная модель памяти – это надстройка над сегментной моделью.

ОП делится на блоки фиксированного размера 4 Кб (должно быть число, кратное степени двойки, чтобы операции сложения можно было бы заменить на операции конкатенации). Каждый такой блок называется страницей.

Основное применение этой модели связано с организацией виртуальной памяти.

Для того, чтобы использовать для работы программ пространство памяти большее, чем объем физической памяти используется механизм виртуальной памяти.

Суть его заключается в том, что у микропроцессора существует возможность по обмену страницами памяти с жестким диском. В случае, если программа требует памяти больше, чем объем физической памяти, редко используемые страницы памяти записываются на жесткий диск в специальный файл виртуальной памяти (файл обмена, или страничный файл, или файл подкачки, чаще swap-файлом,

подчеркивая, что страницы этого файла замещают друг друга в ОП). Файл подкачки может динамически изменять свой размер в зависимости от потребностей системы.

Программа также разбивается на фрагменты - страницы. Все фрагменты программы одинаковой длины, кроме последней страницы. Говорят, что память разбивается на физические страницы, а программа - на виртуальные страницы.

Трансляция (отображение) виртуального адресного пространства задачи на физическую память осуществляется с помощью таблицы страниц.

Для каждой текущей задачи создается таблица страниц.

Диспетчер памяти для каждой страницы формирует соответствующий дескриптор. Дескриптор содержит так называемый бит присутствия.

Если он = 1, это означает, что данная страница сейчас размещена в ОП.

Если он = 0, то страница расположена во внешней памяти.

Защита страничной памяти основана на контроле уровня доступа к каждой странице.

Каждая страница снабжается кодом уровня доступа (только чтение; чтение и запись; только выполнение). При работе со страницей сравнивается значение кода разрешенного уровня доступа с фактически требуемым. При несовпадении работа программы прерывается.

Страничная модель памяти поддерживается только в защищенном режиме работы микропроцессора.

Основное достоинство страничного способа распределения памяти - минимально возможная фрагментация (эффективное распределение памяти).

Недостатки:

- 1) потери памяти на размещение таблиц страниц
- 2) потери процессорного времени на обработку таблиц страниц (диспетчер памяти).
- 3) программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющих в коде (межстраничные переходы осуществляются чаще, чем межсегментные + трудности в организации разделения программных модулей между выполняющими процессами).

Сегментно-страничный способ распределения памяти

Программа разбивается на сегменты. Адрес, по-прежнему, состоит из двух частей: сегмент + смещение. Но смещение относительно начала сегмента может состоять из двух полей: виртуальной страницы и индекса.

Для доступа к памяти необходимо:

- 1) вычислить адрес дескриптора сегмента и прочитать его;
- 2) вычислить адрес элемента таблицы страниц этого сегмента и извлечь из памяти необходимый элемент;
- 3) к номеру (адресу) физической страницы приписать номер (адрес) ячейки в странице.

Задержка в доступе к памяти (в три раза больше, чем при прямой адресации).

Чтобы избежать этого вводится кэширование (кэш строится по ассоциативному принципу).

Плоская модель памяти

Если считать, что задача состоит из одного сегмента, который, в свою очередь, разбит на страницы, то фактически мы получаем только один страничный механизм работы с виртуальной памятью.

Это подход называется плоской памятью.

Достоинства:

- 1) При использовании плоской модели памяти упрощается создание и ОС, и систем программирования.
- 2) уменьшаются расходы памяти на поддержку системных информационных структур

В абсолютном большинстве современных 32-разрядных ОС (для микропроцессоров Intel) используется плоская модель памяти.

Задание модели памяти

Модели памяти задаются директивой `.MODEL`, которая определяет модель памяти, используемую программой. Директива должна находиться перед любой из директив объявления сегментов (`.DATA`, `.STACK`, `.CODE`, `SEGMENT`). Параметр `модель_памяти` является обязательным.

`.MODEL` [модификатор] модель_памяти [,соглашение_о_вызовах] [,тип_ОС] [,параметр_стека]

Основные модели памяти Ассемблера:

Таблица 3

Модель памяти	Адресация кода	Адресация данных	Операционная система	Чередование кода и данных
TINY	NEAR	NEAR	MS-DOS	Допустимо
SMALL	NEAR	NEAR	MS-DOS, Windows	Нет
MEDIUM	FAR	NEAR	MS-DOS, Windows	Нет
COMPACT	NEAR	FAR	MS-DOS, Windows	Нет
LARGE	FAR	FAR	MS-DOS, Windows	Нет
HUGE	FAR	FAR	MS-DOS, Windows	Нет
FLAT	NEAR	NEAR	Windows NT, Windows 2000, Windows XP, Windows Vista	Допустимо

Модель `small` поддерживает один сегмент кода и один сегмент данных. Данные и код при использовании этой модели адресуются как `near` (ближние).

Модель `large` поддерживает несколько сегментов кода и несколько сегментов данных. По умолчанию все ссылки на код и данные считаются дальними (`far`).

Модель `huge` практически эквивалентна модели памяти `large`.

Модель `medium` поддерживает несколько сегментов программного кода и один сегмент данных, при этом все ссылки в сегментах программного кода по умолчанию считаются дальними (`far`), а ссылки в сегменте данных — ближними (`near`).

Модель `compact` поддерживает несколько сегментов данных, в которых используется дальняя адресация данных (`far`), и один сегмент кода с ближней адресацией (`near`).

Модель `tiny` работает только в 16-разрядных приложениях MS-DOS. В этой модели все данные и код располагаются в одном физическом сегменте. Размер программного файла в этом случае не превышает 64 Кбайт.

Модель `flat` предполагает несегментированную конфигурацию программы и используется только в 32-разрядных операционных системах. Эта модель подобна модели `tiny` в том смысле, что данные и код размещены в одном сегменте, только 32-разрядном. Для разработки программы для модели `flat` перед директивой `.model flat` следует разместить одну из директив: `.386`, `.486`, `.586` или `.686`. Желательно указывать тот тип процессора, который используется в машине, хотя на машинах с Intel Pentium можно указывать директивы `.386` и `.486`. Операционная система автоматически инициализирует сегментные регистры при загрузке программы, поэтому модифицировать их нужно, только если необходимо смешивать в одной программе 16- и 32-разрядный код. Адресация данных и кода является ближней (`near`), при этом все адреса и указатели являются 32-разрядными.

Параметр модификатор используется для определения типов сегментов и может принимать значения `use16` (сегменты выбранной модели используются как 16-битные) или `use32` (сегменты выбранной модели используются как 32-битные).

Параметр `соглашение_о_вызовах` используется для определения способа передачи параметров при вызове процедуры из других языков, в том числе и языков высокого уровня (C++, Pascal). Параметр может принимать следующие значения: `C`, `BASIC`, `FORTTRAN`, `PASCAL`, `SYSCALL`, `STDCALL`. При разработке модулей на ассемблере, которые будут применяться в программах, написанных на языках высокого уровня, обращайтесь внимание на то, какие соглашения о вызовах поддерживает тот или иной язык. Более подробно соглашения о вызовах мы будем рассматривать при анализе интерфейса программ на ассемблере с программами на языках высокого уровня.

Параметр `тип_ОС` равен `OS_DOS` по умолчанию, и на данный момент это единственное поддерживаемое значение этого параметра.

Параметр `параметр_стека` устанавливается равным `NEARSTACK` (регистр `SS` равен `DS`, области данных и стека размещаются в одном и том же физическом сегменте) или `FARSTACK` (регистр `SS` не равен `DS`, области данных и стека размещаются в разных физических сегментах). По умолчанию принимается значение `NEARSTACK`.

СТРУКТУРА ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Формат кодирования в языке Ассемблера

Программирование на уровне машинных команд — это тот минимальный уровень, на котором еще возможно программирование компьютера. Система машинных команд должна быть достаточной для того, чтобы реализовать требуемые действия, выдавая указания аппаратуре машины. Каждая машинная команда состоит из двух частей: *операционной*, определяющей «что делать» и *операндной*, определяющей объекты обработки, то есть то «над чем делать».

Машинная команда микропроцессора, записанная на языке Ассемблера, представляет собой одну строку, имеющую следующий вид:

```
[метка]      команда/директива      [операнд(ы)] ; комментарии
```

Метка (если имеется), команда/директива и операнд (если имеется) разделяются по крайней мере одним символом пробела или табуляции. Если команду или директиву необходимо продолжить на следующей строке, то используется символ «обратный слеш»: «\». По умолчанию Ассемблер не различает заглавные и строчные буквы.

Примеры кодирования:

```
Count db 1          ;Имя, директива, один операнд
        mov  eax,0   ;Команда, два операнда
        cbw          ; Команда
```

Метка в языке ассемблера может содержать следующие символы:

- все буквы латинского алфавита;
- цифры от 0 до 9;
- спецсимволы: `_`, `@`, `$`, `?`.

В качестве первого символа метки может использоваться точка, но некоторые компиляторы не рекомендуют применять этот знак. В качестве меток нельзя использовать зарезервированные имена Ассемблера (директивы, операторы, имена команд).

Первым символом в метке должна быть буква или спецсимвол (но не цифра). Максимальная длина метки – 31 символ. Все метки, которые записываются в строке, не содержащей директиву ассемблера, должны заканчиваться *двоеточием* «:».

Команда указывает транслятору, какое действие должен выполнить микропроцессор. В сегменте данных команда (или директива) определяет поле, рабочую область или константу. В сегменте кода команда определяет действие, например, пересылка (mov) или сложение (add).

Ассемблер имеет ряд операторов, которые позволяют управлять процессом ассемблирования и формирования листинга. Эти операторы называются **директивами**. Они действуют только в процессе ассемблирования программы и, в отличие от команд, не генерируют машинных кодов.

Операнд определяет начальное значение данных (в сегменте данных) или элементы, над которыми выполняется действие по команде (в сегменте кода).

Команда может иметь один или два операнда, или вообще не иметь операндов. Число операндов неявно задается кодом команды.

Примеры:

Нет операндов	ret	;Вернуться
Один операнд	inc cx	;Увеличить cx
Два операнда	add ax,12	;Прибавить 12 к ax

Метка, команда (директива) и операнд не обязательно должны начинаться с какой-либо определенной позиции в строке. Однако рекомендуется записывать их в колонку для большей удобочитаемости программы.

Использование **комментариев** в программе улучшает ее ясность, особенно там, где назначение набора команд непонятно. Комментарии начинаются на любой строке исходного модуля с символа «точка с запятой» (;), и Ассемблер полагает в этом случае, что все символы, находящиеся справа от «;» до конца строки, являются комментарием. Комментарий может содержать любые печатные символы, включая «пробел». Комментарий может занимать всю строку или следовать за командой на той же строке.

Предложения ассемблера формируются из лексем, представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора. Лексемами являются:

- **Идентификаторы** – последовательности допустимых символов, используемые для обозначения таких объектов программы, как коды операций, имена переменных и названия меток. Правило записи идентификаторов заключается в следующем. Идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки: `_`, `?`, `$`, `@`. Идентификатор не может начинаться символом цифры. Длина идентификатора может быть до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные игнорирует.
- **Цепочки символов** – последовательности символов, заключенные в одинарные или двойные кавычки;
- **Целые числа** в одной из следующих систем счисления: двоичной, десятичной, восьмеричной, шестнадцатеричной.

Структура программы на ассемблере

Программа, написанная на ассемблере MASM, может состоять из нескольких частей, называемых модулями, в каждом из которых могут быть определены один или несколько сегментов данных, стека и кода. Любая законченная программа на ассемблере должна включать один главный, или основной, модуль, с которого начинается ее выполнение. Модуль может содержать программные сегменты, сегменты данных и стека, объявленные при помощи соответствующих директив. Кроме того, перед объявлением сегментов нужно указать модель памяти при помощи директивы `.MODEL`.

Пример «ничего не делающей» программы:

```
; prog.asm
.686P
.MODEL FLAT, STDCALL
.DATA
.CODE
START:
    RET
END START
```

В данной программе представлена всего одна команда микропроцессора. Эта команда RET. Она обеспечивает правильное окончание работы программы. В общем случае эта команда используется для выхода из процедуры.

Остальная часть программы относится к работе транслятора.

.686P - разрешены команды защищенного режима Pentium 6 (Pentium II).

Данная директива выбирает поддерживаемый набор команд ассемблера, указывая модель процессора. Буква P, указанная в конце директивы, сообщает транслятору о работе процессора в защищенном режиме.

.MODEL FLAT, stdcall - плоская модель памяти. Эта модель памяти используется в операционной системе Windows. stdcall - используемое соглашение о вызовах процедур.

.DATA - блок программы, содержащий данные. В MS-DOS это называлось сегментом. В Windows, для прикладной программы, сегменты отсутствуют, точнее есть один большой плоский сегмент. Такие блоки здесь называются **секциями**. Секции можно задать различные свойства. Например, запретить запись в нее, или сделать секцию доступной для других программ, используя стандартные директивы сегментации.

Данная программа не использует стек, поэтому секция .STACK отсутствует.

.CODE - блок программы, содержащей код.

START - метка. В ассемблере метки играют большую роль, что не скажешь о современных языках высокого уровня.

END START - конец программы и сообщение компилятору, что начинать выполнение программы надо с метки START. Каждая программа должна содержать директиву END, отмечающую конец исходного кода программы. Все строки, которые следуют за директивой END, игнорируются. Если вы опустите директиву END, то генерируется ошибка. Метка, указанная после директивы END, сообщает транслятору имя главного модуля, с которого начинается выполнение программы. Если программа содержит один модуль, метку после директивы END можно не указывать.

Чтобы представленный текст превратить в исполняемый модуль, будем использовать пакет MASM32 8.0.

```
ML /c /coff prog.asm  
LINK /subsystem:windows prog.obj
```

В результате выполнения этих строк появится исполняемый модуль prog.exe. Правда, выполнение его не даст никакого эффекта.

Простейшая программа в ОС Windows

Операционная система Windows является многозадачной средой. Каждая задача имеет свое адресное пространство и свою очередь сообщений. Более того, даже в рамках одной программы может быть осуществлена многозадачность - любая процедура может быть запущена как самостоятельная задача.

Программирование в Windows основывается на использовании функций API (Application Program Interface, т.е. интерфейс программного приложения). Их количество достигает двух тысяч. Программа для Windows в значительной степени состоит из таких вызовов. Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций.

Операционная система Windows использует плоскую модель памяти. Другими словами, всю память можно рассматривать как один сегмент. Для программиста на языке ассемблера это означает, что адрес любой ячейки памяти будет определяться содержимым одного 32-битного регистра, например EBX. Следовательно, мы фактически не ограничены в объеме данных, кода или стека (объеме локальных переменных). Выделение в тексте программы сегмента кода и сегмента данных является теперь простой формальностью, улучшающей читаемость программы.

Вызов функций API. В файле помощи любая функция API представлена в виде (например):

```
int MessageBox ( HWND hWnd, LPCTSTR lpText,  
                LPCTSTR lpCaption, UINT uType );
```

Данная функция выводит на экран окно с сообщением и кнопкой (или кнопками) выхода. Смысл параметров:

hWnd - дескриптор окна, в котором будет появляться окно-сообщение,

lpText - текст, который будет появляться в окне,

lpCaption - текст в заголовке окна,

uType - тип окна, в частности можно определить количество кнопок выхода.

Теперь о типах параметров. Практически все параметры API-функций в действительности 32-битные целые числа: HWND — 32-битное целое, LPCTSTR — 32-битный указатель на строку, UINT — 32-битное целое. К имени функций часто добавляется суффикс "A" для перехода к более новым версиям функций. Кроме того, при использовании MASM необходимо также в конце имени добавить @16 – количество байт, которое занимают в стеке переданные аргументы. Для функций Win32 API это число можно определить как количество аргументов n, умноженное на 4 (байта в каждом аргументе): 4*n. Для вызова функции используется команда CALL ассемблера. При этом все аргументы функции передаются в нее через стек (команда PUSH). Направление передачи аргументов: СЛЕВА НАПРАВО — СНИЗУ ВВЕРХ. В соответствии с этим, первым будет помещаться в стек аргумент uType. Таким образом, вызов указанной функции будет выглядеть так:

```
CALL MessageBoxA@16.
```

Результат выполнения любой API функции — это, как правило, целое число, которое возвращается в регистре EAX.

```
.686P .MODEL FLAT, STDCALL
.STACK 4096
.DATA
MB_OK EQU 0
STR1 DB "Моя первая программа",0
STR2 DB "Привет всем!",0
HW DD ?
EXTERN MessageBoxA@16:NEAR
.CODE
START:
PUSH MB_OK
PUSH OFFSET STR1
PUSH OFFSET STR2
PUSH HW
CALL MessageBoxA@16
RET
END START

.686P .MODEL FLAT, STDCALL
;include C:\masm32\include\user32.inc
MessageBoxA PROTO :DWORD, :DWORD, :DWORD, :DWORD
.STACK 4096
.DATA
MB_OK EQU 0
STR1 DB "Моя первая программа",0
STR2 DB "Привет всем!",0
HW DD ?
.CODE
START:
INVOKE MessageBoxA, HW,
OFFSET STR2, OFFSET STR1, MB_OK
RET
END START
```

Директива OFFSET представляет собой «смещение в секции», или, переводя в понятия языков высокого уровня, «указатель» начала строки.

Директива EQU подобно #define в языке СИ определяет константу.

Транслятор языка MASM позволяет также упростить вызов функций с использованием макросредства – директивы INVOKE:

INVOKE функция, параметр1, параметр2,...

В этом случае нет необходимости добавлять @16 к вызову функции. Кроме того, параметры записываются точно в том порядке, в котором приведены в описании функции. Макросредствами транслятора параметры помещаются в стек. Для использования директивы INVOKE необходимо иметь описание прототипа функции с использованием директивы PROTO в виде:

```
MessageBoxA PROTO :DWORD, :DWORD, :DWORD, :DWORD
```

Если в программе используется множество функций Win32 API, целесообразно воспользоваться директивой

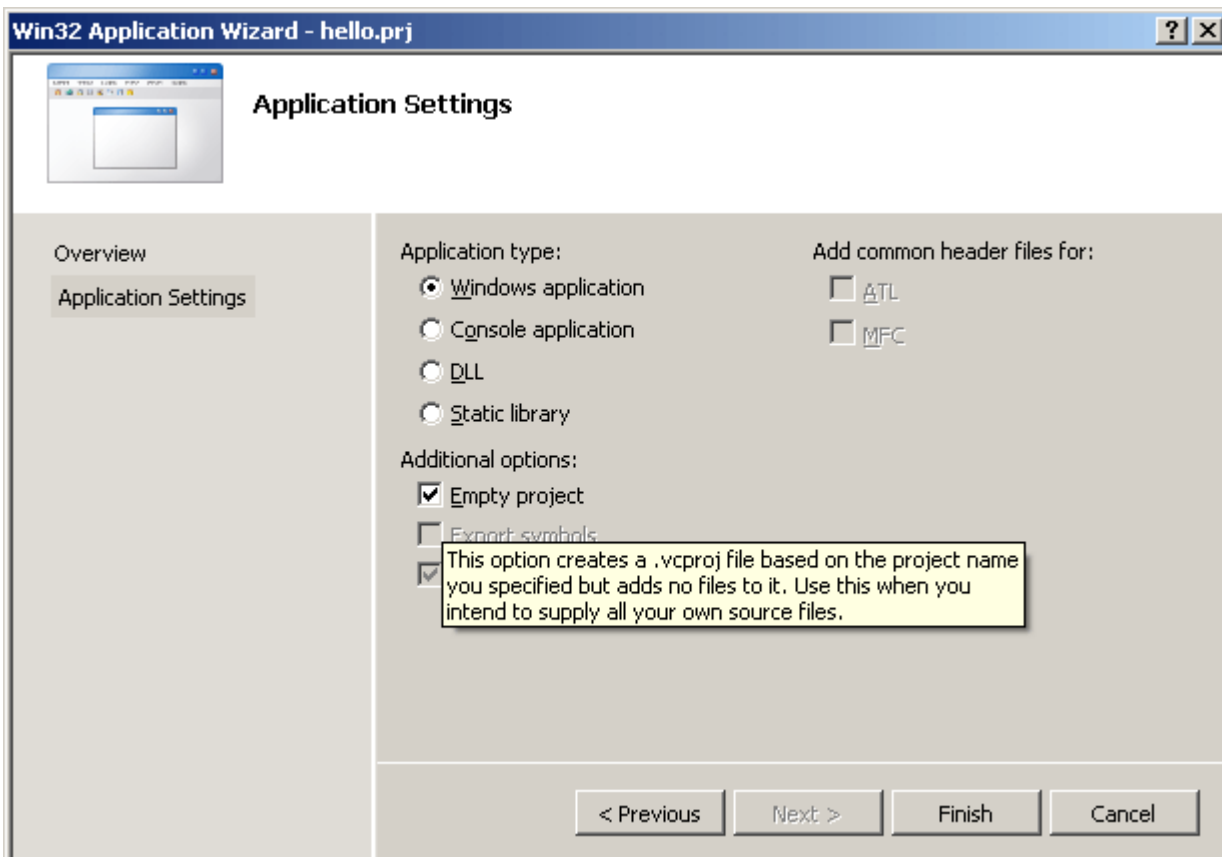
```
include C:\masm32\include\user32.inc
```

Функция MessageBoxA вызывается из системной библиотеки Windows user32.dll.

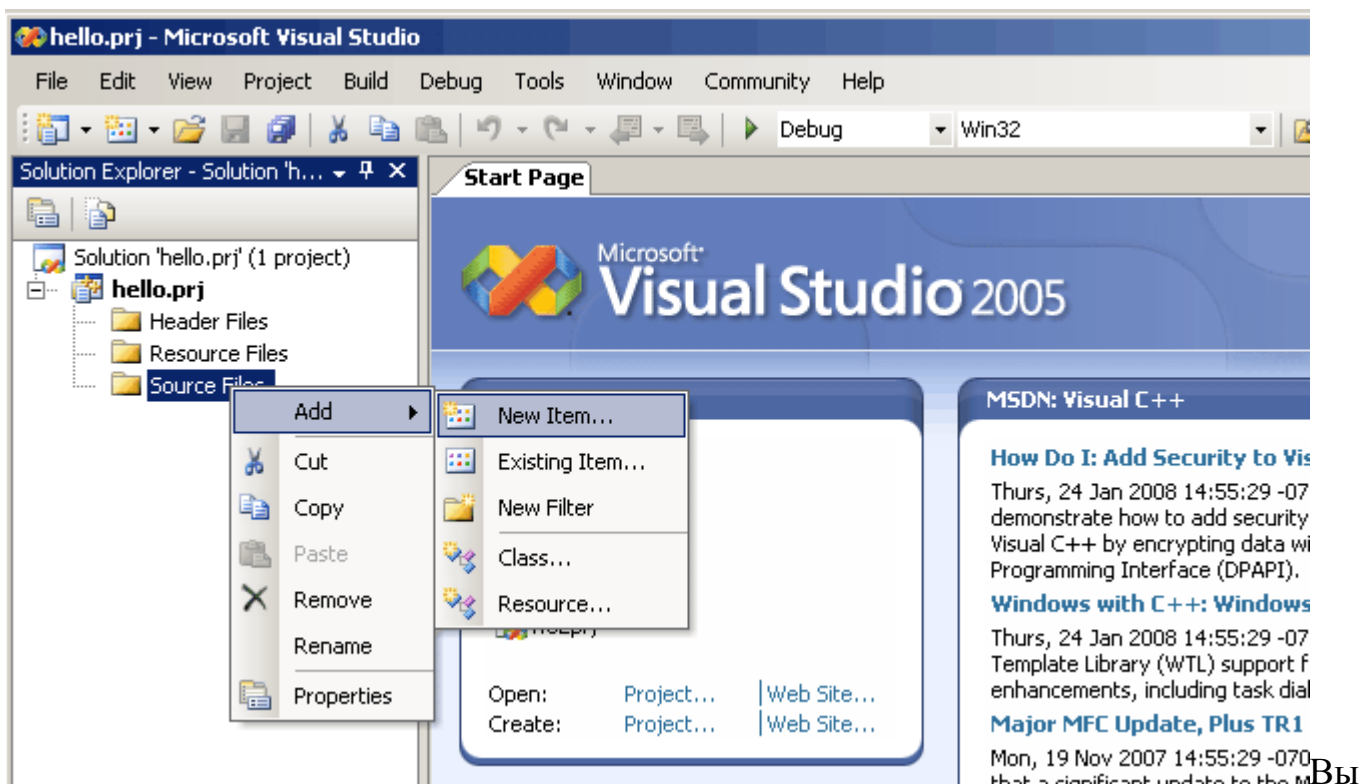
include-файлы MASM с соответствующими названиями библиотек содержат описания прототипов всех функций данной библиотеки.

Для трансляции приведенного выше текста программы в исполнимый модуль можно воспользоваться компилятором Microsoft Visual Studio. Для этого необходимо:

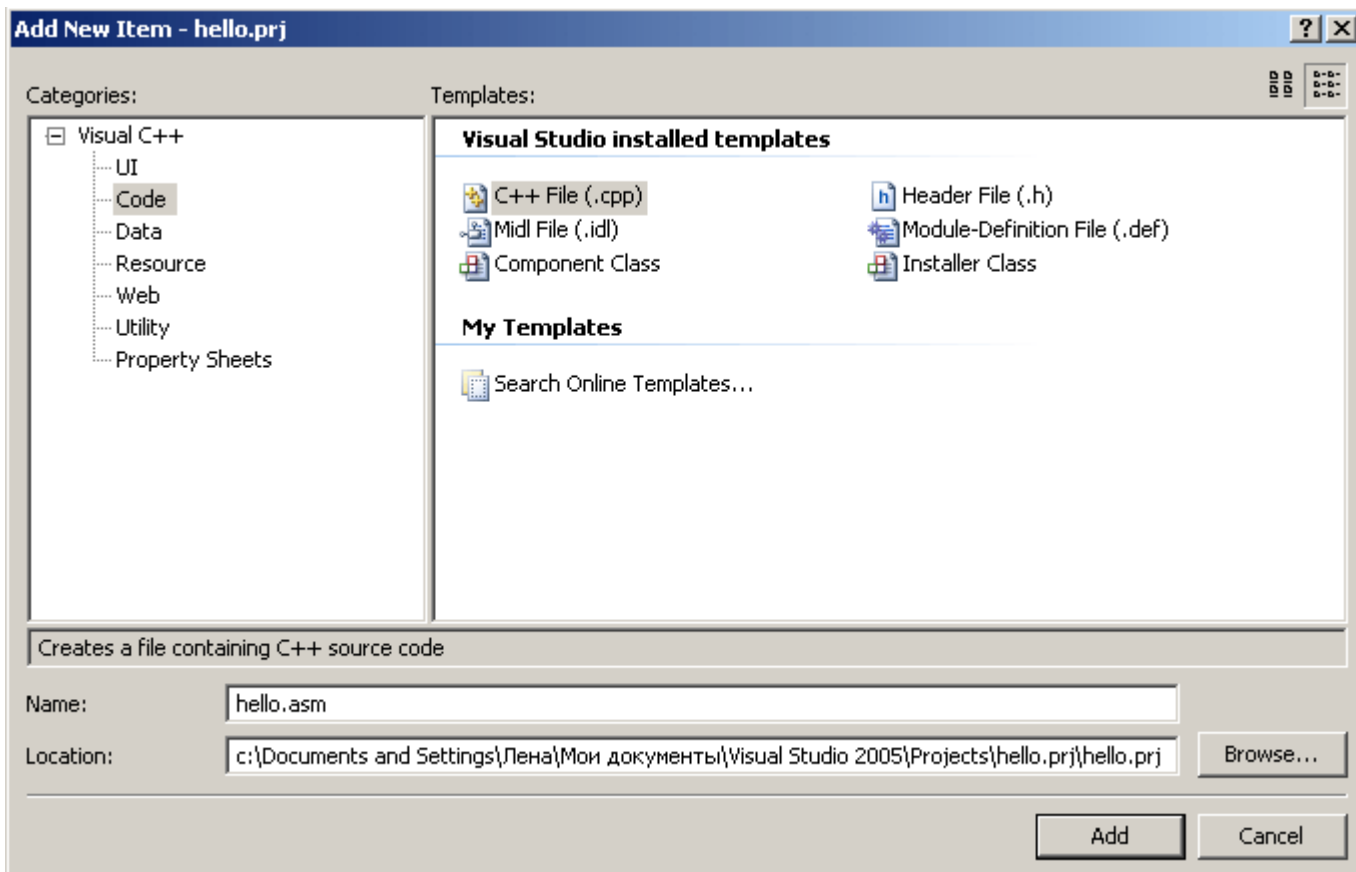
- 1) Создать проект, выбрав меню File->New->Project и указав имя проекта (hello.prj) и тип проекта: Win32 Project. В дополнительных опциях мастера проекта указать "Empty Project".



2) В дереве проекта (View->Solution Explorer) добавить файл, в котором будет содержаться текст программы: SourceFiles->Add->NewItem



Выбрать тип файла Code C++, но указать имя с расширением .asm:

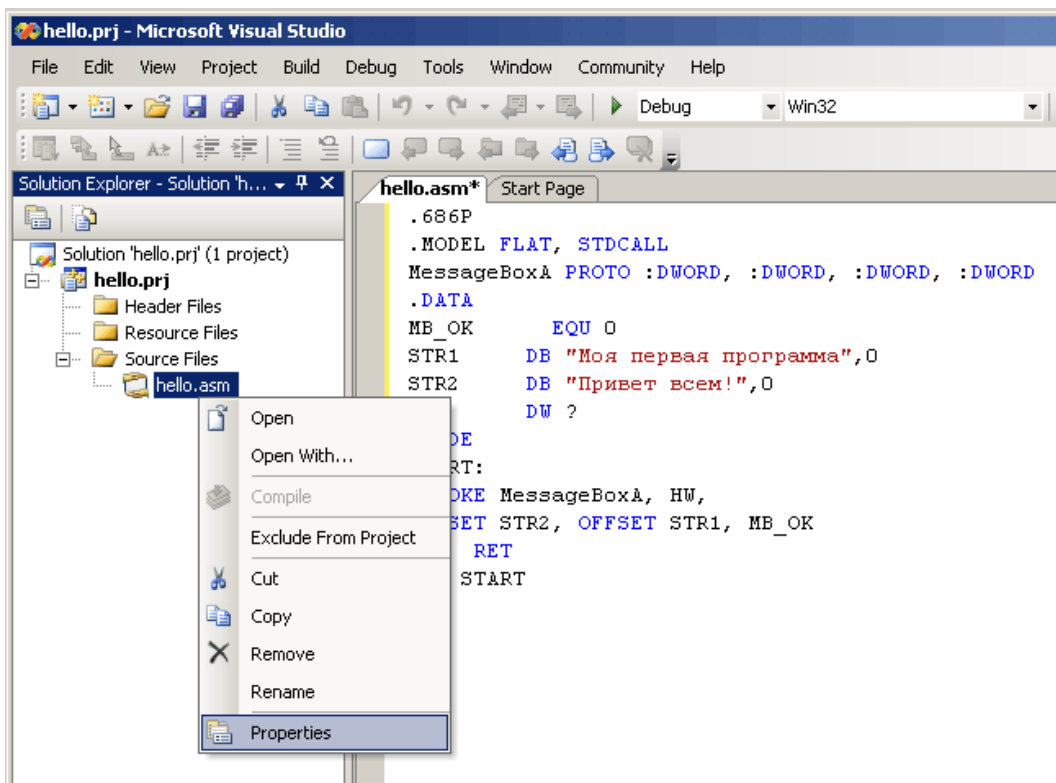


3) В появившемся окне набрать текст программы.

4) Установить параметры компилятора. Для этого

1 СПОСОБ: выбрать по правой кнопке в файле hello.asm дерева проекта меню

Properties.



В появившемся диалоговом окне выбрать Custom Build Step и прописать в правой части окна Command Line в виде:

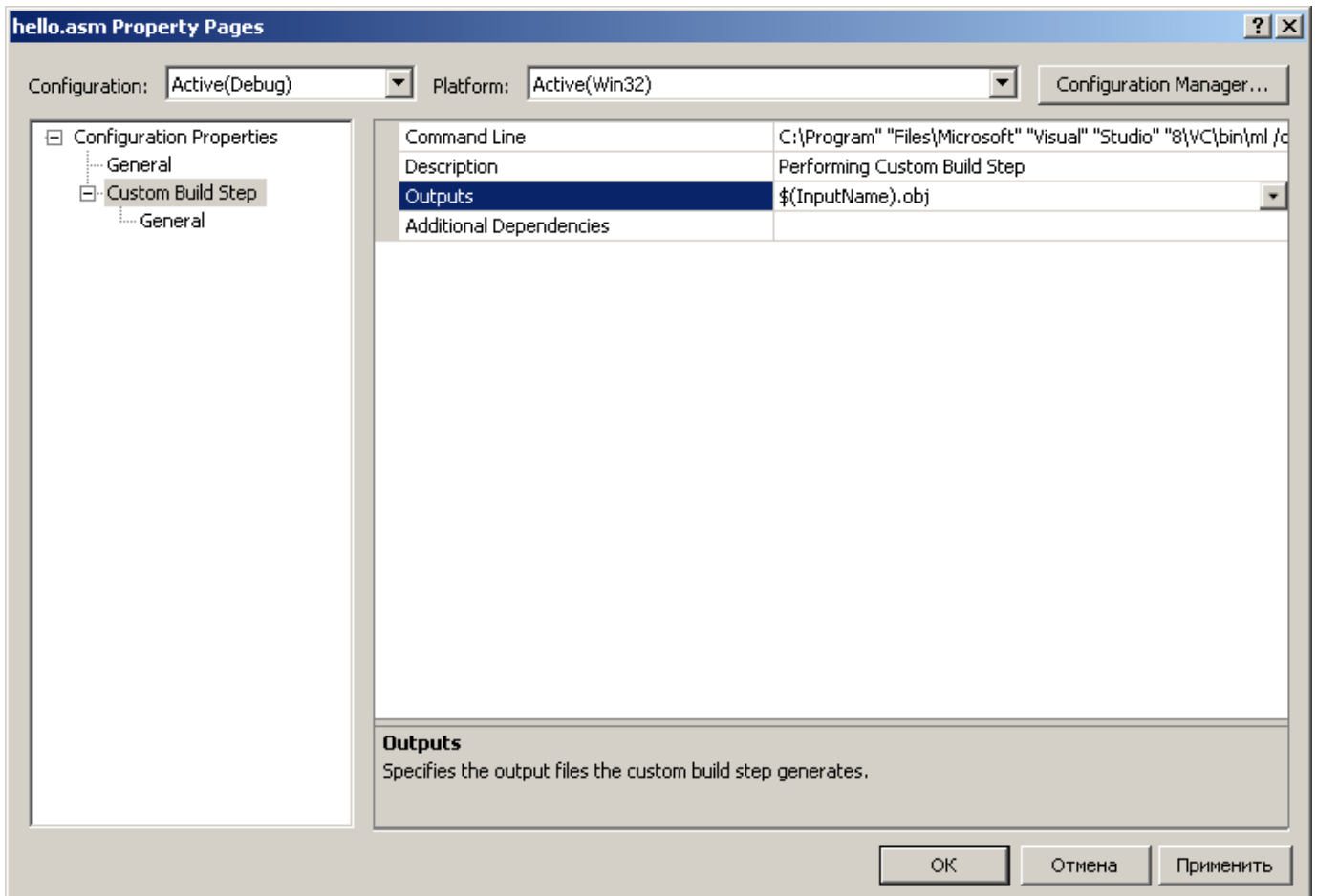
```
C:\путь\ml /c /coff /Zi $(InputName).asm
```

В указании пути все пробелы должны быть взяты в кавычки:

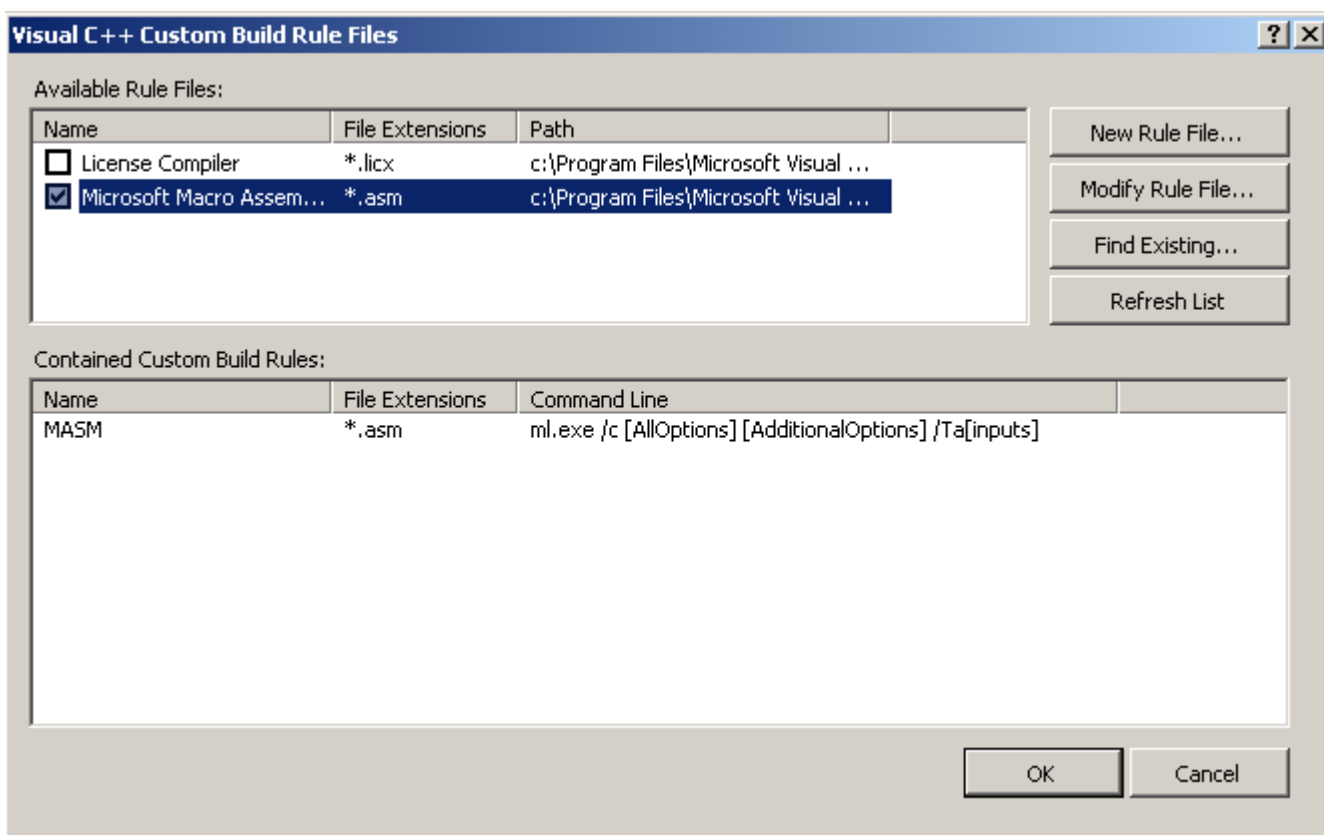
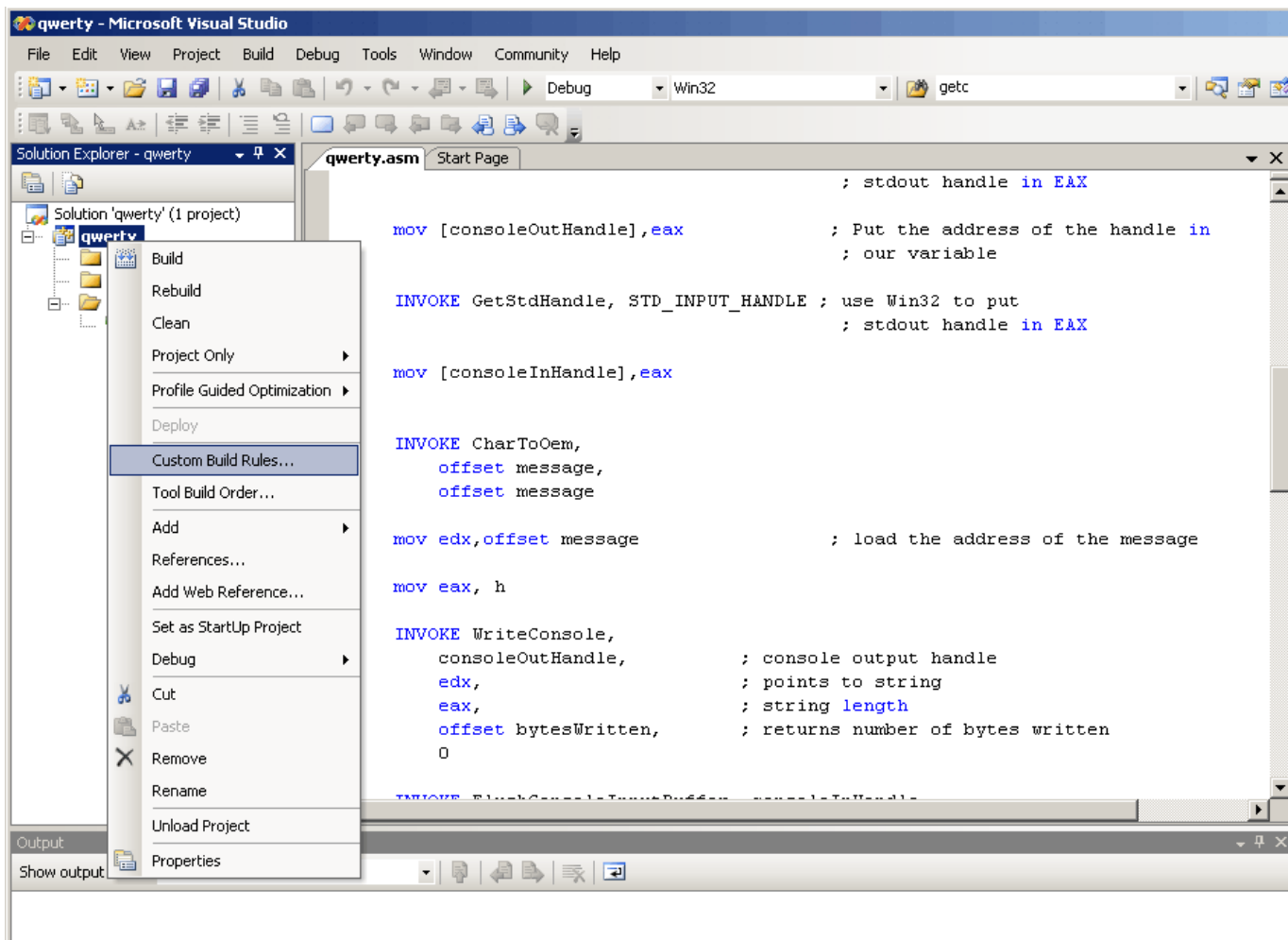
```
C:\Program "Files\Microsoft" "Visual" "Studio" "8\VC\bin\ml  
/c /coff /Zi $(InputName).asm
```

В строке Outputs указать:

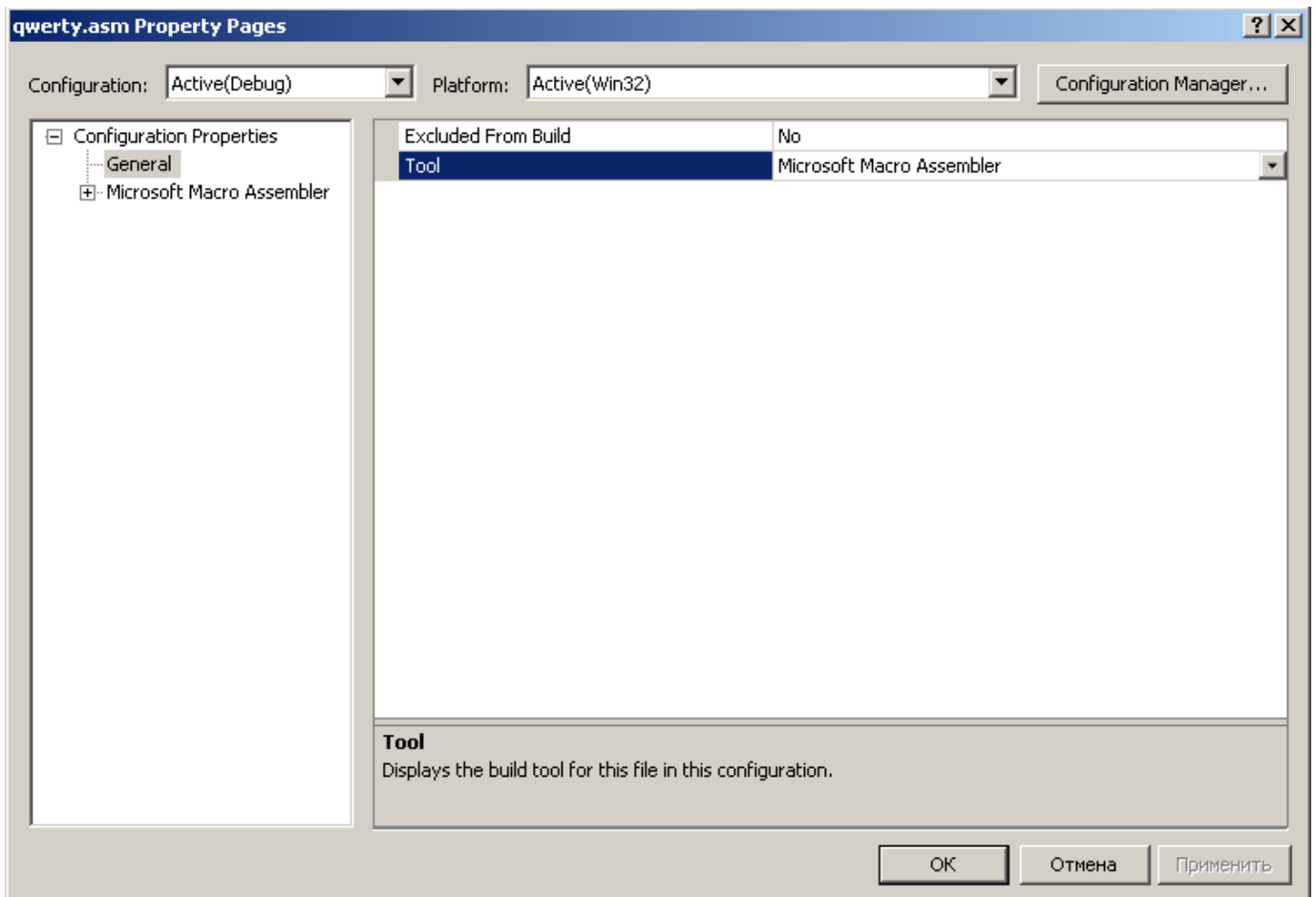
```
$(InputName).obj
```



2 СПОСОБ: выбрать по правой кнопке в файле проекта меню Custom Build Rules... и в появившемся окне выбрать Microsoft Macro Assembler.



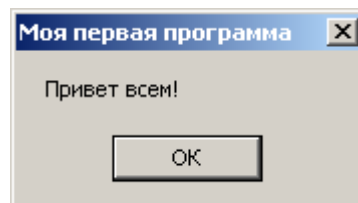
выбрать по правой кнопке в файле hello.asm дерева проекта меню Properties и установить General->Tool: Microsoft Macro Assembler.



5) Откомпилировать файл, выбрав Build->Build hello.prj.

6) Запустить программу, нажав F5 или выбрав меню Debug->Start Debugging.

В результате появится окно сообщения:



ТИПЫ И ФОРМАТЫ ДАННЫХ В АССЕМБЛЕРЕ

Данные – числа и закодированные символы, используемые в качестве операндов команд.

Основными типами данных процессора с архитектурой IA-32 являются байт, слово, двойное слово, четырехбайтное слово и восьмибайтное слово.

Тип данных	Директива	Кол-во байт	Способ представления				
Байт	DB	1	<div style="display: flex; justify-content: space-between; width: 100px;"> 7 0 </div> <div style="border: 1px solid black; width: 100%; height: 15px; margin: 2px 0;"></div> <div style="text-align: center;">N</div>				
Слово	DW	2	<div style="display: flex; justify-content: space-between; width: 100px;"> 15 8 7 0 </div> <table border="1" style="margin: 2px auto; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50px;">старший байт</td> <td style="text-align: center; width: 50px;">младший байт</td> </tr> <tr> <td style="text-align: center;">N+1</td> <td style="text-align: center;">N</td> </tr> </table>	старший байт	младший байт	N+1	N
старший байт	младший байт						
N+1	N						
Двойное слово	DD	4	<div style="display: flex; justify-content: space-between; width: 100px;"> 31 16 15 0 </div> <table border="1" style="margin: 2px auto; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50px;">старшее слово</td> <td style="text-align: center; width: 50px;">младшее слово</td> </tr> <tr> <td style="text-align: center;">N+3</td> <td style="text-align: center;">N</td> </tr> </table>	старшее слово	младшее слово	N+3	N
старшее слово	младшее слово						
N+3	N						
4 слова	DQ	8	<div style="display: flex; justify-content: space-between; width: 100px;"> 63 32 31 0 </div> <table border="1" style="margin: 2px auto; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50px;">старшее двойное слово</td> <td style="text-align: center; width: 50px;">младшее двойное слово</td> </tr> <tr> <td style="text-align: center;">N+7</td> <td style="text-align: center;">N</td> </tr> </table>	старшее двойное слово	младшее двойное слово	N+7	N
старшее двойное слово	младшее двойное слово						
N+7	N						
10 байт	DT	10	80 бит				


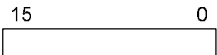
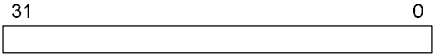
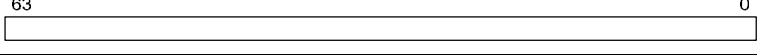
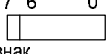
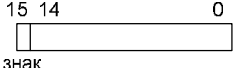
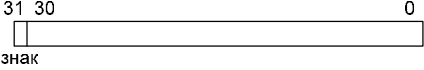
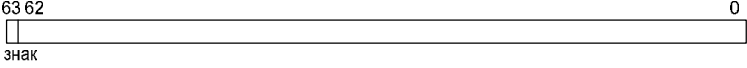
Данные, обрабатываемые вычислительной машиной, можно разделить на 4 группы:

- целочисленные;
- вещественные.
- символные;
- логические;

Целочисленные данные могут представляться в знаковой и беззнаковой форме.

Беззнаковые целые числа представляются в виде последовательности битов в диапазоне от 0 до 2^n , где n-количество занимаемых битов.

Знаковые целые числа представляются в диапазоне $-2^{n-1} \dots +2^{n-1}$. При этом старший бит данного отводится под знак числа (0 соответствует положительному числу, 1 – отрицательному).

Тип	Диапазон изменения	Форма представления	Тип в Си
беззнаковое целое (байт)	0...255		unsigned char
беззнаковое целое (слово)	0...65535		unsigned int
беззнаковое целое (4-байт)	0...4294967295		unsigned long int
беззнаковое целое (8-байт)	0...1,84467e+19		
знаковое целое (байт)	-128...127		char
знаковое целое (слово)	-32768...32767		int
знаковое целое (4-байт)	-2147483648...2147483647		long int
знаковое целое (8-байт)	-9,22e+18...9,22e+18		

Тип DT может быть только вещественным.

Вещественные данные обрабатываются сопроцессором и будут рассмотрены ниже.

Логические данные представляют собой бит информации и могут записываться в виде последовательности битов. Каждый бит может принимать значение 0 или 1 (ЛОЖЬ или ИСТИНА). Логические данные могут начинаться с любой позиции в байте.

Символьные данные задаются в кодах. **Кодировка символов** (часто называемая также кодовой страницей) – это набор числовых значений, которые ставятся в соответствие группе алфавитно-цифровых символов, знаков пунктуации и специальных символов. В Windows первые 128 символов всех кодовых страниц состоят из базовой таблицы символов ASCII (American Standard Code for Interchange of Information). Первые 32 кода базовой таблицы, начиная с нулевого, размещают управляющие коды. Символы с номерами от 128 до 255 представляют дополнительные символы и варьируются в зависимости от набора скриптов, представленных кодировкой символов. Основные таблицы кодировки представлены в приложении 1.

Для того чтобы полноценно поддерживать помимо английского и другие языки, фирма IBM ввела в употребление несколько кодовых таблиц, ориентированных на

конкретные страны. Так для скандинавских стран была предложена таблица 865 (Nordic), для арабских стран - таблица 864 (Arabic), для Израиля - таблица 862 (Israel) и так далее. В этих таблицах часть кодов из второй половины кодовой таблицы использовалась для представления символов национальных алфавитов (за счет исключения некоторых символов псевдографики).

С русским языком ситуация развивалась особым образом. Очевидно, что замену символов во второй половине кодовой таблицы можно произвести разными способами. Вот и появились для русского языка несколько разных таблиц кодировки символов кириллицы: KOI8-R, IBM-866, CP-1251, ISO-8551-5. Все они одинаково изображают символы первой половины таблицы (от 0 до 127) и различаются представлением символов русского алфавита и псевдографики.

Для таких же языков, как китайский или японский, вообще 256 символов недостаточно. Кроме того, всегда существует проблема вывода или сохранения в одном файле одновременно текстов на разных языках (например, при цитировании). Поэтому была разработана универсальная кодовая таблица UNICODE, содержащая символы, применяемые в языках всех народов мира, а также различные служебные и вспомогательные символы (знаки препинания, математические и технические символы, стрелки, диакритические знаки и т.д.). Очевидно, что одного байта недостаточно для кодирования такого большого множества символов. Поэтому в UNICODE используются 16-битовые (2-байтовые) коды, что позволяет представить 65536 символов. К настоящему времени задействовано около 49000 кодов (последнее значительное изменение - введение символа валюты EURO в сентябре 1998 г.).

Для совместимости с предыдущими кодировками первые 256 кодов совпадают со стандартом ASCII.

Для перекодировки символов и русскоязычного вывода в окно консоли можно использовать функцию Windows API

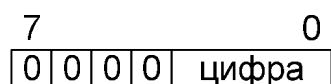
```
BOOL CharToOem(LPCTSTR lpszSrc, LPSTR lpszDst);
```

`lpszSrc` – указатель на строку-источник

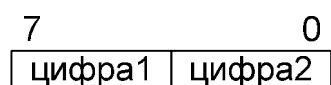
`lpszDst` – указатель на строку-приемник.

Числа в двоично-десятичном формате

В двоично-десятичном коде представляются беззнаковые целые числа, кодирующие цифры от 0 до 9. Числа в двоично-десятичном формате могут использоваться в упакованном и неупакованном виде. В случае неупакованных чисел в каждом байте хранится одна цифра, размещенная в младшей половине байта (биты 3...0). Упакованный вид допускает хранение двух десятичных цифр в одном байте, причем старшая половина байта отводится под старший разряд.



– неупакованный вид;



старший младший разряды

– упакованный вид

Как и в языках высокого уровня (СИ, Паскаль), в Ассемблере в качестве данных могут выступать константы и переменные.

Числовые константы используются для обозначения арифметических операндов и адресов памяти. Для числовых констант в Ассемблере могут использоваться следующие числовые форматы.

Десятичный формат – допускает использование десятичных цифр от 0 до 9 и обозначается последней буквой *d*, которую можно не указывать, например, 125 или 125*d*. Ассемблер сам преобразует значения в десятичном формате в объектный шестнадцатеричный код и записывает байты в обратной последовательности.

Шестнадцатеричный формат – допускает использование шестнадцатеричных цифр от 0 до *F* и обозначается последней буквой *h*, например 7*Dh*. Так как ассемблер полагает, что с буквы начинаются идентификаторы, то первым символом шестнадцатеричной константы должна быть цифра от 0 до 9. Например, 0*Eh*. Ассемблер сам записывает байты в обратной последовательности.

Двоичный формат – допускает использование цифр 0 и 1 и обозначается последней буквой *b*. Двоичный формат обычно используется для более четкого представления битовых значений в логических командах (AND, OR, XOR).

Восьмеричный формат – допускает использование цифр от 0 до 7 и обозначается последней буквой *q* или *o*, например, 253*q*.

Десятичное 12, шестнадцатеричное 0Ch, восьмеричное 14q и двоичное 1100b генерируют один и тот же код. Отрицательные числа ассемблер сам представляет в виде дополнительного кода.

Примеры:

```
_bb    DB    00001100b    ; _bb указывает на байт памяти
                               ; с двоичным значением 00001100
_ww_   DW    1234o       ; _ww указывает на слово памяти
                               ; с восьмеричным значением 1234
_dd_   DD    12345678h    ; _dd указывает на двойное слово
                               ; памяти со значением 12345678h
```

Вещественные числа задаются с помощью директив DD, DQ и DT.

Примеры:

```
_dd_   DD    5.65
_dq_   DQ    -122.87
_dt_   DT    168.987E-21
```

Для инициализации **массивов (цепочек)** можно использовать следующие строки

```
My_array DW 0,1,2,3,4,5,6,7,8,9
```

или

```
My_array DW 0,1,2,3
          DW 4,5,6,7
          DW 8,9
```

Данная строка выделяет десять последовательных слов памяти и записывает в них значения 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Метка My_array определяет смещение начала этой области в сегменте .DATA.

Для инициализации блоков памяти одними и теми же значениями можно использовать оператор DUP. Например,

```
Block_array DW 100 DUP (12h)
```

В тех случаях, когда необходимо выделить память, но не инициализировать ее, используется знак ?. Например,

```
no_init    DD    ?
massiv     DW    20 DUP (?)
```

Символьные строки представляют собой набор символов для вывода на экран. Содержимое строки отмечается одиночными кавычками, например, 'pc' или двойными кавычками ("pc"). Ассемблер переводит символьные строки в объектный

код в обычном формате текущей кодовой страницы. Символьная строка определяется только директивой DB, в которой указывается более двух символов в нормальной последовательности слева направо. Символьная строка должна заканчиваться символом '\0' (для системы MS-DOS строка заканчивается символом '\$'). Для перевода строки могут использоваться символы 13 (возврат каретки 0Dh) и 10 (перевод строки 0Ah).

```
Stroka DB 'Привет',13,10,0
```

При записи символьных констант следует помнить, что символьная константа, определенная как DB '12', представляет собой символы ASCII и генерирует шестнадцатеричный код 3132h, а числовая константа, определенная как DB 12, представляет двоичное число и генерирует шестнадцатеричный код 0Ch.

В кодовой странице ASCII (UNICODE) символы цифр '0'... '9' соответствуют значениям кодов 30h...39h соответственно. Используя это, можно легко переводить числа, вводимые с клавиатуры, в эквивалент их целочисленного значения.

БАЗОВАЯ СИСТЕМА КОМАНД МИКРОПРОЦЕССОРА IA-32

Базовую систему команд микропроцессора можно условно разделить на несколько групп по функциональному назначению (см. табл.8).

- команды передачи данных
- команды работы со стеком
- команды ввода-вывода
- арифметические команды
- логические команды
- сдвиговые команды
- команды управления флагами
- команды прерываний
- команды передачи управления
- команды поддержки языков высокого уровня
- команды синхронизации работы процессора
- команды обработки цепочки бит
- строковые команды
- команды управления защитой (не рассматриваются)
- команды управления кэшированием (не рассматриваются)

Кроме базовой системы команд процессора существуют также команды расширений:

X87 – расширение, содержащее команды математического сопроцессора (работа с вещественными числами)

MMX – расширение, содержащее команды для кодирования/декодирования потоковых аудио/видео данных;

SSE – расширение включает в себя набор инструкций, который производит операции со скалярными и упакованными типами данных;

SSE2 – модификация SSE, содержит инструкции для потоковой обработки целочисленных данных, что делает это расширение более предпочтительным для целочисленных вычислений, нежели использование набора инструкций MMX, появившегося гораздо раньше;

SSE3, SSE4 – содержат дополнительные инструкции расширения SSE.

В таблице приняты следующие обозначения

r – регистр

m – ячейка памяти

c – константа

8, 16, 32 – размер в битах

W – запись бита в регистре признаков

U – значение бита в регистре признаков неизвестно после выполнения операции

– – значение бита в регистре признаков не изменилось

+ – значение бита в регистре признаков меняется в соответствии с результатом

На все базовые команды процессора накладываются следующие ограничения:

1. Нельзя в одной команде оперировать двумя областями памяти одновременно. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения.
2. Нельзя оперировать сегментным регистром и значением непосредственно из памяти. Поэтому для выполнения такой операции нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек.
3. Нельзя оперировать двумя сегментными регистрами. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных регистры общего назначения. Например,

```
mov ax, ds  
mov es, ax ; es=ds
```
4. Нельзя использовать сегментный регистр CS в качестве операнда приемника, поскольку в архитектуре микропроцессора пара CS:EIP всегда содержит адрес команды, которая должна выполняться следующей. Изменение содержимого регистра CS фактически означало бы операцию перехода, а не модификации, что недопустимо.
5. Операнды команды, если это не оговаривается дополнительно в описании команды, должны быть одного размера.

Таблица 4

Команда	Операнды	Флаги										Пояснение	Описание
		O	D	I	T	S	Z	A	P	C			
Команды передачи данных													
MOV	r/m8, r8 r/m16, r16 r/m32, r32 r8, r/m8 r16, r/m16 r32, r/m32 r/m8, imm8 r/m16, c16 r/m32, c32	-	-	-	-	-	-	-	-	-	-	r/m8=r8 r/m16=r16 r/m32=r32 r8=r/m8 r16=r/m16 r32=r/m32 r/m8=c8 r/m16=c16 r/m32=c32	Пересылка операндов
XCHG	r/m8, r8 r8, r/m8 r/m16, r16 r16, r/m16 r/m32, r32 r32, r/m32	-	-	-	-	-	-	-	-	-	r/m8 ↔ r8 r8 ↔ r/m8 r/m16 ↔ r16 r16 ↔ r/m16 r/m32 ↔ r32 r32 ↔ r/m32	Обмен операндов	
BSWAP	r32	-	-	-	-	-	-	-	-	-	TEMP ← r32 r32[7..0] ← TEMP(31..24] r32[15..8] ← TEMP(23..16] r32[23..16] ← TEMP(15..8] r32[31..24] ← TEMP(7..0]]	Перестановка байтов из порядка "младший – старший" в порядок "старший – младший". (с 486 проц.)	
MOVSX	r16, r/m8 r32, r/m8 r32, r/m16	-	-	-	-	-	-	-	-	-	r16, r/m8 DW ← DB r32, r/m8 DD ← DB r32, r/m16 DD ← DW	Пересылка с расширением формата и дублированием знакового бита (с 386 проц.)	
MOVZX	r16, r/m8 r32, r/m8 r32, r/m16	-	-	-	-	-	-	-	-	-	r16, r/m8 DW ← DB r32, r/m8 DD ← DB r32, r/m16 DD ← DW	Пересылка с расширением формата и дублированием нулевого бита (с 386 проц.)	
XLAT XLATB	m8	-	-	-	-	-	-	-	-	-	AL=DS:[(E)BX + unsigned AL]	Загрузить в AL байт из таблицы в сегменте данных, на начало которой указывает EBX (BX); начальное значение AL играет роль смещения.	
LEA	r16, m r32, m	-	-	-	-	-	-	-	-	-	r16=offset m r32=offset m	Загрузка эффективного адреса.	
LDS	r16, m16 r32, m16	-	-	-	-	-	-	-	-	-	DS:r=offset m	Загрузить пару регистров из памяти.	
LSS											SS:r=offset m		
LES											ES:r=offset m		
LFS											FS:r=offset m		
LGS											GS:r=offset m		

Команда	Операнды	Флаги										Пояснение	Описание
		O	D	I	T	S	Z	A	P	C			
<i>Команды установки единичного значения</i>													
SETA SETNBE	r/m8	-	-	-	-	-	-	-	-	-	-	CF=0 и ZF=0	Проверяет условие состояния битов регистра EFLAGS и если условие выполняется, то первый бит байта устанавливается в 1, в противном случае в 0. Условия аналогичны условным переходам (je, jc). Например, SETE AL. (с 386 микропроцессора)
SETAE SETNB SETNC		-	-	-	-	-	-	-	-	-	-	CF=0	
SETB SETC SETNAE		-	-	-	-	-	-	-	-	-	-	CF=1	
SETBE SETNA		-	-	-	-	-	-	-	-	-	-	CF=1 или ZF=1	
SETE SETZ		-	-	-	-	-	-	-	-	-	-	ZF=1	
SETG SETNLE		-	-	-	-	-	-	-	-	-	-	ZF=0 и SF=OF	
SETGE SETNL		-	-	-	-	-	-	-	-	-	-	SF=OF	
SETL SETNGE		-	-	-	-	-	-	-	-	-	-	SF<>OF	
SETLE SETNG		-	-	-	-	-	-	-	-	-	-	ZF=1 или SF<>OF	
SETNE SETNZ		-	-	-	-	-	-	-	-	-	-	ZF=0	
SETNO		-	-	-	-	-	-	-	-	-	-	OF=0	
SETNP SETPO		-	-	-	-	-	-	-	-	-	-	PF=0	
SETNS		-	-	-	-	-	-	-	-	-	-	SF=0	
SETO		-	-	-	-	-	-	-	-	-	-	OF=1	
SETP SETPE		-	-	-	-	-	-	-	-	-	-	PF=1	
SETS	-	-	-	-	-	-	-	-	-	-	SF=1		
<i>Команды работы со стеком</i>													
PUSH	r/m32 c32 r/m16	-	-	-	-	-	-	-	-	-	-	ESP=ESP-4; SS:ESP=r/m32/c SP=SP-2; SS:SP=r/m16	Поместить в стек двойное слово (слово).
POP PUSHA PUSHAD	r/m32 r/m16 -	-	-	-	-	-	-	-	-	-	-	r/m32=SS:ESP ESP=ESP+4 r/m16=SS:SP; SP=SP+4 16-разр. 32-разр.	Извлечь из стека двойное слово (слово). Поместить в стек регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. (с 386 проц.)
POPA POPAD	-	-	-	-	-	-	-	-	-	-	-	16-разр. 32-разр.	Извлечь из стека регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. (с 386 проц.)
PUSHF	-	W	W	W	W	W	W	W	W	W	W		Поместить в стек регистр EFLAGS
POPF	-	W	W	W	W	W	W	W	W	W	W		Извлечь из стека регистр EFLAGS

Команда	Операнды	Флаги										Пояснение	Описание	
		O	D	I	T	S	Z	A	P	C				
Команды ввода-вывода														
IN	AL, c8 AX, c8 EAX, c8 AL, DX AX, DX EAX, DX	-	-	-	-	-	-	-	-	-	-	-	AL= port byte AX= port word EAX= port dword AL= [DX-port] AX= [DX-port] EAX= [DX-port]	Ввод из порта
OUT	c8, AL c8, AX c8, EAX DX, AL DX, AX DX, EAX	-	-	-	-	-	-	-	-	-	-	-	port byte=AL port word=AX port dword=EAX [DX-port]=AL [DX-port]=AX [DX-port]=EAX	Вывод в порт
INSB INSW INSD	-	-	-	-	-	-	-	-	-	-	-	-	ES:(E)DI = [DX-port]	Выводит данные из порта, адресуемого DX в ячейку памяти ES:(E)DI. После ввода байта, слова или двойного слова производится коррекция EDI/DI на 1,2,4. При наличии префикса REP процесс продолжается, пока CX>0.
OUTSB OUTSW OUTSD	-	-	-	-	-	-	-	-	-	-	-	-	[DX-port]=DS:(E)DI	Вводит данные из ячейки памяти, определяемой регистрами DS:(E)SI, в выходной порт, адрес которого находится в регистре DX. После вывода данных производится коррекция указателя ESI/SI на 1,2,4.

Команда	Операнды	Флаги									Пояснение	Описание
		O	D	I	T	S	Z	A	P	C		
Команды целочисленной арифметики												
ADD	r/m8, c8 r/m16, c16 r/m32, c32 r/m8, r8 r/m16, r16 r/m32, r32	+	-	-	-	+	+	+	+	+	r/m8=r/m8+c8 r/m16=r/m16+c16 r/m32=r/m32+c32 r/m8= r/m8+r8 r/m16= r/m16+r16 r/m32= r/m32+r32	Сложение целых чисел
ADC	r8, r/m8 r16, r/m16 r32, r/m32	+	-	-	-	+	+	+	+	+	r8=r8+r/m8 r16=r16+r/m16 r32=r32+r/m32	Сложение с учетом флага переноса CF
XADD	r/m8, r8 r/m16, r16 r/m32, r32	+	-	-	-	+	+	+	+	-	r/mxx↔rxxx; r/mxx= r/mxx+rxxx	Обмен операндами; сложение (с 486 проц.)
INC	r/m8 r/m16	+	-	-	-	+	+	+	+	-	r/m8 =r/m8±1 r/m16 =r/m16±1	Увеличение на 1
DEC	r/m32	+	-	-	-	+	+	+	+	+	r/m32 =r/m32±1	Уменьшение на 1
SUB	r/m8, c8 r/m16, c16 r/m32, c32 r/m8, r8 r/m16, r16 r/m32, r32	+	-	-	-	+	+	+	+	+	r/m8=r/m8-c8 r/m16=r/m16-c16 r/m32=r/m32-c32 r/m8= r/m8-r8 r/m16= r/m16-r16 r/m32= r/m32-r32	Вычитание целых чисел
SBB	r8, r/m8 r16, r/m16 r32, r/m32	+	-	-	-	+	+	+	+	+	r8=r8-r/m8 r16=r16-r/m16 r32=r32-r/m32	Вычитание с учетом флага переноса CF
CMP	r/m8, c8 r/m16, c16 r/m32, c32 r/m8, r8 r/m16, r16 r/m32, r32 r8, r/m8 r16, r/m16 r32, r/m32	+	-	-	-	+	+	+	+	+	r/m8-c8 r/m16-c16 r/m32-c32 r/m8-r8 r/m16-r16 r/m32-r32 r8-r/m8 r16-r/m16 r32-r/m32	Сравнение целых чисел
CMPXCHG	r/m8, r8 r/m16, r16 r/m32, r32	-	-	-	-	-	+	-	-	-	AL=r/m8→ r/m8=r8 AX=r/m16→r/m16=r16 EAX=r/m32→r/m32=r32	Обмен операндов по результату сравнения (с 486)
CMPXCHG 8B	m64	+	-	-	-	+	+	+	+	+	EDX: EAX=m64→ m64=ECX: EBX	Сравнение и обмен 8 байт (с Pentium)
NEG	r/m8 r/m16 r/m32	U	-	-	-	U	U	+	U	+	r/m8=-r/m8 r/m16=-r/m16 r/m32=- r/m32	Изменение знака
MUL	r/m8 r/m16 r/m32	+	-	-	-	U	U	U	U	+	AX=AL*r/m8 DX: AX=AX*r/m18 EDX: EAX=EAX*r/m32	Умножение без знака
IMUL	r/m8 r/m16 r/m32 r16, r/m16 r32, r/m32 r16, r/m16, c r32, r/m32, c r16, c r32, c	U	-	-	-	U	U	U	U	U	AX=AL*r/m8 DX: AX=AX*r/m18 EDX: EAX=EAX*r/m32 r16=r16*r/m16 r32=r32*r/m32 r16=r/m16*c16 r32=r/m32*c32 r16=r16*c16 r32=r32*c32	Умножение со знаком
DIV	r/m8	U	-	-	-	U	U	U	U	U	AL=AX/r/m8, AH=mod	Деление без знака
IDIV	r/m16 r/m32	U	-	-	-	U	U	U	U	U	AX=DX: AX/r/m18 DX=mod EAX=EDX: EAX*r/m32 EDX=mod	Деление со знаком

Команда	Операнды	Флаги									Пояснение	Описание
		O	D	I	T	S	Z	A	P	C		
AAA	-	U	-	-	-	U	U	+	U	+	If((AL&0Fh)>9 AF) { AH=AH+1; AL=(AL+6) & 0Fh; CF:=1; AF:=1;}	Коррекция AL после сложения двух неупакован- ных BCD чисел
AAS	-	U	-	-	-	+	+	U	+	U	if((AL&0Fh)>9) AF) { AH=AH-1; AL=(AL-6)& 0Fh; CF=1; AF=1;}	Коррекция AH после вычитания двух неупакован- ных BCD чисел
AAM	-	O	D	I	T	S	Z	A	P	C	AH=AL/10; AL=mod(AL/10);	Коррекция AL после умножения двух неупакован- ных BCD чисел
AAD		U	-	-	-	+	+	+	+	+	AL=AH*10+AL; AH=0;	Коррекция AL перед делением двух неупакован- ных BCD чисел
DAA		U	-	-	-	+	+	+	+	+		Коррекция AL при сложении двух упакованных BCD чисел
DAS		+	-	-	-	U	U	U	U	+		Коррекция AL при вычитании двух упакованных BCD чисел
CBW	-	-	-	-	-	-	-	-	-	-	AX=(DW)AL	Преобразование типов
CWDE	-	-	-	-	-	-	-	-	-	-	EAX=(DD)AX	
CWD	-	-	-	-	-	-	-	-	-	-	DX:AX=(DD)AX	
CDQ	-	-	-	-	-	-	-	-	-	-	EDX:EAX=(DQ)EAX	
Логические команды												
NOT	r/m8 r/m16 r/m32	0	-	-	-	+	+	U	+	0	r/m8=!r/m8 r/m16=!r/m16 r/m32=!r/m32	инверсия
AND	r/m8,c8 r/m16,c16 r/m32,c32	0	-	-	-	+	+	U	+	0	r/m8=r/m8 φ c8 r/m16=r/m16 φ c16 r/m32=r/m32 φ c32	Логическое умножение (И)
OR	r/m8,r8 r/m16,r16 r/m32,r32	0	-	-	-	+	+	U	+	0	r/m8= r/m8 φr8 r/m16= r/m16 φr16 r/m32= r/m32 φ r32	Логическое сложение (ИЛИ)
XOR	r8,r/m8 r16,r/m16 r32,r/m32	0	-	-	-	+	+	U	+	0	r8=r8 φ r/m8 r16=r16 φ r/m16 r32=r32 φ r/m32	Исключающее ИЛИ
TEST	r/m8,c8 r/m16,c16 r/m32,c32 r/m8,r8 r/m16,r16 r/m32,r32 r8,r/m8 r16,r/m16 r32,r/m32	0	-	-	-	+	+	U	+	0	r/m8&c8 r/m16&c16 r/m32&c32 r/m8&r8 r/m16&r16 r/m32&r32 r8&r/m8 r16&r/m16 r32&r/m32	Логическое умножение без сохранения результата

Команда	Операнды	Флаги									Пояснение	Описание
		O	D	I	T	S	Z	A	P	C		
Сдвиговые команды												
SHR	r/m8 r/m8, CL r/m8, c r/m16 r/m16, CL r/m16, c r/m32 r/m32, CL r/m32, c	+	-	-	-	+	+	U	+	+	r/m8 сдвиг 1 r/m8 сдвиг CL r/m8 сдвиг c r/m16 сдвиг 1 r/m16 сдвиг CL r/m16 сдвиг c r/m32 сдвиг 1 r/m32 сдвиг CL r/m32 сдвиг c	Логический сдвиг вправо
		+	-	-	-	+	+	U	+	+		Арифметический сдвиг вправо (старшие биты заполняются значением знакового)
SAR		+	-	-	-	+	+	U	+	+		Арифметический/логический сдвиг влево
SAL SHL		+	-	-	-	-	-	-	-	+		Арифметический/логический сдвиг влево
ROR		+	-	-	-	-	-	-	-	+		Циклический сдвиг вправо
ROL		+	-	-	-	-	-	-	-	+		Циклический сдвиг влево
RCR		+	-	-	-	-	-	-	-	+		Циклический сдвиг вправо через перенос CF
RCL		+	-	-	-	-	-	-	-	+		Циклический сдвиг влево через перенос CF
Команды управления флагами												
CLC		-	0	-	-	-	-	-	-	-	CF=0;	Сброс бита переноса
CLD		-	-	0	-	-	-	-	-	-	DF=0;	Сброс бита направления
CLI		-	-	-	-	-	-	-	-	+	IF=0;	Сброс бита прерывания
CMC		-	-	-	-	-	-	-	-	1	CF=!CF;	Инверсия бита переноса
STC		-	1	-	-	-	-	-	-	-	CF=1;	Установка бита переноса
STD		-	-	1	-	-	-	-	-	-	DF=1;	Установка бита направления
STI		-	-	1	-	-	-	-	-	-	IF=1;	Установка бита прерывания
Команды прерываний												
INT	3 c8	-	-	0	0	-	-	-	-	-	Отладчик Вектор c8	Программное прерывание
INTO	-	W	W	W	W	W	W	W	W	W	OF=1	
IRET IRETD		W	W	W	W	W	W	W	W	W	16 бит 32 бит	Возврат из прерывания

Команда	Операнды	Флаги										Пояснение	Описание
		O	D	I	T	S	Z	A	P	C			
Команды передачи управления													
JMP	label r/m16 r/m32 ptr16:16 ptr16:32 m16:16 m16:32	-	-	-	-	-	-	-	-	-	-	метка адрес - в r/m16 near адрес - в r/m32 near far far far far	Безусловный переход. При работе в Windows используется в основном внутрисегментный переход (NEAR) в пределах 32-битного сегмента.
CALL	label r/m ptr	-	-	-	-	-	-	-	-	-	-		Вызов процедуры
RET	- c16	-	-	-	-	-	-	-	-	-	-	Удалить с байт из стека	Возврат из процедуры
LOOP	label	-	-	-	-	-	-	-	-	-	-	CX=CX-1; If(CX!=0) EIP=label;	Переход если CX!=0
LOOPE LOOPZ	label	-	-	-	-	-	-	-	-	-	-	CX=CX-1; If(CX!=0 & ZF=1) EIP=label;	Переход если равно ZF=1
LOOPNE LOOPNZ	label	-	-	-	-	-	-	-	-	-	-	CX=CX-1; If(CX!=0 & ZF=0) EIP=label;	Переход если не равно ZF=0
JCXZ	label	-	-	-	-	-	-	-	-	-	-	if(CX==0) EIP=label;	Переход если CX=0
JC	label	-	-	-	-	-	-	-	-	-	-	if(CF==1) EIP=label;	Переход при переносе
JNC	label	-	-	-	-	-	-	-	-	-	-	if(CF==0) EIP=label;	Переход при отсутствии переноса
JS	label	-	-	-	-	-	-	-	-	-	-	if(SF==1) EIP=label;	Переход при отрицательном результате
JNS	label	-	-	-	-	-	-	-	-	-	-	if(SF==0) EIP=label;	Переход при положительном результате
JE JZ	label	-	-	-	-	-	-	-	-	-	-	if(ZF==1) EIP=label;	Переход при нулевом результате
JNE JNZ	label	-	-	-	-	-	-	-	-	-	-	if(ZF==0) EIP=label;	Переход при ненулевом результате
JP JPE	label	-	-	-	-	-	-	-	-	-	-	if(PF==1) EIP=label;	Переход, если PF=1
JNP JPO	label	-	-	-	-	-	-	-	-	-	-	if(PF==0) EIP=label;	Переход, если PF=0
JO	label	-	-	-	-	-	-	-	-	-	-	if(OF==1) EIP=label;	Переход при переполнении
JNO	label	-	-	-	-	-	-	-	-	-	-	if(OF==0) EIP=label;	Переход при отсутствии переполнения

Команда	Операнды	Флаги										Пояснение	Описание	
		O	D	I	T	S	Z	A	P	C				
<i>Беззнаковые переходы</i>														
JB JNAE	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(CF==1)//m1<m2 EIP=label;	Переход если ниже операнд1<операнд 2	
JBE JNA	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(CF==1 ZF==1) //m1<=m2 EIP=label;	Переход если ниже или равно операнд1<=операн д2	
JAE JNB	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(CF==0) //m1>=m2 EIP=label;	Переход если выше или равно операнд1>=операн д2	
JA/JNBE	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(CF==0&ZF==0) //m1>m2 EIP=label;	Переход если выше операнд1>операнд 2	
<i>Знаковые переходы</i>														
JL JNGE	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(SF^OF)//m1<m2 EIP=label;	Переход если меньше операнд1<операнд 2	
JLE JNG	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if((SF^OF) ZF) //m1<=m2 EIP=label;	Переход если меньше или равно операнд1<=операн д2	
JGE JNL	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(SF==OF) //m1>=m2 EIP=label;	Переход если больше или равно операнд1>=операн д2	
JG JNLE	label	-	-	-	-	-	-	-	-	-	-	cmp m1,m2 if(SF==OF & !ZF) //m1>m2 EIP=label;	Переход если больше операнд1>операнд 2	
<i>Команды поддержки языков высокого уровня</i>														
ENTER	c16,0 c16,1 c16,c8	-	-	-	-	-	-	-	-	-	-	PUSH EBP MOV EBP, ESP	Подготовка стека при входе в про- цедуру. Константа c16 указывает ко- личество байт, ре- зервируемых в сте- ке для локальных идентификаторов, константа 0, 1,...31 определяется вло- женностью проце- дур	
LEAVE	-	-	-	-	-	-	-	-	-	-	-	POP EBP	Приведение стека в исходное состояние	
BOUND	r16, m16&16 r32, m32&32	-	-	-	-	-	-	-	-	-	-	m16<r16<m16&16 m32<r16<m32&32	Проверка индекса массива	

Команда	Операнды	Флаги										Пояснение	Описание
		O	D	I	T	S	Z	A	P	C			
Команды синхронизации процессора													
HLT	-	-	-	-	-	-	-	-	-	-	-	-	Остановка процессора до внешнего прерывания
LOCK	-	-	-	-	-	-	-	-	-	-	-	-	Префикс блокировки шины. Заставляет процессор сформировать сигнал LOCK# на время выполнения находящейся за префиксом команды. Этот сигнал блокирует запросы шины другими процессорами в мультипроцессорной системе.
WAIT	-	-	-	-	-	-	-	-	-	-	-	-	Ожидание завершения команды сопроцессора. Большинство команд сопроцессора автоматически вырабатывают эту команду.
NOP	-	-	-	-	-	-	-	-	-	-	-	-	Пустая операция
CPUID	-	-	-	-	-	-	-	-	-	-	-	-	Получение информации о процессоре. Возвращаемое значение зависит от параметра в EAX.
Команды обработки цепочки бит													
BSR	r16, r/m16 r32, r/m32	U	-	-	-	U	+	U	U	U	i"1"[r/m16]→r16 i"1"[r/m32]→r32	Ищет «1» в операнде 2, начиная со старшего бита. Если 1 найдена, ее индекс записывается в операнд 1.	
BSF	r16, r/m16 r32, r/m32	U	-	-	-	U	U	U	U	+	i"1"[r/m16]→r16 i"1"[r/m32]→r32	Ищет 1 в операнде 2, начиная с младшего бита.	
BT	r/m16, r16 r/m32, r32	U	-	-	-	U	U	U	U	+		Тестирование бита с номером из операнда 2 в операнде 1 и перенос его значения во флаг CF.	
BTC		U	-	-	-	U	U	U	U	+		То же с инвертированием бита	
BTR		U	-	-	-	U	U	U	U	+		То же со сбросом бита	
BTS		U	-	-	-	U	U	U	U	+		То же с установкой бита	

Команда	Операнды	Флаги										Пояснение	Описание	
		O	D	I	T	S	Z	A	P	C				
Строковые команды														
MOVSB	-												Байт	Пересылка строки из адреса DS:(E)SI в адрес ES:(E)DI.
MOVSW													Слово	
MOVSD													Двойное слово	
LODSB	-												DS:(E)SI → AL.	Загрузить строку из адреса DS:(E)SI в регистр AL/AX/EAX
LODSW													DS:(E)SI → AX.	
LODSD														
STOSB	-												AL → ES:(E)DI	Сохранить строку из регистра AL/AX/EAX в адрес ES:(E)DI.
STOSW		+	-	-	-	+	+	+	+	+			AX → ES:(E)DI	
STOSD														
SCASB	-												AL (ES:(E)DI)	Поиск в строке по адресу ES:(E)DI байта/слова/двойного слова из регистра AL/AX/EAX
SCASW		+	-	-	-	+	+	+	+	+			AX (ES:(E)DI)	
SCASD														
CMPSB	-												Байт	Сравнение строк с адресами DS:(E)SI и ES:(E)DI.
CMPSW													Слово	
CMPSD														
Префиксы														
REP		-	-	-	-	-	-	-	-	-	-	-	Префикс для MOVSB, STOSB, LODSB	Повторение
REPE REPZ		-	-	-	-	-	-	-	-	-	-	-	Префикс для CMPSB, SCASB	Повторение пока байты/слова равны
REPNE REPNZ													Префикс для CMPSB, SCASB	Повторение пока байты/слова не равны

Пересылочные команды

Команда MOV – это основная команда пересылки данных.

Синтаксис:

MOV приемник, источник

Команда MOV присваивает значению операнда приемника значение операнда источника. В качестве приемника могут выступать регистр общего назначения, сегментный регистр или ячейка памяти, в качестве источника могут выступать число, регистр общего назначения, сегментный регистр или ячейка памяти. Оба операнда должны быть одного размера (байт, слово или двойное слово).

Примеры:

```
mov ax, 2345h
```

```
mov bx, ax
```

```
mov my_mem, ax ; my_mem – переменная в сегменте данных
```

На команду MOV накладываются следующие ограничения:

- командой mov нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения.
- нельзя загрузить в сегментный регистр значение непосредственно из памяти. Поэтому для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек.
- нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных регистры общего назначения. Например,

```
mov ax, ds  
mov es, ax ; es=ds
```
- нельзя использовать сегментный регистр CS в качестве операнда приемника, поскольку в архитектуре микропроцессора пара CS : IP всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой mov

содержимого регистра CS фактически означало бы операцию перехода, а не пересылки, что недопустимо.

Команда XCHG применяется для двунаправленной пересылки данных.

Синтаксис:

XCHG выражение_1, выражение_2

В качестве операндов команды могут быть заданы два регистра общего назначения или регистр общего назначения и ячейка памяти. Для этой операции можно, конечно, применить последовательность из нескольких команд mov, но из-за того, что операция обмена используется довольно часто, разработчики системы команд микропроцессора посчитали нужным ввести отдельную команду обмена xchg. Операнды должны иметь один тип. Не допускается (как и для всех команд ассемблера) обменивать между собой содержимое двух ячеек памяти.

Примеры:

XCHG ax, bx ;обменять содержимое регистров ax и bx

XCHG ax, word ptr [si] ;обменять содержимое регистра ax
и слова в памяти по адресу в [si]

Команда XLAT/XLATB применяется для передачи данных из таблицы.

Синтаксис:

XLAT метка

XLATB

Команда XLAT осуществляет пересылку байта из таблицы в регистр AL. Перед вызовом команды в регистр BX должно быть занесено смещение начала таблицы в сегменте данных (адрес начала таблицы определяется парой регистров DS:BX). Регистр AL перед вызовом команды содержит смещение от начала таблицы (номер элемента в таблице). Различают два варианта записи: без операндов или с одним операндом, в качестве которого может выступать метка начала таблицы.

Команда PUSH/PUSHF применяется для записи операнда в вершину стека.

Синтаксис:

PUSH источник

Операнд, заданный в команде, помещается в вершину стека. При этом указатель SP автоматически декрементируется на 2 или 4 (в зависимости от определенного ранее размера указателя стека: 16 бит или 32 бита). Операндом в данной команде могут быть число, регистр общего назначения, сегментный регистр или ячейка памяти.

Пример:

PUSH AX ; SP=SP-2; [SP]=AX;

Команда PUSHF позволяет записать в стек содержимое регистра FLAGS. Принцип ее работы полностью аналогичен команде PUSH.

Команда POP/POPF применяется для чтения операнда из вершины стека.

Синтаксис:

POP приемник

Операнд, заданный в команде, перемещается из вершины стека в регистр или ячейку памяти, указанную в качестве приемника. При этом указатель SP автоматически инкрементируется на 2 или 4 (в зависимости от определенного ранее размера указателя стека: 16 бит или 32 бита).

Пример:

POP AX ; AX=[SP]; SP=SP+2;

Команда POPF позволяет записать в регистр FLAGS содержимое вершины стека. Принцип ее работы полностью аналогичен команде POP.

Команда LDS/LES применяется для загрузки указателя типа far в пару DS:dst/ES:dst.

Синтаксис:

LDS приемник, источник

Загружает указатель типа far, заданный источником (сегмент: смещение) в пару DS:приемник / ES:приемник.

Команда LEA применяется для загрузки исполнительного адреса.

Синтаксис:

LEA приемник, источник

Вычисляет физический адрес операнда источника и загружает его в операнд приемника. Операнд источника является адресом ячейки памяти, операнд приемника – регистром общего назначения.

Пример:

```
LEA bx, m1
```

Данная команда загружает эффективный адрес, на который указывает переменная m1 в сегменте данных в регистр bx. Те же действия можно выполнить с использованием команды MOV:

```
MOV bx, OFFSET m1
```

Команда LAHF пересылает регистр FLAGS в регистр AH.

Команда SAHF записывает регистр AH в регистр FLAGS.

```
FLAGS (SF:ZF:0:AF:0:PF:1:CF) := AH
```

Микропроцессор может передавать данные в порты ввода-вывода, которые поддерживаются аппаратно и используют соответствующие своим предназначениям вывода процессора.

Аппаратное адресное пространство ввода-вывода процессора не является физическим адресным пространством памяти. Адресное пространство ввода-вывода состоит из 64Кбайт индивидуально адресуемых 8-битных портов ввода-вывода, имеющих адреса 0...FFFFh. Адреса 0F8h...0FFh являются резервными. Любые два последовательных 8-битных порта могут быть объединены в 16-битный порт, 4 последовательных 8-битных порта – в 32-битный порт.

Для работы с портами ввода-вывода используются команды IN и OUT.

Команда IN применяется для чтения данных из порта.

Синтаксис:

```
IN AX, источник
```

```
IN AX, DX
```

Копирует данные порта ввода-вывода, заданного в качестве источника, в аккумулятор. Операнд источника может быть задан непосредственно как номер порта или через регистр DX. Если адрес порта меньше 256, то он может быть задан в команде, например

`in al,60h` ; данные из порта 60h копируются в al

Если адрес имеет значение больше 255, то это значение указывается через регистр DX.

Команда OUT применяется для записи данных в порт.

Синтаксис:

`OUT приемник, AX`

`OUT DX, AX`

Копирует данные аккумулятора AX в порт ввода-вывода, заданный как операнд приемника. Операнд источника может быть задан непосредственно как номер порта или через регистр DX.

Команды целочисленной арифметики

Команда INC – прибавление 1

Синтаксис:

`INC операнд`

Команда INC прибавляет 1 к операнду, не изменяя состояние бита переноса CF в регистре признаков. Операндом может быть регистр или ячейка памяти. Признаки OF, SF, ZF, AF, PF устанавливаются в соответствии с результатом.

Пример:

`inc ax` ; $ax=ax+1$;

Команда DEC – вычитание 1

Синтаксис:

`DEC операнд`

Команда DEC вычитает 1 из операнда, не изменяя состояние бита переноса CF в регистре признаков. Операндом может быть регистр или ячейка памяти. Признаки OF, SF, ZF, AF, PF устанавливаются в соответствии с результатом.

Пример:

`dec ax` ; $ax=ax-1$;

Команда ADD – сложение целых чисел

Синтаксис:

`ADD приемник, источник`

Команда ADD прибавляет операнд источника к операнду приемника и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Может оперировать с двумя знаковыми или беззнаковыми целыми числами разрядностью 8, 16 или 32 бита. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом. Если установлены признаки OF и CF, то результат операции вышел за разрядную сетку операнда.

Примеры:

```
add ax, 1          ; ax=ax+1
add ax, bx         ; ax=ax+bx
```

Команда SUB – вычитание целых чисел

Синтаксис:

SUB приемник, источник

Команда SUB вычитает операнд источника из операнда приемника и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Может оперировать с двумя знаковыми или беззнаковыми целыми числами разрядностью 8, 16 или 32 бита. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом. Если установлены признаки OF и CF, то результат операции вышел за разрядную сетку операнда. Флаг SF отображает знак результата.

Примеры:

```
sub ax, 2          ; ax=ax-2
sub ax, bx         ; ax=ax-bx
```

Команда CMP – сравнение целых чисел

Синтаксис:

CMP источник_1, источник_2

Команда CMP вычитает операнд источник_2 из операнда источник_1 и не сохраняет результат в каком-либо операнде, а устанавливает биты регистра признаков в соответствии с результатом. Операнд источник_1 может быть регистром или ячейкой памяти. Операнд источник_2 может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Команда CMP не может оперировать двумя ячейками памяти. Операнды команды должны иметь одинаковый формат. Если в качестве операнда используется число, то оно автоматически преобразуется к формату операнда источник_1. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом.

Иными словами, команда CMP отличается от команды SUB тем, что не сохраняет результат операции вычитания.

Команда CMP чаще всего используется совместно с командами условных переходов.

Пример:

```
cmp ax,2          ; если ax > 2
ja 11             ; перейти к метке 11
```

Команда IMUL/MUL – умножение целых чисел

Синтаксис:

IMUL источник

MUL источник

Команда IMUL производит умножение двух чисел со знаком. Команда MUL производит умножение двух чисел без знака. Операнд источник может быть регистром, ячейкой памяти или числом. Второй операнд содержится в регистре AL или AX (в зависимости от формата операнда источника). Результат умножения сохраняется в регистре AX или в паре DX|AX соответственно. Признаки OF, CF устанавливаются в случае, если старший бит (включая знаковый бит) переходит в старший регистр результата. Значения битов SF, ZF, AF, PF не определены.

Пример:

```
mov al,2          ;
```

```
mul 4 ; ax=8
```

Команда IDIV/DIV – деление целых чисел

Синтаксис:

```
IDIV источник
```

```
DIV источник
```

Команда IDIV производит деление со знаком. Команда DIV производит деление без знака. Делитель (источник) может быть регистром, ячейкой памяти или числом. Делимое содержится в регистре AX или паре DX | AX (в зависимости от формата операнда источника). Результат деления сохраняется в регистре AL или AX, а остаток – в регистрах AH или DX соответственно. Знак частного всегда совпадает со знаком делимого. Абсолютная величина частного всегда меньше абсолютной величины делимого. Значения битов OF, SF, ZF, AF, PF, CF не определены.

Пример:

```
mov ax, 21
```

```
div 4 ; al=5, ah=1
```

Команда CBW – преобразование байта в слово

Синтаксис:

```
CBW
```

Команда CBW удваивает размер операнда источника, содержащегося в регистре AL (используется по умолчанию). Копирует знаковый бит (бит 15) регистра AL во все биты регистра AH.

Пример:

```
mov al, -21h ; ah=?, al=11011111b
```

```
cbw ; ah=11111111b, al=11011111b
```

Команда CWD – преобразование слова в двойное слово

Синтаксис:

```
CWD
```

Команда CWD удваивает размер операнда источника, содержащегося в регистре AX (используется по умолчанию). Копирует знаковый бит (бит 31) регистра AX во все биты регистра DX.

Пример:

```
mov ax, -1234h ; ah=11101101h, al=11001100b
```

```
cwd ; dx=11111111 11111111b
```

Команда ADC – сложение целых чисел с переносом

Синтаксис:

ADD приемник, источник

Команда ADC складывает операнд источника с операндом приемника и битом переноса CF и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Команда ADC не делает различий между знаковыми и беззнаковыми числами. Может оперировать с целыми числами разрядностью 8, 16 или 32 бита. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом. Если установлены признаки OF и CF, то результат операции вышел за разрядную сетку операнда.

Пример:

```
mov ax, 21      ; ax=21
stc             ; CF=1
adc ax, 9       ; ax=21+9+1=31
```

Команда SBB – вычитание целых чисел с заемом

Синтаксис:

SBB приемник, источник

Команда SBB складывает операнд источника с битом переноса и вычитает из операнда приемника, а затем сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Команда SBB не делает различий между знаковыми и беззнаковыми числами. Может оперировать с целыми числами разрядностью 8, 16 или 32 бита. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в

соответствии с результатом. Признаки OF и CF указывают на переполнение разрядной сетки операнда приемника. Флаг SF отображает знак результата.

Пример:

```
mov ax, 21      ; ax=21
stc             ; CF=1
sbb ax, 9       ; ax=21-(9+1)=11
```

Команда AAA – коррекция после сложения двух неупакованных BCD чисел

Синтаксис:

AAA

Команда AAA корректирует результат сложения двух двоично-десятичных неупакованных чисел для представления его в двоично-десятичном формате. Операнд источник содержится по умолчанию в регистре AX. Команда AAA должна непосредственно следовать за командой сложения ADD, два числа в которой представлены в двоично-десятичной неупакованной форме. Признаки AF и CF устанавливаются, если была произведена корректировка в соответствии с алгоритмом. Признаки OF, SF, ZF, и PF после выполнения команды могут находиться в любом состоянии.

Алгоритм действия команды AAA следующий.

```
if(((AL & 0x0F)>9) || AF==1)
{
    AL = AL + 6;
    AH = AH + 1;
    AF = 1;
    CF = 1;
}
else
{
    AF = 0;
    CF = 0;
}
```

Пример:

```
mov ax, 0208h   ; ax=00000010 00001000b (28 BCD)
mov bx, 0104h   ; bx=00000001 00000100b (14 BCD)
add ax, bx      ; ax=00000011 00001100b (3h 0Ch)
aaa             ; ax=00000100 00000010b (42 BCD)
```

Команда AAS – коррекция после вычитания неупакованных BCD чисел

Синтаксис:

AAS

Команда AAS корректирует разность двух двоично-десятичных неупакованных чисел для представления его в двоично-десятичном формате. Операнд источник содержится по умолчанию в регистре AX. Команда AAS должна непосредственно следовать за командой вычитания SUB, два числа в которой представлены в двоично-десятичной неупакованной форме. Признаки AF и CF устанавливаются, если была произведена корректировка в соответствии с алгоритмом. Признаки OF, SF, ZF, и PF после выполнения команды могут находиться в любом состоянии.

Алгоритм действия команды AAS следующий.

```
if(((AL & 0x0F)>9) || AF==1)
{
    AL = AL - 6;
    AH = AH - 1;
    AF = 1;
    CF = 1;
}
else
{
    AF = 0;
    CF = 0;
}
```

Пример:

```
mov ax,0304h      ; ax=00000011 00000100b   (34 BCD)
mov bx,0109h      ; bx=00000001 00001001b   (19 BCD)
sub ax,bx         ; ax=00000010 11111011b   (1h 0FBh)
aas              ; ax=00000001 00000101b   (15 BCD)
```

Команда AAM – коррекция после умножения двух неупакованных BCD чисел

Синтаксис:

AAM

Команда AAM корректирует произведение двух двоично-десятичных неупакованных чисел для представления его в двоично-десятичном формате. Операнд источник содержится по умолчанию в регистре AX. Команда AAM должна непосредственно следовать за командой умножения MUL, два числа в которой представлены в двоично-десятичной неупакованной форме. Признаки SF, ZF, и PF устанавливаются в соответствии с полученным значением в регистре AL.

Признаки AF, CF и OF после выполнения команды могут находиться в любом состоянии.

Алгоритм действия команды AAM следующий.

$AH = AL / 10;$
 $AL = AL - AH * 10;$

Пример:

```
mov ax,0009      ; ax=00000000 00001001b (09 BCD)
mul 0008         ; ax=00000000 01001000b (48h)
aam             ; ax=00000111 00000010b (07 02 BCD)
```

Команда AAD – коррекция перед делением неупакованных BCD чисел

Синтаксис:

AAD

Команда AAD корректирует делимое перед делением двух двоично-десятичных неупакованных чисел так, чтобы результат деления был корректным двоично-десятичным числом. Операнд источник содержится по умолчанию в регистре AX. Команда AAD, как правило предшествует непосредственно команде деления DIV. . Признаки SF, ZF, и PF устанавливаются в соответствии с полученным значением в регистре AL. Признаки AF, CF и OF после выполнения команды могут находиться в любом состоянии.

Алгоритм действия команды AAD следующий.

$AL = AL + AH * 10;$
 $AH = 0;$

Пример:

```
mov ax,0702h    ; ax=00000111 00000010b (0702h)
aad            ; ax=00000000 01001000b (48h)
div 0008       ; ax=00000000 00001001b (9h)
```

Команда DAA – коррекция после сложения двух упакованных BCD чисел

Синтаксис:

DAA

Команда DAA корректирует результат сложения двух упакованных двоично-десятичных чисел. Операнд источник содержится по умолчанию в регистре AL. Команда DAA должна непосредственно следовать за командой сложения ADD, два числа в которой представлены в двоично-десятичной упакованной форме. Признаки

AF и CF устанавливаются в случае появления переноса. Признаки SF, ZF, и PF устанавливаются в соответствии с результатом. Признак OF после выполнения команды может находиться в любом состоянии.

Алгоритм действия команды DAA следующий.

```
old_AL = AL;
old_CF = CF;
if(((AL & 0x0F)>9) || AF==1)
{
    AL = AL + 6;           ; результат формирует CF
    CF = old_CF | CF;
    AF = 1;
}
else
    AF = 0;
if((old_AL > 99h) || (CF==1))
{
    AL = AL + 60h;
    CF = 1;
}
else
    CF = 0;
```

Пример:

```
mov al,79h           ; al=01111001b   (79h)
add al,35h           ; al=10101110b   (AEh)
daa                  ; al=00010100b   (14h), CF=1
```

Команда DAS – коррекция после вычитания упакованных BCD чисел

Синтаксис:

DAS

Команда DAS корректирует разность двух упакованных двоично-десятичных чисел. Операнд источник содержится по умолчанию в регистре AL. Команда DAS должна непосредственно следовать за командой вычитания SUB, два числа в которой представлены в двоично-десятичной упакованной форме. Признаки AF и CF устанавливаются в случае появления переноса. Признаки SF, ZF, и PF устанавливаются в соответствии с результатом. Признак OF после выполнения команды может находиться в любом состоянии.

Алгоритм действия команды DAS следующий.

```
old_AL = AL;
```

```

old_CF = CF;
if(((AL & 0x0F)>9) || AF==1)
{
    AL = AL - 6;           ; результат формирует CF
    CF = old_CF | CF;
    AF = 1;
}
else
    AF = 0;
if((old_AL > 99h) || (CF==1))
{
    AL = AL + 60h;
    CF = 1;
}
else
    CF = 0;

```

Пример:

```

mov al,35h           ; al=00110101b   (35h)
sub al,47h           ; al=11101110b   (EEh)
das                  ; al=10001000b   (88h), CF=1

```

Команда NOT – логическое отрицание

Синтаксис:

NOT приемник

Команда NOT производит побитовое отрицание операнда приемника.

Операнд приемника может быть регистром или ячейкой памяти.

Пример:

```

mov al,11010000b
not al               ; al=00101111b

```

Команда AND – логическое умножение двух целых чисел

Синтаксис:

AND приемник, источник

Команда AND производит побитовое умножение двух операндов и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки

OF, CF устанавливаются в 0, признаки SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
and al,10001000b      ; al=00001000b
```

Команда TEST – сравнение двух целых чисел логическим умножением

Синтаксис:

```
TEST источник_1, источник_2
```

Команда TEST производит побитовое умножение двух операндов и не сохраняет результат, а устанавливает только биты SF, ZF, и PF в регистре FLAGS в зависимости от результата. Операнд источник_1 может быть регистром или ячейкой памяти. Операнд источник_2 может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
test al,5           ; SF=0;ZF=0;PF=1
```

Команда OR – логическое сложение двух целых чисел

Синтаксис:

```
OR приемник, источник
```

Команда OR производит побитовое сложение двух операндов и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, признаки SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
```

```
or al,10001000b ; al=11001101b
```

Команда XOR – логическое исключающее ИЛИ двух целых чисел

Синтаксис:

XOR приемник, источник

Команда XOR производит побитовое исключающее ИЛИ двух операндов и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
xor al,10001000b ; al=11000101b
```

Команда SAR/SAL/SHR/SHL – арифметический сдвиг вправо/влево, логический сдвиг вправо/влево

Синтаксис:

SAR приемник, число_разрядов

SAL приемник, число_разрядов

SHR приемник, число_разрядов

SHL приемник, число_разрядов

Команды SAR/SAL/SHR/SHL сдвигает биты в операнде приемника на указанное число_разрядов вправо/влево. Каждый сдвигаемый бит помещается в признак CF и при следующем сдвиге бита опускается. Операнд приемника может быть регистром или ячейкой памяти. Число сдвигаемых разрядов варьируется от 1 до 5. Если число сдвигаемых разрядов не указано, то производится сдвиг на 1 разряд.

Команды SAL/SHL производят одинаковые действия. Старший бит сдвигается в признак переноса CF, а младший бит сбрасывается.

Команды SAR/SHR производят сдвиг вправо. Младший бит сдвигается в признак переноса CF, а старший бит сбрасывается командой SHR или

устанавливается равным предыдущему значению (знаку числа) командой SAR. Таким образом, команда SHR может применяться для беззнакового деления на 2, 4, 8, 16, 32, а команда SAR – для знакового деления.

При сдвиге влево признак OF сбрасывается, если признак CF равен старшему биту результату, устанавливается в 1 в противном случае. При сдвиге вправо использование команды SHR устанавливает бит OF при сдвиге каждого бита. При использовании команды SAR признак OF сбрасывается. Признак OF изменяется только при сдвиге на 1 бит.

Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

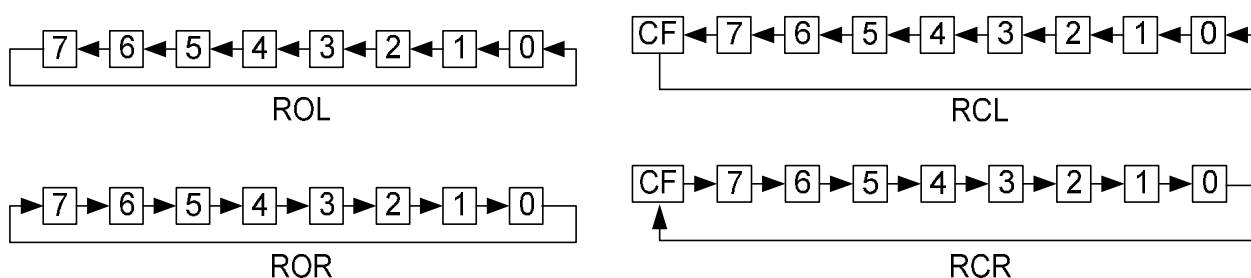
```
mov al,11001101b
sar al,2          ; al=11110011b
shr al,1         ; al=01111001b
```

Команда ROR/ROL/RCL/RCL – циклический сдвиг вправо/влево, циклический сдвиг вправо/влево через перенос

Синтаксис:

```
ROR приемник, число_разрядов
ROL приемник, число_разрядов
RCR приемник, число_разрядов
RCL приемник, число_разрядов
```

Команды ROR/ROL/RCL/RCL циклически сдвигает биты в операнде приемника на указанное число_разрядов вправо/влево в соответствии со схемой.



Операнд приемника может быть регистром или ячейкой памяти. Число сдвигаемых разрядов варьируется от 1 до 5.

Признак CF устанавливается равным последнему сдвигаемому биту. Признак OF изменяется только при сдвиге на 1 бит. Признаки SF, ZF, AF и PF не изменяются.

Примеры:

```
mov al,11001101b
rol al,2           ; al=00110111b
rcr al,2           ; al=11001101b CF=1
```

Команда JMP – безусловный переход

Синтаксис:

JMP адрес

Команда JMP осуществляет передачу программного управления в другую точку кода, не запоминая информацию для возврата. Операнд адрес определяет адрес команды, которой должно быть передано управление. Операндом в данной команде может быть число, регистр общего назначения или ячейка памяти.

Команда JMP может применяться для осуществления следующих четырех типов переходов:

- *near* – переход к команде в том же сегменте кода (физический адрес сегмента кода содержится в регистре CS);
- *short* – переход типа *near* к команде, находящейся в адресном пространстве -128 ...+127 байт от текущего положения программного счетчика IP;
- *far* – переход к команде, находящейся в другом сегменте кода, который имеет такой же уровень приоритета, как и текущий сегмент;
- *task* – переход к команде, находящейся в другой задаче.

Последний тип может использоваться только в защищенном режиме.

При переходах типа *near* и *short* операнд адрес задается в виде абсолютного (смещение от начала сегмента кода) или относительного (смещение от текущего счетчика команд IP) смещения. При этом значение регистра CS не изменяется.

При переходах типа `far` операнд адрес задается в виде сегмент:смещение, в регистр `CS` записывается физический адрес нового сегмента кода, а в регистр `IP` – адрес смещения в новом сегменте кода.

Пример:

```
jmp m1          ; переход на метку m1 (тип near)
...
m1: add ax,bx   ; ax=ax+bx
```

Команда CALL – вызов подпрограммы

Синтаксис:

`CALL` адрес

Команда `CALL` сохраняет информацию о вызывающей программе в стеке и передает управление вызываемой подпрограмме, адрес первой команды которой содержится в операнде. Операндом в данной команде может быть число, регистр общего назначения или ячейка памяти.

Команда `CALL` может применяться для осуществления следующих четырех типов переходов:

- `near` – переход к подпрограмме в том же сегменте кода (физический адрес сегмента кода содержится в регистре `CS`);
- `far` – переход к подпрограмме, находящейся в другом сегменте кода, который имеет такой же уровень приоритета, как и текущий сегмент;
- `inter-privilege-level far` – переход к подпрограмме, находящейся в другом сегменте кода, который имеет другой уровень приоритета;
- `task` – переход к подпрограмме, находящейся в другой задаче.

Последние два типа могут использоваться только в защищенном режиме.

При переходах типа `near` процессор помещает в стек значение счетчика команд `IP` (который содержит смещение в сегменте кода команды, следующей за командой `CALL`). Затем процессор передает управление команде, находящейся по адресу операнда. Операнд адрес задается в виде абсолютного (смещение от начала сегмента кода) или относительного (смещение от текущего счетчика команд `IP`) смещения. При этом значение регистра `CS` не изменяется.

При переходах типа `far` процессор помещает в стек значение сегмента кода вызываемой программы `CS` и значение счетчика команд `IP` (который содержит смещение в сегменте кода команды, следующей за командой `CALL`). Операнд адрес задается в виде `сегмент:смещение`, в регистр `CS` записывается физический адрес нового сегмента кода, а в регистр `IP` – адрес смещения в новом сегменте кода.

Пример:

```
call m1          ; переход к подпрограмме m1 (тип near)
mov dx, ax
...
m1: add ax, bx    ; ax=ax+bx
...
ret
```

Команда RET – возврат из подпрограммы

Синтаксис:

`RET [число]`

Команда `RET` передает программное управление по адресу возврата, хранящемуся в вершине стека. Адрес обычно помещен в вершину стека командой `CALL`, и возврат производится к команде, стоящей после `CALL`.

Операнд `число` (если имеется) задает число байтов стека, которые были переданы в вызываемую подпрограмму перед вызовом команды `CALL` и больше не используются. Эти байты должны быть извлечены из стека вызываемой программой.

Команда `RET` может применяться для осуществления следующих трех типов возврата:

- `near` – переход к вызываемой подпрограмме в том же сегменте кода (физический адрес сегмента кода содержится в регистре `CS`);
- `far` – переход к вызываемой подпрограмме, находящейся в другом сегменте кода, который имеет такой же уровень приоритета, как и текущий сегмент;
- `inter-privilege-level far` – переход к вызываемой подпрограмме, находящейся в другом сегменте кода, который имеет другой уровень приоритета;

Последний тип может использоваться только в защищенном режиме.

При переходах типа `near` процессор извлекает из вершины стека значение счетчика команд, помещает его в регистр `IP` и передает управление команде, находящейся по новому адресу счетчика команд. Значение регистра `CS` не изменяется.

При переходах типа `far` процессор извлекает из вершины стека значение счетчика команд, помещая его в регистр `IP`, затем извлекает значение сегмента кода вызывающей программы и помещает его в регистр `CS`. Процессор начинает выполнять команды по новому адресу в новом сегменте кода.

Команда `INT/INT0` – вызов обработчика прерывания

Синтаксис:

`INT n`

Команда `INT` вызывает обработчик указанного операндом `n` прерывания. Операнд `n` определяет номер вектора системного прерывания BIOS от 0 до 255, представленный в виде беззнакового 8-битного целого числа. При вызове обработчика прерывания в стеке сохраняются регистры `IP`, `CS` и `FLAGS`.

Прерывание – это, как правило, асинхронная остановка работы процессора, вызванная началом работы устройства ввода-вывода. Исключением являются синхронные прерывания, возникающие при определении некоторых предопределенных условий в процессе выполнения команды.

Когда поступает сигнал о прерывании, процессор останавливает выполнение текущей программы и переключается на выполнение обработчика прерывания, заранее записанного для каждого прерывания.

Архитектура IA-32 поддерживает 17 векторов аппаратных прерываний и 224 пользовательских. Прерывание по переполнению вызывается отдельной командой `INT0` и имеет вектор `04h`. Номера векторов прерываний приведены в приложении 2.

Команда `IRET` – возврат из обработчика прерывания

Команда `IRET` возвращает программное управление из обработчика прерывания в прерванную программу. При возврате из обработчика прерывания из стека извлекаются значения регистров `FLAGS`, `CS` и `IP`, сохраненные командой `INT`.

Команды условных переходов Jcc

Синтаксис:

Jcc адрес

Команды условных переходов проверяют состояние одного или нескольких битов регистра признаков и при выполнении условия осуществляют передачу программного управления в другую точку кода, задаваемую операндом. Указанный класс команд не запоминает информацию для возврата. Операнд адрес определяет адрес команды, которой должно быть передано управление. Операндом в данной команде может быть число, регистр общего назначения или ячейка памяти.

Команды условных переходов не поддерживают переходы типа far. Для осуществления таких переходов используются конструкции вида

```
JNcc M1
```

```
JMP far_label
```

```
M1: ...
```

Команды организации циклов LOOPcc

Синтаксис:

LOOPcc адрес

Команды организации циклов используют регистр CX в качестве счетчика числа повторений цикла. Каждый раз при выполнении команды LOOPcc значение регистра CX уменьшается на 1, а затем сравнивается с 0. Если CX=0, выполнение цикла заканчивается, и продолжает выполняться код программы, записанный после команды LOOPcc. Если CX содержит ненулевое значение, то осуществляется переход к адресу операнда команды LOOPcc.

Команды циклов поддерживают переходы типа short.

Некоторые формы команд организации циклов используют нулевой признак в качестве условия окончания цикла. Сама команда LOOPcc не влияет на бит ZF.

Команды операций с флагами – сбрасывают, устанавливают или инвертируют соответствующий бит в регистре FLAGS.

Команды синхронизации работы сопроцессора

Команда ESC передает управление команде сопроцессора. Все команды сопроцессора x87 FPU имеют одинаковый формат кодирования, где первый байт команды представляется одним из чисел D8h . . . DFh.

Команда WAIT ожидает, пока сопроцессор выполнит свои операции и сможет вернуть управление центральному процессору.

Команда LOCK в мультипроцессорной системе дает процессору исключительную возможность пользования некоторой областью памяти пока установлен сигнал LOCK#. Сигнал LOCK# формируется, если команда LOCK используется с одной из следующих инструкций: ADD, ADC, AND, DEC, INC, NOT, OR, SBB, SUB, XOR, XCHG.

Команда HLT производит остановку работы процессора. Возобновить работу процессора могут прерывания, выход из отладчика или сигналы INIT#, RESET#.

Команда MOVS /MOVSB /MOVSW –команда пересылки строк.

Синтаксис:

MOVS приемник, источник

Команда MOVS копирует байт или слово из цепочки, адресуемой операндом источник, в цепочку, адресуемую операндом приемник. Размер пересылаемых элементов ассемблер определяет, исходя из атрибутов идентификаторов, указывающих на области памяти приемника и источника. К примеру, если эти идентификаторы были определены директивой db, то пересылаться будут байты, если идентификаторы были определены с помощью директивы dw, то пересылке подлежат 16-битные элементы, то есть слова. Для цепочечных команд с операндами, к которым относится и команда пересылки MOVS, не существует машинного аналога. При трансляции, в зависимости от типа операндов, транслятор преобразует ее в одну из команд: MOVSB или MOVSW.

Если перед командой написать префикс REP, то одной командой можно переслать до 64 Кбайт данных (если размер адреса в сегменте 16 бит — use16). Число пересылаемых элементов должно быть загружено в счетчик – регистр CX.

Для того, чтобы выполнить пересылку последовательности элементов из одной области памяти в другую с помощью команды MOV_S необходимо выполнить следующий набор действий:

1. Установить значение флага DF в зависимости от того, в каком направлении будут обрабатываться элементы цепочки — в направлении возрастания или убывания адресов.
2. Загрузить указатели на адреса цепочек в памяти в пары регистров DS : SI и ES : DI.
3. Загрузить в регистр CX количество элементов, подлежащих обработке.
4. Выполнить команду MOV_S с префиксом REP.

Команда LODS / LODSB / LODSW – загрузить элемент из цепочки в аккумулятор.

Синтаксис:

LODS источник

Эта команда позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор AL или AH. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Возможный размер извлекаемого элемента определяется применяемой командой.

Команда имеет один операнд, обозначающий строку в основном сегменте данных. Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому пары регистров DS : SI, и поместить его в регистр AH/AL. При этом содержимое SI увеличивается или уменьшается (в зависимости от состояния флага DF) на значение, равное размеру элемента.

Команда STOS / STOSB / STOSW – сохранить аккумулятор в элементе цепочки.

Синтаксис:

STOS приемник

Эта команда позволяет произвести действие, обратное команде LODS, то есть сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать вместе с операциями поиска (сканирования) SCANS и загрузки

LODS с тем, чтобы, найдя нужный элемент, извлечь его в регистр и записать на его место новое значение.

Команда имеет один операнд-приемник, адресующий цепочку в дополнительном сегменте данных. Работа команды заключается в том, что она пересылает элемент из аккумулятора (регистра AX/AL) в элемент цепочки по адресу, соответствующему содержимому пары регистров ES:DI. При этом содержимое DI подвергается инкременту или декременту (в зависимости от состояния флага DF) на значение, равное размеру элемента цепочки.

Команда CMPS/CMPSB/CMPSW – сравнить строки.

Синтаксис:

CMPS источник1, источник2

Эти команды производят сравнение элементов цепочки-источника1 с элементами цепочки-источника2 и устанавливают биты в регистре FLAGS в зависимости от результата. Оба операнда находятся в памяти. Адрес источника1 определяет цепочку-источник в сегменте данных. Адрес цепочки должен быть заранее загружен в пару DS:SI. Адрес источника2 определяет цепочку-источник2, которая должна находиться в дополнительном сегменте, и ее адрес должен быть заранее загружен в пару ES:DI.

Алгоритм работы команды CMPS заключается в последовательном выполнении вычитания (источник1 – источник2) над очередными элементами обеих цепочек. Принцип выполнения вычитания командой CMPS аналогичен команде сравнения CMP. Она, так же, как и CMP, производит вычитание элементов, не записывая при этом результата, и устанавливает флаги CF, OF, SF, ZF, AF и PF в соответствии с результатом. После выполнения вычитания очередных элементов цепочек командой CMPS индексные регистры SI и DI автоматически изменяются в соответствии со значением флага DF на значение, равное размеру элемента сравниваемых цепочек.

Чтобы заставить команду CMPS выполняться несколько раз, то есть производить последовательное сравнение элементов цепочек, необходимо перед

этой командой определить префикс повторения. С командой CMPS можно использовать префиксы повторения REPE/REPZ или REPNE/REPZ:

- REPE/REPZ — если необходимо проводить сравнение до тех пор, пока не будет выполнено одно из двух условий:
 - достигнут конец цепочки (CX=0);
 - в цепочках встретились разные элементы (ZF=0).
- REPNE/REPZ — если нужно проводить сравнение до тех пор, пока не будет:
 - достигнут конец цепочки (CX=0);
 - в цепочках встречены одинаковые элементы (ZF=1).

Таким образом, выбрав подходящий префикс, удобно использовать команду CMPS для поиска одинаковых или различающихся элементов цепочек. Выбор префикса определяется причиной, которая приводит к выходу из цикла. Таких причин может быть две для каждого из префиксов. Для определения конкретной причины наиболее подходящим является способ, использующий команду условного перехода JCXZ. Ее работа заключается в анализе содержимого регистра CX, и если оно равно нулю, то управление передается на метку, указанную в качестве операнда JCXZ. Так как в регистре CX содержится счетчик повторений для цепочечной команды, имеющей любой из префиксов повторения, то, анализируя этот регистр, можно определить причину выхода из заикливания цепочечной команды. Если значение в CX не равно нулю, то выход произошел по причине совпадения либо несовпадения очередных элементов цепочек.

Для того, чтобы определить местоположение очередных совпавших или несовпавших элементов в цепочках необходимо проанализировать состояние индексных регистров после выхода из цикла. Поскольку после каждой итерации цепочечная команда автоматически осуществляет инкремент/декремент значения адреса в соответствующих индексных регистрах, после выхода из цикла в этих регистрах будут находиться адреса элементов, находящихся в цепочке после элементов, которые послужили причиной выхода из цикла. Для получения истинного адреса этих элементов необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив значение в них на длину элемента цепочки.

Команда SCAS/SCASB/SCASW – сканирование цепочки.

Синтаксис:

SCAS приемник

Эти команды производят поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8 или 16 бит. Искомое значение предварительно должно быть помещено в регистр AL/AH. Выбор конкретного регистра из этих трех должен быть согласован с размером элементов цепочки, в которой осуществляется поиск.

Команда имеет один операнд, обозначающий местонахождение цепочки в дополнительном сегменте (адрес цепочки должен быть заранее сформирован в ES:DI). Транслятор анализирует тип идентификатора приемник, который обозначает цепочку в сегменте данных, и формирует одну из машинных команд, *scasb*, *scasw*. Условие поиска для каждой из этих команд находится в строго определенном месте. Так, если цепочка описана с помощью директивы *db*, то искомый элемент должен быть байтом и находиться в AL, а сканирование цепочки осуществляется командой *scasb*; если цепочка описана с помощью директивы *dw*, то это — слово в *ax*, и поиск ведется командой *scasw*. Принцип поиска тот же, что и в команде сравнения *CMPS*, то есть производится последовательное выполнение вычитания (регистр аккумулятора – элемент цепочки), не записывая при этом результата, и устанавливаются флаги *CF*, *OF*, *SF*, *ZF*, *AF* и *PF* в соответствии с результатом.. При этом сами операнды не изменяются. Так же, как и в случае команды *CMPS*, с командой *SCAS* удобно использовать префиксы *REPЕ/REPZ* или *REPNE/REPNZ*:

Таким образом, команда *SCAS* с префиксом *REPЕ/REPZ* позволяет найти элемент цепочки, отличающийся по значению от заданного в аккумуляторе. Команда *SCAS* с префиксом *REPNE/REPNZ* позволяет найти элемент цепочки, совпадающий по значению с элементом в аккумуляторе.

ОПЕРАНДЫ ЯЗЫКА АССЕМБЛЕР

Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков **операций** и некоторых **зарезервированных слов**.

Операнды могут комбинироваться с арифметическими, логическими, побитовыми и атрибутивными операторами для расчета некоторого значения или определения ячейки памяти, на которую будет воздействовать данная команда или директива.

Под способами адресации понимаются существующие способы задания адреса хранения операндов.

- *Операнд задается на микропрограммном уровне:* в этом случае команда явно не содержит операнда, алгоритм выполнения команды использует некоторые объекты по умолчанию (регистры, признаки и т.д.).

`mul bx` ; неявно использует регистр `ax`

- *Операнд задается в самой команде (непосредственный операнд):* операнд является частью кода команды. Для хранения такого операнда в команде выделяется поле длиной до 32 бит. Непосредственный операнд может быть только вторым операндом (источником). Операнд-получатель может находиться либо в памяти, либо в регистре.

`mov ax, 0ffffh` ; пересылает в регистр `ax` константу `ffff`

`add sum, 2` ; складывает содержимое поля по адресу `sum`

; с целым числом 2 и записывает результат в `sum`.

- *Операнд находится в одном из регистров (регистровый операнд):* в коде команды указываются именами регистров. В качестве регистров могут использоваться:

- 32-разрядные регистры `EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP`;

- 16-разрядные регистры `AX, BX, CX, DX, SI, DI, SP, BP`;

- 8-разрядные регистры `AH, AL, BH, BL, CH, CL, DH, DL`;

- сегментные регистры `CS, DS, SS, ES, FS, GS`.

`add ax, bx` ; складывает содержимое регистров `ax` и `bx`

; и записывает результат в `ax`

`dec si` ; уменьшает содержимое `si` на 1.

- *Операнд располагается в памяти.* Данный способ позволяет реализовать два основных вида адресации: *прямую адресацию* и *косвенную адресацию*.

Прямая адресация: эффективный адрес берется непосредственно из поля смещения машинной команды, которое может иметь размер 8, 16 или 32 бита.

`mov ax, es:0001` ; помещает в `ax` слово из сегмента, на который

; указывает `es` со смещением от начала сегмента `0001`.

`mov ax, word_var` ; ассемблер заменяет `word_var` на соответствующий

; адрес, использует сегмент по умолчанию `ds`.

Косвенная адресация:

Косвенная базовая (регистровая) адресация.

При такой адресации эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме `sp/esp` и `bp/ebp` (это специфические регистры для работы с сегментом стека). Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки [].

`mov ax, [ecx]` ; помещает в регистр `ax` содержимое слова по адресу

; из сегмента данных со смещением,

; хранящимся в регистре `ecx`.

Данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство полезно для организации циклических вычислений и для работы со структурами данных (таблицы, массивы).

Косвенная базовая (регистровая) адресация со смещением

Предназначена для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее, на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически, на стадии выполнения программы. Модификация содержимого базового регистра позволяет обратиться к одноименным элементам различных экземпляров однотипных структур данных.

`mov ax, [edx+3h]` ; пересылает в регистр `ax` слова из области памяти

; по адресу содержимое `[edx + 3h]`.

`mov ax, mas[dx]` ; пересылает в регистр `ax` слово по адресу

; содержимое `dx` + значение идентификатора

; `mas` (смещение от начала сегмента данных)

Косвенная индексная адресация со смещением

Для формирования эффективного адреса используется один из регистров общего назначения, но обладает возможностью масштабирования содержимого индексного регистра.

`mov ax, mas[si*2]` ; значение эффективного адреса второго операнда

; вычисляется выражением $mas + (si) * 2$.

Наличие возможности масштабирования существенно помогает в решении проблемы индексации при условии, что размер элементов массива постоянен и составляет 1, 2, 4 или 8 байт.

Косвенная базовая индексная адресация

Эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто используется масштабирование содержимого индексного регистра.

`mov eax, [esi][edx]` ; эффективный адрес второго операнда

; формируется как $(esi) + (edx)$.

Косвенная базовая индексная адресация со смещением

Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде.

`mov eax, [esi+5][edx]` ; пересылает в регистр `eax` двойное слово

; по адресу: $(esi) + 5 + (edx)$.

`add ax, array[esi][ebx]` ; производит сложение содержимого

; регистра `ax` с содержимым слова по адресу

; идентификатор $array + (esi) + (ebx)$.

- *Операндом является порт ввода/вывода.* Помимо адресного пространства оперативной памяти микропроцессор поддерживает адресное пространство ввода-вывода, которое используется для доступа к устройствам ввода-вывода.

Объем адресного пространства ввода-вывода составляет 64 Кбайт. Для любого устройства компьютера в этом пространстве выделяются адреса. Конкретное значение адреса в пределах этого пространства называется портом ввода-вывода. Физически порту ввода-вывода соответствует аппаратный регистр (не путать с регистром микропроцессора), доступ к которому осуществляется с помощью специальных команд ассемблера `in` и `out`.

```
in  al, 60h ; ввести байт из порта 60h
```

Регистры, адресуемые с помощью порта ввода-вывода, могут иметь разрядность 8, 16 или 32 бит, но для конкретного порта разрядность регистра фиксирована. В качестве источника информации или получателя применяются регистры-аккумуляторы `eax`, `ax`, `al`. Выбор регистра определяется разрядностью порта. Номер порта может задаваться непосредственным операндом в командах `in` и `out` или значением в регистре `dx`. Последний способ позволяет динамически определить номер порта в программе.

```
mov  dx, 20h ; записать номер порта 20h в регистр dx
```

```
mov  al, 21h ; записать значение 21h в регистр al
```

```
out  dx, al ; вывести значение 21h в порт 20h
```

- *Операнд находится в стеке.* Команды могут совсем не иметь операндов, иметь один или два операнда. Большинство команд требуют двух операндов, один из которых является операндом-источником, а второй — операндом назначения. Один операнд может располагаться в регистре или памяти, а второй операнд обязательно должен находиться в регистре или непосредственно в команде. Непосредственный операнд может быть только операндом-источником. В двухоперандной машинной команде возможны следующие сочетания операндов:
- *Счетчик адреса* – специфический вид операнда. Он обозначается знаком `$`. Специфика этого операнда в том, что когда транслятор ассемблера встречается в исходной программе этот символ, то он подставляет вместо него текущее значение счетчика адреса (регистр `EIP`). Значение счетчика адреса представляет собой смещение текущей машинной команды относительно начала сегмента кода. При обработке транслятором очередной команды

ассемблера счетчик адреса увеличивается на длину сформированной машинной команды. Обработка директив ассемблера не влечет за собой изменения счетчика. В качестве примера использования в команде значения счетчика адреса можно привести следующий фрагмент:

```
jmp $+3 ; безусловный переход на команду mov
nop      ; длина команды еld составляет 1 байт
mov al,1
```

При использовании подобного выражения для перехода нельзя забывать о длине самой команды, в которой это выражение используется, так как значение счетчика адреса соответствует смещению в сегменте кода данной, а не следующей за ней команды. В приведенном выше примере команда `jmp` занимает 2 байта. Длина этой и некоторых других команд может зависеть от того, какие в ней используются операнды. Команда с регистровыми операндами будет короче команды, один из операндов которой расположен в памяти. В большинстве случаев эту информацию можно получить, зная формат машинной команды.

- *Структурные операнды* используются для доступа к конкретному элементу сложного типа данных, называемого структурой.
- *Записи* (аналогично структурному типу) используются для доступа к битовому полю некоторой записи. Для доступа к битовому полю записи используется директива `RECORD`.

Операнды являются элементарными компонентами, из которых формируется часть машинной команды, обозначающая объекты, над которыми выполняется операция. В более общем случае операнды могут входить как составные части в более сложные образования, называемые **выражениями**. Выражения представляют собой комбинации операндов и операторов, рассматриваемые как единое целое. Результатом вычисления выражения может быть адрес некоторой ячейки памяти или некоторое константное (абсолютное) значение.

Выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами (см. табл. 6). Операции с одинаковыми приоритетами выполняются последовательно слева направо. Изменение порядка

выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

Таблица 5

Оператор	Приоритет
<code>length, size, width, mask, (), [], < ></code>	1
<code>.</code>	2
<code>:</code>	3
<code>ptr, offset, seg, this</code>	4
<code>high, low</code>	5
<code>+, - (унарные)</code>	6
<code>*, /, mod, shl, shr</code>	7
<code>+, -, (бинарные)</code>	8
<code>eq, ne, lt, le, gt, ge</code>	9
<code>not</code>	10
<code>and</code>	11
<code>or, xor</code>	12
<code>short, type</code>	13

Дадим краткую характеристику основных операторов.

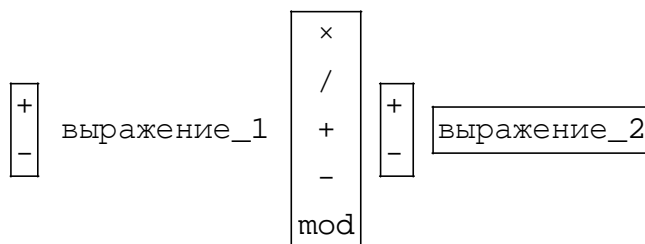
- **Арифметические операторы.** К ним относятся унарные операторы «+» и «-», бинарные «+» и «-», операторы умножения «*», целочисленного деления «/», получения остатка от деления «mod». Например,

```

tabsize equ 50 ;размер массива в байтах
sizeel equ 2 ;размер элементов
;вычисляется число элементов массива и заносится в регистр cx
mov cx,tab_size / sizeel ;оператор "/"

```

Синтаксис арифметических операторов



- **Операторы сдвига** выполняют сдвиг выражения на указанное количество разрядов Например:

```

mask_b equ 10111011
...
mov al,mask_b shr 3 ; al=00010111

```

Синтаксис операторов сдвига

выражение shr числодвигаемыхразрядов
 shl

- **Операторы сравнения** (возвращают значение «истина» или «ложь») предназначены для формирования логических выражений. Логическое значение «истина» соответствует логической единице, а «ложь» – логическому нулю. Логическая единица – значение, все биты которого равны 1. Соответственно, логический нуль – значение, все биты которого равны 0.

```
tab_size        equ 30    ;размер таблицы
...
mov al,tab_size ge 50 ;загрузка размера таблицы в al
cmp al,0        ;если tabsize < 50, то
je  m1            ;переход на m1
...
m1: ...
```

В этом примере, если значение `tab_size` больше или равно 50, то результат в `al` равен `FFh`, а если `tab_size` меньше 50, то `al` равно `00h`. Команда `cmp` сравнивает значение `al` с нулем и устанавливает соответствующие флаги в `FLAGS/EFLAGS`. Команда `je` на основе анализа этих флагов передает или не передает управление на метку `m1`.

Синтаксис операторов сравнения

```
                                          eq
                                          ne
                                          lt
                  выражение_1            выражение_2
                                          le
                                          gt
                                          ge
```

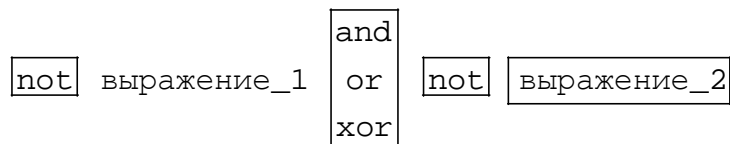
Таблица 6

Оператор	Значение	Условие
<code>eq</code>	«ИСТИНА»	выражение_1 равно выражению_2
<code>ne</code>	«ИСТИНА»	выражение_1 не равно выражению_2
<code>lt</code>	«ИСТИНА»	выражение_1 меньше, чем выражение_2
<code>le</code>	«ИСТИНА»	выражение_1 меньше или равно выражению_2
<code>gt</code>	«ИСТИНА»	выражение_1 больше, чем выражение_2
<code>ge</code>	«ИСТИНА»	выражение_1 больше или равно выражению_2

- **Логические операторы** выполняют над выражениями побитовые операции. Выражения должны быть абсолютными, то есть такими, численное значение которых может быть вычислено транслятором. Например,

```
L1 equ 10010011
...
mov al,L1 xor 01h ;al=10010010
```

Синтаксис логических операторов

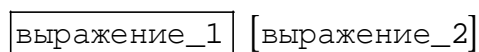


- **Индексный оператор []**. Транслятор воспринимает наличие квадратных скобок как указание сложить значение выражения_1 за этими скобками со значением выражения_2, заключенным в скобки. Например,

```
mov ax,mas[si] ;ax=*(mas+(si))
```

В литературе принято следующее обозначение: когда в тексте речь идет о содержимом регистра, то его название берут в круглые скобки.

Синтаксис индексного оператора []



- **Оператор переопределения типа ptr** применяется для переопределения или уточнения типа метки или переменной, определяемых выражением. Тип может принимать одно из следующих значений: byte, word, dword, qword, tbyte, near, far. Например,

```
d_wrd dd 0000000100100011b
...
mov al,byte ptr d_wrd+1 ;al=0001
```

Переменная d_wrd имеет тип двойного слова. Допустим, возникнет необходимость обращения не ко всему значению переменной, а только к одному из входящих в нее байтов (в примере – ко второму). Если попытаться сделать это командой `mov al,d_wrd+1`, то транслятор выдаст сообщение о несовпадении типов операндов. Оператор ptr позволяет непосредственно в команде переопределить тип и выполнить команду.

Синтаксис оператора ptr

```
byte
word
dword ptr выражение
qword
tword
```

- **Оператор переопределения сегмента «:» (двоеточие)** заставляет вычислять физический адрес относительно конкретно задаваемой сегментной составляющей: «имя сегментного регистра», «имя сегмента» из соответствующей директивы SEGMENT или «имя группы». Для выборки на выполнение очередной команды микропроцессор должен обязательно посмотреть содержимое сегментного регистра CS. А в этом регистре, как мы знаем, содержится (пока еще не сдвинутый) физический адрес начала сегмента команд. Для получения адреса конкретной команды микропроцессору остается умножить содержимое CS на 16 (что означает сдвиг на четыре разряда) и сложить полученное 20-битное значение с 16-битным содержимым регистра IP. Примерно то же самое происходит и тогда, когда микропроцессор обрабатывает операнды в машинной команде. Если он видит, что операнд – это адрес (эффективный адрес, который является только частью физического адреса), то он знает, в каком сегменте его искать (по умолчанию это сегмент, адрес начала которого записан в сегментном регистре DS).

```
.code
jmp met1      ; обход обязателен, иначе поле ind будет
              ; трактоваться как очередная команда
ind db 5      ; описание поля данных в сегменте команд
met1:  mov al,cs:ind    ; al=5
```

Синтаксис оператора переопределения сегмента

```
cs
ds
es          : выражение
ss
имя сегмента
имя группы
```

- **Оператор именованного типа структуры «.» (точка)** также заставляет транслятор производить определенные вычисления, если он встречается в выражении.

- **Оператор получения сегментной составляющей адреса выражения** `seg` возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.

Синтаксис оператора получения сегментной составляющей

`seg` выражение

- **Оператор получения смещения выражения** `offset` позволяет получить значение смещения выражения в байтах относительно начала того сегмента, в котором выражение определено. Например,

```
.data
pole    dw      5
.code
mov ax,seg pole
mov es,ax
mov dx,offset pole    ;es:dx = полный адрес pole
```

Синтаксис оператора получения смещения выражения

`offset` выражение

- **Оператор определения длины массива** `length` возвращает число элементов, определенных операндом `dup`. Если операнд `dup` отсутствует, то оператор `length` возвращает значение 01. Например,

```
tablea  dw  10 dup(?)
...
mov     dx,length tablea    ; dx=000a (10)
```

- **Оператор** `type` возвращает число байтов, соответствующее определению указанной переменной:

```
fldb    db  ?
tablea  dw  10 dup(?)    ;Определение 20 слов
...
mov ax,type fldb        ;ax = 0001
mov ax,type tablea      ;ax = 0002
```

Так как область `tablea` определена как `dw`, то оператор `type` возвращает `0002h`.

- **Оператор** `size` возвращает произведение длины `length` и типа `type` и используется при ссылках на переменную с операндом `dup`. Синтаксис оператора:

size переменная

Для предыдущего примера

```
mov dx,size tablea ;dx = 0014h (20) (10*2)
```

- **Оператор short** –модификация атрибута near в команде jmp, если переход не превышает границы +127 и -128 байт. Например,

```
jmp short метка
```

В результате ассемблер сокращает машинный код операнда от двух до одного байта. Эта возможность оказывается полезной для коротких переходов вперед, так как в этом случае ассемблер не может сам определить расстояние до адреса перехода и резервирует два байта при отсутствии оператора short.

- **Оператор width** возвращает размер в битах объекта типа RECORD или его поля. Например,

```
mov ax,WIDTH MY_REC ;ax=13  
mov bx,WIDTH Bit8_11 ;cx=4
```

СТАНДАРТНЫЕ ДИРЕКТИВЫ СЕГМЕНТАЦИИ

Для задания сегментов/секций в тексте программы наряду с упрощенными директивами (.STACK, .DATA, .CODE) может также использоваться директива SEGMENT, которая определяет начало любого сегмента/секции. Синтаксис:

```
name SEGMENT align combine dim 'class'  
...  
name ENDS
```

Директива ENDS определяет конец сегмента.

Физически сегмент представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Функциональное назначение сегмента несколько шире, чем простое разбиение программы на блоки кода, данных и стека. Сегментация является частью более общего механизма, связанного с концепцией модульного программирования. Она предполагает унификацию оформления объектных модулей, создаваемых компилятором, в том числе с разных языков программирования. Это позволяет объединять программы, написанные на разных языках. Именно для реализации различных вариантов такого объединения и предназначены операнды в директиве SEGMENT.

- Атрибут выравнивания сегмента (тип выравнивания) `align` сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i8086 выполняется быстрее. Допустимые значения этого атрибута следующие:

BYTE — выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;

WORD — сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);

DWORD — сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита равны 0 (выравнивание на границу двойного слова);

PARA — сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);

PAGE — сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу страницы размером 256 байт);

MEMPAGE — сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей страницы памяти размером 4 Кбайт);

По умолчанию тип выравнивания имеет значение PARA.

- Атрибут комбинирования сегментов (комбинаторный тип) `combine` сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. По умолчанию атрибут комбинирования принимает значение `PRIVATE`. Значениями атрибута комбинирования сегмента могут быть:

`PRIVATE` — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;

`PUBLIC` — заставляет компоновщик соединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;

`COMMON` — располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;

`AT xxxx` — располагает сегмент по абсолютному адресу параграфа (параграф — объем памяти, кратный 16, поэтому последняя шестнадцатеричная

цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением `xxxx`. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;

```
seg_name SEGMENT AT 0FE00h  
;располагает seg_name по адресу 0FE00h
```

STACK — определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра `SS`. Комбинированный тип **STACK** (стек) аналогичен комбинированному типу **PUBLIC**, за исключением того, что регистр `SS` является стандартным сегментным регистром для сегментов стека. Регистр `SP` устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип **STACK** не используется, программист должен явно загрузить в регистр `SS` адрес сегмента (подобно тому, как это делается для регистра `DS`).

- Атрибут размера сегмента `dim`. Для процессоров `i80386` и выше сегменты могут быть 16- или 32-разрядными. Это влияет прежде всего на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:

USE16 — это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;

USE32 — сегмент будет 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

- Атрибут класса сегмента (тип класса) `'class'` — это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при сборке программы из сегментов нескольких модулей. Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента). Типичным примером использования имени класса является объединение в группу всех сегментов кода программы (обычно для этого используется класс «code»). С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных.

Все сегменты сами по себе равноправны, так как директивы `SEGMENT` и `ENDS` не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву `ASSUME`.

Синтаксис:

```
nothing
CS: имясегмента
    nothing
DS: имясегмента
assume  nothing
ES: имясегмента
    nothing
SS: имясегмента
    nothing
```

Эта директива сообщает транслятору о том, какой сегмент к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах. Привязка сегментов к сегментным регистрам осуществляется с помощью операндов этой директивы, в

которых `имя_сегмента` должно быть именем сегмента, определенным в исходном тексте программы директивой `SEGMENT` или ключевым словом `nothing`. Если в качестве операнда используется только ключевое слово `nothing`, то предшествующие назначения сегментных регистров аннулируются, причем сразу для всех шести сегментных регистров. Но ключевое слово `nothing` можно использовать вместо аргумента имя сегмента; в этом случае будет выборочно разрываться связь между сегментом с именем `имя_сегмента` и соответствующим сегментным регистром.

Директива `SEGMENT` может применяться с любой моделью памяти. При использовании директивы `SEGMENT` с моделью `flat` потребуется указать компилятору на то, что все сегментные регистры устанавливаются в соответствии с моделью памяти `flat`. Это можно сделать при помощи директивы `ASSUME`:

```
ASSUME CS:FLAT, DS:FLAT, SS:FLAT, ES:FLAT, FS:ERROR, GS:ERROR
```

Регистры `FS` и `GS` программами не используются, поэтому для них указывается атрибут `ERROR`.

МАКРОКОМАНДЫ

При программировании достаточно серьезной задачи, появляются повторяющиеся участки кода. Они могут быть небольшими, а могут занимать и достаточно много места. В последнем случае эти фрагменты будут существенно затруднять чтение текста программы, снижать ее наглядность, усложнять отладку и служить неисчерпаемым источником ошибок. В языке ассемблера есть несколько средств, решающих проблему дублирования участков программного кода. К ним относятся:

- макроассемблер;
- механизм процедур;
- механизм прерываний.

Макрокоманда представляет собой строку, содержащую некоторое символическое имя — имя макрокоманды, предназначенную для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами. Для написания макрокоманды вначале необходимо задать ее шаблон-описание, который называют **макроопределением**.

Синтаксис макроопределения следующий:

```
имя_макрокоманды macro список_формальных_аргументов
тело макроопределения
endm
```

Макроопределение обрабатывается транслятором особым образом. Для того чтобы использовать описанное макроопределение в нужном месте программы, оно должно быть активизировано с помощью макрокоманды указанием следующей синтаксической конструкции:

```
имя_макрокоманды список_фактических_аргументов
```

Результатом применения данной синтаксической конструкции в исходном тексте программы будет ее замещение строками из конструкции тела макроопределения. Но это не простая замена. Обычно макрокоманда содержит некоторый список аргументов — список_фактических_аргументов, которыми корректируется макроопределение. Места в теле макроопределения, которые будут замещаться фактическими аргументами из макрокоманды, обозначаются с помощью так называемых формальных аргументов. Таким образом, в результате применения

макрокоманды в программе формальные аргументы в макроопределении замещаются соответствующими фактическими аргументами; в этом и заключается учет контекста. Процесс такого замещения называется макрогенерацией, а результатом этого процесса является макрорасширение.

Макрокоманды в ассемблере схожи с директивой `#define` в языке Си, но, в отличие от нее, могут принимать аргументы.

Есть три варианта размещения макроопределений:

- в начале исходного текста программы, до кода и данных с тем, чтобы не ухудшать читаемость программы. Этот вариант следует применять в случаях, если определяемые макрокоманды актуальны только в пределах одной этой программы;
- в отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву `include имя_файла`, например:

```
.586  
.model flat, stdcall  
include show.inc ;сюда вставляется текст файла show.inc
```
- в макробιβлиотеке. Универсальные макрокоманды, которые используются практически во всех программах целесообразно записать в так называемую макробιβлиотеку. Сделать актуальными макрокоманды из этой бιβлиотеки можно также с помощью директивы `include`. Недостаток этого и предыдущего способов в том, что в исходный текст программы включаются абсолютно все макроопределения. Для исправления ситуации можно использовать директиву `purge`, в качестве операндов которой через запятую перечисляются имена макрокоманд, которые не должны включаться в текст программы. К примеру:

```
include iomac.inc  
purge outstr,exit
```

В данном случае в исходный текст программы перед началом трансляции MASM вместо строки `include iomac.inc` вставит строки из файла

iomac.inc. Но вставленный текст будет отличаться от оригинала тем, что в нем будут отсутствовать макроопределения `outstr` и `exit`.

Каждый фактический аргумент представляет собой строку символов, для формирования которой применяются следующие правила:

1. Строка может состоять:

- из последовательности символов без пробелов, точек, запятых, точек с запятой;
- из последовательности любых символов, заключенных в угловые скобки: `<...>`. В этой последовательности можно указывать как пробелы, так и точки, запятые, точки с запятыми.

2. Для того чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр, является собственно символом, а не чем-то иным, например, некоторым разделителем или ограничивающей скобкой, применяется специальный оператор «!». Этот оператор ставится непосредственно перед описанным выше символом, и его действие эквивалентно заключению данного символа в угловые скобки.

3. Если требуется вычисление в строке некоторого константного выражения, то в начале этого выражения нужно поставить знак %:

`%константное_выражение` — значение `константное_выражение` вычисляется и подставляется в текстовом виде в соответствии с текущей системой счисления.

В процессе генерации макрорасширения транслятор ассемблера ищет в тексте тела макроопределения последовательности символов, совпадающие с теми последовательностями символов, из которых состоят формальные параметры. После обнаружения такого совпадения формальный параметр из тела макроопределения замещается соответствующим фактическим параметром из макрокоманды. Этот процесс называется **подстановкой аргументов**. В общем случае список формальных аргументов содержит не только перечисление формальных элементов через запятую, но и некоторую дополнительную информацию. Полный синтаксис формального аргумента следующий:

`имя_формального_аргумента [: тип]`

где тип может принимать значения:

- REQ – требуется обязательное явное задание фактического аргумента при вызове макрокоманды;
- =<любая_строка> — если аргумент при вызове макрокоманды не задан, то в соответствующие места в макрорасширении будет вставлено значение по умолчанию, соответствующее значению любая_строка. Символы, входящие в любая_строка, должны быть заключены в угловые скобки.

Но не всегда ассемблер может распознать в теле макроопределения формальный аргумент. Это, например, может произойти в случае, когда он является частью некоторого идентификатора. В этом случае последовательность символов формального аргумента отделяют от остального контекста с помощью специального символа &. Этот прием часто используется для задания модифицируемых идентификаторов и кодов операций. Например,

```
.686p
.model flat, stdcall
def_table macro t:REQ, len:=<1>
tabl_&t d&t len dup(5)
endm
.data
def_table d, 10
def_table b
.code
main proc
mov al, [tabl_b]
mov ah, [tabl_b+1]
mov ebx, [tabl_d]
ret
main endp
end main
```

После трансляции текста программы, содержащего строки секции данных, получится

```
def_table d, 10          ;tabl_d dd 10 dup(5)
def_table b              ;tabl_b db 1 dup(5)
```

Заметим, что строка программы `mov ah, [tabl_b+1]` поместит в `ah` число отличное от 5, поскольку память для `tabl_b` распределена только под 1 элемент массива длиной 1 байт со значением 5.

Символ & можно применять и для распознавания формального аргумента в строке, заключенной в кавычки " ".

Если тело макроопределения содержит метку или имя в директиве резервирования и инициализации данных, и в программе данная макрокоманда вызывается несколько раз, то в процессе макрогенерации возникнет ситуация, когда в программе один идентификатор будет определен несколько раз, что, естественно, будет распознано транслятором как ошибка. Для выхода из подобной ситуации применяют директиву `local`, которая имеет следующий синтаксис:

```
local список_идентификаторов
```

Эту директиву необходимо задавать непосредственно за заголовком макроопределения. Результатом работы этой директивы будет генерация в каждом экземпляре макрорасширения уникальных имен для всех идентификаторов, перечисленных в список_идентификаторов. Эти уникальные имена имеют вид `??xxxx`, где `xxxx` — шестнадцатеричное число. Для первого идентификатора в первом экземпляре макрорасширения `xxxx=0000`, для второго — `xxxx=0001` и т. д. Контроль за правильностью размещения и использования этих уникальных имен берет на себя ассемблер.

Для примера использования макроопределений рассмотрим программу вывода имени в диалоговое окно.

```
.686P
.MODEL FLAT, STDCALL
PrintName macro Name
local STR1, STR2, МЕТКА
    jmp МЕТКА
STR1    DB "Программа",0
STR2 DB "Меня зовут: &Name ",0
МЕТКА:
    PUSH    0
    PUSH    OFFSET STR1
    PUSH    OFFSET STR2
    PUSH    0
    CALL    MessageBoxA@16
endm
Init macro
EXTERN MessageBoxA@16:NEAR
endm
Init
.CODE
START:
PrintName <Лена>
PrintName <Таня>
PrintName <Алёша>
    RET
END START
```

При использовании макроопределений код программы становится более читаемым.

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз где-то описать, а затем вызывать их многократно специальным образом. Различия макроопределений и процедур в зависимости от целевой установки можно рассматривать и как достоинства, и как недостатки:

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении менее гибки;
- при каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

ПРОЦЕДУРЫ (ФУНКЦИИ)

Процедура (подпрограмма) — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого приоритета и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Процедуру можно определить и как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Функция – процедура, способная возвращать некоторое значение.

Процедуры ценны тем, что могут быть активизированы в любом месте программы. Процедурам могут быть переданы некоторые аргументы, что позволяет, имея одну копию кода в памяти, изменять ее для каждого конкретного случая использования.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC и ENDP.

Синтаксис описания процедуры:

```
имя_процедуры PROC [язык] [расстояние]
    команды, директивы ; тело процедуры
                                ; языка ассемблера
[имя_процедуры] ENDP
```

В заголовке процедуры (директиве PROC) обязательным является только задание имени процедуры. Атрибут [расстояние] может принимать значения *near* или *far* и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут [расстояние] принимает значение *near*.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока и, соответственно, будет осуществлять выполнение команд процедуры. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

- в начале программы (до первой исполняемой команды);

- в конце (после команды, возвращающей управление операционной системе);
- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы;
- в другом модуле.

Размещение процедуры в начале сегмента кода предполагает, что последовательность команд, ограниченная парой директив PROC и ENDP, будет размещена до метки, обозначающей первую команду, с которой начинается выполнение программы. Эта метка должна быть указана как параметр директивы END, обозначающей конец программы:

```
...
.code
myproc proc near
ret
my_proc endp
start proc
call myproc
...
start endp
end start
```

В этом фрагменте после загрузки программы в память управление будет передано первой команде процедуры с именем `start`.

Объявление имени процедуры в программе равнозначно объявлению метки, поэтому директиву PROC в частном случае можно рассматривать как форму определения метки в программе.

Размещение процедуры в конце программы предполагает, что последовательность команд, ограниченная директивами PROC и ENDP, будет размещена после команды, возвращающей управление операционной системе.

Промежуточный вариант расположения тела процедуры предполагает ее размещение внутри другой процедуры или основной программы. В этом случае возможна необходимость предусмотреть обход тела процедуры, ограниченного директивами PROC и ENDP, с помощью команды безусловного перехода `jmp`:

```
...
.code
start proc
jmp ml
my_proc proc near
ret
myproc endp
```

```
ml:  
...  
start endp  
end start
```

Последний вариант расположения описаний процедур — в отдельном сегменте кода — предполагает, что часто используемые процедуры выносятся в отдельный файл (модуль). Файл с процедурами должен быть оформлен как обычный исходный файл и подвергнут трансляции для получения объектного кода. Впоследствии этот объектный файл утилитой LINK можно объединить с файлом, где процедуры используются. Этот способ предполагает наличие в исходном тексте программы еще некоторых элементов, связанных с особенностями реализации концепции модульного программирования в языке ассемблера. Вариант расположения процедур в отдельном модуле используется также при построении Windows-приложений на основе выхода API-функций.

Так как имя процедуры обладает теми же атрибутами, что и обычная метка в команде перехода, то обратиться к процедуре можно с помощью любой команды перехода. Но есть одно важное свойство, которое можно использовать благодаря специальному механизму вызова процедур. Суть состоит в возможности сохранения информации о контексте программы в точке вызова процедуры. Под контекстом понимается информация о состоянии программы в точке вызова процедуры. В системе команд микропроцессора есть две команды, осуществляющие работу с контекстом. Это команды `call` и `ret`:

`call имя_процедуры@num` — вызов процедуры (подпрограммы).

`ret [число]` — возврат управления вызывающей программе.

[число] — необязательный параметр, обозначающий количество элементов, удаляемых из стека при возврате из процедуры.

@num – количество байтов, которое занимают в стеке переданные аргументы для процедуры (параметр является особенностью использования компилятора MASM).

Особого внимания заслуживает вопрос размещения процедуры в другом модуле. Так как отдельный модуль — это функционально автономный объект, то он ничего не должен, знать о внутреннем устройстве других модулей, и наоборот, другим модулям также ничего не известно о внутреннем устройстве данного модуля. Но каждый модуль должен иметь такие средства, с помощью которых он извещал бы

транслятор о том, что некоторый объект (процедура, переменная) должен быть видимым вне этого модуля. И наоборот, нужно объяснить транслятору, что некоторый объект находится вне данного модуля. Это позволит транслятору правильно сформировать машинные команды, оставив некоторые их поля незаполненными. Позднее, на этапе компоновки, программа LINK или программа компоновки языка высокого уровня произведут настройку модулей и разрешат все внешние ссылки в объединяемых модулях.

Для того чтобы объявить о видимых извне объектах, программа должна использовать две директивы MASM: `extern` и `public`. Директива `extern` предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве `public`. Директива `public` предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях. Синтаксис этих директив следующий:

```
extern имя:тип, . . . , имя:тип
public имя, . . . , имя
```

Здесь `имя` — идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

- имена переменных;
- имена процедур;
- имена констант.

Тип определяет тип идентификатора. Указание типа необходимо для того, чтобы транслятор правильно сформировал соответствующую машинную команду. Действительные адреса будут вычислены на этапе редактирования, когда будут разрешаться внешние ссылки. Возможные значения типа определяются допустимыми типами объектов для этих директив:

- если `имя` — это имя переменной, то тип может принимать значения `byte`, `word`, `dword`, `qword` и `tbyte`;
- если `имя` — это имя процедуры, то тип может принимать значения `near` или `far`; в компиляторе MASM после имени процедуры необходимо указывать число байтов в стеке, которые занимают аргументы функции:

```
extern p1@0:near
```

- если имя — это имя константы, то тип должен быть `abs`.

Пример использования директив `extern` и `public` для двух модулей

```

;Модуль 1
.586
.model flat, stdcall
.data
extern p1@0:near
.code
start proc
call p1@0
ret
start endp
end start

;Модуль 2
.586
.model flat, stdcall
public p1
.data
.code
p1 proc
ret
p1 endp
end

```

Исполняемый модуль находится в программе *Модуль 1*, поскольку содержит метку `start`, с которой начинается выполнение программы (эта метка указана после директивы `end` в программе *Модуль 1*). Программа вызывает процедуру `p1`, внешнюю, содержащуюся в файле *Модуль 2*. Процедура `p1` не имеет аргументов, поэтому описывается в программе *Модуль 1* с помощью директивы

```
extern p1@0:near
```

`@0` – количество байт, переданных функции в качестве аргументов

`near` – тип функции (для плоской модели памяти всегда имеет тип `near`).

Вызов процедуры осуществляется командой `call p1@0`.

Организация интерфейса с процедурой

Аргумент — это ссылка на некоторые данные, которые требуются для выполнения возложенных на модуль функций и размещенных вне этого модуля. По аналогии с макрокомандами рассматривают понятия формального и фактического аргументов. Исходя из этого, формальный аргумент можно рассматривать не как непосредственные данные или их адрес, а как «местодержатель» для действительных данных, которые будут подставлены в него с помощью фактического аргумента. Формальный аргумент можно рассматривать как элемент интерфейса модуля, а фактический аргумент — это то, что фактически передается на место формального аргумента.

Переменные — это данные размещенные в регистре или ячейке памяти, которые могут в дальнейшем подвергаться изменению.

Константы — данные, значения которых не могут изменяться.

Сигнатура процедуры (функции) — это имя функции, тип возвращаемого значения и список аргументов с указанием порядка их следования и типов.

Семантика процедуры (функции) — это описание того, что данная функция делает. Семантика функции включает в себя описание того, что является результатом вычисления функции, как и от чего этот результат зависит. Обычно результат выполнения зависит только от значений аргументов функции, но в некоторых модулях есть понятие состояния. Тогда результат функции может зависеть от этого состояния, и, кроме того, результатом может стать изменение состояния. Логика этих зависимостей и изменений относится к семантике функции. Полным описанием семантики функций является исполняемый код функции или математическое определение функции.

Если переменная находится за пределами модуля (процедуры) и должна быть передана в него, то для модуля она является формальным аргументом. Значение переменной передается в модуль для замещения соответствующего параметра при помощи фактического аргумента.

Как правило, один и тот же модуль можно использовать многократно для разных наборов значений для формальных аргументов. Для передачи аргументов в языке ассемблера существуют следующие способы:

- через регистры;
- через общую область памяти;
- через стек;
- с помощью директив `extrn` и `public`.

Передача аргументов через регистры — это наиболее простой в реализации способ передачи данных. Данные, переданные подобным способом, становятся доступными немедленно после передачи управления процедуре. Этот способ очень популярен при небольшом объеме передаваемых данных.

Ограничения на способ передачи аргументов через регистры:

- небольшое число доступных для пользователя регистров;
- нужно постоянно помнить о том, какая информация в каком регистре находится;

- ограничение размера передаваемых данных размерами регистра. Если размер данных превышает 8, 16 или 32 бита, то передачу данных посредством регистров произвести нельзя. В этом случае передавать нужно не сами данные, а указатели на них.

Передача аргументов через общую область памяти – предполагает, что вызывающая и вызываемая программы используют некоторую область памяти как общую. Для организации такой области памяти используется атрибут комбинирования сегментов. Наличие этого атрибута указывает компоновщику LINK, как нужно комбинировать сегменты, имеющие одно имя. Значение `common` означает, что все сегменты, имеющие одинаковое имя в объединяемых модулях, будут располагаться компоновщиком, начиная с одного адреса оперативной памяти. Это значит, что они будут просто перекрываться в памяти и, следовательно, совместно использовать выделенную память. Данные в сегментах `common` могут иметь одинаковые имена. Главное – структура общих сегментов. Она должна быть идентична во всех модулях использующих обмен данными через общую память.

Недостатком этого способа в реальном режиме работы микропроцессора является отсутствие средств защиты данных от разрушения, так как нельзя проконтролировать соблюдение правил доступа к этим данным.

Передача аргументов через стек

Этот способ наиболее часто используется для передачи аргументов при вызове процедур. Суть его заключается в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего передает управление вызываемой процедуре. При передаче управления процедуре микропроцессор автоматически записывает в вершину стека четыре байта. Эти байты являются адресом возврата в вызывающую программу. Если перед передачей управления процедуре командой `call` в стек были записаны переданные процедуре данные или указатели на них, то они окажутся под адресом возврата.

Стек обслуживается тремя регистрами: `ESS`, `ESP` и `EBP`. Микропроцессор автоматически работает с регистрами `ESS` и `ESP` в предположении, что они всегда указывают на дно и вершину стека соответственно. По этой причине их содержимое изменять не рекомендуется. Для осуществления произвольного доступа к данным в

стеке архитектура микропроцессора имеет специальный регистр EBP (Base Point — указатель базы). Так же, как и для регистра ESP, использование EBP автоматически предполагает работу с сегментом стека. Перед использованием этого регистра для доступа к данным стека его содержимое необходимо правильно инициализировать, что предполагает формирование в нем адреса, который бы указывал непосредственно на переданные данные. Для этого в начало процедуры рекомендуется включить дополнительный фрагмент кода. Он имеет свое название — **пролог процедуры**. Код пролога состоит всего из двух команд. Первая команда `push ebp` сохраняет содержимое `ebp` в стеке с тем, чтобы исключить порчу находящегося в нем значения в вызываемой процедуре. Вторая команда пролога `mov ebp, esp` настраивает `ebp` на вершину стека. После этого мы можем не волноваться о том, что содержимое `esp` перестанет быть актуальным, и осуществлять прямой доступ к содержимому стека.

Конец процедуры также должен содержать действия, обеспечивающие корректный возврат из процедуры. Фрагмент кода, выполняющего такие действия, имеет свое название — **эпилог процедуры**. Код эпилога должен восстановить контекст программы в точке вызова процедуры из вызывающей программы. При этом, в частности, нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, передававшиеся в процедуру. Это можно сделать несколькими способами:

- используя последовательность из n команд `pop xh`. Лучше всего это делать в вызывающей программе сразу после возврата управления из процедуры;
- откорректировать регистр указателя стека `esp` на величину $4 * n$, например, командой `add esp, NN`, где $NN = 4 * n$ (n — количество аргументов). Это также лучше делать после возврата управления вызывающей процедуре;
- используя машинную команду `ret n` в качестве последней исполняемой команды в процедуре, где n — количество байт, на которое нужно увеличить содержимое регистра `esp` после того, как со стека будут сняты составляющие адреса возврата. Этот способ аналогичен предыдущему, но выполняется автоматически микропроцессором.

Программа, содержащая вызов процедуры с передачей аргументов через стек:

```
.586
.model flat, stdcall
.stack 4096
.data
.code
proc_1 proc          ; начало процедуры
push ebp           ; пролог: сохранение ЕВР
mov ebp, esp       ; пролог: инициализация ЕВР
mov eax, [ebp+8]   ; доступ к аргументу 4
mov ebx, [ebp+12]  ; доступ к аргументу 3
mov ecx, [ebp+16]  ; доступ к аргументу 2
pop ebp            ; эпилог: восстановление ЕВР
ret 12
proc_1 endp
main proc
push 2
push 3
push 4
call proc_1
ret
main endp
end main
```

Для доступа к аргументу «4» достаточно сместиться от содержимого `ebp` на 8 (4 байта хранят адрес возврата в вызывающую процедуру, и еще 4 байта хранят значение регистра `ebp`, помещенное в стек данной процедурой), для аргумента «3» — на 12 и т. д.

Пролог и эпилог процедуры можно также заменить командами поддержки языков высокого уровня:

- Команда `enter` подготавливает стек для обращения к аргументов, имеет 2 операнда :
 - 1 – определяет количество байт в стеке, используемых для хранения локальных идентификаторов процедуры;
 - 2 – определяет уровень вложенности процедуры.
- Команда `leave` подготавливает стек к возврату из процедуры, не имеет операндов.

```
proc_1 proc
enter 0,0
mov eax, [ebp+8]
mov ebx, [ebp+12]
mov ecx, [ebp+16]
leave
ret 12
proc_1 endp
```

В процедуру могут передаваться либо данные, либо их адреса (указатели на данные). В языке высокого уровня это называется передачей по значению и адресу, соответственно.

Наиболее простой способ передачи аргументов в процедуру — передача по значению. Этот способ предполагает, что передаются сами данные, то есть их значения. Вызываемая программа получает значение аргумента через регистр или через стек. При передаче переменных через регистр или стек на их размерность накладываются ограничения, связанные с размерностью используемых регистров или стека. При передаче аргументов по значению в вызываемой процедуре обрабатываются их копии. Поэтому значения переменных в вызывающей процедуре не изменяются.

Способ передачи аргументов по адресу предполагает, что вызываемая процедура получает не сами данные, а их адреса. В процедуре нужно извлечь эти адреса тем же методом, как это делалось для данных, и загрузить их в соответствующие регистры. После этого, используя адреса в регистрах, следует выполнить необходимые операции над самими данными. В отличие от способа передачи данных по значению, при передаче данных по адресу в вызываемой процедуре обрабатывается не копия, а оригинал передаваемых данных. Поэтому при изменении данных в вызываемой процедуре они автоматически изменяются и в вызывающей программе, так как изменения касаются одной области памяти.

Передача аргументов с помощью директив `extern` и `public` используется в случаях, если

- оба модуля используют сегмент(секцию) данных вызывающей программы;
- у каждого модуля есть свой собственный сегмент(секция) данных;
- модули используют атрибут комбинирования (объединения) сегментов `public` в директиве сегментации `segment`.

Рассмотрим пример вывода на экран двух символов, описанных в вызывающей программе. Два модуля используют только сегмент данных вызывающей программы. В этом случае не требуется переопределения сегмента данных в вызываемой процедуре.

```

;Модуль 1
.586
.model flat, stdcall
.data
STR1 DB "Программа",0
STR2 db "Меня зовут Лена ",0
extern p1@0:near
public STR1, STR2
.code
start proc
call p1@0
ret
start endp
end start

;Модуль 2
.586
.model flat, stdcall
public p1
extern STR1:BYTE, STR2:BYTE
EXTERN MessageBoxA@16:NEAR
.code
p1 proc
    PUSH    0
    PUSH    OFFSET STR1
    PUSH    OFFSET STR2
    PUSH    0
    CALL    MessageBoxA@16
ret
p1 endp
end

```

Возврат результата из процедуры

В общем случае программист располагает тремя вариантами возврата значений из процедуры:

- С использованием регистров. Ограничения здесь те же, что и при передаче данных, — это небольшое количество доступных регистров и их фиксированный размер. Данный способ является наиболее быстрым, поэтому его есть смысл использовать для организации критичных по времени вызова процедур.
- С использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования для устранения неоднозначностей.
- С использованием стека. Здесь, подобно передаче аргументов через стек, также нужно использовать регистр `ebp`. При этом возможны следующие варианты:
 - использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру. То есть предполагается замещение ставших ненужными входных аргументов выходными данными;
 - предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения. При использовании этого варианта процедура, конечно же, не должна пытаться очистить стек командой `ret`. Эту операцию придется делать в вызывающей программе, например командой `pop`.

СВЯЗЬ АССЕМБЛЕРА С ЯЗЫКАМИ ВЫСОКОГО УРОВНЯ

Существуют следующие формы комбинирования программ на языках высокого уровня с ассемблером:

- Использование ассемблерных вставок (встроенный ассемблер, режим `inline`). Ассемблерные коды в виде команд ассемблера вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как команды ассемблера и без изменений включает в формируемый им объектный код. Эта форма удобна, если надо вставить небольшой фрагмент.
- Использование внешних процедур и функций. Это более универсальная форма комбинирования. У нее есть ряд преимуществ:
 - написание и отладку программ можно производить независимо;
 - написанные подпрограммы можно использовать в других проектах;
 - облегчаются модификация и сопровождение подпрограмм.

Встроенный ассемблер

При написании ассемблерных вставок используется следующий синтаксис:

1 способ:

```
_asm код_операции операнды ; // комментарии
```

`код_операции` задает команду ассемблера, `операнды` – это операнды команды. В конце записывается «;», как и в любой команде языка Си. Комментарии записываются в той форме, которая принята для языка Си.

2 способ:

```
_asm
{
    текст программы на ассемблере ; комментарии
}
```

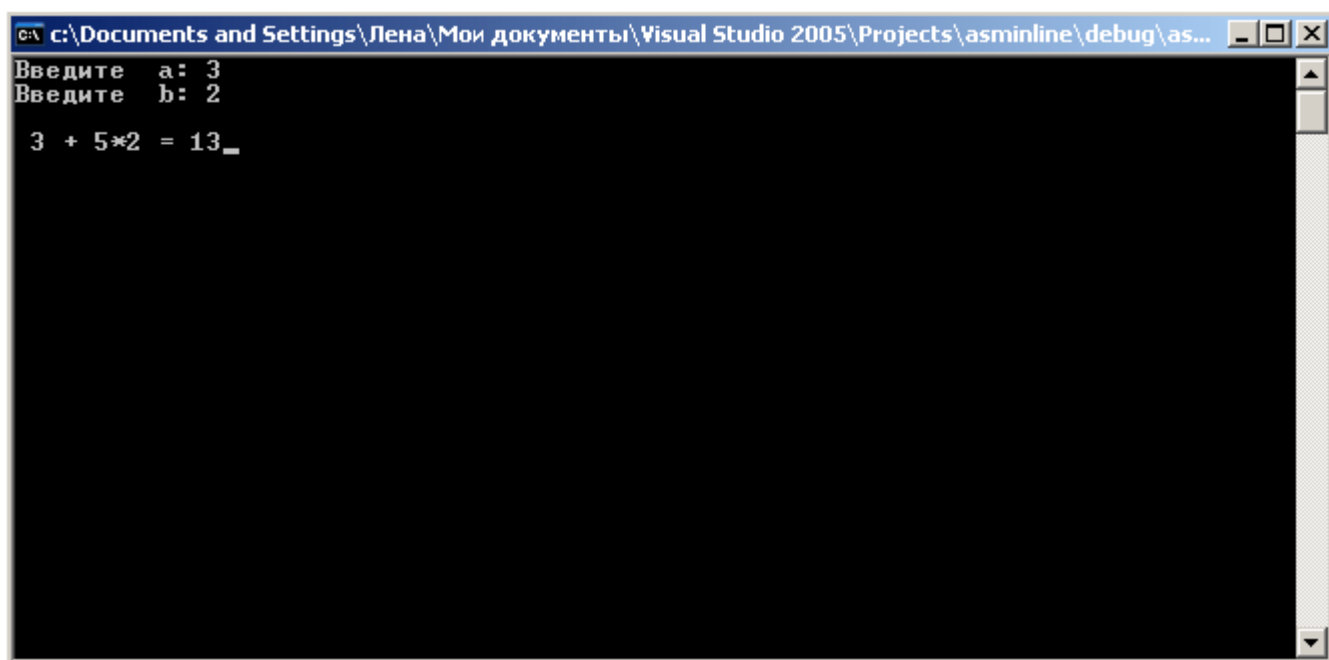
Текст программы пишется с использованием синтаксиса ассемблера, при необходимости можно использовать метки и идентификаторы. Комментарии в этом случае можно записывать как после «;», так и после «//».

В качестве примера рассмотрим программу, которая запрашивает ввод чисел `a` и `b` и вычисляет выражение `a+5b`. Для вывода приглашений `Введите a:` и `Введите b:` используем функцию `CharToOem(_T("Введите "), s)`, где `s` –

указатель на перекодированную строку, которая перекодирует русскоязычные сообщения.

```
#include <windows.h>
#include <tchar.h>
void main()
{
char s[20];
int a,b, sum;
CharToOem(_T("Введите "),s);
printf("%s a: ", s);
scanf("%d",&a);
printf("%s b: ",s);
scanf("%d",&b);
_asm mov eax, a;
_asm
{
mov ecx, 5
m:   add eax, b
loop m
mov sum, eax
}
printf("\n %d + 5*%d = %d",a,b,sum);
getch();
}
```

Для компиляции программы создаем проект File → New → Project типа Win32 Console Application. Далее в мастере указываем создание пустого проекта (Empty Project) и добавляем файл программы с расширением .c . В результате выполнения программы появится консольное окно:



```
c:\Documents and Settings\Лена\Мои документы\Visual Studio 2005\Projects\asminline\debug\as...
Введите a: 3
Введите b: 2
3 + 5*2 = 13_
```

Использование внешних процедур

Для связи посредством внешних процедур возможны два варианта вызова:

- программа на языке высокого уровня вызывает процедуру на языке ассемблера;
- программа на языке ассемблера вызывает процедуру на языке высокого уровня.

Чаще используется первый способ.

Транслятор MASM генерирует подчеркивание перед именем процедуры автоматически (в отличие от TASM), если в начале программы устанавливается тип вызова `stdcall` (Standard Call, т.е. стандартный вызов).

В таблице ниже представлены основные соглашения по передаче параметров в процедуру. Во всех предыдущих ассемблерных программах указывался тип передачи параметров как `stdcall`. Однако, по сути, это никак и нигде не использовалось – так передача и извлечение параметров делалась нами явно, без помощи транслятора. Когда мы имеем дело с языками высокого уровня, это необходимо учитывать и знать, как работают те или иные соглашения.

Соглашение	Параметры	Очистка стека	Регистры
Pascal (конвенция языка Паскаль)	Слева направо	Процедура	Нет
c (конвенция C)	Справа налево	Вызывающая программа	Нет
Fastcall (быстрый или регистровый вызов)	Слева направо	Процедура	Задействованы три регистра (EAX, EDX, ECX), далее стек
Stdcall (стандартный вызов)	Справа налево	Процедура	Нет

Конвенция Pascal заключается в том, что параметры из программы на языке высокого уровня передаются в стеке и возвращаются в регистре AX/EAX, — это способ, принятый в языке PASCAL (а также в BASIC, FORTRAN, ADA, OBERON, MODULA2), — просто поместить параметры в стек в естественном порядке. В этом случае запись

```
some_proc(a, b, c, d, e)
```

превращается в

```
push    a
push    b
push    c
```

```

push    d
push    e
call    some_proc

```

Процедура `some_proc`, во-первых, должна очистить стек по окончании работы (например, завершившись командой `ret 20`) и, во-вторых, параметры, переданные ей, находятся в стеке в обратном порядке:

```

some_proc proc
push     ebp
mov      ebp,esp           ; создать стековый кадр
a       equ     [ebp+24]
b       equ     [ebp+20]
c       equ     [ebp+16]
d       equ     [ebp+12]
e       equ     [ebp+8]
...
pop      ebp
ret      20
some_proc endp

```

Этот код в точности соответствует усложненной форме директивы `proc`, которую поддерживают все современные ассемблеры:

```

some_proc proc PASCAL, a:dword, b:dword, c:dword, d:dword, e:dword
...
ret
some_proc endp

```

Главный недостаток этого подхода — сложность создания функции с изменяемым числом параметров, аналогичных функции языка C `printf`. Чтобы определить число параметров, переданных `printf`, процедура должна сначала прочитать первый параметр, но она не знает его расположения в стеке. Эту проблему решает подход, используемый в C, где параметры передаются в обратном порядке.

Конвенция C используется, в первую очередь, в языках C и C++, а также в PROLOG и других. Параметры помещаются в стек в обратном порядке, и, в противоположность PASCAL-конвенции, удаление параметров из стека выполняет вызывающая процедура. Запись

```
some_proc(a, b, c, d, e)
```

превращается в

```

push    e
push    d
push    c
push    b
push    a

```

```

call    some_proc
add     esp,20    ; освободить стек

```

Вызванная таким образом процедура может инициализироваться так:

```

some_proc proc
    push    ebp
    mov     ebp,esp ; создать стековый кадр
a equ [ebp+8]
b equ [ebp+12]
c equ [ebp+16]
d equ [ebp+20]
e equ [ebp+24]
    ...
    pop    ebp
    ret   20
some_proc endp

```

Ассемблеры поддерживают и такой формат вызова при помощи усложненной формы директивы `proc` с указанием языка C:

```

some_proc proc C, a:dword, b:dword, c:dword, d:dword, e:dword
    ...
    ret
some_proc endp

```

Регистр `EBP` используется для хранения параметров, и его ни в коем случае нельзя изменять.

Преимущество по сравнению с `PASCAL`-конвенцией заключается в том, что освобождение стека от параметров в `C` возлагается на вызывающую процедуру, что позволяет лучше оптимизировать код программы. Например, если мы должны вызвать несколько функций, принимающих одни и те же параметры подряд, можно не заполнять стек каждый раз заново, и это — одна из причин, почему компиляторы с языка `C` создают более компактный и быстрый код по сравнению с компиляторами с других языков.

Смешанные конвенции

Существует конвенция передачи параметров `STDCALL`, отличающаяся и от `C`, и от `PASCAL`-конвенций, которая применяется для всех системных функций `Win32 API`. Здесь параметры помещаются в стек в обратном порядке, как в `C`, но процедуры должны очищать стек сами, как в `PASCAL`.

Еще одно отклонение от `C`-конвенции — это быстрое или регистровое соглашение. В этом случае параметры в функции также передаются по возможности через регистры. Например, при вызове функции с шестью параметрами


```
some_proc(a,b,c,d,e,f);
```

первые три параметра передаются соответственно в EAX, EDX, ECX, а только начиная с четвертого, параметры помещают в стек в обычном обратном порядке:

```
d    equ    [bp+8]
e    equ    [bp+12]
f    equ    [bp+16]
```

В этом случае если стек был задействован, освобождение его возлагается на вызываемую процедуру. Есть еще один нюанс. В случае быстрого вызова транслятор Си добавляет к имени значок @ спереди, что искажает имена при обращении к ним в ассемблерном модуле.

Чтобы вернуть результат в программу на С из процедуры на ассемблере, перед возвратом управления в программу на С в программе на ассемблере необходимо поместить результат в соответствующий регистр.

Тип возвращаемого значения	Регистр
unsigned char	al
char	al
unsigned int	eax
int	eax
unsigned long int	edx:eax
long int	edx:eax

Как видим из таблицы, возвращаемое значение функции по умолчанию находится в регистре AL/ AX/ EAX/ EDX:EAX.

Рассмотрим пример: Умножить на 2 первый элемент массива.

```
//Вызывающая программа
#include <windows.h>
#include <tchar.h>
void main()
{
extern int MAS_FUNC (int *,int);
int *mas, i, n, k;
char s[30];
CharToOem(_T("Введите размер массива: "),s);
printf(s);
scanf("%d",&n);
mas = (int*) calloc(n,sizeof(int));
CharToOem(_T("Введите элементы массива: "),s);
printf(s);
for(i=0; i<n; i++)
{
printf("mas[%d]= ",i);
scanf("%d",mas+i);
}
k = MAS_FUNC(mas, n);
```

```

printf("mas[1]*2 = %d",k);
getch();
}

;Вызываемая функция
.586
.MODEL FLAT, C
.CODE
MAS_FUNC PROC C mas:dword, n:dword
    mov esi,mas
    mov eax, [esi+4]
    shl eax, 1
    ret
MAS_FUNC ENDP
END

```

Результатом работы программы будет окно консоли.

```

c:\Documents and Settings\Лена\Мои документы\Visual Studio 2005\Projects\mkj\debug\mkj.exe
Введите размер массива: 5
Введите элементы массива: mas[0]= 2
mas[1]= -3
mas[2]= 4
mas[3]= 5
mas[4]= 3
mas[1]*2 = -6_

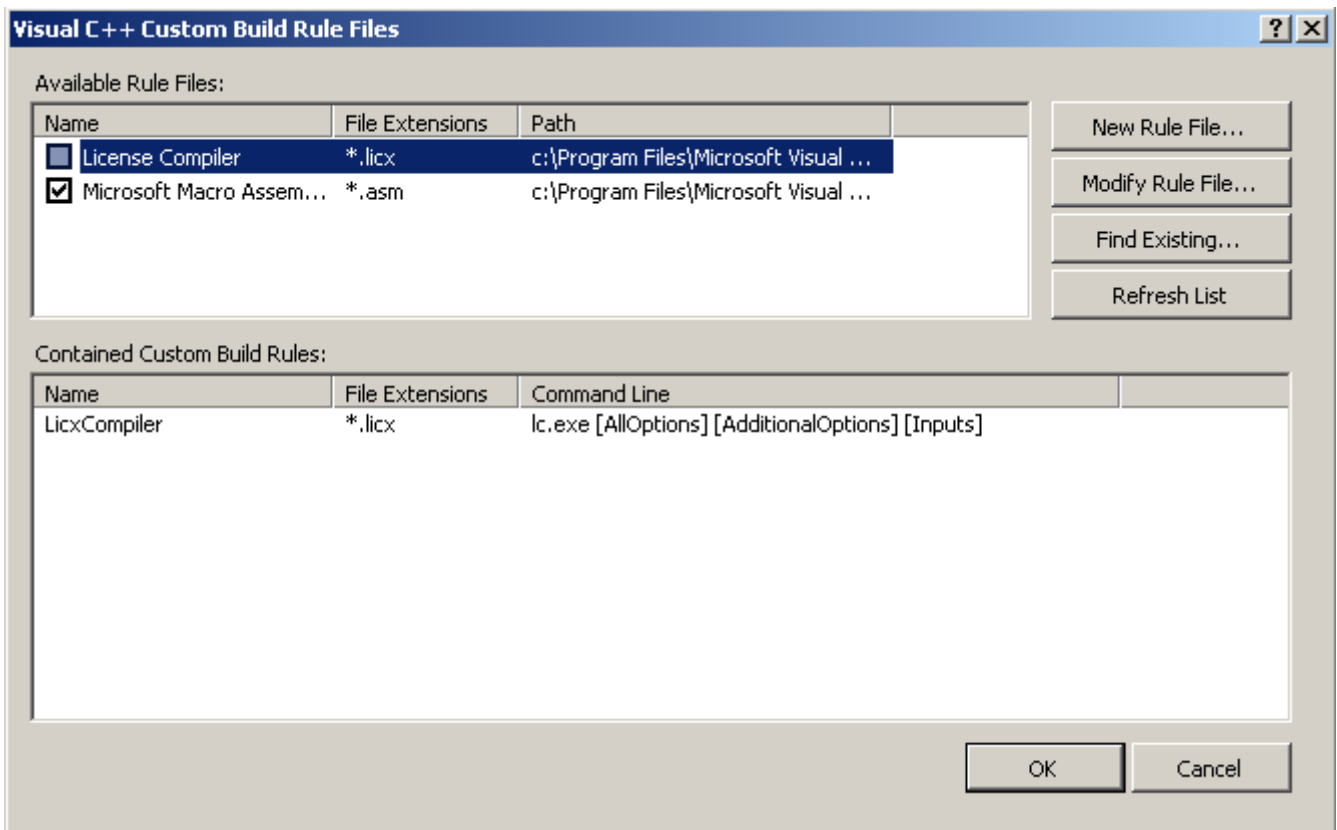
```

Перед вызовом процедуры всегда нужно сохранять содержимое регистров `ebp`, `esp`, а перед выходом из процедуры – восстанавливать содержимое этих регистров. Это делается компилятором языка Си. Остальные регистры нужно сохранять при необходимости (если содержимое регистра подвергается изменению в вызванной процедуре, а далее может использоваться в вызывающей программе) Это может быть сделано с помощью команды `pusha`.

Передача аргументов в процедуру на ассемблере из программы на Си осуществляется через стек. При этом вызывающая программа записывает передаваемые параметры в стек, а вызываемая программа извлекает их из стека.

Для компиляции проекта, состоящего из двух модулей, написанных на разных языках, в Visual Studio необходимо:

- 1) Создать проект File → New Project типа Windows Console Application с дополнительной опцией Empty Project.
- 2) Добавить файлы Source Files → Add Item в дереве проекта. Файл вызывающей программы (на языке Си) должен иметь расширение .c , а не .cpp. У файла с функцией на ассемблере также явно указывается расширение .asm.
- 3) Для установки «инструмента» трансляции файла на ассемблере выбираем в дереве проекта по правой кнопке Custom Build Rules... и указываем Microsoft Macro Assembler галочкой.



- 4) Скомпилировать проект Build → Build имя проекта.
- 5) Запустить программу Debug → Start Debugging или F5.

ОБРАБОТКА ПРЕРЫВАНИЙ

Прерывание — это временное прекращение основного процесса вычислений для выполнения некоторых запланированных или незапланированных действий, вызванных работой аппаратуры или программы.

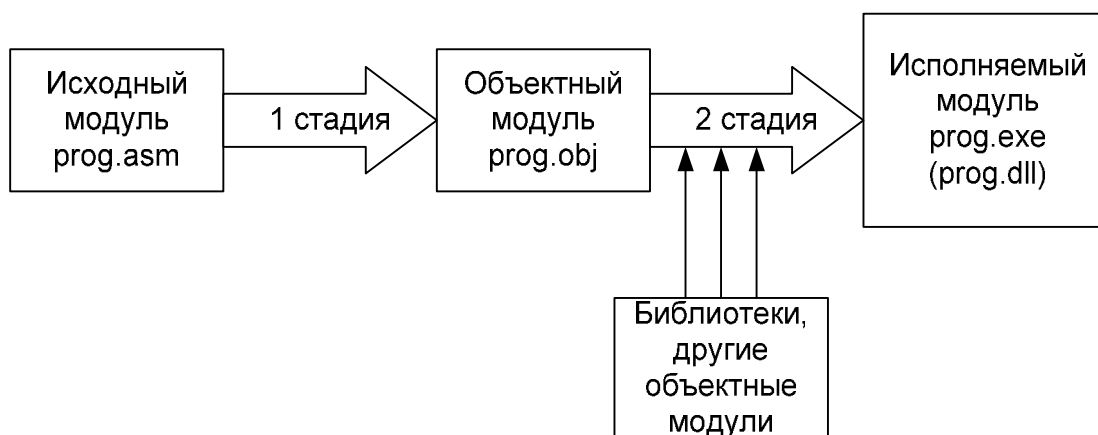
Механизм прерываний поддерживается на аппаратном уровне и позволяет реализовать как эффективное взаимодействие программ с операционной системой, так и эффективное управление программой со стороны аппаратной части компьютера.

В зависимости от источника прерывания классифицируют так:

- *аппаратные* — возникают как реакция микропроцессора на физический сигнал от некоторого устройства компьютера (клавиатура, системный таймер, жесткий диск и т.д.); по времени возникновения эти прерывания *асинхронны*, то есть возникают в случайные моменты времени;
- *программные* — вызываются искусственно с помощью соответствующей команды из программы (команда `int`); предназначены для выполнения некоторых действий операционной системы; являются *синхронными*;
- *исключения* — программные, являющиеся реакцией микропроцессора на нештатную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды программы.

СОЗДАНИЕ ИСПОЛНЯЕМОГО ФАЙЛА

Трансляция модуля на ассемблере проходит две стадии.



Двум стадиям трансляции соответствуют две основные программы MASM: ассемблер ML.EXE и редактор связей LINK.EXE.

Исполняемым форматом в Windows является формат PE. Сокращение PE означает Portable Executable, т.е. переносимый исполняемый формат. Этот формат имеют как EXE-файлы, так и динамические библиотеки (.DLL). Фирма Microsoft ввела «новый» формат и для объектных модулей - это COFF-формат (COFF - Common Object File Format).

1 стадия

Программа ассемблера ML.EXE имеет синтаксис:

```
ML [параметры] имя_файла.asm [/link опции_редактора_связей]
```

Атрибут [параметры] может принимать следующие значения:

Параметр	Комментарий
/?, /help	Вывод помощи.
/AT	Создать файл в формате .com. Для программирования в Windows этот ключ, естественно, бесполезен.
/Bl<linker>	Использовать альтернативный компоновщик. Предполагается автоматический запуск компоновщика.
/c	Компиляция без компоновки.
/Cp	Сохранение регистров пользовательских идентификаторов. Может использоваться для дополнительного контроля.
/Cu	Приведение всех пользовательских идентификаторов к верхнему регистру.
/Cx	Сохранение регистров пользовательских идентификаторов, объявленных PUBLIC и EXTERNAL.
/coff	Создание объектных файлов в стандарте coff. Применение обязательно.
/D<name>=[строка]	Задание текстового макроса. Очень удобен для отладки с использованием условной компиляции.
/EP	Листинг: текст программы с включаемыми файлами.
/F <hex>	Размер стека в байтах. Размер стека по умолчанию равен 1 Мб.
/Fe<file>	Имя исполняемого файла. Имеет смысл без параметра /c.

Параметр	Комментарий
/Fl<file>	Создать файл листинга.
/Fm<file>	Создать map-файл. Имеет смысл без опции /c.
/Fo<file>	Задать имя объектного файла.
/Fpi	Включение кода эмулятора сопроцессора. Начиная с 486-ого микропроцессора, данный параметр потерял актуальность.
/Fr<file>	Включить ограниченную информацию браузера.
/FR<file>	Включить полную информацию браузера.
/G<c d z>	Использовать соглашение вызова Паскаль, Си, stdcall.
/H<number>	Установить максимальную длину внешних имен.
/I<name>	Добавить путь для inc-файлов. Допускается до 10 опций /I.
/link <opt>	Опции командной строки компоновщика. Имеет смысл без опции /c.
/nologo	Не показывать заголовочный текст компилятора.
/Sa	Листинг максимального формата.
/Sc	Включить в листинг синхронизацию.
/Sf	Листинг первого прохода.
/Sl<number>	Длина строки листинга.
/Sn	Не включать в листинг таблицу символов.
/Sp<number>	Высота страницы листинга.
/Ss<string>	Текст подзаголовка листинга.
/St<string>	Текст заголовка листинга.
/Sx	Включить в листинг фрагменты условной компиляции.
/Ta<file>	Для компилирования файлов, расширение которых не .asm.
/W<number>	Устанавливает перечень событий компиляции, трактуемые как предупреждения.
/WX	Трактовать предупреждения как ошибки.
/w	Тоже что /w0 /wX.
/X	Игнорировать путь, установленный переменной окружения INCLUDE.
/zd	Отладочная информация состоит только из номеров строк.
/zf	Объявить все имена PUBLIC.
/zi	Включить полную отладочную информацию.
/zm	Включить совместимость с MASM 5.01.
/Zp<n>	Установить выравнивание структур.
/Zs	Выполнять только проверку синтаксиса.

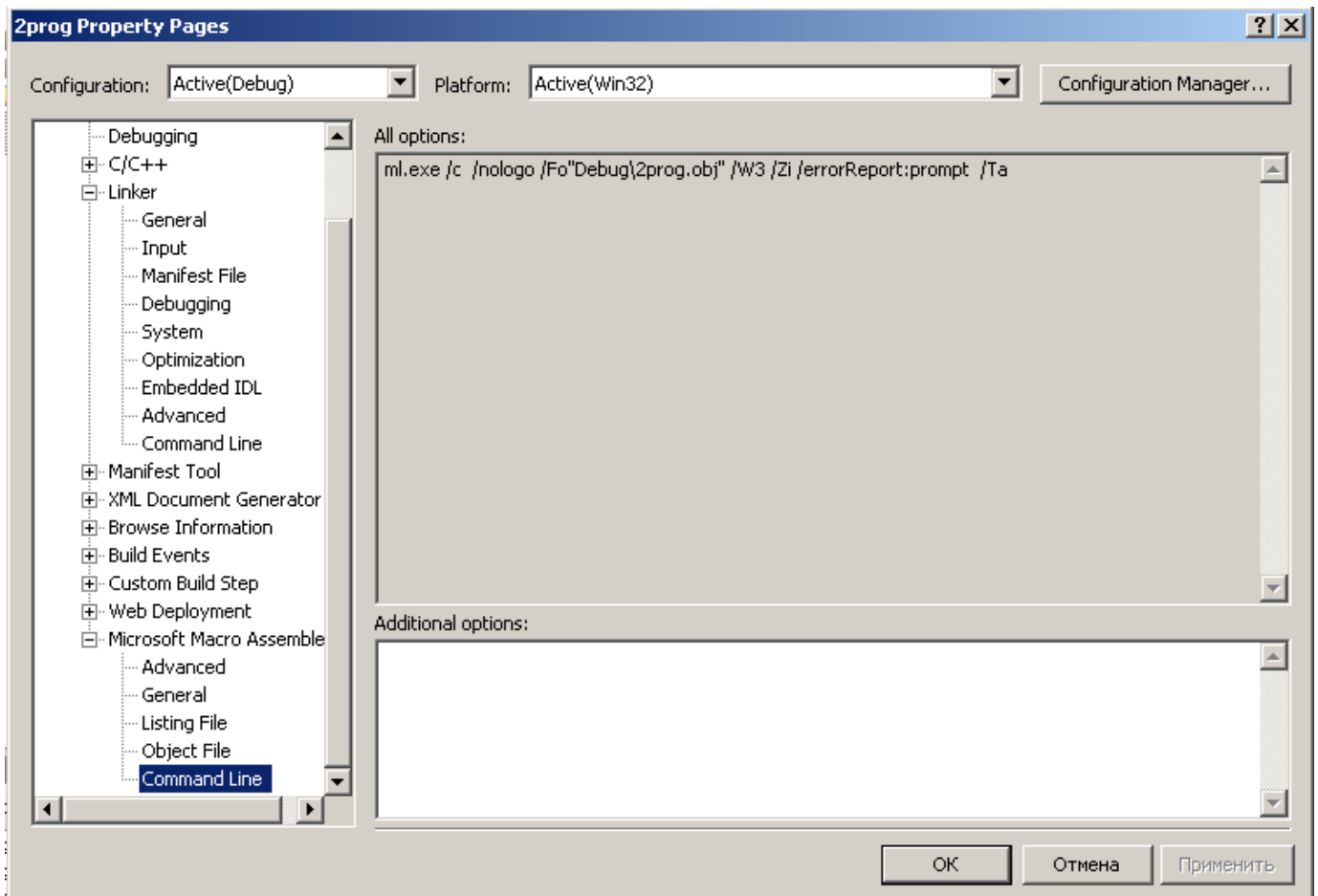
Параметр [/link опции_редактора_связей] предполагает автоматический запуск редактора связей (2 стадия трансляции) при успешном получении объектного модуля. Компиляция без компоновки осуществляется с использованием параметра /c.

В Visual Studio параметры командной строки ассемблера можно посмотреть в окне Properties проекта после выбора «инструмента» Microsoft Macro Assembler.

2 стадия

Программа LINK.EXE имеет синтаксис:

LINK [параметры] [файлы] [@файл_параметров]



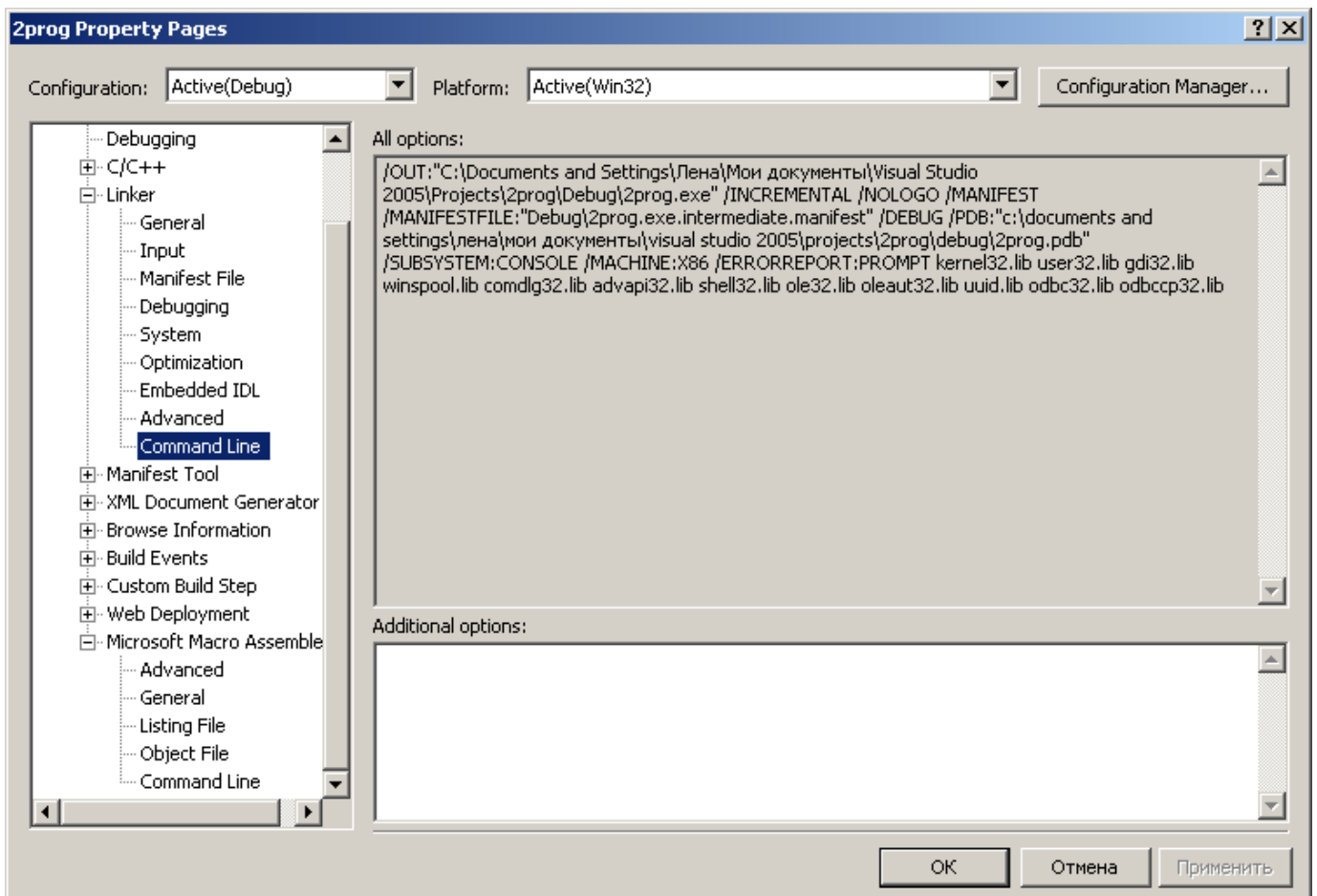
Атрибут [параметры] может принимать следующие значения:

Параметр	Комментарий
/ALIGN: number	Определяет выравнивание секций в линейной модели. По умолчанию 4096.
/BASE: { address @filename, key }	Определяет базовый адрес (адрес загрузки). По умолчанию для EXE-программы адрес 0x400000, для DLL — 0x10000000.
/COMMENT: ["] comment ["]	Определяет комментарий, помещаемый в заголовок EXE- и DLL-файлов.
/DEBUG	Создает отладочную информацию для EXE- и DLL-файлов. Отладочная информация помещается в pdb-файл.
/DEBUGTYPE: { CV COFF BOTH }	CV — отладочная информация в формате Microsoft, COFF — отладочная информация в формате COFF (Common Object File Format), BOTH — создаются оба вида отладочной информации.
/DEF: filename	Определяет DEF-файл (файл определений).
/DEFAULTLIB: library	Добавляет одну библиотеку к списку используемых библиотек.
/DLL	Создать DLL-файл.
/DRIVER [: { UPONLY WDM }]	Используется для создания NT-драйвера (Kernel Mode Driver).
/ENTRY: symbol	Определяет стартовый адрес для EXE- и DLL-файлов.
/EXETYPE: DYNAMIC	Данная опция используется при создании VxD-драйвера.
/EXPORT: entryname [=internalname] [, @ordinal [, NONAME]] [, DATA]	Данная опция позволяет экспортировать функцию из вашей программы так, чтобы она была доступна для других программ. При этом создается import-библиотека.
/FIXED [: NO]	Данная опция фиксирует базовый адрес, определенный в опции /BASE.

Параметр	Комментарий
/FORCE[: {MULTIPLE UNRESOLVED}]	Позволяет создавать исполняемый файл, даже если не найдено внешнее имя или имеется несколько разных определений.
/GPFSIZE: number	Определяет размер общих переменных для MIPS и Alpha платформ.
/HEAP: reserve[, commit]	Определяет размер кучи (HEAP) в байтах. По умолчанию этот размер равен одному мегабайту.
/IMPLIB: filename	Определяет имя import-библиотеки, если она создается.
/INCLUDE: symbol	Добавляет идентификатор к таблице имен.
/INCREMENTAL: {YES NO}	Если установлена опция /INCREMENTAL: YES, то в EXE добавляется дополнительная информация, позволяющая быстрее перекомпилировать этот файл. По умолчанию эта информация не добавляется.
/LARGEADDRESSAWARE[: NO]	Указывает, что приложение оперирует адресами, большими 2 Гб.
/LIBPATH: dir	Определяет библиотеку, которая в первую очередь разыскивается компоновщиком.
/MACHINE: {ALPHA ARM IX86 MIPS MIPS16 MIPSR41XX PPC SH3 SH4}	Определяет платформу. В большинстве случаев это делать не приходится.
/MAP[: filename]	Дает команду создания MAP-файла.
/MAPINFO: {EXPORTS FIXUPS LINES}	Указывает компоновщику включить соответствующую информацию в MAP-файл.
/MERGE: from=to	Объединить секцию "from" с секцией "to" и присвоить имя "to".
/NODEFAULTLIB[: library]	Игнорирует все или конкретную библиотеку.
/NOENTRY	Необходимо для создания DLL-файла.
/NOLOGO	Не выводить начальное сообщение компоновщика.
/OPT: {ICF[, iterations] NOICF NOREF NOWIN98 REF WIN98}	Определяет способ оптимизации, которую выполняет компоновщик.
/ORDER: @filename	Оптимизация программы путем вставки определенных инициализированных данных (COMDAT).
/OUT: filename	Определяет выходной файл.
/PDB: {filename NONE}	Определить имя файла, содержащего информацию для отладки.
/PDBTYPE: {CON[SOLIDATE] SEPT[YPES]}	Определяет тип PDB-файла.
/PROFILE	Используется для работы с профайлером (анализатором работы программы).
/RELEASE	Помещает контрольную сумму в выходной файл.
/SECTION: name, [E][R][W][S][D][K][L][P][X]	Данная опция позволяет изменить атрибут секции.
/STACK: reserve[, commit]	Определяет размер выделяемого стека. Commit — определяет размер памяти, интерпретируемый операционной системой.
/STUB: filename	Определяет STUB-файл, запускающийся в системе MS DOS.
/SUBSYSTEM: {NATIVE WINDOWS CONSOLE WINDOWSCE POSIX} [, # [. ##]]	Определяет, как запускать EXE-файл. CONSOLE — консольное приложение, WINDOWS — обычные WINDOWS-приложения, NATIVE — приложение для Windows NT, POSIX — создает приложение в POSIX-подсистеме

Параметр	Комментарий
	WINDOWS NT.
/SWAPRUN: { CD NET }	Сообщает операционной системе скопировать выходной файл в swap-файл (WINDOWS NT).
/VERBOSE[:LIB]	Заставляет выводить информацию о процессе компоновки.
/VERSION: # [. #]	Помещает информацию о версии в EXE-заголовок.
/VXD	Создать VXD-драйвер.
/WARN[:warninglevel]	Определяет количество возможных предупреждений, выдаваемых компоновщиком.
/WS: AGGRESSIVE	Несколько уменьшает скорость выполнения приложения (Windows NT). Операционная система удаляет данное приложение из памяти в случае его простоя.

В Visual Studio параметры командной строки редактора связей можно посмотреть в окне Properties проекта.



Пусть файл с текстом программы на языке ассемблера называется PROG.ASM, тогда две стадии трансляции будут выглядеть следующим образом:

`c:\masm32\bin\ml /c /coff PROG.ASM` - в результате появляется модуль PROG.OBJ,

`c:\masm32\bin\link /SUBSYSTEM:WINDOWS PROG.OBJ` - в результате появляется исполняемый модуль PROG.EXE.

Отладка программы

Отладка программы осуществляется с использованием специализированной программной среды, называемой отладчиком. Отладчик позволяет решить следующие задачи:

- определить место логической ошибки;
- определить причину логической ошибки.

С помощью отладчика также возможно:

- выполнение трассировки программы в прямом направлении, то есть последовательное исполнение программы, при котором за один шаг выполняется одна машинная инструкция;
- просмотр и изменение состояния аппаратных ресурсов микропроцессора во время покомандного выполнения программы.

Для организации процесса получения исполняемого модуля для отладки необходимо выполнить следующие действия.

- В исходной программе должна быть обязательно определена метка для первой команды, с которой начнется выполнение программы. Такая метка может быть собственно меткой или именем процедуры. Имя этой метки обязательно нужно указать в конце программы в качестве операнда директивы END:

```
END        имя_метки
```

- Исходный модуль должен быть оттранслирован с опцией /zi:

```
c:\masm32\bin\ml /c /coff /zi PROG.ASM
```

Применение опции /zi разрешает транслятору сохранить связь символических имен в программе и их смещений в памяти, что позволит отладчику производить отладку, используя оригинальные имена идентификаторов.

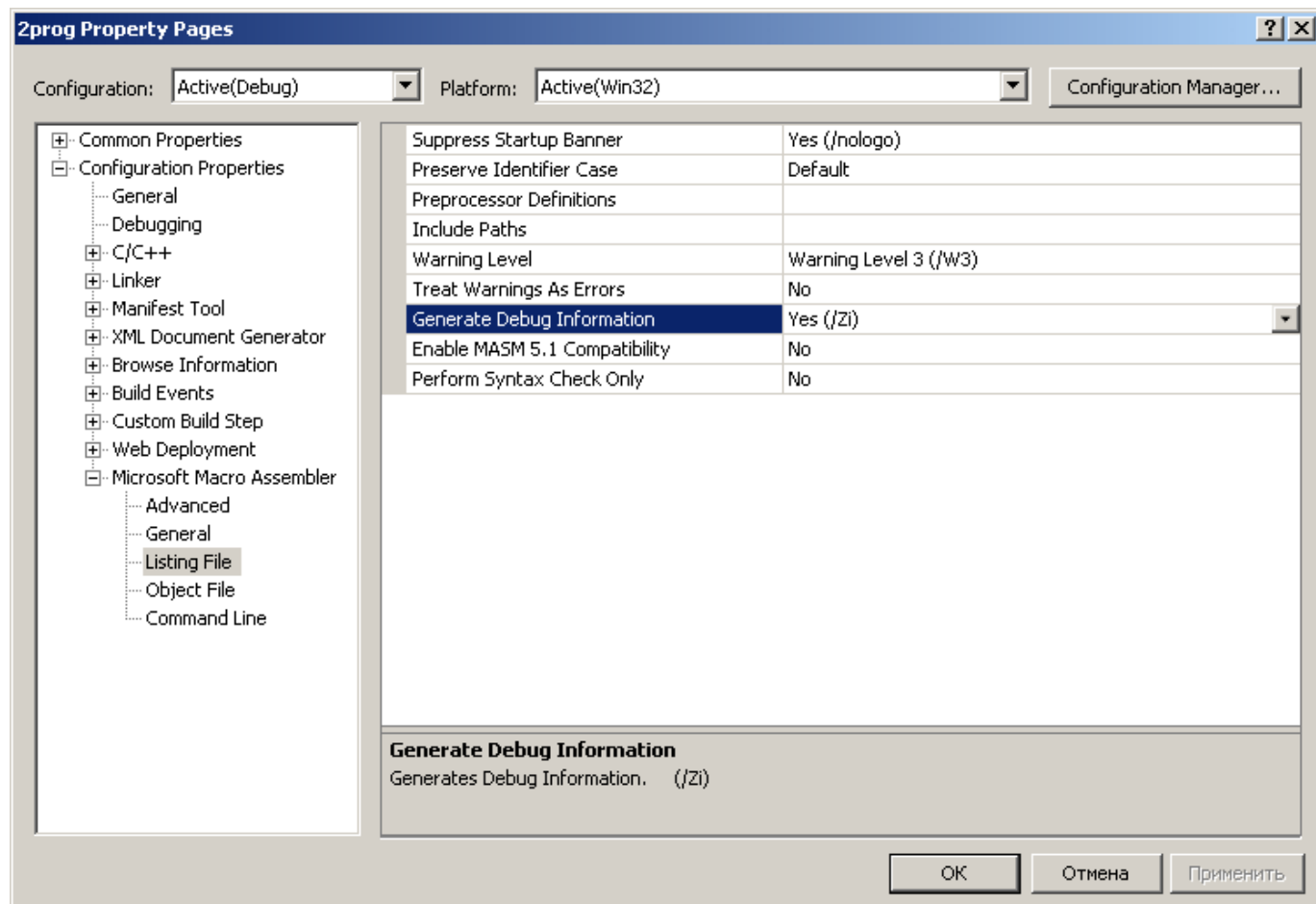
- Компоновка модуля должна быть произведена с опцией /DEBUG:

```
c:\masm32\bin\link /SUBSYSTEM:WINDOWS /DEBUG PROG.OBJ
```

Опция /DEBUG указывает на необходимость сохранения отладочной информации в исполняемом файле.

Для отладки программы как в комбинированном проекте с использованием нескольких языков программирования, так и в проекте программы на ассемблере, можно использовать встроенную среду отладки Microsoft Visual Studio.

Для того, чтобы отладка программы была возможна необходимо включить отладочную информацию в исполнимый модуль.



Запустить программу на выполнение в отладчике можно в одном из четырех режимов:

- режим безусловного выполнения;
- выполнение по шагам;
- выполнение до текущего положения курсора;
- выполнение с установкой точек прерывания.

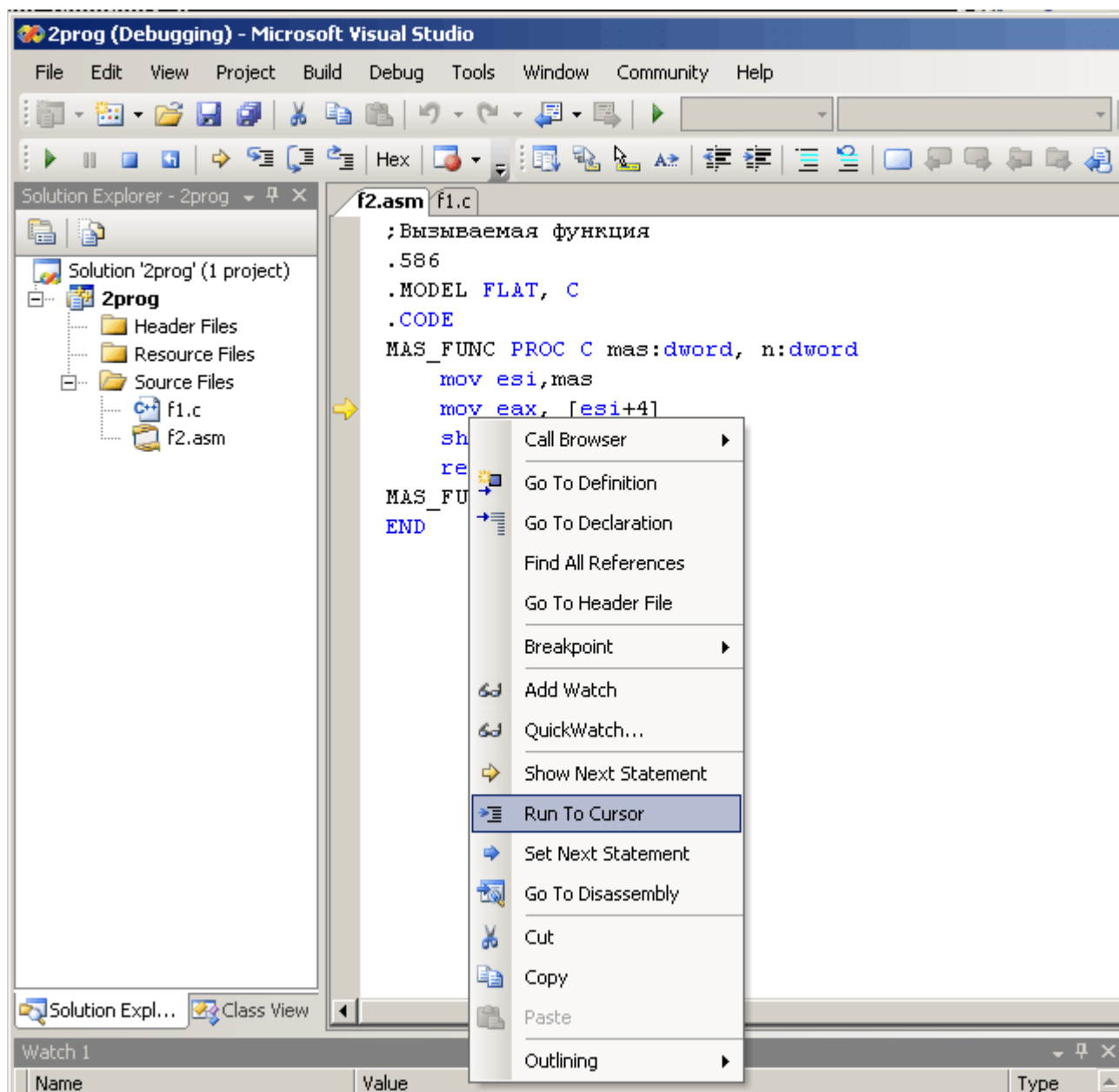
Режим безусловного выполнения целесообразно применять, когда требуется посмотреть на общее поведение программы. Для запуска программы в этом режиме необходимо нажать клавишу F5 или выбрать меню **Debug** → **Start Debugging**. В точках, где необходимо ввести данные, отладчик, в соответствии с логикой работы

применяемого средства ввода, будет осуществлять определенные действия. Аналогичные действия отладчик выполняет для вывода данных.

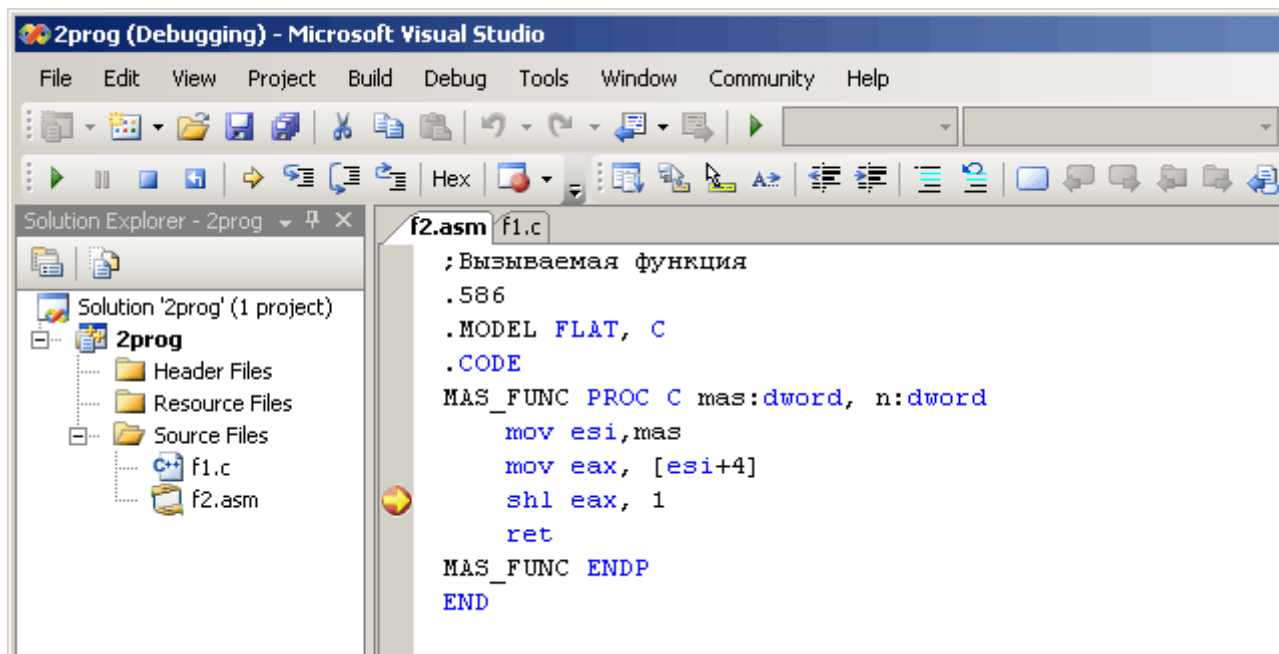
В случае, если нужно более детально изучить работу программы, применяются три следующих режима отладки.

Выполнение по шагам применяется для детального изучения работы программы. В этом режиме можно выполнить программу по командам. При этом можно наблюдать результат исполнения каждой команды. Для активизации этого режима нужно нажать клавишу F11 (меню Drbug → Step Into) или F10 (меню Drbug → Step Over). Обе эти клавиши активизируют пошаговый режим; отличие их проявляется в том случае, когда в потоке команд встречаются команды перехода в процедуру call или на прерывание int. При использовании Step Into отладчик осуществит переход на процедуру или прерывание и выполнит их по шагам. Если же используется Step Over, то вызов процедуры или прерывания отрабатывается как одна обычная команда, и управление передается следующей команде программы.

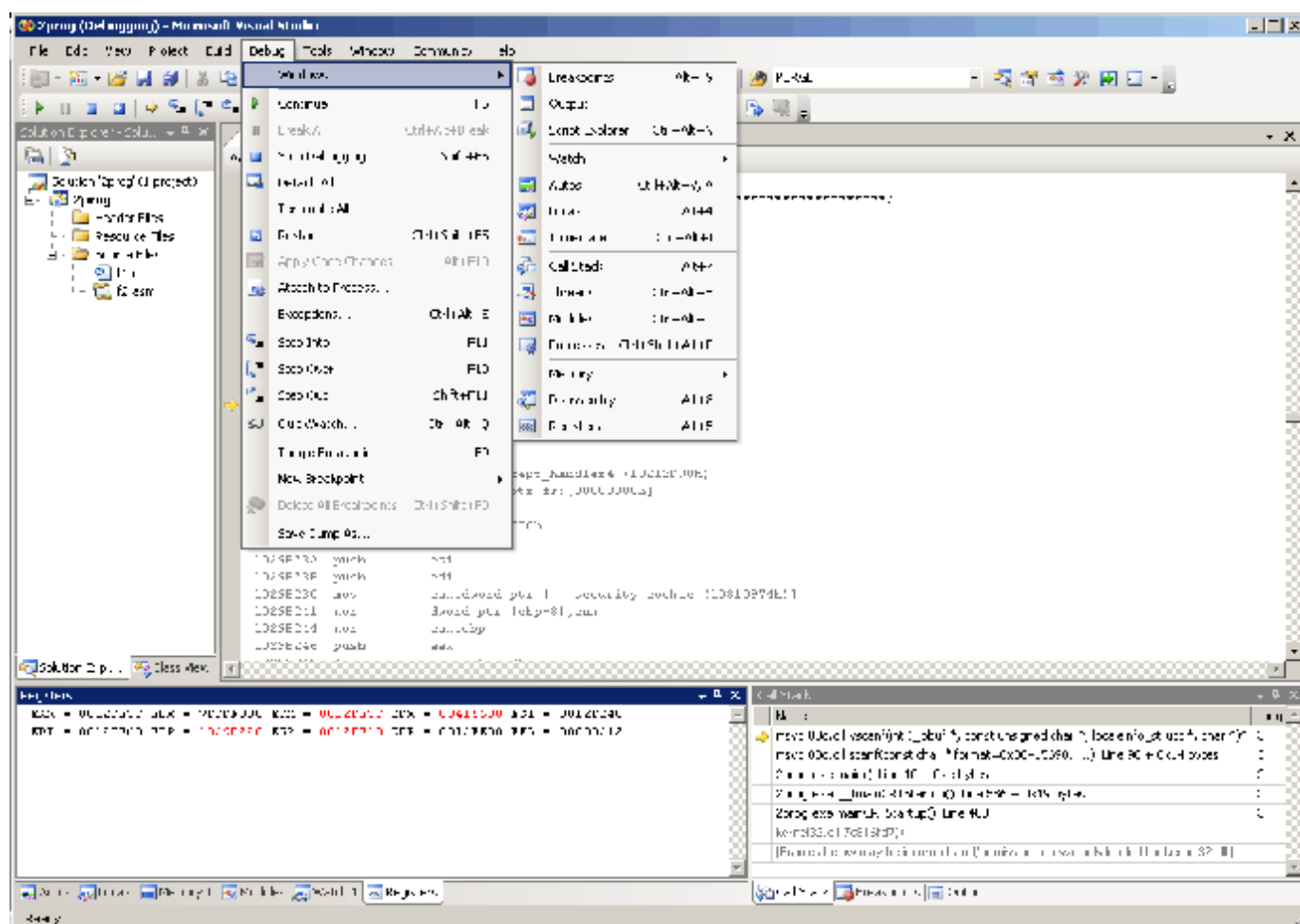
Выполнение до текущего положения курсора позволяет выполнить программу по шагам, начиная с произвольного места программы. Этот режим целесообразно использовать в том случае, если интересует только правильность функционирования некоторого участка программы. Для активизации этого режима необходимо установить курсор на нужную строку программы и во всплывающем меню выбрать Run To Cursor. Программа начнет выполнение и остановится на отмеченной команде, не выполнив ее.



Выполнение с установкой точек прерывания позволяет выполнить программу с остановкой ее в строго определенных точках прерывания (breakpoints). Перед выполнением программы необходимо установить эти точки в программе, для чего следует перейти в нужную строку и во всплывающем меню выбрать **Breakpoint** → **Insert Breakpoint**. В выбранной строке слева появится красный кружок. Установленные ранее точки прерывания можно убрать — для этого нужно выбрать в нужной строке во всплывающем меню **Breakpoint** → **Delete Breakpoint** или **Breakpoint** → **Disable Breakpoint** (неактивная точка останова). После установки точек прерывания программа запускается на безусловное выполнение клавишей F5. На первой точке прерывания программа остановится. Теперь можно посмотреть состояние микропроцессора и памяти, а затем продолжить выполнение программы.



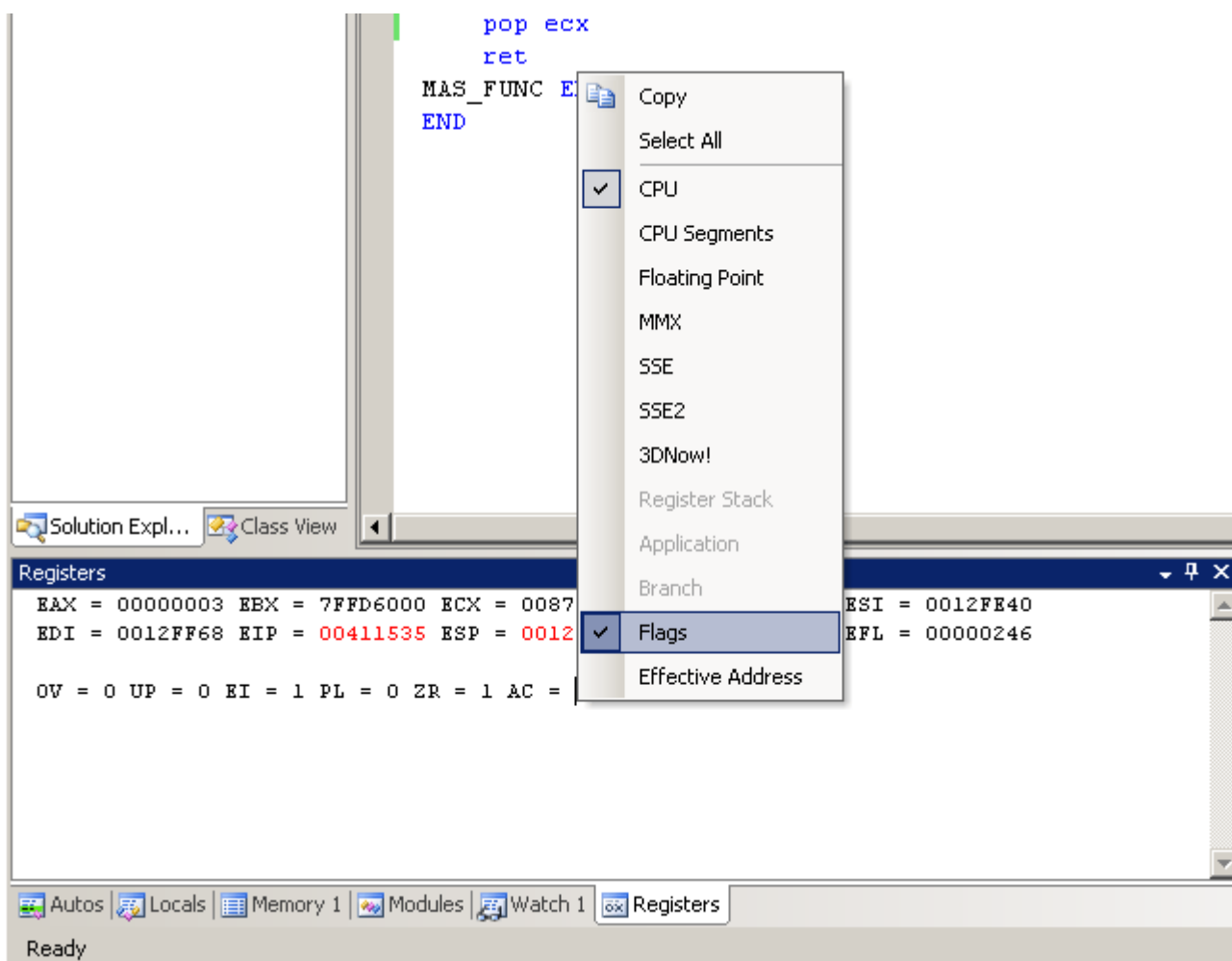
Использование одной из перечисленных возможностей отладчика позволяет просматривать значения регистров и идентификаторов, а также ассемблерный код исполняемой программы.



- Для просмотра ассемблерного кода исполняемой программы выбрать меню **Debug** → **Windows** → **Disassembly**;

- для просмотра содержимого регистра выбрать Debug → Windows → Registers (содержимое регистров содержится в левом нижнем окне, при изменении значения регистра его цвет становится красным);
- для просмотра значения идентификатора используется меню Debug → Windows → Watch, где вводится имя идентификатора;
- для просмотра области памяти используется меню Debug → Windows → Memory, где вводится адрес ячейки памяти.

Для того, чтобы вывести в отладчике информацию о содержимом регистра признаков EFLAGS необходимо во всплывающем меню окна Registers выбрать соответствующую опцию Flags.



МАТЕМАТИЧЕСКИЙ СОПРОЦЕССОР

Важной частью архитектуры микропроцессоров Intel является наличие устройства для обработки числовых данных в формате с плавающей точкой. До этого момента мы рассматривали команды и алгоритмы обработки целочисленных данных (чисел с фиксированной точкой).

Архитектура компьютеров на базе микропроцессоров вначале опиралась исключительно на целочисленную арифметику. С ростом мощи, а главное с осознанием разработчиками микропроцессорной техники того факта, что их устройства могут составить достойную конкуренцию своим «большим» предшественникам, в архитектуре компьютеров на базе микропроцессоров стали появляться устройства для обработки чисел с плавающей точкой. В архитектуре семейства микропроцессоров Intel 80x86 устройство для обработки чисел с плавающей точкой появилось в составе компьютера на базе микропроцессора i8086/88 и получило название математический сопроцессор (далее просто сопроцессор). Выбор такого названия был обусловлен тем, что, во-первых, это устройство было предназначено для расширения вычислительных возможностей основного процессора, а, во-вторых, оно было реализовано в виде отдельной микросхемы, то есть его присутствие было необязательным. Микросхема сопроцессора для микропроцессора i8086/88 имела название i8087. С появлением новых моделей микропроцессоров Intel совершенствовались и сопроцессоры, хотя их программная модель осталась практически неизменной. Как отдельные (а, соответственно, необязательные в конкретной комплектации компьютера) устройства, сопроцессоры сохранялись вплоть до модели микропроцессора i386 и имели название i287 и i387 соответственно. Начиная с модели i486, сопроцессор выполняется в одном корпусе с основным микропроцессором и, таким образом, является неотъемлемой частью компьютера.

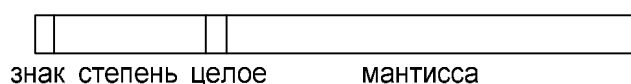
Перечислим основные возможности сопроцессора:

- полная поддержка стандартов IEEE-754 и 854 на арифметику с плавающей точкой. Эти стандарты описывают как форматы данных, с которыми должен работать сопроцессор, так и набор реализуемых им функций;

- поддержка численных алгоритмов для вычисления значений тригонометрических функций, логарифмов и т. п.;
- обработка десятичных чисел с точностью до 18 разрядов, что позволяет сопроцессору выполнять арифметические операции без округления над целыми десятичными числами со значениями до 1018;
- обработка вещественных чисел из диапазона $3.37 \times 10^{-4932} \dots 1.18 \times 10^{+4932}$.

Представление чисел с плавающей точкой в разрядной сетке вычислительной машины

Числа с плавающей точкой представляются в следующей форме



Числа с плавающей точкой могут быть представлены в одном из трех форматов.

знаковое с плавающей точкой	$\pm 1.18e-38 \dots$ $3.4e+38$	<div style="display: flex; justify-content: space-between; font-size: small;"> 31 30 23 22 0 </div> <p style="text-align: center; font-size: x-small;">знак</p>	float (DD)
знаковое с плавающей точкой двойной точности	$\pm 2.23e-308 \dots$ $1.79e+308$	<div style="display: flex; justify-content: space-between; font-size: small;"> 63 62 52 51 0 </div> <p style="text-align: center; font-size: x-small;">знак</p>	double (DQ)
знаковое с плавающей точкой двойной расширенной точности	$\pm 3.37e-4932 \dots$ $1.18e+4932$	<div style="display: flex; justify-content: space-between; font-size: small;"> 7978 6463 62 0 </div> <p style="text-align: center; font-size: x-small;">знак целое</p>	long double (DT)

Числа простой и двойной точности (float и double, DD и DQ) могут быть представлены только в нормированной форме. При этом бит целой части числа является скрытым и «подразумевает» 1. Остальные 23 (52) разряда хранят двоичную мантиссу числа.

Числа двойной расширенной точности могут быть представлены как в нормированной, так и в ненормированной форме, поскольку бит целой части числа не является скрытым и может принимать значения как 0, так и 1.

Основным типом данных, которыми оперирует математический сопроцессор, являются 10-байтные данные (DT).

Ниже приведены способы кодировки чисел с плавающей точкой в формате DT.

Для представления числа в двоичной системе счисления преобразуем отдельно целую и дробную части:

$$12345678_{10} = 1011\ 1100\ 0110\ 0001\ 0100\ 1110_2.$$

$$0,1234567_{10} = 0,0001\ 1111\ 1001\ 1010\ 1101\ 1011\ 1011_2.$$

Тогда

$$12345678,1234567_{10} = 101111000110000101001110,00011111100110101101101110111_2$$

или $1,0111\ 1000\ 1100\ 0010\ 1001\ 1100\ 0011\ 1111\ 0011\ 0101\ 1011\ 0111\ 0111_2 E_2 10111$

(сдвиг на 23 разрядов влево).

Для определения степени числа применяем сдвиг

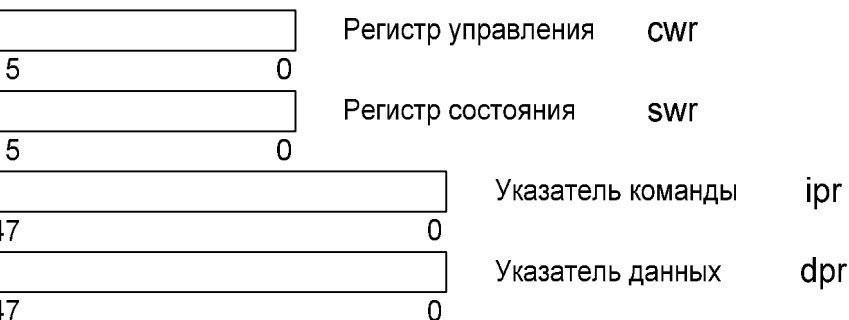
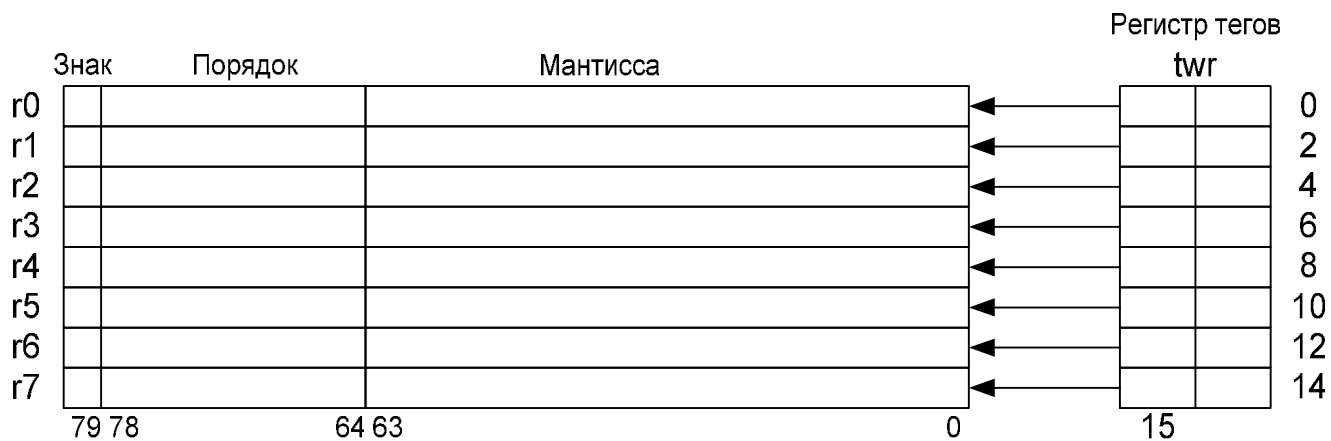
$$011\ 1111\ 1111 + 10111 = 10000010110.$$

Таким образом, число $12345678,1234567_{10}$ запишется в разрядной сетке следующим образом:

s	степень											мантисса																			
0	1	0	0	0	0	0	1	0	1	1	0	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	0	1	0	0	1
мантисса																															
1	1	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0	1	0	1	1	0	1	1	0	1	1	1	0	1	1	1

Архитектура сопроцессора

Как и в случае с основным процессором, интерес для нас представляет программная модель сопроцессора. С точки зрения программиста, сопроцессор представляет собой совокупность регистров, каждый из которых имеет свое функциональное назначение.



В программной модели сопроцессора можно выделить три группы регистров.

1. Восемь регистров $r0\dots r7$, составляющих основу программной модели сопроцессора — стек сопроцессора. Размерность каждого регистра 80 битов. Такая организация характерна для устройств, специализирующихся на обработке вычислительных алгоритмов.
2. Три служебных регистра:
 - регистр состояния сопроцессора swr (Status Word Register — регистр слова состояния) — отражает информацию о текущем состоянии сопроцессора. В регистре swr содержатся поля, позволяющие определить: какой регистр является текущей вершиной стека сопроцессора, какие исключения возникли после выполнения последней команды, каковы особенности выполнения последней команды (некий аналог регистра флагов основного процессора) и т. д.;
 - управляющий регистр сопроцессора cwr (Control Word Register — регистр слова управления) — управляет режимами работы сопроцессора. С помощью полей в этом регистре можно регулировать точность выполнения численных вычислений, управлять округлением, маскировать исключения;

- регистр слова тегов `twr` (Tags Word Register — слово тегов) — используется для контроля за состоянием каждого из регистров `r0...r7`. Команды сопроцессора используют этот регистр, например, для того, чтобы определить возможность записи значений в эти регистры.
3. Два регистра указателей — данных `dpr` (Data Point Register) и команд `ipr` (Instruction Point Register). Они предназначены для запоминания информации об адресе команды, вызвавшей исключительную ситуацию и адресе ее операнда. Эти указатели используются при обработке исключительных ситуаций (но не для всех команд).

Все эти регистры являются программно доступными. Однако к одним из них доступ получить достаточно легко, для этого в системе команд сопроцессора существуют специальные команды. К другим регистрам получить доступ сложнее, так как специальных команд для этого нет, поэтому необходимо выполнить дополнительные действия.

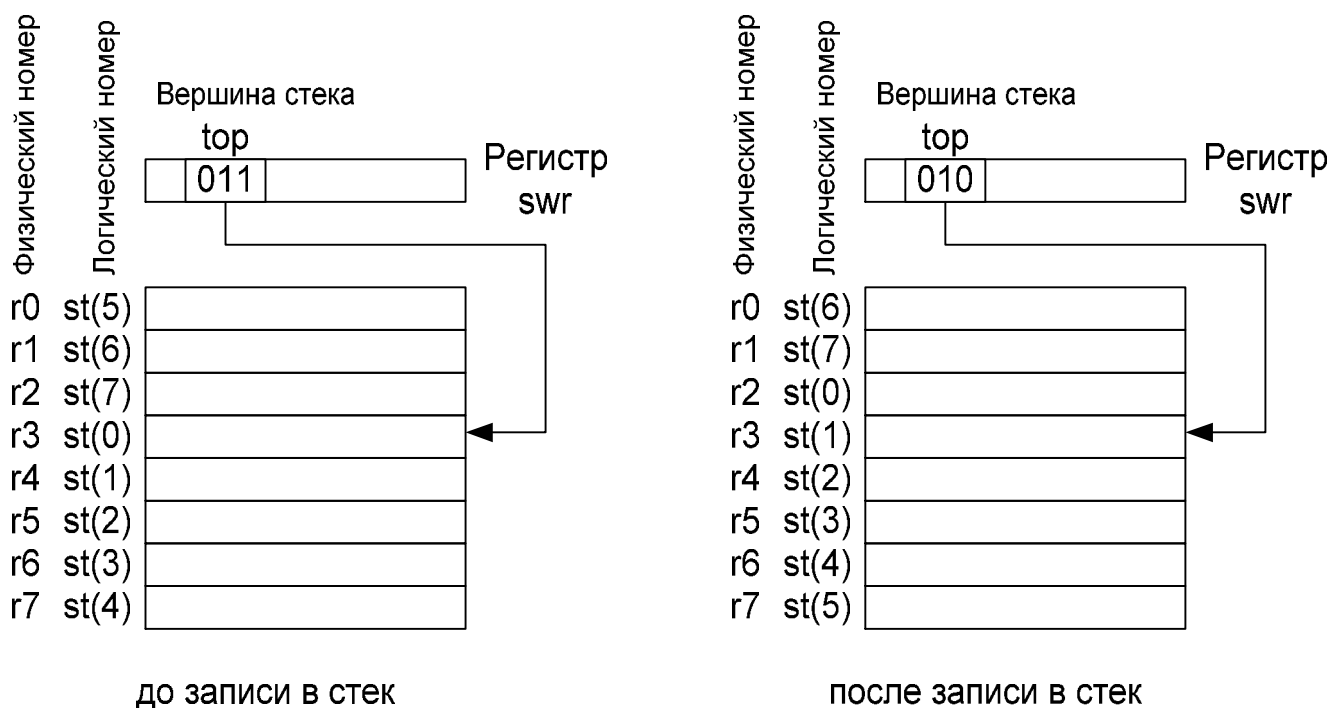
Рассмотрим общую логику работы сопроцессора и более подробно охарактеризуем перечисленные регистры.

Регистровый стек сопроцессора организован по принципу кольца. Это означает, что среди всех регистров, составляющих стек, нет такого, который является вершиной стека. Напротив, все регистры стека с функциональной точки зрения абсолютно одинаковы и равноправны. Но, как известно, в стеке всегда должна быть вершина. И она действительно есть, но является плавающей. Контроль текущей вершины осуществляется аппаратно с помощью трехбитового поля `top` регистра `swr`. В поле `top` фиксируется номер регистра стека `0...7` (`r0...r7`), являющегося в данный момент текущей вершиной стека.

Команды сопроцессора не оперируют физическими номерами регистров стека `r0...r7`. Вместо этого они используют логические номера этих регистров `st(0)...st(7)`. С помощью логических номеров реализуется относительная адресация регистров стека сопроцессора. На рис. 9 показан пример, когда текущей вершиной до записи в стек является физический регистр стека `r3`, а после записи в стек текущей вершиной становится физический регистр стека `r2`. То есть, по мере записи в стек, указатель его вершины движется по направлению к младшим номерам

физических регистров (уменьшается на единицу). Если текущей вершиной является $r0$, то после записи очередного значения в стек сопроцессора его текущей вершиной станет физический регистр $r7$. Что касается логических номеров регистров стека $st(0) \dots st(7)$, то они «плавают» вместе с изменением текущей вершины стека. Таким образом, реализуется принцип кольца.

Такая организация стека обладает большой гибкостью. Это хорошо видно на примере передачи параметров подпрограмме. Для повышения гибкости разработки и использования подпрограмм не желательно привязывать их по передаваемым параметрам к аппаратным ресурсам (физическим номерам регистров сопроцессора). Гораздо удобнее задавать порядок следования передаваемых параметров в виде логических номеров регистров. Такой способ передачи был бы однозначным и не требовал от разработчика знания лишних подробностей об аппаратных реализациях сопроцессора. Логическая нумерация регистров сопроцессора, поддерживаемая на уровне системы команд, идеально реализует эту идею. При этом не имеет значения, в какой физический регистр стека сопроцессора были помещены данные перед вызовом подпрограммы, определяющим является только порядок следования параметров в стеке. По этой причине подпрограмме важно знать уже не место, а только порядок размещения передаваемых параметров в стеке.



Процессор и сопроцессор имеют свои, несовместимые друг с другом системы команд и форматы обрабатываемых данных. Несмотря на то, что сопроцессор архитектурно представляет собой отдельное вычислительное устройство, он не может существовать отдельно от основного процессора.

Процессор и сопроцессор, являясь двумя самостоятельными вычислительными устройствами, могут работать параллельно. Но этот параллелизм касается только их внутренней работы над исполнением очередной команды. Оба процессора подключены к общей системной шине и имеют доступ к одинаковой информации. Иницирует процесс выборки очередной команды всегда основной процессор. После выборки команда попадает одновременно в оба процессора. Любая команда сопроцессора имеет код операции, первые пять бит, которого имеют значение 11011. Когда код операции начинается этими битами, то основной процессор по дальнейшему содержанию кода операции выясняет, требует ли данная команда обращения к памяти. Если это так, то основной процессор формирует физический адрес операнда и обращается к памяти, после чего содержимое ячейки памяти выставляется на шину данных. Если обращение к памяти не требуется, то основной процессор заканчивает работу над данной командой (не делая попытки ее исполнения!) и приступает к декодированию следующей команды из текущего входного командного потока. Что же касается сопроцессора, то выбранная команда попадает в него одновременно с основным процессором. Сопроцессор, определив по первым пяти битам, что очередная команда принадлежит его системе команд, начинает ее исполнение. Если команда требовала операнд из памяти, то сопроцессор обращается к шине данных за чтением содержимого ячейки памяти, которое к этому моменту предоставлено основным процессором. Из этой схемы взаимодействия следует, что в определенных случаях необходимо согласовывать работу обоих устройств. К примеру, если во входном потоке сразу за командой сопроцессора следует команда основного процессора, использующая результаты работы предыдущей команды, то сопроцессор не успеет выполнить свою команду за то время, когда основной процессор, пропустив сопроцессорную команду, выполнит свою. Очевидно, что логика работы программы будет нарушена. Возможна и другая ситуация. Если входной поток команд содержит последовательность из нескольких

команд сопроцессора, то процессор, в отличие от сопроцессора, проскочит их очень быстро, чего он не должен делать, так как он обеспечивает внешний интерфейс для сопроцессора. Эти и другие, более сложные ситуации, приводят к необходимости синхронизации между собой работы двух процессоров. В первых моделях микропроцессоров это делалось путем вставки перед или после каждой команды сопроцессора специальной команды `wait` или `fwait`. Работа данной команды заключалась в приостановке работы основного процессора до тех пор, пока сопроцессор не закончит работу над последней командой. В моделях микропроцессора (начиная с i486) подобная синхронизация выполняется командами `wait/fwait`, которые введены в алгоритм работы большинства команд сопроцессора. Но для некоторых команд из группы команд управления сопроцессором оставлена возможность выбора между командами с синхронизацией (ожиданием) и без нее.

Для организации эффективной работы с сопроцессором программист должен хорошо разобраться со структурой регистров и логикой их использования. Поэтому перед тем как приступить к рассмотрению команд и данных, с которыми работает сопроцессор, приведем описание структуры некоторых регистров сопроцессора.

Регистр состояния `swr` – отражает текущее состояние сопроцессора после выполнения последней команды.

B	C3	TOP			C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Структурно регистр `swr` состоит из:

- 6 флагов исключительных ситуаций;
- бита `sf` (Stack Fault) — ошибка работы стека сопроцессора. Бит устанавливается в единицу, если возникает одна из трех исключительных ситуаций — `PE`, `UE` или `IE`. В частности, его установка информирует о попытке записи в заполненный стек, или, напротив, попытке чтения из пустого стека. После того как вы проанализировали этот бит, его нужно снова установить в ноль, вместе с битами `PE`, `UE` и `IE` (если они были установлены);

- бита `es` (Error Summary) — суммарная ошибка работы сопроцессора. Бит устанавливается в единицу, если возникает любая из шести перечисленных ниже исключительных ситуаций;
- четырех битов `c0...c3` (Condition Code) — кода условия. Назначение этих битов аналогично флагам в регистре `EFLAGS` основного процессора — отразить результат выполнения последней команды сопроцессора.
- трехбитного поля `top`. Поле содержит указатель регистра текущей вершины стека.

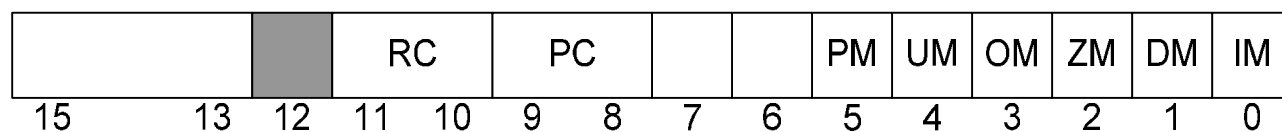
Почти половину регистра `swr` занимают биты (флаги) для регистрации исключительных ситуаций. **Исключения** — это разновидность прерываний, с помощью которых процессор информирует программу о некоторых особенностях ее реального исполнения. Сопроцессор также обладает способностью возбуждения подобных прерываний при возникновении определенных ситуаций (не обязательно ошибочных). Все возможные исключения сведены к шести типам, каждому из которых соответствует один бит в регистре `swr`. Программисту совсем не обязательно писать обработчик для реакции на ситуацию, приведшую к некоторому исключению. Сопроцессор умеет самостоятельно реагировать на многие из них. Это так называемая обработка исключений по умолчанию. Для того чтобы «заказать» сопроцессору обработку определенного типа исключения по умолчанию, необходимо это исключение замаскировать. Такое действие выполняется с помощью установки в единицу нужного бита в управляющем регистре сопроцессора `swr` (рис. 10). Приведем типы исключений, фиксируемые с помощью регистра `swr`:

- `IE` (Invalid operation Error) — недействительный код операция;
- `DE` (Denormalized operand Error) — ненормированный операнд;
- `ZE` (divide by Zero Error) — ошибка деления на нуль;
- `OE` (Overflow Error) — ошибка переполнения. Возникает в случае выхода порядка числа за максимально допустимый диапазон;
- `UE` (Underflow Error) — ошибка антипереполнения. Возникает, когда результат слишком мал;

- PE (Precision Error) — ошибка точности. Устанавливается, когда сопроцессору приходится округлять результат из-за того, что его точное представление невозможно. Так, сопроцессору никогда не удастся точно разделить 10 на 3.

При возникновении любого из этих шести типов исключений устанавливается в единицу соответствующий бит в регистре `swr`, вне зависимости от того, было ли замаскировано это исключение в регистре `swr` или нет.

Регистр управления работой сопроцессора `swr` – определяет особенности обработки числовых данных.



Он состоит из:

- шести масок исключений;
- поля управления точностью PC (Precision Control);
- поля управления округлением RC (Rounding Control).

Шесть масок предназначены для маскирования исключительных ситуаций, возникновение которых фиксируется с помощью шести бит регистра `swr`. Если какие-то маскирующие биты исключений в регистре `swr` установлены в единицу, то это означает, что соответствующие исключения будут обрабатываться самим сопроцессором. Если для какого-либо исключения в соответствующем бите масок исключений регистра `swr` содержится нулевое значение, то при возникновении исключения этого типа будет возбуждено прерывание 16 (10h). Операционная система должна содержать (или программист должен написать) обработчик этого прерывания. Он должен выяснить причину прерывания, после чего, если это необходимо, исправить ее, а также выполнить другие действия.

Поле управления точностью PC предназначено для выбора длины мантиссы. Возможные значения в этом поле означают:

- PC=00 — длина мантиссы 24 бита;
- PC=10 — длина мантиссы 53 бита;
- PC=11 — длина мантиссы 64 бита.

По умолчанию устанавливается значение поля $RC=11$.

Поле управления округлением RC позволяет управлять процессом округления чисел в процессе работы сопроцессора. Необходимость операции округления может появиться в ситуации, когда после выполнения очередной команды сопроцессора получается не представимый результат, например, периодическая дробь $3,333\dots$. Установив одно из значений в поле RC , можно выполнить округление в необходимую сторону.

Значения поля RC с соответствующим алгоритмом округления:

- 00 — значение округляется к ближайшему числу, которое можно представить в разрядной сетке регистра сопроцессора;
- 01 — значение округляется в меньшую сторону;
- 10 — значение округляется в большую сторону;
- 11 — производится отбрасывание дробной части числа. Используется для приведения значения к форме, которая может использоваться в операциях целочисленной арифметики.

Регистр тегов twr – представляет собой совокупность двухбитовых полей.

Каждое поле соответствует определенному физическому регистру стека и характеризует его текущее состояние. Изменение состояния любого регистра стека отражается на содержимом соответствующего этому регистру поля регистра тега.

Возможны следующие значения в полях регистра тега:

- 00 — регистр стека сопроцессора занят допустимым ненулевым значением;
- 01 — регистр стека сопроцессора содержит нулевое значение;
- 10 — регистр стека сопроцессора содержит одно из специальных численных значений, за исключением нуля;
- 11 — регистр пуст и в него можно производить запись. Нужно отметить, что это значение в одном из двухбитовых полей регистра тегов не означает, что все биты соответствующего регистра стека должны быть обязательно нулевыми.

Поскольку при написании программы разработчик манипулирует не абсолютными, а относительными номерами регистров стека, у него могут возникнуть трудности при попытке интерпретации содержимого регистра тегов twr , с соответствующими физическими регистрами стека. В качестве связующего звена необходимо привлекать информацию из поля top регистра swr .

Система команд сопроцессора

Система команд сопроцессора включает в себя около 80 машинных команд, включающих в себя

- команды передачи данных;
- команды сравнения данных;
- арифметические команды;
- трансцендентные команды;
- команды управления.

Мнемоническое обозначение команд сопроцессора характеризует особенности их работы и в связи с этим может представлять определенный интерес. Поэтому коротко рассмотрим основные моменты образования названий команд:

- все мнемонические обозначения начинаются с символа *f* (*float*);
- вторая буква мнемонического обозначения определяет тип операнда в памяти, с которым работает команда:

i — целое двоичное число;

b — целое десятичное число;

отсутствие буквы — вещественное число;

- последняя буква мнемонического обозначения команды *r* означает, что последним действием команды обязательно является извлечение операнда из стека;
- последняя или предпоследняя буква *r* (*reversed*) означает реверсивное следование операндов при выполнении команд вычитания и деления, так как для них важен порядок следования операндов.

Система команд сопроцессора отличается большой гибкостью в выборе вариантов задания команд, реализующих определенную операцию, и их операндов. Минимальная длина команды сопроцессора — 2 байта.

Все команды сопроцессора оперируют регистрами стека сопроцессора. Если операнд в команде не указывается, то по умолчанию используется вершина стека сопроцессора (логический регистр *st(0)*). Если команда выполняет действие с двумя операндами по умолчанию, то эти операнды – регистры *st(0)* и *st(1)*.

Команда	Операнды	Флаги				Пояснение	Описание
		C3	C2	C1	C0		
Команды передачи данных							
<i>Вещественного типа</i>							
FLD	src	U	U	+	U	TOP-=1;ST(0)=src;	Загрузка в стек
FST	dst	U	U	+	U	dst=ST(0);	Пересылка в память
FSTP	dst	U	U	+	U	dst=ST(0);TOP+=1;	Пересылка в память с выталкиванием из стека
FXCH	[ST(i)]	U	U	+	U	ST(0)↔ST(i)	Обмен значений ST(0) и ST(i)
<i>Целого типа</i>							
FILD	src	U	U	+	U	TOP-=1;ST(0)=src;	Загрузка в стек
FIST	src	U	U	+	U	dst=ST(0);	Пересылка в память
FISTP	dst	U	U	+	U	dst=ST(0);TOP+=1;	Пересылка в память с выталкиванием из стека
<i>Двоично-десятичного типа</i>							
FBLD	src	U	U	+	U	TOP-=1;ST(0)=src;	Загрузка в стек
FBSTP	dst	U	U	+	U	dst=ST(0);TOP+=1;	Пересылка в память с выталкиванием из стека
Команды загрузки констант							
FLDZ		U	U	+	U	TOP-=1;ST(0)=0;	Загрузка 0
FLD1		U	U	+	U	TOP-=1;ST(0)=1;	Загрузка 1
FLDPI		U	U	+	U	TOP-=1;ST(0)=3.1415926535;	Загрузка π
FLDL2T		U	U	+	U	TOP-=1;ST(0)=3,3219280948;	Загрузка log ₂ 10
FLDL2E		U	U	+	U	TOP-=1;ST(0)=1,4426950408;	Загрузка log ₂ e
FLDLG2		U	U	+	U	TOP-=1;ST(0)=0,3010299956;	Загрузка lg 2
FLDLN2		U	U	+	U	TOP-=1;ST(0)=0,6931471805;	Загрузка ln 2
Команды сравнения данных							
<i>Вещественного типа</i>							
FCOM FUCOM	[src]	+	+	+	+	ST(0)-src;	Сравнение вещественное
FCOMP FUCOMP	[src]	+	+	+	+	ST(0)-src; TOP+=1;	Сравнение вещественное с выталкиванием
FCOMPP FUCOMPP		+	+	+	+	ST(0)-ST(1); TOP+=2;	Сравнение вещественное с выталкиванием
FCOMI FUCOMI	ST,[ST(i)]	-	-	-	+	ST(0)-ST(i);	Сравнение вещественное с модификацией EFLAGS
FCOMIP FUCOMIP	ST,[ST(i)]	-	-	-	+	ST(0)-ST(i); TOP+=1;	Сравнение вещественное с выталкиванием и модификацией EFLAGS
FXAM		+	+	+	+		Анализ ST(0)
<i>Целого типа</i>							
FICOM	src	+	+	+	+	ST(0)-src;	Сравнение целочисленное
FICOMP	src	+	+	+	+	ST(0)-src; TOP+=1;	Сравнение целочисленное с выталкиванием
FTST		+	+	+	+	ST(0)-0.0;	Сравнение с нулем
Арифметические команды							
<i>Целого типа</i>							
FIADD	src	U	U	+	U	ST(0)=ST(0)+src;	Сложение целочисленное
FISUB	src	U	U	+	U	ST(0)=ST(0)-src;	Вычитание целочисленное
FISUBR	src	U	U	+	U	ST(0)=src-ST(0);	Вычитание целочисленное реверсивное
FIMUL	src	U	U	+	U	ST(0)=ST(0)*src;	Умножение целочисленное
FIDIV	src	U	U	+	U	ST(0)=ST(0)/src;	Деление целочисленное
FIDIVR	src	U	U	+	U	ST(0)=src/ST(0);	Деление целочисленное реверсивное

Команда	Операнды	Флаги				Пояснение	Описание
		C3	C2	C1	C0		
<i>Вещественного типа</i>							
FADD	[dst], src	U	U	+	U	dst=dst+src;	Сложение вещественное
FADDP	[ST(i), ST(0)]	U	U	+	U	ST(i)=ST(i)+ST(0); TOP+=1;	Сложение вещественное с выталкиванием
FSUB	[dst], src	U	U	+	U	dst=dst-src;	Вычитание вещественное
FSUBR	[dst], src	U	U	+	U	dst=src-dst;	Вычитание вещественное реверсивное
FSUBP	[ST(i), ST(0)]	U	U	+	U	ST(i)=ST(i)-ST(0); TOP+=1;	Вычитание вещественное с выталкиванием
FSUBRP	[ST(i), ST(0)]	U	U	+	U	ST(i)=ST(0)-ST(i); TOP+=1;	Вычитание вещественное реверсивное с выталкиванием
FMUL	[dst], src	U	U	+	U	dst=dst*src;	Умножение вещественное
FMULP	[ST(i), ST(0)]	U	U	+	U	ST(i)=ST(i)*ST(0); TOP+=1;	Умножение вещественное с выталкиванием
FDIV	[dst], src	U	U	+	U	dst=dst/src;	Деление вещественное
FDIVR	[dst], src	U	U	+	U	dst=src/dst;	Деление вещественное реверсивное
FDIVP	[ST(i), ST(0)]	U	U	+	U	ST(i)=ST(i)/ST(0); TOP+=1;	Деление вещественное с выталкиванием
FDIVRP	[ST(i), ST(0)]	U	U	+	U	ST(i)=ST(0)/ST(i); TOP+=1;	Деление вещественное реверсивное с выталкиванием
<i>Дополнительные арифметические команды</i>							
FSQRT		U	U	+	U	ST(0) = $\sqrt{ST(0)}$	Вычисление квадратного корня
FABS		U	U	+	U	ST(0) = ST(0)	Вычисление модуля
FCHS		U	U	+	U	ST(0) = -ST(0)	Изменение знака
FEXTRACT		U	U	+	U	temp = ST(0); ST(0)=порядок(temp); TOP-=1; ST(0)=мантисса(temp);	Выделение порядка и мантиссы
FSCALE		U	U	+	U	ST(0) = ST(0) * 2 ^{ST(1)}	Масштабирование по степеням 2
FRNDINT		U	U	+	U	ST(0) = (ST(0))	Округление ST(0)
FPREM		+	+	+	+	ST(0) = ST(0) - Q * ST(1);	Частичный остаток от деления
FPREM1		+	+	+	+	ST(0) = ST(0) - Q * ST(1), ST(0) <= ST(1) / 2	Частичный остаток от деления
<i>Команды трансцендентных функций</i>							
FCOS		U	+	+	U	ST(0) = cos(ST(0));	Вычисление косинуса
FSIN		U	+	+	U	ST(0) = sin(ST(0));	Вычисление синуса
FSINCOS		U	+	+	U	temp = ST(0); ST(0) = sin(temp); TOP -= 1; ST(1) = cos(temp);	Вычисление синуса и косинуса
FPTAN		U	+	+	U	ST(0) = tg(ST(0)); TOP -= 1; ST(1) = 1.0;	Вычисление тангенса
FPATAN		U	U	+	U	ST(1) = atan(ST(1)/ST(0)); TOP += 1;	Вычисление арктангенса
F2XM1		U	U	+	U	ST(0) = 2 ^{ST(0)} - 1;	Вычисление выражения $y = 2^x - 1$
FYL2X		U	U	+	U	x = ST(0); y = ST(1); TOP += 1; ST(0) = y * log ₂ x;	Вычисление выражения $z = y \cdot \log_2 x$
FYL2XP1		U	U	+	U	x = ST(0); y = ST(1); TOP += 1; ST(0) = y * log ₂ x;	Вычисление выражения $z = y \cdot \log_2 x$

Команда	Операнды	Флаги				Пояснение	Описание
		C3	C2	C1	C0		
Команды управления сопроцессором							
FWAIT		-	-	-	-		Синхронизация работы с основным процессором
FINIT FNINIT		-	-	-	-	CWR=037Fh; SWR=0; TWR=FFFFh; DPR=0; IPR=0;	Инициализация сопроцессора
FSTSW FSTNSW	dst AX	-	-	-	-	dst=SWR; AX = SWR;	Считать слово состояния сопроцессора в память
FSTCW FSTNCW	Dst AX	U	U	U	U	dst=CWR; AX = CWR;	Считать слово управления сопроцессора в память
FLDCW	src	U	U	U	U	CWR=src;	Загрузить слово управления сопроцессора
FCLEX FNCLEX		U	U	U	U	SWR=SWR & 7F00h	Сброс флагов исключений
FINCSTP		U	U	U	U	TOP+=1;	Увеличение указателя стека сопроцессора на 1
FDECSTP		U	U	U	U	TOP-=1;	Уменьшение указателя стека сопроцессора на 1
FFREE	ST(i)	U	U	U	U	TAG(i)=11b	Очистка указанного регистра
FNOP		-	-	-	-		Пустая операция
FSAVE FNSAVE	dst	0	0	0	0	...	Сохранение состояния среды сопроцессора
FRSTOR	src	+	+	+	+	...	Восстановление состояния среды сопроцессора
FSTENV FNSTENV	dst	U	U	U	U	...	Частичное сохранение состояния среды сопроцессора
FLDENV	src	+	+	+	+	...	Восстановление частичного состояния среды сопроцессора

Команды передачи данных

Группа команд передачи данных предназначена для организации обмена между регистрами стека, вершиной стека сопроцессора и ячейками оперативной памяти. Команды этой группы имеют такое же значение для процесса программирования сопроцессора, как и команда `mov` основного процессора. С помощью этих команд осуществляются все перемещения значений операндов в сопроцессор и из него. По этой причине для каждого из трех типов данных, с которыми может работать сопроцессор, существует своя подгруппа команд передачи данных. Собственно на этом уровне все его умения по работе с различными форматами данных и заканчиваются. Главной функцией всех команд загрузки данных в сопроцессор является преобразование их к единому представлению в виде вещественного числа расширенного формата. Это же касается и обратной операции — сохранения в памяти данных из сопроцессора.

Команды передачи данных можно разделить на следующие группы:

- команды передачи данных в вещественном формате;
- команды передачи данных в целочисленном формате;
- команды передачи данных в двоично-десятичном формате.

Команды передачи данных в вещественном формате:

Команда FLD – загрузка вещественного числа из области памяти в вершину стека сопроцессора.

Синтаксис: FLD источник

Помещает операнд `источник` в вершину стека сопроцессора. Операнд может быть простой, двойной или двойной расширенной точности. Если операнд задан в формате простой или двойной точности, он автоматически преобразуется к формату двойной расширенной точности перед помещением в стек.

Устанавливает в 1 признак `C1` при переполнении стека.

Команда FST/FSTP – сохранение вещественного числа из вершины стека сопроцессора в память.

Синтаксис: FST/FSTP приемник

Копирует значение из вершины стека сопроцессора в операнд приемник. Операнд может быть ячейкой памяти или другим регистром стека сопроцессора. При сохранении значения в памяти оно автоматически преобразуется в формат с простой или двойной точностью. Если операндом является ячейка памяти, то она определяет адрес первого байта операнда-приемника. Если операндом является регистр, то он определяет регистр в стеке сопроцессора относительно вершины стека, определенной полем `top` регистра `swr`.

Признак `C1` определяет направление округления.

`C1=1` – округление вверх, `C1=0` – округление вниз.

Команда `FSTP` – в отличие от `FST` осуществляет выталкивание вещественного числа из стека после его сохранения в памяти. Команда изменяет поле `top`, увеличивая его на единицу. Вследствие этого, вершиной стека становится следующий, больший по своему физическому номеру, регистр стека сопроцессора.

Команды передачи данных в целочисленном формате:

Команда `FILD` – загрузка целого числа из памяти в вершину стека сопроцессора.

Синтаксис: `FILD источник`

Помещает знаковый целочисленный операнд `источник` в вершину стека сопроцессора. Операнд может быть словом, двойным словом или 8-байтным словом. Значение операнда перед помещением в стек преобразуется к формату с плавающей точкой двойной расширенной точности.

Устанавливает в 1 признак `C1` при переполнении стека.

Команда `FIST/FISTP` – сохранение целого числа из вершины стека сопроцессора в память.

Синтаксис: `FIST/FISTP приемник`

Преобразует значение из вершины стека сопроцессора в знаковое целое (в соответствии с таблицей) и сохраняет его в операнде приемник. Значение может быть сохранено в целочисленном формате слова или двойного слова. Операнд определяет адрес первого байта сохраняемого значения.

ST(0)	DEST
$-\infty$ или очень маленькое число	недопустимая операция #IA

$ST(0) \leq -1$	int со знаком «-»
$-1 < ST(0) < -0$	0 или -1 в зависимости от режима округления
-0, +0	0
$+0 < ST(0) < +1$	0 или +1 в зависимости от режима округления
$ST(0) \geq +1$	int со знаком «+»
$+\infty$ или очень большое число	недопустимая операция #IA
нечисло (NaN)	недопустимая операция #IA

Признак C1 определяет направление округления.

C1=1 – округление вверх,

C1=0 – округление вниз.

Команды передачи данных в десятичном формате:

Команда FBLD – загрузка десятичного числа из памяти в вершину стека сопроцессора.

Синтаксис: FBLD источник

Помещает знаковый двоично-десятичный операнд источник в вершину стека сопроцессора, преобразуя его в формат с плавающей точкой двойной расширенной точности. Знак операнда сохраняется за исключением значения -0.

Устанавливает в 1 признак C1 при переполнении стека.

Команда FBSTP – сохранение десятичного числа из вершины стека сопроцессора в области памяти.

Синтаксис: FBSTP приемник

Значение выталкивается из стека после преобразования его в формат двоично-десятичного упакованного числа и сохраняется в операнде приемник. Для двоично-десятичных чисел нет команды сохранения значения в памяти без выталкивания из стека.

Признак C1 определяет направление округления.

C1=1 – округление вверх,

C1=0 – округление вниз.

К группе команд передачи данных можно отнести команду обмена вершины регистрового стека $st(0)$ с любым другим регистром стека.

Команда FXCH – обмен значений между текущей вершиной стека и регистром стека сопроцессора $st(i)$.

Синтаксис: FXCH [операнд]

Производит обмен значений вершины стека $ST(0)$ и стекового регистра $ST(i)$. Если операнд не задан, то команда производит обмен регистров $ST(0)$ и $ST(1)$.

Сбрасывает в 0 признак C1 при пустом стеке.

Действие команд загрузки FLD, FILD, FBLD можно сравнить с командой push основного процессора. Аналогично ей (push уменьшает значение в регистре sp), команды загрузки сопроцессора перед сохранением значения в регистровом стеке сопроцессора вычитают из содержимого поля top регистра состояния swr единицу. Это означает, что вершиной стека становится регистр с физическим номером на единицу меньше. При этом возможно переполнение стека. Так как стек сопроцессора состоит из ограниченного числа регистров, то в него может быть записано максимум восемь значений. Из-за кольцевой организации стека, девятое записываемое значение затрет первое. Программа должна иметь возможность обработать такую ситуацию. По этой причине почти все команды, помещающие свой операнд в стек сопроцессора, после уменьшения значения поля top, проверяет регистр — кандидат на новую вершину стека — на предмет его занятости. Для анализа этой и подобных ситуаций используется регистр twr, содержащий слово тегов. Наличие регистра тегов в архитектуре сопроцессора позволяет избежать разработки программистом сложной процедуры распознавания содержимого регистров сопроцессора и дает самому сопроцессору возможность фиксировать определенные ситуации, например, попытку чтения из пустого регистра или запись в непустой регистр. Возникновение таких ситуаций фиксируется в регистре состояния swr, предназначенном для сохранения общей информации о сопроцессоре.

Команды загрузки констант

Основным назначением сопроцессора является поддержка вычислений с плавающей точкой. В математических вычислениях достаточно часто встречаются предопределенные константы. Сопроцессор хранит значения некоторых из них. Другая причина использования этих констант заключается в том, что для

определения их в памяти (в расширенном формате) требуется 10 байт, а это для хранения, например, единицы, расточительно (сама команда загрузки константы, хранящейся в сопроцессоре, занимает два байта). В формате, отличном от расширенного, эти константы хранить не имеет смысла, так как теряется время на их преобразование в тот же расширенный формат. Для каждой predetermined константы существует своя специальная команда, которая производит загрузку ее в вершину регистрового стека сопроцессора.

Команды загрузки констант помещают одну из 7 часто используемых констант в вершину стека. Значение константы преобразуется к формату с плавающей точкой двойной повышенной точности.

Устанавливают в 1 признак C1 при переполнении стека.

Команда FLDZ – загрузка нуля в вершину стека сопроцессора;

Команда FLD1 – загрузка единицы в вершину стека сопроцессора;

Команда FLDP1 – загрузка числа π в вершину стека сопроцессора;

Команда FLDL2T – загрузка $\log_2 10$ в вершину стека сопроцессора;

Команда FLDL2E – загрузка $\log_2 e$ в вершину стека сопроцессора;

Команда FLDLG2 – загрузка $\lg 2$ в вершину стека сопроцессора;

Команда FLDLN2 – загрузка $\ln 2$ в вершину стека сопроцессора;

Команды сравнения данных

Команды данной группы выполняют сравнение значений числа в вершине стека и операнда, указанного в команде.

Команды сравнения данных в вещественном формате:

Команда F(U)COM/F(U)COMP/F(U)COMPP – сравнение значения в вершине стека с операндом.

Синтаксис: FCOM/FCOMP [источник]
 FCOMPP

Сравнивает содержимое регистра ST(0) со значением операнда источник. По умолчанию (если операнд не задан) производит сравнение регистров ST(0) и ST(1). В качестве операнда может быть задана ячейка памяти или регистр. Команда устанавливает биты C0, C2, C3 регистра swt в соответствии с таблицей. Сбрасывает в 0 признак C1 при опустошении стека.

Условие	C3	C2	C0
$ST(0) > src$	0	0	0
$ST(0) < src$	0	0	1
$ST(0) = src$	1	0	0
Недопустимая операция (#IA)	1	1	1

Команда FCOMP дополнительно выталкивает значение из ST(0).

Команда FCOMPP — сравнивает значения ST(0) и ST(1) и, после сравнения, выталкивает оба эти значения из стека.

Команда FCOMI/FUCOMI/FCOMIP/FUCOMIP – сравнение значения в вершине стека с операндом.

Синтаксис: FCOMI/FUCOMI/FCOMIP/FUCOMIP [ST(i)]

Сравнивает содержимое регистра ST(0) со значением операнда ST(i). Команда устанавливает биты ZF, PF, CF регистра EFLAGS в соответствии с таблицей. Сбрасывает в 0 признак C1 при опустошении стека.

Условие	ZF	PF	CF	Переход
$ST(0) > ST(i)$	0	0	0	ja
$ST(0) < ST(i)$	0	0	1	jb
$ST(0) = ST(i)$	1	0	0	je
Недопустимая операция (#IA)	1	1	1	
$ST(0) \geq ST(i)$	(1)	0	0	jae
$ST(0) \leq ST(i)$	(1)	0	(1)	jbe

Команда FCOMIP/FUCOMIP последним действием осуществляет выталкивание значения из ST(0).

Команда FXAM – проверка значения в вершине стека.

Синтаксис: FXAM

Проверяет содержимое регистра ST(0) и устанавливает биты C0, C2, C3 регистра swt в соответствии с таблицей. Бит C1 устанавливается равным знаковому биту ST(0).

Класс	C3	C2	C1
Неподдерживаемый формат	0	0	0
Нечисло (NaN)	0	0	1
Конечное число	0	1	0
Бесконечность	0	1	1
Ноль	1	0	0
Пустой регистр	1	0	1
Ненормированное число	1	1	0

Команды сравнения данных в целочисленном формате:

Команда FICOM/FICOMP – сравнение значения в вершине стека с целочисленным операндом.

Синтаксис: FICOM/FICOMP источник

Сравнивает содержимое регистра ST(0) с целочисленным значением операнда источник. Длина целого операнда – 16 или 32 бита. Перед выполнением сравнения целочисленный операнд преобразуется к вещественному типу двойной расширенной точности. Команда устанавливает биты C0, C2, C3 регистра swr в соответствии с таблицей 15. Устанавливает в 1 признак C1 при переполнении стека.

Команда FICOMP последним действием выталкивает значения из ST(0).

Команда FTST — сравнение значения в вершине стека с нулем.

Синтаксис: FTST

Команда не имеет операндов и сравнивает значения в ST(0) со значением 0.0 и устанавливает биты C0, C2, C3 регистра swr в соответствии с таблицей 15. Сбрасывает в 0 признак C1 при опустошении стека.

Арифметические команды

Команды сопроцессора, входящие в данную группу, реализуют четыре основные арифметические операции — сложение, вычитание, умножение и деление. Имеется также несколько дополнительных команд, предназначенных для повышения эффективности использования основных арифметических команд. С точки зрения типов операндов, арифметические команды сопроцессора можно разделить на команды, работающие с вещественными и целыми числами.

Целочисленные арифметические команды

Целочисленные арифметические команды предназначены для работы на тех участках вычислительных алгоритмов, где в качестве исходных данных используются целые числа в памяти в формате слово и короткое слово, имеющие размерность 16 и 32 бит. Перед произведением операции целочисленное значение преобразуется в вещественному формату двойной повышенной точности.

Синтаксис: FIxxx m

Команда FIADD – сложение $ST(0) = ST(0) + m$.

Команда FISUB – вычитание $ST(0) = ST(0) - m$.

Команда FISUBR – реверсивное вычитание $ST(0) = m - ST(0)$.

Команда FIMUL – умножение $ST(0) = ST(0) * m$.

Команда FIDIV – деление $ST(0) = ST(0) / m$.

Команда FIDIVR – реверсивное деление $ST(0) = m / ST(0)$.

Команды целочисленной арифметики сбрасывает в 0 признак C1 при пустом стеке, устанавливается в 1 при округлении результата.

Вещественные арифметические команды

Схема расположения операндов вещественных команд традиционна для команд сопроцессора. Один из операндов располагается в вершине стека сопроцессора — регистре $ST(0)$, куда после выполнения команды записывается и результат. Другой операнд может быть расположен либо в памяти, либо в другом регистре стека сопроцессора. Допустимыми типами операндов в памяти являются вещественные форматы простой и двойной точности.

В отличие от целочисленных арифметических команд, вещественные арифметические команды допускают большее разнообразие в сочетании местоположения операндов и самих команд для выполнения конкретного арифметического действия.

Синтаксис:	Fxxx m	; $ST(0) = ST(0) + m$
	Fxxx ST(0), ST(i)	; $ST(0) = ST(0) + ST(i)$
	Fxxx(P) ST(i), ST(0)	; $ST(i) = ST(i) + ST(0)$
	FxxxP	; $ST(0) = ST(0) + ST(1)$

Команда FADD/FADDP – сложение

Команда FSUB/FSUBP – вычитание

Команда FSUBR/FSUBRP – реверсивное вычитание

Команда FMUL/FMULP – умножение

Команда FDIV/FDIVR – деление

Команда FDIVR/FDIVRP – реверсивное деление

Команды вещественной арифметики сбрасывает в 0 признак C1 при пустом стеке, устанавливается в 1 при округлении результата.

Дополнительные арифметические команды

Команды этой группы не имеют операндов, производят действие с операндом в вершине стека сопроцессора. Результат выполнения операции сохраняется в регистре $ST(0)$. Сбрасывают в 0 признак C1 при пустом стеке, устанавливают в 1 при округлении результата.

Команда FSQRT вычисление квадратного корня: $ST(0) = \sqrt{ST(0)}$.

Команда FABS – вычисление модуля: $ST(0) = |ST(0)|$.

Команда FCHS – изменение знака: $ST(0) = -ST(0)$.

Команда FXTRACT – выделение порядка и мантиссы. Операнд-источник по умолчанию, хранящийся в регистре $ST(0)$, разделяется на порядок и мантиссу, порядок сохраняется в $ST(0)$, а затем мантисса помещается в стек, меняя при этом указатель вершины стека (поле top). Для операнда, хранящего мантиссу, знак и мантисса остаются неизменными по сравнению с операндом источника. Вместо порядка записывается 3FFFh. После выполнения команды регистр $ST(1)$ хранит значение порядка исходного операнда.

Команда FSCALE – команда масштабирования: изменяет порядок значения, находящегося в вершине стека сопроцессора $ST(0)$ на величину $ST(1)$. Команда не имеет операндов. Величина в $ST(1)$ рассматривается как число со знаком. Его прибавление к полю порядка вещественного числа в $ST(0)$ означает его умножение на величину $2^{ST(1)}$. С помощью данной команды удобно масштабировать на степень двойки некоторую последовательность чисел в памяти. Для этого достаточно последовательно загружать числа последовательности в вершину стека, после чего применять команду FSCALE и сохранять значения обратно в памяти. Команда FSCALE является обратной по алгоритму работы команде FXTRACT.

Сопроцессор имеет программно-аппаратные средства для выполнения операции округления тех результатов работы команд, которые не могут быть точно представлены. Но операция округления может быть проведена и принудительно к значению в регистре $ST(0)$, для этого предназначена последняя команда в группе дополнительных команд — команда округления.

Команда FRNDINT – округляет значение, находящееся в вершине стека сопроцессора $ST(0)$. Команда не имеет операндов.

Возможны четыре режима округления величины в $ST(0)$, которые определяются значениями в двухбитовом поле RC управляющего регистра сопроцессора. Для изменения режима округления используются команды FSTCWR и FLDCWR, которые, соответственно, записывают в память содержимое управляющего регистра сопроцессора и восстанавливают его обратно. Таким образом, пока содержимое этого регистра находится в памяти, вы имеете возможность установить необходимое значение поля RC.

Команда FPREM – получение частичного остатка от деления. Исходные значения делимого и делителя размещаются в стеке — делимое в $ST(0)$, делитель в $ST(1)$. Команда производит вычисления по формуле

$$ST(0) = ST(0) - Q * ST(1),$$

где Q – целочисленное частное от деления.

Делитель рассматривается как некоторый модуль. Поэтому в результате работы команды получается остаток от деления по модулю. Физически работа команды заключается в реализации хорошо известного всем действия: деление в столбик. При этом каждое промежуточное деление осуществляется отдельной командой FPREM. Цикл, центральное место в котором занимает команда FPREM, завершается, когда очередная полученная разность в $ST(0)$ становится меньше значения модуля в $ST(1)$. Судить об этом можно по состоянию флага C2 в регистре состояния swr:

- если $C2=0$, то работа команды fprem полностью завершена, так как разность в $ST(0)$ меньше значения модуля в $ST(1)$;
- если $C2=1$, то необходимо продолжить выполнение команды fprem, так как разность в $ST(0)$ больше значения модуля в $ST(1)$.

Таким образом, необходимо анализировать флаг C2 в теле цикла. Для этого C2 записывается в регистр флагов основного процессора с последующим анализом его командами условного перехода. Другой способ заключается в сравнении $ST(0)$ и $ST(1)$.

Команда `fprem` не соответствует последнему стандарту на вычисления с плавающей точкой IEEE-754. По этой причине в систему команд сопроцессора i387 была введена команда `fprem1`

Команда `FPREM1` – отличается от команды `FPREM` тем, что накладывается дополнительное требование на значение остатка в $ST(0)$. Это значение не должно превышать половины модуля в $ST(1)$.

После полного завершения работы команды `FPREM/FPREM1` (когда $C2=0$), биты $C0, C3, C1$ содержат значения трех младших бит частного.

Команды трансцендентных функций

Сопроцессор имеет ряд команд, предназначенных для вычисления значений тригонометрических функций, таких как синус, косинус, тангенс, арктангенс, а также значений логарифмических и показательных функций.

Значения аргументов в командах, вычисляющих результат тригонометрических функций, должны задаваться в радианах. Для нахождения радианной меры угла по его градусной мере необходимо число градусов умножить на $\frac{P}{180}=0,017453$, число минут на $\frac{P}{180} \cdot 60=0,000291$, а число секунд на $\frac{P}{180} \cdot 60^2=0,000005$ и найденные произведения сложить.

Данная группа команд не имеет операндов. Результат сохраняется в регистре $ST(0)$. Сбрасывает в 0 признак $C1$ при пустом стеке, устанавливает в 1 при округлении. Признак $C2$ устанавливается в 1 при выходе значения угла за границы диапазона $-2^{63} < src < 2^{63}$.

Команда `FCOS` – вычисляет косинус угла: $ST(0)=\cos(ST(0))$.

Команда `FSIN` – вычисляет синус угла: $ST(0)=\sin(ST(0))$.

Команда `FSINCOS` – вычисляет синус и косинус угла: $ST(1)=\sin(ST(0))$, $ST(0)=\cos(ST(0))$.

Команда `FPTAN` – вычисляет частичный тангенс угла: $ST(1)=\text{tg}(ST(0))$, $ST(0)=1$. Это сделано для совместимости с сопроцессорами 8087 и 287. Выполнение данной команды в них имело следующую особенность: результат

команды возвращался в виде двух значений — в регистрах $ST(0)$ и $ST(1)$. Ни одно из этих значений не является истинным значением тангенса. Истинное его значение получается лишь после операции деления $ST(0)/ST(1)$. Таким образом, для получения тангенса требовалась еще команда деления. Синус и косинус в ранних версиях сопроцессоров вычислялись через тангенс половинного угла.

$$\sin \alpha = \frac{2 \cdot \operatorname{tg} \frac{\alpha}{2}}{1 + \operatorname{tg}^2 \frac{\alpha}{2}} \qquad \cos \alpha = \frac{1 - \operatorname{tg}^2 \frac{\alpha}{2}}{1 + \operatorname{tg}^2 \frac{\alpha}{2}}$$

В микропроцессоре i387 появились самостоятельные команды для вычисления синуса и косинуса, вследствие чего отпала необходимость составлять соответствующие подпрограммы. Что же до команды `fptan`, то она по-прежнему выдает два значения в $st(0)$ и $st(1)$. Но значение в $st(0)$ всегда равно единице, это означает, что в $st(1)$ находится истинное значение тангенса числа, находившегося в $st(0)$ до выполнения команды `fptan`.

Команда `FPTAN` — вычисляет частичный арктангенс угла:

$ST(0) = \operatorname{arctg} \left(\frac{ST(1)}{ST(0)} \right)$. Команда не имеет операндов. Результат возвращается в регистр $ST(1)$, после чего производится выталкивание вершины стека.

Команда `FPTAN` широко применяется для вычисления значений обратных тригонометрических функций таких, как: `arcsin`, `arccos`, `arcctg`, `arccosec`, `arcsec`. Например, для вычисления функции `arcsin` используется следующая формула:

$$\operatorname{arcsin}(a) = \operatorname{arctg} \left(\frac{a}{\sqrt{1 - a^2}} \right)$$

Для вычисления этой формулы необходимо выполнить следующую последовательность шагов:

- Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру.
- Поместить в стек a в радианной мере.
- Вычислить значение выражения $\sqrt{1 - a^2}$ и поместить его в стек.
- Выполнить команду `fpatan` с аргументами в $st(0) = \sqrt{1 - a^2}$ и $st(1) = a$.

В результате этих действий в регистре $ST(0)$ будет сформировано значение, которое и будет являться значением $\arcsin(a)$.

Аналогично вычисляются значения других обратных тригонометрических функций.

$$\arccos(a) = 2 \cdot \arctg\left(\frac{\sqrt{1-a}}{\sqrt{1+a}}\right) \quad \operatorname{arcc} \operatorname{tg}(a) = \arctg\left(\frac{1}{a}\right)$$

Команда F2XM1 – вычисляет значение функции: $ST(0) = 2^{ST(0)} - 1$. Исходное значение x размещается в вершине стека сопроцессора $ST(0)$ и должно лежать в диапазоне $-1 \leq x \leq +1$. Результат замещает значение в регистре $ST(0)$. Эта команда может быть использована для вычисления показательных функций.

Единица вычитается для того, чтобы получить точный результат, когда x близок к нулю. Поскольку нормированное число всегда содержит в качестве первой значащей цифры единицу, если в результате вычисления функции 2^x получается число 1,000000000456..., то команда F2XM1, вычитая 1 из этого числа, и затем, нормируя результат, формирует больше значащих цифр, то есть делает его более точным. Неявное вычитание единицы командой F2XM1 компенсируется командой FADD с единичным операндом.

Команда FYL2X – вычисляет значение функции $ST(0) = ST(1) \cdot \log_2 ST(0)$. Значение x должно лежать в диапазоне $0 < x < +\infty$. Перед тем, как осуществить запись результата в вершину стека, команда FYL2X выталкивает значения x и y из стека, и только после этого производит запись результата в стек.

Команда FYL2XP1 – вычисляет $ST(0) = ST(1) \cdot \log_2 (ST(0) + 1)$.

Значение x должно лежать в диапазоне $-\left(1 - \frac{\sqrt{2}}{2}\right) < x < \left(1 - \frac{\sqrt{2}}{2}\right)$. Перед тем, как осуществить запись результата в вершину стека, команда FYL2XP1 выталкивает значения x и y из стека, и только после этого производит запись результата в стек.

Поскольку специальной команды в сопроцессоре для операции возведения в степень нет, возведение в произвольную степень числа с любым основанием производится по формуле:

$$x^y = 2^{y \cdot \log_2 x}$$

Вычисление значения выражения $y \cdot \log_2 x$ для любых y и $x > 0$ производится специальной командой сопроцессора `FYL2X`. Вычисление 2^x производится командой `F2XM1`. Лишнее действие вычитания 1 можно компенсировать сложением с единицей. Но в последнем действии есть тонкий момент, который связан с тем, что величина аргумента x должна лежать в диапазоне: $-1 \leq x \leq +1$. А как быть в случае, если x превышает это значение (например, для вычисления 16^3). При вычислении выражения $3 \cdot \log_2 16$ командой `FYL2X`, получим в стеке значение 12. Попытка вычислить значение 2^{12} командой `F2XM1` ни к чему не приведет — результат будет не определен. В этой ситуации используется команда сопроцессора `FSCALE`, которая также вычисляет значение выражения 2^x , но для целых значений x со знаком. Применив формулу $2^{a+b} = 2^a \cdot 2^b$, получаем решение проблемы. Разделяем дробный показатель степени больший 1 по модулю на две части — целую и дробную. После этого вычисляем отдельно командами `FSCALE` и `F2XM1` степени двойки и перемножаем результаты (не забываем компенсировать вычитание единицы в команде `F2XM1` последующим сложением ее результата с единицей).

Команды управления математическим сопроцессором

Данная группа команд предназначена для общего управления работой сопроцессора. Команды этой группы имеют особенность — перед началом своего выполнения они не проверяют наличие незамаскированных исключений. Однако такая проверка может понадобиться, в частности для того, чтобы при параллельной работе основного процессора и сопроцессора предотвратить разрушение информации, необходимой для корректной обработки исключений, возникших в сопроцессоре. Поэтому некоторые команды управления имеют аналоги, выполняющие те же действия плюс одну дополнительную функцию — проверку наличия исключения в сопроцессоре. Эти команды имеют одинаковые мнемокоды (и машинные коды тоже), отличающиеся только вторым символом — символом *n*:

- мнемокод, не содержащий второго символа *n*, обозначает команду, которая перед началом своего выполнения проверяет наличие незамаскированных исключений;
- мнемокод, содержащий второй символ *n*, обозначает команду, которая перед началом своего выполнения не проверяет наличия незамаскированных исключений, то есть выполняется немедленно, что позволяет сэкономить несколько машинных тактов.

Эти команды имеют одинаковый машинный код. Отличие лишь в том, что перед командами, не содержащими символа *n*, транслятор ассемблера вставляет команду `wait`. Команда `wait` является полноценной командой основного процессора и ее, при необходимости, можно указывать явно. Команда `wait` имеет аналог среди команд сопроцессора — `fwait`. Общим этим командам соответствует код операции `9bh`.

Команда `FWAIT` — команда ожидания. Предназначена для синхронизации работы процессора и сопроцессора. Так как основной процессор и сопроцессор работают параллельно, то может создаться ситуация, когда за командой сопроцессора, изменяющей данные в памяти, следует команда основного процессора, которой эти данные требуются. Чтобы синхронизировать работу этих команд, необходимо включить между ними команду `wait/fwait`. Встретив данную команду в потоке команд, основной процессор приостановит свою работу до

тех пор, пока не поступит аппаратный сигнал о завершении очередной команды в сопроцессоре.

Команда FINIT/FNINIT — инициализация сопроцессора. Данная команда инициализирует управляющие регистры сопроцессора определенными значениями:

CWR = 037Fh

- RC=00b; округление к ближайшему целому;
- PM, UM, OM, ZM, DM, IM=1; все исключения замаскированы;
- PC=11b; максимальная точность 64 бита.

SWR = 0h – отсутствие исключений и указание на то, что физический регистр стека сопроцессора r0 является вершиной стека и соответствует логическому регистру ST(0);

TWR = FFFFh – все регистры стека сопроцессора пусты;

DPR=0; IPR=0.

Данную команду используют перед первой командой сопроцессора в программе и в других случаях, когда необходимо привести сопроцессор в начальное состояние.

Команда FSTSW/FNSTSW — сохранение содержимого регистра состояния swr в регистре ax или в ячейке памяти размером 2 байта. Эту команду целесообразно использовать для подготовки к условным переходам по описанной при рассмотрении команд сравнения схеме.

Команда FSTCW/FNSTCW — сохранение содержимого регистра управления swr в ячейке памяти размером 2 байта. Эту команду целесообразно использовать для анализа полей маскирования исключений, управления точностью и округления. Следует заметить, что в качестве операнда назначения не используется регистр ax, в отличие от команды FSTSW/FNSTSW.

Команда FLDCW — загрузки значения ячейки памяти размером 16 бит в регистр управления swr. Эта команда выполняет действие, противоположное командам FSTCW/FNSTCW. Команду целесообразно использовать для задания или изменения режима работы сопроцессора. Если в регистре состояния swr установлен любой бит

исключения, то попытка загрузки нового содержимого в регистр управления `swr` приведет к возбуждению исключения. По этой причине необходимо перед загрузкой регистра `swr` сбросить все флаги исключений в регистре `swr`.

Команда FCLEX/FNCLEX — позволяет сбросить флаги исключений, которые, в частности, необходимы для корректного выполнения команды `FLDCW`. Ее также применяют и в случаях, когда необходимо сбрасывать флаги исключений в регистре `swr`, например, в конце подпрограмм обработки исключений. Если этого не делать, то при исполнении первой же команды сопроцессора прерванной программы (кроме тех команд, которые имеют в названии своего мнемокода второй символ `n`) будет опять возбуждено исключение.

`PE, UE, OE, ZE, DE, IE, ES, SF` и `B` биты регистра `SWR` равны `0`.

Команда FINCSTP — увеличение указателя стека на единицу (поле `top`) в регистре `swr`. Команда не имеет операндов. Действие команды `fincstp` подобно команде `FST`, но она извлекает значение операнда из стека «в никуда». Таким образом, эту команду можно использовать для выталкивания, ставшего ненужным операнда, из вершины стека. Команда работает только с полем `top` и не изменяет соответствующее данному регистру поле в регистре тегов `twr`, то есть регистр остается занятым и его содержимое из стека не извлекается.

Команда FDECSTP — уменьшение указателя стека (поле `top`) в регистре `swr`. Команда не имеет операндов. Действие команды `FDECSTP` подобно команде `FLD`, но она не помещает значения операнда в стек. Таким образом, эту команду можно использовать для проталкивания внутрь стека операндов, ранее включенных в него. Команда работает только с полем `top` и не изменяет соответствующее данному регистру поле в регистре тегов `twr`, то есть регистр остается пустым.

Команда FFREE — помечает любой регистр стека сопроцессора как пустой.

Синтаксис: `FFREE ST(i)`

Команда записывает в поле регистра тегов, соответствующего регистру `ST(i)`, значение `11b`, что соответствует, пустому регистру. При этом указатель стека (поле

top) в регистре swr и содержимое самого регистра не изменяются. Необходимость в этой команде может возникнуть при попытке записи в регистр ST(i), который помечен, как непустой. В этом случае будет возбуждено исключение. Для предотвращения этого применяется команда FFREE.

Команда FNOP — пустая операция. Не производит никаких действий и влияет только на регистр указателя команды IPR.

Команда FSAVE/FNSAVE — сохранения полного состояния среды сопроцессора в память по адресу, указанному операндом приемник. Размер области памяти зависит от размера операнда сегмента кода use16 или use32:

- use 16 — область памяти должна быть 94 байта: 80 байт для восьми регистров из стека сопроцессора и 14 байт для остальных регистров сопроцессора с дополнительной информацией;
- use32 — область памяти должна быть 108 байт: 80 байт для восьми регистров из стека сопроцессора и 28 байт для остальных регистров сопроцессора с дополнительной информацией.

После выполнения сохранения состояния среды сопроцессора производится инициализация сопроцессора.

use16		use32		
CWR	0	0	CWR	0
SWR	2	0	SWR	4
TWR	4	0	TWR	8
IRP(0-15)	6	IRP(0-31)		12
CS	8	0	IPR(31-47)	16
DPR(0-15)	10	DPR(0-31)		20
IP	12	0	DPR(31-47)	24
ST(0)	14	ST(0)		28
ST(1)	24	ST(1)		38
...		...		
ST(7)	84	ST(7)		98
15	0	31	15	0

Команда FRSTOR — используется для восстановления полного состояния среды сопроцессора из области памяти, адрес которой указан операндом источник.

Сопроцессор будет работать в новой среде сразу после окончания работы команды `frstor`.

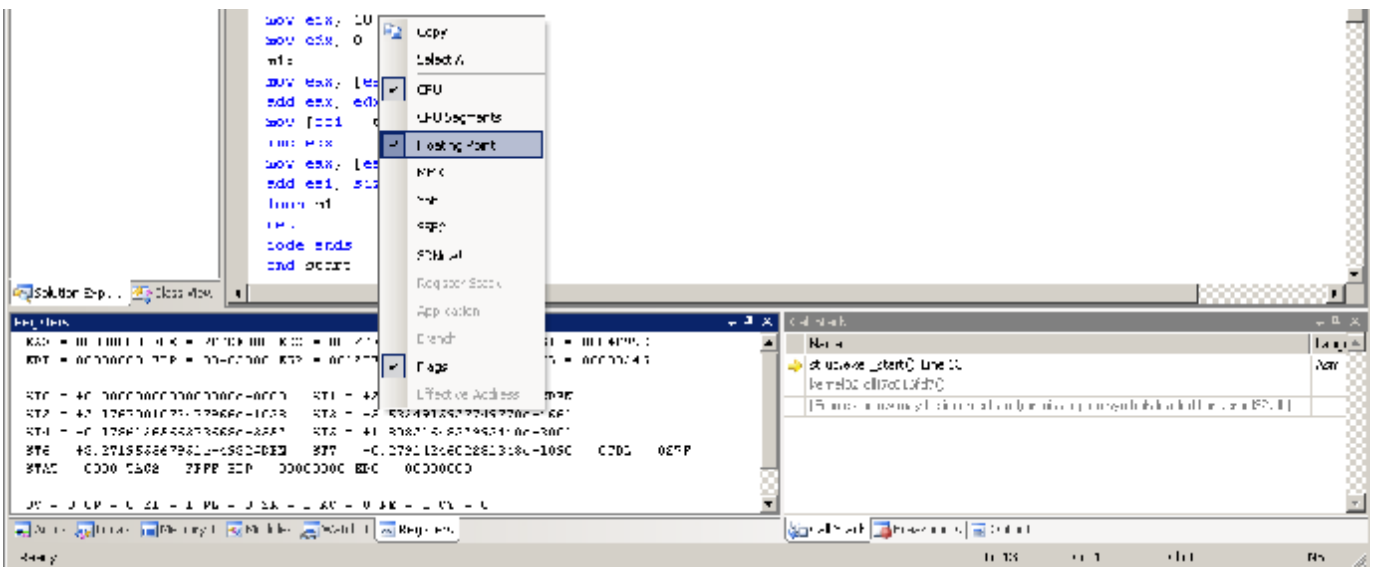
Команда `FSTENV/FNSTENV` — сохранение частичного состояния среды сопроцессора в область памяти, адрес которой указан операндом приемник. Размер области памяти зависит от размера операнда сегмента кода `use16` или `use32`. Формат области частичной среды сопроцессора совпадает с форматом области полной среды, за исключением содержимого стека сопроцессора (80 байт).

Команда `FLDENV` — восстановление частичного состояния среды сопроцессора содержимым из области памяти, адрес которой указан операндом источник. Информация в данной области памяти была ранее сохранена командой `FSTENV/FNSTENV`.

Команды сохранения среды целесообразно применять в обработчиках исключений, так как только с помощью данных команд можно получить доступ, например, к регистрам `DPR` и `IPR`. Не исключено использование этих команд при написании подпрограмм или при переключении контекстов программ в многозадачной среде.

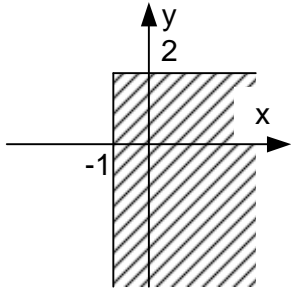
Отладка программ, использующих функции сопроцессора

Для просмотра содержимого регистров сопроцессора в среде Microsoft Visual Studio в окне регистров выбрать всплывающее меню `Floating Point`.



Пример программы с использованием команд сопроцессора

Определить, принадлежит ли точка заштрихованной части плоскости.



Программа запрашивает вещественные координаты точки X и Y, после чего с помощью команды сопроцессора сравнивает координату Y с 2, а X – с (-1). В соответствии с результатом сравнения выдается сообщение о

принадлежности или непринадлежности точки.

```
#include <windows.h>
#include <tchar.h>
void main()
{
float x,y;
int cons=2;
char s[40];
CharToOem(_T("Введите координаты точки X и Y: "),s);
printf(s);
scanf("%f%f",&x,&y);
_asm
{
    fild cons
    fld y
    fcomip ST,ST(1)
    jae m
    fstp cons ; освобождение стека
    mov cons, -1
    fild cons
    fld x
    fcomip ST,ST(1)
    jb m
}
CharToOem(_T("Точка принадлежит плоскости."),s);
_asm jmp ext;
m:
CharToOem(_T("Точка не принадлежит плоскости."),s);
ext:
printf(s);
getch();
}
```

СЛОЖНЫЕ ТИПЫ ДАННЫХ

Структуры

Часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип. Это очень актуально, например, для программ баз данных, где необходимо связывать совокупность данных разного типа с одним объектом. Такой объект обычно описывается с помощью специального типа данных — структуры. С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

По определению **структура** — это тип данных, состоящий из фиксированного числа элементов разного типа. Для использования структур в программе необходимо выполнить три действия:

- Задать шаблон структуры. По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
- Определить экземпляр структуры. Этот этап подразумевает инициализацию конкретной переменной с заранее определенной (с помощью шаблона) структурой.
- Организовать обращение к элементам структуры.

Описать структуру в программе означает лишь указать ее схему или шаблон; память при этом не выделяется. Этот шаблон можно рассматривать лишь как информацию для транслятора о расположении полей и их значении по умолчанию. Определить структуру — значит дать указание транслятору выделить память и присвоить этой области памяти символическое имя. Описать структуру в программе можно только один раз, а определить — любое количество раз.

Описание шаблона структуры

Описание шаблона структуры имеет следующий синтаксис:

```
имя_структуры STRUC  
<описание полей>  
имя_структуры ENDS
```

Здесь <описание полей> представляет собой последовательность директив описания данных db, dw, dd, dq и dt. Их операнды определяют размер полей и при

необходимости начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

Местоположение шаблона в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, он должен быть расположен до того места, где определяется переменная с типом данной структуры. То есть при описании в секции данных переменной с типом некоторой структуры ее шаблон необходимо поместить в начале секции данных либо перед ней.

Рассмотрим работу со структурами на примере моделирования базы данных о сотрудниках некоторого отдела. Для простоты, чтобы уйти от проблем преобразования информации при вводе, условимся, что все поля символьные. Определим структуру записи этой базы данных следующим шаблоном:

```
worker  struc                ; информация о сотруднике
nam      db  30 dup ( " " ) ; фамилия, имя, отчество
position db  30 dup ( " " ) ; должность
age      dd  20                ; возраст
worker  ends
```

Определение данных типа «структура»

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данной структуры. Для этого используется следующая синтаксическая конструкция:

```
[имя_переменной] имя_структуры <[список_значений]>
```

имя_переменной — идентификатор переменной данного структурного типа.

Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры;

список_значений — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми. Его задание также необязательно. Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы. Допускается инициализация отдельных полей, но в этом случае пропущенные поля должны отделяться запятыми. Пропущенные поля будут инициализированы значениями из шаблона структуры. Если при определении новой переменной с

типом данной структуры все поля структуры инициализируются значениями по умолчанию, то нужно просто написать угловые скобки.

Для примера определим несколько переменных с типом структуры `worker`:

```
data segment
sotr1 worker<"Гурко Андрей Вячеславович", 'художник', 33>
sotr2 worker<"Степанова Ольга Валерьевна", 'связист'>
sotr3 worker<> ;все значения по умолчанию
data ends
```

Методы работы со структурой

Идея введения структурного типа в любой язык программирования состоит в объединении разнотипных переменных в один объект. В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того, чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор — символ «.» (точка). Он используется в следующей синтаксической конструкции:

адресное_выражение.имя_поля_структуры

адресное_выражение — идентификатор переменной структурного типа;

имя_поля_структуры — имя поля из шаблона структуры, представляет собой смещение поля от начала структуры. Таким образом оператор «.» вычисляет выражение:

(адресное_выражение) + (имя_поля_структуры)

Продемонстрируем на примере структуры `worker` некоторые приемы работы со структурами. Пусть требуется извлечь в `eax` и `ebx` значения поля с возрастом.

```
code segment
start:
mov eax, sotr1.age
mov ebx, sotr2.age
...
ret
code ends
```

ИЛИ

```
code segment
start:
lea ebx, sotr1
mov eax, [ebx + worker.age]
mov edx, [ebx + size(worker) + worker.age]
ret
code ends
```

Язык ассемблера позволяет определять не только отдельную переменную с типом структуры, но и массив структур:

```
mas_sotr worker 10 dup (<>)
```

Дальнейшая работа с массивом структур производится так же, как и с одномерным массивом.

Аналогично другим идентификаторам, определенным в программе, транслятор назначает имени типа структуры и имени переменной с типом структуры атрибут типа. Значением этого атрибута является размер в байтах, занимаемый полями этой структуры. Извлечь это значение можно с помощью оператора `type`. После того, как стал известен размер экземпляра структуры, организовать индексацию в массиве структур можно следующим образом:

```
...
data segment
mas_worker worker 10 dup(<>)
data ends
code segment
start:
lea esi, mas_worker
mov ecx, 10
mov edx, 0
m1:
mov eax, [esi + worker.age]
add eax, edx
mov [esi + worker.age], eax
inc edx
mov eax, [esi + worker.age]
add esi, size(worker)
loop m1
ret
code ends
end start
```

Объединения

Язык ассемблера предоставляет возможность переопределить некоторую область памяти для объекта с другим типом и именем. Для этого используется специальный тип данных, называемый объединением. **Объединение** — тип данных, позволяющий трактовать одну и ту же область памяти как имеющую разные типы и имена.

Описание объединений в программе напоминает описание структур, то есть сначала описывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей:

```
[имя_объединения] UNION  
<описание полей>  
[имя_объединения] ENDS
```

Отличие объединений от структур состоит, в частности, в том, что при определении переменной типа объединения память выделяется в соответствии с размером максимального элемента. Обращение к элементам объединения происходит по их именам, но при этом нужно, конечно, помнить о том, что все поля в объединении накладываются друг на друга. Одновременная работа с элементами объединения исключена.

```
MY UNION  
mb    db    ?  
mw    dw    ?  
md    dd    ?
```

Для описания объекта объединения в тексте программы используется конструкция вида

```
.data  
...  
MyData MY <> ;определение экземпляра объединения
```

Для обращения к полям объединения используется оператор «.» (точка)

```
mov MyData.mw, 1  
mov al, MyData.mb      ; al=1
```


ПРОГРАММИРОВАНИЕ ДЛЯ WINDOWS

Любая программа на языке самого высокого уровня в своем внутреннем виде представляет собой последовательность машинных кодов. А раз так, то всегда остается теоретическая возможность написать ту же программу, но уже на языке ассемблера. Чем обосновать необходимость разработки Windows-приложений на языке ассемблера?

- язык ассемблера позволяет программисту полностью контролировать создаваемый им программный код и оптимизировать его по своему усмотрению;
- компиляторы языков высокого уровня помещают в загрузочный модуль программы избыточную информацию. Эквивалентные исполняемые модули, исходный текст которых написан на языке ассемблера, имеют в несколько раз меньший размер;
- при программировании на ассемблере сохраняется полный доступ к аппаратным ресурсам компьютера;
- приложение, написанное на языке ассемблера, как правило, быстрее загружается в оперативную память компьютера;
- приложение, написанное на языке ассемблера, обладает, как правило, более высокой скоростью работы и реактивностью ответа на действия пользователя.

Основы программирования в ОС Windows

Возможны 3 типа структур программ для Windows:

- диалоговая (основное окно — диалоговое),
- консольная, или безоконная структура,
- классическая структура

Программирование в Windows основывается на использовании функций API (Application Program Interface, т.е. интерфейс программного приложения). Их количество достигает двух тысяч. Программа для Windows в значительной степени состоит из таких вызовов. Все взаимодействие с внешними устройствами и ресурсами операционной системы будет происходить посредством таких функций.

Система Windows является многозадачной. Если программа DOS после своего запуска должна быть постоянно активной, и если ей что-то требуется (к примеру, получить очередную порцию данных с устройства ввода-вывода), то она сама должна выполнять соответствующие запросы к операционной системе, то в Windows все наоборот. Программа пассивна, после запуска она ждет, когда ей уделит внимание операционная система. Операционная система делает это посылкой специально оформленных групп данных, называемых **сообщениями**. Сообщения могут быть разного типа, они функционируют в системе достаточно хаотично, и приложение не знает, какого типа сообщение придет следующим. Логика построения Windows-приложения должна обеспечивать корректную и предсказуемую работу при поступлении сообщений любого типа. Для обеспечения нормального функционирования своей программы программист должен уметь эффективно использовать функции интерфейса прикладного программирования (API, Application Program Interface) операционной системы, количество которых насчитывает более двух тысяч.

Диалоговые приложения для Windows имеют минимальный интерфейс связи с пользователем и передают информацию пользователю посредством стандартных диалоговых окон (например, окно сообщения MessageBox). Работа программы происходит «вслепую».

Неоконные приложения, также называемые *консольными*, представляет собой программу, работающую в текстовом режиме. Работа консольного приложения напоминает работу программы MS-DOS. Но это лишь внешнее впечатление. Консольное приложение обеспечивается специальными функциями Windows. Примером консольного приложения является Far.

Оконные приложения строятся на базе специального набора функций (API), составляющих графический интерфейс пользователя (GUI, Graphic User Interface). Главным элементом такого приложения является окно. Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы, по сути, также являются окнами, но обладающими особыми свойствами. События, происходящие с этими элементами (и самим окном), приводят к приходу сообщений в процедуру окна.

Разница между оконными и консольными приложениями Windows состоит в том, с каким типом информации они работают.

Все приложения Windows имеют 32-битную архитектуру и используют плоскую модель памяти FLAT. Использование ключевого слова STDCALL при описании модели памяти подразумевает передачу параметров в вызываемые API-функции справа налево (в соответствии с конвенцией STDCALL соглашения о вызовах) независимо от языка, на котором написана вызываемая API-функция.

Консольные приложения Windows

Консольные приложения представляют собой систему средств взаимодействия пользователя с компьютером, основанную на использовании текстового (буквенно-цифрового) режима дисплея или аналогичных (командная строка MS-DOS, Far). Консольные приложения очень компактны не только в откомпилированном виде, но и в текстовом варианте, и имеют такие же возможности обращаться к ресурсам Windows посредством API-функций, как и обычные графические приложения.

Для работы с консольными приложениями Windows используются соответствующие функции Windows API. В программе на ассемблере записываются имена вызываемых функций как внешние, а сами функции расположены в системном каталоге Windows. Для обращения к ним используется подключаемый файл библиотеки (*.lib), в котором указывается, в каком системном .dll файле расположена вызываемая функция и файл описания прототипов функций (*.inc). Для подключения файла библиотеки используется директива `includelib`, для подключения файла прототипа – директива `include`:

```
include C:\masm32\include\kernel32.inc
includelib C:\masm32\lib\kernel32.lib
```

Подключаемые библиотеки могут указываться в программе при помощи директивы `includelib` или при сборке приложения (команда `link`).

Пример простейшего консольного приложения.

```
.586
.MODEL flat, stdcall
STD_OUTPUT_HANDLE EQU -11 ; константа Win API

include C:\masm32\include\kernel32.inc
include C:\masm32\include\user32.inc
```

```

include C:\masm32\include\msvcrt.inc
includelib C:\masm32\lib\msvcrt.lib
.data
consoleOutHandle dd ? ; дескриптор консоли вывода
bytesWritten dd ? ; количество байт(вспомогательная)
message db "Привет всем!",13,10,0 ; текст (13 = \r, 10=\n 0=\0)
h EQU $-message; ; длина текстовой строки (константа)

.code
main PROC ; начало функции
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE ; Получить дескриптор
консоли ввода
    mov consoleOutHandle,eax ; поместить его в EAX

    INVOKE CharToOem,offset message,offset message ; перекодировка
сообщения

    mov eax, h
    INVOKE WriteConsole, ; функция вывода в консоль
        consoleOutHandle, ; дескриптор консоли
        offset message, ; указатель строки
        eax, ; длина строки
        offset bytesWritten, ; количество выведенных байт
        0 ; возвращается функцией

    INVOKE crt__getch
    INVOKE ExitProcess,0 ; Окончание программы
main ENDP ; окончание процедуры (функции)
END main ; окончание модуля

```

Функция получения дескриптора стандартного устройства ввода, вывода или ошибки в зависимости от переданного константного параметра:

```
HANDLE WINAPI GetStdHandle(__in DWORD nStdHandle);
```

nStdHandle может принимать значения

STD_INPUT_HANDLE = -10	устройство ввода
STD_OUTPUT_HANDLE = -11	устройство вывода
STD_ERROR_HANDLE = -12	ошибка

Функция перекодировки русского текста, введенного в Win-кодировке, в код, читаемый в консоли (DOS-кодировку). Возвращаемое значение 1 в случае успешной перекодировки.

```
BOOL CharToOem(LPCTSTR lpszSrc, LPSTR lpszDst)
```

lpszSrc – указатель на строку-источник;

lpszDst – указатель на строку-приемник;

Функция вывода текстовой информации в консоль.

```
BOOL WINAPI WriteConsole(
```

```

__in      HANDLE hConsoleOutput,
__in      const VOID* lpBuffer,
__in      DWORD nNumberOfCharsToWrite,
__out     LPDWORD lpNumberOfCharsWritten,
__reserved LPVOID lpReserved);

```

`hConsoleOutput` – дескриптор буфера вывода консоли, который может быть получен при помощи функции `GetStdHandle`.

`lpBuffer` – указатель на буфер, где находится выводимый текст.

`nNumberOfCharsToWrite` – количество выводимых символов.

`lpNumberOfCharsWritten` – указывает на переменную DD, куда будет помещено количество действительно выведенных символов.

`lpReserved` – резервный параметр, должен быть равен 0.

Буфер, где находится выводимый текст, не обязательно должен заканчиваться нулем, поскольку для данной функции указывается количество выводимых символов.

Функция завершения Windows-приложения.

```
VOID WINAPI ExitProcess( __in UINT uExitCode);
```

параметр `uExitCode` – код завершения.

Для трансляции консольного приложения из командной строки используются следующие команды

```
ml /c /coff cons1.asm
```

```
link /subsystem:console cons1.obj
```

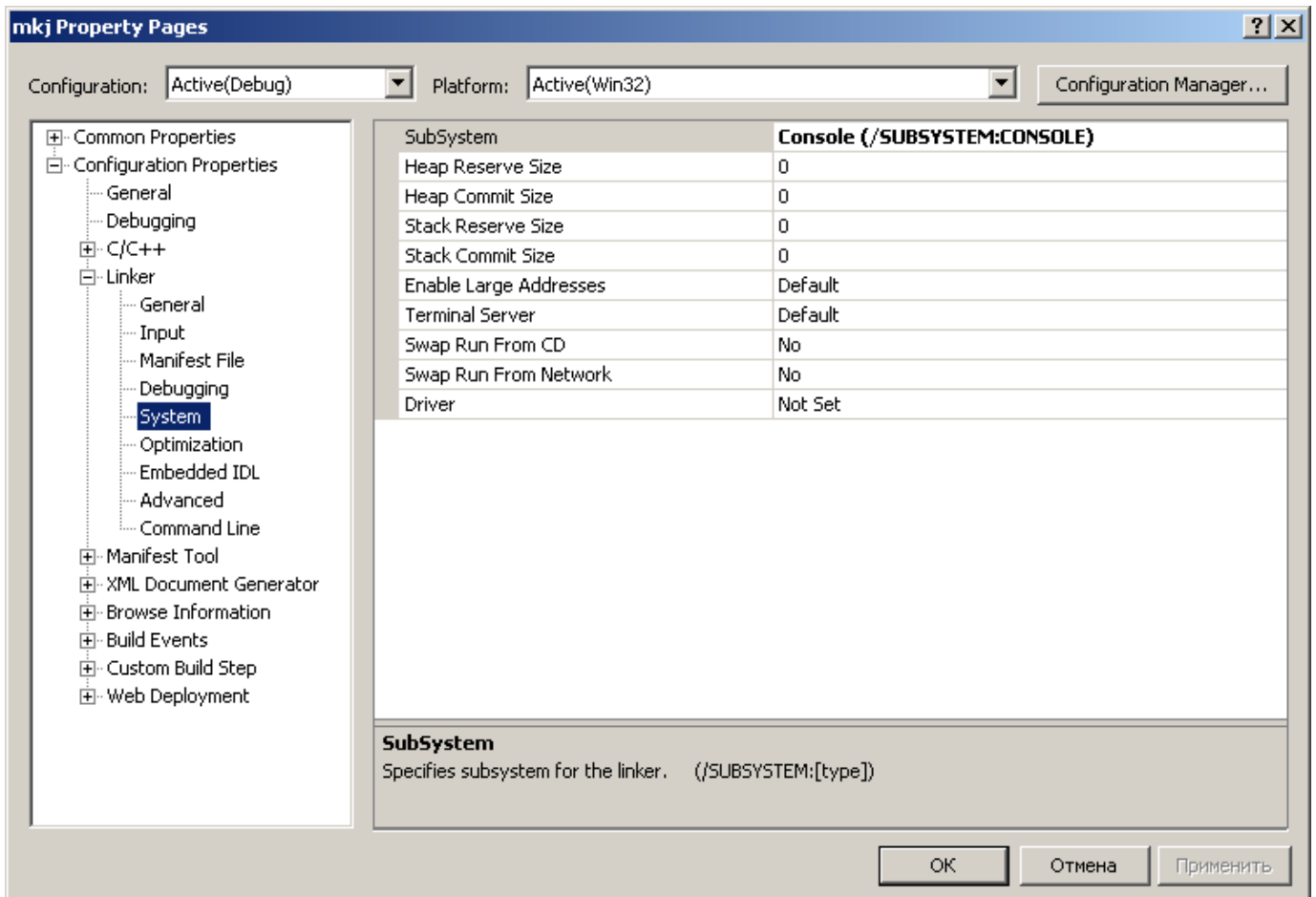
При этом необходимо объявить все вызываемые API-функции как внешние с помощью директивы `EXTERN`.

Трансляция из Visual Studio осуществляется аналогично диалоговому приложению, но необходимо указать платформу `subsystem:console` либо при создании проекта (тип Win32 Console Application), либо указать в уже существующем проекте в окне свойств.

Основные динамические библиотеки `kernel32.lib` `user32.lib` `gdi32.lib` `winspool.lib` `comdlg32.lib` `advapi32.lib` `shell32.lib` `ole32.lib` `oleaut32.lib` `uuid.lib` `odbc32.lib` `odbcsp32.lib` подключаются в командной строке редактора связей `link.exe` в Visual Studio, поэтому отсутствие директив типа

```
includelib C:\masm32\lib\kernel32.lib
```

не будет ошибкой.



Библиотеки `kernel32.lib`, `user32.lib` подключаются линковщиком по умолчанию, поэтому нет необходимости подключать их директивой `includelib`.

Консольные приложения могут создать свою консоль. В этом случае весь ввод-вывод будет производиться в эту консоль. Если приложение консоль не создает, то здесь может возникнуть двоякая ситуация: либо наследуется консоль, в которой программа была запущена (например, консоль `Fara`), либо `Windows` создает для приложения свою консоль.

Для того чтобы создать свою консоль, используется функция

```
BOOL WINAPI AllocConsole(void);
```

По завершении программы все выделенные консоли автоматически освобождаются. Однако это можно сделать и принудительно, используя функцию

```
BOOL WINAPI FreeConsole(void);
```

Дополнительные функции для работы с консолью

Функция чтения строки из буфера консоли

```
BOOL WINAPI ReadConsole(  
    __in    HANDLE hConsoleInput,  
    __out   LPVOID lpBuffer,  
    __in    DWORD nNumberOfCharsToRead,  
    __out   LPDWORD lpNumberOfCharsRead,  
    __in_opt LPVOID pInputControl);
```

`hConsoleInput` – дескриптор буфера ввода консоли, полученного функцией `GetStdHandle`.

`lpBuffer` – указатель на буфер, куда помещается вводимый текст.

`nNumberOfCharsToRead` – длина буфера ввода.

`lpNumberOfCharsRead` – указатель на переменную DD (32 бит), куда будет помещено количество действительно введенных символов.

`pInputControl` – резервный параметр, должен быть равен нулю.

Функция определения размеров окна консоли

```
BOOL WINAPI SetConsoleScreenBufferSize(  
    __in HANDLE hConsoleOutput,  
    __in COORD dwSize);
```

`hConsoleOutput` – дескриптор буфера вывода консоли;

`dwSize` – структура координат, задающая размер окна консоли:

```
COORD STRUC  
X  DW   ?  
Y  DW   ?  
COORD ENDS
```

Функция установки позиции курсора в окне консоли

```
BOOL WINAPI SetConsoleCursorPosition(  
    __in HANDLE hConsoleOutput,  
    __in COORD dwCursorPosition);
```

`hConsoleOutput` – дескриптор буфера выходной консоли;

`dwCursorPosition` – структура координат `COORD`, определяющая позицию курсора.

Функция определения заголовка окна консоли

```
BOOL WINAPI SetConsoleTitle(__in LPCTSTR lpConsoleTitle);
```

`lpConsoleTitle` — указатель на строку имени консоли, заканчивающуюся нуль-символом.

Функция определения атрибутов вводимых символов в окне консоли

```
BOOL WINAPI SetConsoleTextAttribute(  
    __in HANDLE hConsoleOutput,  
    __in WORD wAttributes);
```

`hConsoleOutput` – дескриптор буфера вывода консоли;

`wAttributes` – цвет букв и фона, получаемый путем комбинации констант

<code>FOREGROUND_BLUE</code>	<code>equ 1h</code>	<code>; синий</code>
<code>FOREGROUND_GREEN</code>	<code>equ 2h</code>	<code>; зеленый</code>
<code>FOREGROUND_RED</code>	<code>equ 4h</code>	<code>; красный</code>
<code>FOREGROUND_INTENSITY</code>	<code>equ 8h</code>	<code>; интенсивный</code>
<code>BACKGROUND_BLUE</code>	<code>equ 10h</code>	<code>; синий фон букв</code>
<code>BACKGROUND_GREEN</code>	<code>equ 20h</code>	<code>; зеленый фон букв</code>
<code>BACKGROUND_RED</code>	<code>equ 40h</code>	<code>; красный фон букв</code>
<code>BACKGROUND_INTENSITY</code>	<code>equ 80h</code>	<code>; интенсивный фон букв</code>

Функция, задающая цвет фона консоли (путем закрашивания фона отдельных символов)

```
BOOL WINAPI FillConsoleOutputAttribute(  
    __in HANDLE hConsoleOutput,  
    __in WORD wAttribute,  
    __in DWORD nLength,  
    __in COORD dwWriteCoord,  
    __out LPDWORD lpNumberOfAttrsWritten);
```

`hConsoleOutput` – дескриптор буфера вывода консоли;

`wAttribute` – атрибут цвета фона символа в консоли;

`nLength` – количество ячеек символов, фон которых устанавливается заданным цветом;

`dwWriteCoord` – координаты первой закрашиваемой ячейки;

`lpNumberOfAttrsWritten` – указатель на 4-байтный идентификатор, в который записывается количество реально закрашенных ячеек.

Функция, получающая информацию о клавиатуре и мыши в консольном режиме:

```
BOOL WINAPI ReadConsoleInput(  
    __in HANDLE hConsoleInput,  
    __out PINPUT_RECORD lpBuffer,  
    __in DWORD nLength,  
    __out LPDWORD lpNumberOfEventsRead);
```

`hConsoleInput` – дескриптор входного буфера консоли;

`lpBuffer` – указатель на структуру (или массив структур) `PINPUT_RECORD`, в которой содержится информация о событиях, происшедших с консолью;

nLength – количество получаемых информационных записей (структур);

lpNumberOfEventsRead – указатель на двойное слово, содержащее количество реально полученных записей.

Структура PINPUT_RECORD используется для получения события, происшедшего в консоли. Всего системой зарезервировано пять типов событий:

```
KEY_EV          equ 1h          ; клавиатурное событие
MOUSE_EV        equ 2h          ; событие с мышью
WINDOW_BUFFER_SIZE_EV equ 4h    ; изменился размер окна
MENU_EV         equ 8h          ; зарезервировано
FOCUS_EV        equ 10h         ; зарезервировано
```

Остальные байты структуры зависят от происшедшего события.

Событие KEY_EVENT

```
KEY_EVENT_RECORD STRUCT
  bKeyDown      DD ? ; При нажатии клавиши значение поля больше нуля
  wRepeatCount  DW ? ; Количество повторов при удержании клавиши
  wVirtualKeyCode DW ? ; Виртуальный код клавиши
  wVirtualScanCode DW ? ; Скан-код клавиши
  UNION
    UnicodeChar  DW ? ; код символа в формате Unicode
                  ; для функции (ReadConsoleInputW)
    AsciiChar    DB ? ; код символа в формате ASCII
  ENDS
  dwControlKeyState DD ? ; Содержится состояния управляющих клавиш.
; Может являться суммой следующих констант:
  ; RIGHT_ALT_PRESSED equ 1h
  ; LEFT_ALT_PRESSED equ 2h
  ; RIGHT_CTRL_PRESSED equ 4h
  ; LEFT_CTRL_PRESSED equ 8h
  ; SHIFT_PRESSED equ 10h
  ; NUMLOCK_ON equ 20h
  ; SCROLLLOCK_ON equ 40h
  ; CAPSLOCK_ON equ 80h
  ; ENHANCED_KEY equ 100h
KEY_EVENT_RECORD ENDS
```

Событие MOUSE_EVENT

```
MOUSE_EVENT_RECORD STRUCT
  dwMousePosition COORD <> ; координаты курсора мыши
  dwButtonState   DD ?      ; состояние кнопок мыши.
                  ; первый бит – левая кнопка,
                  ; второй бит – правая кнопка,
                  ; третий бит – средняя кнопка.
                  ; Бит установлен – кнопка нажата.
  dwControlKeyState DD ?      ; Состояние управляющих клавиш
  dwEventFlags     DD ?      ; Может содержать значения:
                  ; MOUSE_MOVED equ 1h; было движение мыши
                  ; DOUBLE_CLICK equ 2h; был двойной щелчок
MOUSE_EVENT_RECORD ENDS
```

Событие WINDOW_BUFFER_SIZE_EVENT

```
WINDOW_BUFFER_SIZE_RECORD STRUCT
    dwSize COORD <> ; новый размер консольного окна
WINDOW_BUFFER_SIZE_RECORD ENDS
```

События MENU_EVENT_RECORD и FOCUS_EVENT_RECORD зарезервированы.

```
MENU_EVENT_RECORD STRUCT
    dwCommandId DD ?
MENU_EVENT_RECORD ENDS
```

```
FOCUS_EVENT_RECORD STRUCT
    bSetFocus DD ?
FOCUS_EVENT_RECORD ENDS
```

Структура, объединяющая все типы событий:

```
INPUT_RECORD STRUCT
    EventType DW ?
                DW ?
    MY UNION
        KeyEvent KEY_EVENT_RECORD <>
        MouseEvent MOUSE_EVENT_RECORD <>
        WindowBufferSizeEvent WINDOW_BUFFER_SIZE_RECORD <>
        MenuEvent MENU_EVENT_RECORD <>
        FocusEvent FOCUS_EVENT_RECORD <>
    MY ENDS
INPUT_RECORD ENDS
```

В данном случае структуры всех событий: `KeyEvent`, `MouseEvent`, `WindowBufferSizeEvent`, `MenuEvent`, `FocusEvent` будут находиться в одной и той же области памяти размером 16 байт (по максимальной длине структуры `KeyEvent`, `MouseEvent`).

Рассмотрим пример консольного приложения: посчитать количество символов в строке. Программа состоит из двух файлов. Файл `console.inc` содержит основные конструкции и константы Windows, необходимые при написании программы, а также основные функции. Файл `mmm.asm` содержит текст программы.

```
; файл console.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc

; константы дескрипторов буфера
STD_INPUT_HANDLE equ -10
STD_OUTPUT_HANDLE equ -11
STD_ERROR_HANDLE equ -12
```

```

; структура координат
COORD STRUC
X    DW    ?
Y    DW    ?
COORD    ENDS

;Цвет окна консоли
FOREGROUND_BLUE equ 1h ; синий цвет букв
FOREGROUND_GREEN equ 2h ; зеленый цвет букв
FOREGROUND_RED equ 4h ; красный цвет букв
FOREGROUND_INTENSITY equ 8h ; повышенная интенсивность
BACKGROUND_BLUE equ 10h ; синий свет фона
BACKGROUND_GREEN equ 20h ; зеленый цвет фона
BACKGROUND_RED equ 40h ; красный цвет фона
BACKGROUND_INTENSITY equ 80h ; повышенная интенсивность

; тип события
KEY_EV equ 1h ; клавиатурное
событие
MOUSE_EV equ 2h ; событие с мышью
WINDOW_BUFFER_SIZE_EV equ 4h ; изменился размер
окна
MENU_EV equ 8h ; зарезервировано
FOCUS_EV equ 10h ; зарезервировано

; константы - состояния клавиатуры
RIGHT_ALT_PRESSED equ 1h
LEFT_ALT_PRESSED equ 2h
RIGHT_CTRL_PRESSED equ 4h
LEFT_CTRL_PRESSED equ 8h
SHIFT_PRESSED equ 10h
NUMLOCK_ON equ 20h
SCROLLLOCK_ON equ 40h
CAPSLOCK_ON equ 80h
ENHANCED_KEY equ 100h

; события мыши
MOUSE_MOVED equ 1h; было движение мыши
DOUBLE_CLICK equ 2h; был двойной щелчок

; описание событий структуры PINPUT_RECORD
; событие мыши
MOUSE_EVENT_RECORD STRUCT
    dwMousePosition COORD <>
    dwButtonState DWORD ?
    dwControlKeyState DWORD ?
    dwEventFlags DWORD ?
MOUSE_EVENT_RECORD ENDS

; событие клавиатуры
KEY_EVENT_RECORD STRUCT
    bKeyDown DD ?
    wRepeatCount DW ?
    wVirtualKeyCode DW ?
    wVirtualScanCode DW ?

```

```

UNION
    UnicodeChar    DW ?
    AsciiChar      DB ?
ENDS
dwControlKeyState DD ?
KEY_EVENT_RECORD ENDS

;изменение размера окна консоли
WINDOW_BUFFER_SIZE_RECORD STRUCT
    dwSize    COORD <>
WINDOW_BUFFER_SIZE_RECORD ENDS

MENU_EVENT_RECORD STRUCT
    dwCommandId  DWORD      ?
MENU_EVENT_RECORD ENDS

FOCUS_EVENT_RECORD STRUCT
    bSetFocus    DWORD      ?
FOCUS_EVENT_RECORD ENDS

;Структура PINPUT_RECORD
INPUT_RECORD STRUCT
    EventType DW ?
                DW ?
    UNION
        KeyEvent          KEY_EVENT_RECORD      <>
        MouseEvent        MOUSE_EVENT_RECORD    <>
        WindowBufferSizeEvent  WINDOW_BUFFER_SIZE_RECORD <>
        MenuEvent         MENU_EVENT_RECORD     <>
        FocusEvent        FOCUS_EVENT_RECORD    <>
    ENDS
INPUT_RECORD ENDS

; Секция данных содержит временные переменные
.data
    @CO          DD ?
    @numBytes    DD ?
    KeyEvent     INPUT_RECORD <>
    @SYMBOL      DB ?
.code
;-----
; Функция считывание символа в консоли
; consoleInHandle - дескриптор буфера консоли ввода
; consoleOutHandle - дескриптор буфера консоли вывода
; Display - управление отображением символа:
;   0 - символ отображается
;   1 - символ не отображается
; функция возвращает считанный символ в регистре al
ReadSymbol proc consoleInHandle:DWORD, consoleOutHandle:DWORD,
Display:DWORD
@L1:
    INVOKE ReadConsoleInputA,
        consoleInHandle,
        offset KeyEvent,
        1,

```

```

        offset @CO
        CMP KeyEvent.EventType,KEY_EV
        JNE @L1

; сохранение введенного символа
        MOV AL, KeyEvent.KeyEvent.AsciiChar
        MOV @SYMBOL, AL

        CMP Display,0
        JNE @L2

;ВЫВОД СИМВОЛА
        INVOKE WriteConsoleA,
            consoleOutHandle,
            OFFSET @SYMBOL,
            1,
            OFFSET @numBytes,
            0
; Считывание события клавиатуры отпущения клавиши
@L2:
        INVOKE ReadConsoleInputA,
            consoleInHandle,
            offset KeyEvent,
            1,
            offset @CO
        CMP KeyEvent.EventType,KEY_EV
        JNE @L2

        mov eax,0
        mov al, @SYMBOL
        ret
ReadSymbol endp
;-----
;Представление целого числа в текстовой форме
; Number - целое число
; Str1 - указатель на строку, в которую будет помещено представление
числа
; функция возвращает длину строки символов в регистре EAX
IntToStr proc Number:DWORD, Str1:DWORD
        MOV EAX, Number
        MOV EDI,Str1
        MOV ECX, 0
        CMP EAX,0
        JGE @I1
        MOV DL, '-'
        MOV [EDI],DL
        INC EDI
        NOT EAX
        INC EAX
@I1:
        MOV EBX, 10
        MOV EDX, 0
        IDIV EBX
        PUSH EDX
        INC ECX

```

```

        CMP EAX,0
        JG   @I1
;       MOV EBX,ECX
@I2:
        POP EDX
        ADD DL, 30h
        MOV [EDI],DL
        INC EDI
        LOOP @I2
        MOV DL,0
        MOV [EDI], DL
        INC EDI
        MOV EAX, EDI
        SUB EAX, Str1
        ret

```

IntToStr endp

```

;-----
;Определение длины строки
;Str1 - указатель на строку
; функция возвращает длину строки символов в регистре EAX
LENSTR PROC Str1:DWORD

```

```

        CLD
        CLD
        MOV EDI, Str1
        MOV EBX,EDI
        MOV ECX,100 ; ограничить длину строки
        MOV EAX, 0
        REPNE SCASB ; найти символ 0
        SUB EDI, EBX ; длина строки, включая 0
        MOV EAX,EDI
        DEC EAX
        RET

```

LENSTR ENDP

```

; Вывод строки в окно консоли
; StrPtr - указатель на выводимую строку, оканчивающуюся 0
; consoleOutHandle - дескриптор буфера консоли вывода
PrintStr proc StrPtr:DWORD, Handle:DWORD

```

```

        INVOKE CharToOem, StrPtr, StrPtr
        INVOKE LENSTR, StrPtr ; определение длины строки
        INVOKE WriteConsole,
                Handle,
                StrPtr,
                eax,
                OFFSET @numBytes, 0

```

ret

PrintStr endp

```

; файл mmm.asm
.586
.MODEL FLAT, stdcall
include console.inc
.DATA
    consoleOutHandle DD ? ; дескриптор выходного буфера
    consoleInHandle DD ? ; дескриптор входного буфера
    COUNT DD 0 ; счетчик количество символов
    numBytes DD ?
    TITL DB "Подсчет количества символов в строке",0
    IN_STR DB "Введите строку: ",0
    IN_SYM DB "Введите символ: ",0
    BUF DB 200 dup (?) ; для ввода строки
    Len DD ? ; длина введенной строки
    Yes DB 13,10,"Количество символов: ",0
    No DB 13,10,"Символ не найден.",0
    SymCount DB 10 dup(?) ; строка с количеством символов
    CRD COORD <?> ; структура координат
.CODE
START proc
; образовать консоль, вначале освободить уже существующую
    INVOKE FreeConsole
    INVOKE AllocConsole
; получить дескрипторы
    INVOKE GetStdHandle, STD_INPUT_HANDLE
    MOV consoleInHandle,EAX
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    MOV consoleOutHandle,EAX
; задать заголовок окна консоли
    INVOKE CharToOem, OFFSET TITL, OFFSET TITL
    INVOKE SetConsoleTitle, OFFSET TITL
; задать размер окна консоли
    MOV CRD.X, 80
    MOV CRD.Y, 25
    mov eax, CRD
    INVOKE SetConsoleScreenBufferSize,
        consoleOutHandle, EAX
; задать цвет окна консоли
    INVOKE FillConsoleOutputAttribute,
        consoleOutHandle,
        BACKGROUND_BLUE + BACKGROUND_GREEN,
        2000, 0, offset numBytes
; установить позицию курсора
    MOV CRD.X,0
    MOV CRD.Y,2
    mov eax, CRD
    INVOKE SetConsoleCursorPosition, consoleOutHandle, EAX
; задать цветовые атрибуты выводимого текста
    INVOKE SetConsoleTextAttribute,
        consoleOutHandle,
        FOREGROUND_BLUE + \
        BACKGROUND_BLUE + BACKGROUND_GREEN

```

```

;Вывод сообщения Введите строку:
    INVOKE PrintStr, offset IN_STR, consoleOutHandle
;Ввод строки
    INVOKE ReadConsole,
        consoleInHandle,
        OFFSET BUF,
        200,
        OFFSET numBytes, 0
; Сохранить длину введенной строки
    MOV EAX, numBytes
    MOV Len, EAX
;Вывод сообщения Введите символ:
    INVOKE PrintStr, offset IN_SYM, consoleOutHandle
;Считывание символа
    INVOKE ReadSymbol, consoleInHandle, consoleOutHandle, 0
; поиск символа в строке
; символ содержится в регистре AL
    CLD ; поиск символа с начала строки DF=0
    LEA EDI, BUF ;
    MOV ECX, Len ; количество повторений равно длине строки
FIND:
repne SCASB
    JNE FAILED ; просмотр строки завершен
    INC COUNT
    JMP FIND
; символ не найден
FAILED:
    CMP COUNT, 0
    JNE FOUND ; количество символов не 0 - идем на FOUND
    INVOKE CharToOem, OFFSET No, OFFSET No ; таких символов нет
    INVOKE PrintStr, offset No, consoleOutHandle
    JMP EXIT
; символ найден
FOUND:
    INVOKE PrintStr, offset Yes, consoleOutHandle
    INVOKE IntToStr, COUNT, offset SymCount
    INVOKE PrintStr, offset SymCount, consoleOutHandle
EXIT:
    INVOKE ReadSymbol, consoleInHandle, consoleOutHandle, 1
    INVOKE ExitProcess, 0
    RET
START ENDP
END START

```


Работа с файлами в системе Windows

Создание и открытие файла производится функцией

```
HANDLE WINAPI CreateFile(  
    __in        LPCTSTR lpFileName,  
    __in        DWORD dwDesiredAccess,  
    __in        DWORD dwShareMode,  
    __in_opt    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    __in        DWORD dwCreationDisposition,  
    __in        DWORD dwFlagsAndAttributes,  
    __in_opt    HANDLE hTemplateFile);
```

`lpFileName` – указатель на ASCII-строку с именем (путем) открываемого или создаваемого файла;

`dwDesiredAccess` – тип доступа к файлу:

```
GENERIC_READ = 0x80000000 - доступ для чтения;  
GENERIC_WRITE = 0x40000000 - доступ для записи;  
GENERIC_READ+GENERIC_WRITE = 0xC0000000 - доступ для чтения-  
записи;
```

`dwShareMode` – режим разделения файлов между разными процессами, может принимать значения:

0 — монополизация доступа к файлу;

`FILE_SHARE_READ=0x00000001h` — другие процессы могут открыть файл, но только для чтения, запись в файл монополизирована процессом, открывшим файл;

`FILE_SHARE_WRITE=0x00000002h` — другие процессы могут открыть файл, но только для записи, чтение в файл монополизировано процессом, открывшим файл;

`FILE_SHARE_READ+FILE_SHARE_WRITE=0x00000003h` — другие процессы могут открывать файл для чтения-записи;

`lpSecurityAttributes` – указатель на структуру `SecurityAttributes` (файл `winbase.h`), определяющую защиту связанного с файлом объекта ядра; при отсутствии защиты заносится `NULL`;

`dwCreationDisposition` – действия для случаев, когда файл существует или не существует, данный параметр может принимать значения:

`CREATE_NEW=1` — создать новый файл, если файл не существует; если файл существует, то функция завершается формированием ошибки;

CREATE_ALWAYS=2 — создать новый файл, если файл не существует; если он существует, то заместить новым;

OPEN_EXISTING=3 — открыть файл, если он существует; если файл не существует, то формируется ошибка;

OPEN_ALWAYS=4 — открыть файл при его существовании и создать его если файла нет;

TRUNCATE_EXISTING=5 — открыть файл с усечением его до нулевой длины; если файл не существует, то формируется ошибка;

dwFlagsAndAttributes – флаги и атрибуты; этот параметр используется для задания характеристик создаваемого файла:

FILE_ATTRIBUTE_READONLY=00000001h – файл только для чтения;

FILE_ATTRIBUTE_HIDDEN=00000002h – скрытый файл;

FILE_ATTRIBUTE_SYSTEM=00000004h – системный файл;

FILE_ATTRIBUTE_DIRECTORY=00000010h – каталог;

FILE_ATTRIBUTE_ARCHIVE=00000020h – архивный файл;

FILE_ATTRIBUTE_NORMAL=00000080h – обычный файл для чтения-записи (этот атрибут нельзя комбинировать с другими);

FILE_ATTRIBUTE_TEMPORARY=00000100h – создается временный файл

FILE_FLAG_WRITE_THROUGH=80000000h – не использовать промежуточное кэширование при записи на диск, а все изменения записывать прямо на диск;

FILE_FLAG_NO_BUFFERING=20000000h – не использовать средства буферизации операционной системы;

FILE_FLAG_RANDOM_ACCESS=10000000h – прямой доступ к файлу (установка этого флага или флага

FILE_FLAG_SEQUENTIAL_SCAN=08000000h – последовательный доступ к файлу;

FILE_FLAG_DELETE_ON_CLOSE=04000000h – удалить файл после его закрытия;

FILE_FLAG_OVERLAPPED=40000000h – асинхронный доступ к файлу (синхронность означает то, что программа, вызвавшая функцию для доступа к файлу, приостанавливается до тех пор, пока не закончит работу функция ввода-вывода);

hTemplateFile – при создании нового файла значением данного параметра является дескриптор другого существующего и предварительно открытого файла, а новый файл создается с теми же значениями атрибутов и флагов, что и у файла, дескриптор которого указан в параметре.

При удачном завершении функция возвращает в регистре EAX дескриптор нового файла. В случае неудачи функция возвращает в регистре EAX значение NULL.

Заккрытие файла производится функцией

```
BOOL WINAPI CloseHandle( __in HANDLE hObject);
```

`hObject` – дескриптор, полученный при открытии файла функцией `CreateFile`.

При удачном завершении функция возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

Удаление файла производится функцией

```
BOOL WINAPI DeleteFile( __in LPCTSTR lpFileName);
```

`lpFileName` – указатель на ASCIIZ-строку с именем (путем) удаляемого файла.

Перед удалением файл необходимо закрыть, хотя в некоторых версиях Windows это не является обязательным.

При удачном завершении функция возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

Установка текущей файловой позиции

Доступ к содержимому файла может быть произвольным (прямым) и последовательным. Обычно, функции ввода-вывода работают с файловым указателем. Но необходимо иметь в виду, что файловый указатель связан только с дескриптором файла. Его значение равно текущему номеру позиции в файле, с которой будет производиться чтение-запись данных при очередном вызове функции ввода-вывода. В первый момент после открытия значение указателя равно 0, то есть он указывает на начало файла. Функции, производящие чтение-запись в файле, меняют значение файлового указателя на количество прочитанных или записанных байт. При необходимости, а при организации прямого доступа к файлу без этого не обойтись, значение файлового указателя можно изменять с помощью функции

```
DWORD WINAPI SetFilePointer(  
    __in HANDLE hFile,  
    __in LONG lDistanceToMove,  
    __inout_opt PLONG lpDistanceToMoveHigh,  
    __in DWORD dwMoveMethod);
```

`hFile` – дескриптор файла, в котором производится позиционирование указателя позиции (дескриптор был получен функцией `CreateFile`);

`lDistanceToMove` – расстояние в байтах, на которое необходимо переместить указатель; это число может трактоваться как знаковое и беззнаковое — его

отрицательное значение соответствует случаю, когда указатель требуется перемещать к началу файла;

`lpDistanceToMoveHigh` – используется при необходимости создания 64-битового знакового значения указателя позиции как значение старших 32 битов указателя;

`dwMoveMethod` – определяет трактовку параметров `lDistanceToMove` и `lpDistanceToMoveHigh`.

`FILE_BEGIN=0` — указатель позиции – значение без знака, заданное содержимым полей 2 и 3;

`FILE_CURRENT=1` — текущее значение указателя позиции складывается со знаковым значением, заданным в полях 2 и 3;

`FILE_END=2` — значение, определяемое содержимым полей 2 и 3, должно представлять собой отрицательное значение, и смещение в файле вычисляется как сумма, в которой первое слагаемое определяется полями 2 и 3, а второе является размером файла.

Получение размера файла осуществляет функция

```
DWORD WINAPI GetFileSize(  
    __in    HANDLE hFile,  
    __out_opt LPDWORD lpFileSizeHigh);
```

`hFile` – дескриптор файла, размер которого будет получен;

`lpFileSizeHigh` – указатель на старшее двойное слово, если размер памяти занимает больше 32 бит, если не требуется, данный аргумент равен нулю.

Функция возвращает младшее двойное слово размера файла в регистре EAX. В случае ошибки функция возвращает константу `INVALID_FILE_SIZE = 0xFFFFFFFF`.

Чтение данных из файла осуществляется функцией

```
BOOL WINAPI ReadFile(  
    __in    HANDLE hFile,  
    __out    LPVOID lpBuffer,  
    __in    DWORD nNumberOfBytesToRead,  
    __out_opt LPDWORD lpNumberOfBytesRead,  
    __inout_opt LPOVERLAPPED lpOverlapped);
```

Запись в файл осуществляется функцией

```
BOOL WINAPI WriteFile(  
    __in    HANDLE hFile,  
    __in    LPCVOID lpBuffer,  
    __in    DWORD nNumberOfBytesToWrite,
```

```
__out_opt    LPDWORD lpNumberOfBytesWritten,  
__inout_opt  LPOVERLAPPED lpOverlapped);
```

hFile – дескриптор файла, с которым производится операция чтения-записи;

lpBuffer – указатель на буфер, с которым производится операция чтения-записи;

nNumberOfBytesToWrite – число байт данных для чтения-записи;

lpNumberOfBytesWritten – указатель на буфер, в который записывается число действительно считанных или записанных байт данных;

lpOverlapped – указатель на структуру, используемую в процессе асинхронного ввода-вывода (для синхронного режима NULL).

При удачном завершении функция возвращает ненулевое значение в регистре EAX. В случае неудачи функция возвращает в регистре EAX значение NULL.

Пример программы: считать строку файла и продублировать его в тот же файл со следующей строки, дополнительно выводя на консоль.

```
.486  
.model flat, STDCALL  
; описания констант Windows  
STD_INPUT_HANDLE          equ -10  
STD_OUTPUT_HANDLE        equ -11  
STD_ERROR_HANDLE          equ -12  
  
FILE_SHARE_READ           equ 1h  
FILE_SHARE_WRITE          equ 2h  
FILE_ATTRIBUTE_READONLY   equ 1h  
FILE_ATTRIBUTE_HIDDEN     equ 2h  
FILE_ATTRIBUTE_SYSTEM     equ 4h  
FILE_ATTRIBUTE_DIRECTORY  equ 10h  
FILE_ATTRIBUTE_ARCHIVE    equ 20h  
FILE_ATTRIBUTE_NORMAL     equ 80h  
FILE_ATTRIBUTE_TEMPORARY  equ 100h  
FILE_ATTRIBUTE_COMPRESSED equ 80  
  
FILE_BEGIN                equ 0  
FILE_CURRENT              equ 1  
FILE_END                  equ 2  
  
CREATE_NEW                equ 1  
CREATE_ALWAYS             equ 2  
OPEN_EXISTING             equ 3  
OPEN_ALWAYS              equ 4  
TRUNCATE_EXISTING        equ 5  
  
GENERIC_READ              equ 80000000h  
GENERIC_WRITE            equ 40000000h  
GENERIC_EXECUTE          equ 20000000h
```

GENERIC_ALL

equ 10000000h

include C:\masm32\include\kernel32.inc

.data

file db "myfile.txt",0

hFile dd 0

TitleText db 'Работа с файлами',0

dOut dd 0 ;дескриптор вывода консоли

NumWri dd 0 ;действительное количество символов

buf db 300 dup(?)

n db 13,10 ; перевод строки

.code

start proc

INVOKE GetStdHandle, STD_OUTPUT_HANDLE

mov dOut,eax ;dOut-дескриптор вывода консоли

INVOKE CreateFile, ;открываем файл

offset file,

GENERIC_READ or GENERIC_WRITE,

0,

0, ;защита файла не требуется

OPEN_ALWAYS,

0, ;атрибуты

0

mov hFile,eax ;дескриптор файла

INVOKE ReadFile, ; чтение строки из файла

hFile,

offset buf,

300,

offset NumWri, 0

INVOKE WriteConsole, ; Вывод в консоль

dOut,

offset buf,

NumWri,

offset NumWri, 0

push NumWri ; сохраняем в стек количество считанных байт

INVOKE WriteFile, ; перевод строки

hFile,

offset n,

2,

offset NumWri, 0

pop NumWri

INVOKE WriteFile, ; дублирование в файл строки

hFile,

offset buf,

NumWri,

offset NumWri, 0

INVOKE CloseHandle, hFile ; закрытие файла

INVOKE ExitProcess, 0

start endp

end start

Вывод чисел в консоль

Сложность вывода чисел состоит в том, что для восприятия человеком число в консоли должно состоять из последовательности символов цифр, а в реальности число – это некоторый эквивалент такой последовательности, представленный с разрядной сетке вычислительной машины. Именно с таким числом производятся вычисления, но для отображения результатов вычислений необходимо перевести число в символьную строку. ASCII-код символа ‘0’ равен 30h. Коды остальных символов цифр располагаются по возрастанию. Следовательно, чтобы получить ASCII-код цифры нужно к ней добавить ASCII-код ‘0’. Для целых чисел программа вывода была рассмотрена в примере «Подсчет количества символов в строке»:

```
IntToStr proc Number:DWORD, Str1:DWORD
    MOV EAX, Number
    MOV EDI, Str1
    MOV ECX, 0
    CMP EAX, 0
    JGE @I1
    MOV DL, '-'
    MOV [EDI], DL
    INC EDI
    NOT EAX
    INC EAX
@I1: MOV EBX, 10
    MOV EDX, 0
    IDIV EBX
    PUSH EDX
    INC ECX
    CMP EAX, 0
    JG @I1
@I2: POP EDX
    ADD DL, 30h
    MOV [EDI], DL
    INC EDI
    LOOP @I2
    MOV DL, 0
    MOV [EDI], DL
    INC EDI
    MOV EAX, EDI
    SUB EAX, Str1
    ret
IntToStr endp
```

В результате получим в строке Str1 число (положительное или отрицательное) в символьном представлении, оканчивающееся ‘\0’-символом.

Для вывода вещественного числа вначале выведем функцией IntToStr его целую часть. После этого в выводимую строку поместим символ ‘.’ и вычтем целую часть

из исходного числа. Для преобразования к символам строки дробной части умножим ее на коэффициент требуемой точности 10^n и вновь сохраним целую часть полученного числа. Для преобразования в символьную строку снова вызовем функцию `IntToStr`, но указатель передадим не на начало строки, а на следующий символ строки (регистр `edi`).

```
.586
.model flat, stdcall
include console.inc
.data
consoleInHandle DD ?
consoleOutHandle DD ?
a DD -974.600007
str1 DB 20 dup(?)

Numi dd ? ; целая часть числа
tol dd 10000 ; точность
corr dd 0.00001 ; коррекция «лукавого знака»
cw dw ? ; управление округлением
.code
FloatToStr proc Num:DWORD, strr:DWORD
    fstcw cw
    or cw, 0C00h
    fldcw cw
    mov edi, strr
    mov ecx, 20
@F0: mov byte ptr [edi],0
    inc edi
    loop @F0
    mov edi, strr
    fld Num
    fadd corr
    fldz
    fcomip ST, ST(1)
    jb @F1
    mov byte ptr [edi], '-'
    inc edi
@F1: fsub corr
    fabs
    fadd corr
    fist Numi
    INVOKE IntToStr, Numi, edi
    dec edi
    mov bl, '.'
    mov [edi], bl
    inc edi
    fild Numi
    fsubrp
    fabs
    fimul tol
    fistp Numi
    push tol
```



```

    push corr
    mov corr, 10
@F2: mov edx, 0      ; определение количества нулей после точки
    mov eax, tol
    idiv corr
    mov tol, eax
    cmp Numi, eax
    jge @F3
    mov byte ptr [edi], '0'
    inc edi
    cmp tol, 1
    jg @F2
    cmp Numi, 0
    je @F4
@F3: INVOKE IntToStr, Numi, edi
@F4: pop corr
    pop tol
    ret
FloatToStr endp

start proc
    INVOKE GetStdHandle, STD_INPUT_HANDLE
    MOV consoleInHandle, EAX
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    MOV consoleOutHandle, EAX
    INVOKE FloatToStr, a, offset str1
    INVOKE PrintStr, offset str1, consoleOutHandle
    INVOKE ReadSymbol, consoleInHandle, consoleOutHandle, 1
    INVOKE ExitProcess, 0
start endp
end start

```

Оконные (каркасные) приложения Windows

Главным элементом программы в среде Windows является окно. Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы также являются окнами, но обладающими особым свойством: события, происходящие с этими элементами (и самим окном), приводят к приходу сообщений в процедуру окна. Для каждого окна определяется своя процедура обработки сообщений. Система Windows является многозадачной: каждая задача имеет свое адресное пространство и свою очередь сообщений. Любая процедура в среде Windows может быть запущена как самостоятельная задача.

В структуре каркасного приложения Windows можно выделить следующие разделы, образующие «скелет» программы:

- получение дескриптора приложения;
- регистрация класса окна;
- создание главного окна;
- цикл обработки очереди сообщений;
- процедура главного окна.

Получение дескриптора приложения осуществляется функцией

```
HMODULE WINAPI GetModuleHandle( __in_opt LPCTSTR lpModuleName );
```

`lpModuleName` – имя загруженного модуля (.dll или .exe файла), включая путь; если принимает значение 0, то функция возвращает дескриптор исполняемого .exe – файла.

Регистрация класса окна осуществляется с помощью функции

```
ATOM RegisterClass( CONST WNDCLASS *lpwcc );
```

единственным параметром которой является указатель на структуру WNDCLASS, содержащую информацию об окне. Под регистрацией класса окна понимается совокупность присущих ему характеристик, таких как стиль его границ, форм курсора, пиктограмм, цвета фона, наличия меню, адреса оконной процедуры, обрабатывающей сообщения этого окна.

```
WNDCLASS   struc
style      DD    ?           ; стиль границ окна и его поведение
                                ; при перерисовке
                                ;CS_VREDRAW      = 0001h
                                ;CS_HREDRAW      = 0002h
```

```

;CS_KEYCVTWINDOW      = 0004H
;CS_DBLCLKS           = 0008h
;CS_OWNSDC            = 0020h
;CS_CLASSDC          = 0040h
;CS_PARENTDC         = 0080h
;CS_NOKEYCVT         = 0100h
;CS_SAVEBITS         = 0800h
;CS_NOCLOSE          = 0200h
;CS_BYTEALIGNCLIENT = 1000h
;CS_BYTEALIGNWINDOW = 2000h
;CS_GLOBALCLASS      = 4000h

lpfnWndProc      WNDPROC  ?      ;адрес оконной процедуры
;Все окна, созданные на основе этого класса,
;будут обрабатывать посланные им сообщения
cbClsExtra DD      ?      ;количество дополнительно
cbWndExtra DD      ?      ;резервируемых байт, обычно 0
hInstance DD      ?      ;дескриптор приложения, полученный
;GetModuleHandleA
hIcon DD          ?      ;дескриптор значка LoadIcon
;IDI_APPLICATION = 32512
;IDI_HAND        = 32513
;IDI_QUESTION    = 32514
;IDI_EXCLAMATION = 32515
;IDI_ASTERISK    = 32516
hCursor DD        ?      ;дескриптор курсора LoadCursor
;IDC_ARROW       = 32512
;IDC_IBEAM       = 32513
;IDC_WAIT        = 32514
;IDC_CROSS       = 32515
;IDC_UPARROW    = 32516
;IDC_SIZE        = 32640
;IDC_ICON        = 32641
;IDC_SIZENWSE   = 32642
;IDC_SIZENESW   = 32643
;IDC_SIZEWE     = 32644
;IDC_SIZENS     = 32645
hbrBackground DD   ?      ;дескриптор кисти
;WHITE_BRUSH     =0
;LTGRAY_BRUSH   =1
;GRAY_BRUSH     =2
;DKGRAY_BRUSH   =3
;BLACK_BRUSH    =4
;NULL_BRUSH     =5
;HOLLOW_BRUSH   =5

```

Цвет фона окна можно также задать функцией `CreateSolidBrush`, аргумент которой – это константа, получаемая как комбинация трех цветов (RGB): красного, зеленого и синего. Цвет определяется одним 32-битным числом. В этом числе первый байт - интенсивность красного, второй байт - интенсивность зеленого, третий байт - интенсивность синего цвета. Последний байт равен нулю.

```

lpzMenuName DD 0 ;указатель на ASCIIZ строку
;с именем меню
lpzClassName DD ? ;присвоение данному классу
;уникального имени
ends

```

Создание окна осуществляется с помощью функции, позволяющей создать экземпляр окна.

```

HWND CreateWindowEx(
    DWORD dwExStyle,
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam);

```

dwExStyle – определяет дополнительные стили создаваемого окна:

```

WS_EX_ACCEPTFILES = 10h - поддержка перетаскивания файлов
WS_EX_APPWINDOW = 00040000h - добавление окна на панель задач
WS_EX_CLIENTEDGE = 00000200h - стиль границ окна
WS_EX_COMPOSITED - перерисовка снизу вверх (Windows XP)
WS_EX_CONTEXTHELP = 00000400h - значок «?» в строке заголовка окна
WS_EX_CONTROLPARENT = 00010000h - окно содержит дочернее окно, использующееся в
диалогах
WS_EX_DLGMODALFRAME = 1h - окно с двойной границей
WS_EX_LAYERED = 00080000h - «послойное» окно (Windows XP)
WS_EX_LAYOUTRTL = 00400000h - начало отсчета по горизонтали - справа
WS_EX_LEFT = 00000000h - выравнивание свойств по левому краю (по умолчанию)
WS_EX_LEFTSCROLLBAR = 00004000h - полоса прокрутки слева
WS_EX_LTRREADING = 00000000h - отображение текста слева направо (по умолчанию)
WS_EX_MDICHILD - создание дочернего окна в многооконном интерфейсе (MDI)
WS_EX_NOACTIVATE = 08000000h - созданное окно не активируется
WS_EX_NOINHERITLAYOUT = 00100000h - окно не передает свое положение дочерним окнам
WS_EX_NOPARENTNOTIFY = 4h - дочернее окно не передает сообщение WM_PARENTNOTIFY
родительскому окну при своем создании/закрытии.
WS_EX_OVERLAPPEDWINDOW = WS_EX_CLIENTEDGE + WS_EX_WINDOW.
WS_EX_PALETTEWINDOW = WS_EX_WINDOWEDGE + WS_EX_TOOLWINDOW + WS_EX_TOPMOST.
WS_EX_RIGHT = 00001000h - выравнивание свойств по правому краю
WS_EX_RIGHTSCROLLBAR = 00000000h - вертикальная полоса прокрутки в правой части
окна (по умолчанию)
WS_EXRTLREADING = 00002000h - размещение текста справа налево
WS_EX_STATICEDGE = 00020000h - окно с трехмерным стилем границ
WS_EX_TOOLWINDOW = 00000080h - окно для плавающей панели инструментов
WS_EX_TOPMOST = 8h - расположение окна перед всеми окнами.
WS_EX_TRANSPARENT = 20h - окно без перерисовки
WS_EX_WINDOWEDGE = 00000100h - «приподнятый» стиль границ окна

```

lpClassName — указатель на ASCIIZ-строку с именем класса окна.

Можно использовать predefined классы для создания элементов окна:

```

BUTTON – кнопка;
COMBOBOX – набор переключателей;
EDIT – поле редактирования;
LISTBOX – список;
MDICLIENT – MDI-клиент;
RICHEDIT_CLASS – объект Rich Edit 2.0;
SCROLLBAR – полоса прокрутки;
STATIC – надпись.

```

lpWindowName — указатель на ASCIIZ-строку с именем окна.

dwStyle определяет стиль окна приложения:

WS_OVERLAPPED	=00000000h
WS_ICONICPOPUP	=0C000000h
WS_POPUP	=08000000h
WS_CHILD	=04000000h
WS_MINIMIZE	=02000000h
WS_VISIBLE	=01000000h
WS_DISABLED	=00800000h
WS_CLIPSIBLINGS	=00400000h
WS_CLIPCHILDREN	=00200000h
WS_MAXIMIZE	=00100000h
WS_CAPTION	=000C0000h ;WS_BORDER WS_DLGFRAME
WS_BORDER	=00080000h
WS_DLGFRAME	=00040000h
WS_VSCROLL	=00020000h
WS_HSCROLL	=00010000h
WS_SYSMENU	=00008000h
WS_THICKFRAME	=00004000h
WS_HREDRAW	=00002000h
WS_VREDRAW	=00001000h
WS_GROUP	=00002000h
WS_TABSTOP	=00001000h
WS_MINIMIZEBOX	=00002000h
WS_MAXIMIZEBOX	=00001000h

x — координата левого верхнего угла окна

y — координата левого верхнего угла окна

nWidth — ширина окна (по умолчанию CW_USEDEFAULT (80000000h))

nHeight — высота окна (по умолчанию CW_USEDEFAULT (80000000h))

hWndParent — дескриптор родительского окна. Между двумя окнами Windows-приложения можно устанавливать родовидовые отношения. Дочернее окно всегда должно появляться в области родительского окна.

hMenu — дескриптор главного меню окна

hInstance — дескриптор приложения, создающего окно.

lpParam — используется при создании окна для передачи данных или указателя на них в оконную функцию. Все параметры, передаваемые функции CreateWindowEx, сохраняются в создаваемой Windows внутренней структуре CREATESTRUCT. Поля этой структуры идентичны параметрам функции CreateWindowEx. Указатель на структуру CREATESTRUCT передается оконной функции при обработке сообщения WMCREATE. Сам указатель находится в поле lpParam сообщения. Значение параметра lpParam функции CreateWindowEx находится в поле lpCreateParams структуры CREATESTRUCT.

Функция CreateWindowEx возвращает дескриптор окна.

Для отображения окна используется функция

```
BOOL ShowWindow( HWND hWnd, int nCmdShow);
```

hWnd – дескриптор окна.

nCmdShow – константа, задающая начальный вид окна.

```
SW_HIDE           =0
SW_SHOWNORMAL     =1
SW_NORMAL         =1
SW_SHOWMINIMIZED  =2
SW_SHOWMAXIMIZED  =3
SW_MAXIMIZE       =3
SW_SHOWNOACTIVATE =4
SW_SHOW           =5
SW_MINIMIZE       =6
SW_SHOWMINNOACTIVE =7
SW_SHOWNA        =8
SW_RESTORE        =9
```

Цикл обработки сообщений. После создания данного цикла приложение становится пассивным и начинает взаимодействовать с внешним миром посредством специальным образом оформленных данных — сообщений. Для получения сообщений используется функция GetMessage, выполняющая следующие действия:

- постоянно просматривать очередь сообщений;
- выбирать сообщения, удовлетворяющие заданным в функции параметрам;
- заносит информацию о событии в экземпляр структуры MSG;
- передает управление в цикл обработки сообщений.

```
BOOL GetMessage( LPMSG lpMsg,
                HWND hWnd,
                UINT wMsgFilterMin,
                UINT wMsgFilterMax);
```

lpMsg – указатель на экземпляр структуры MSG. Во время работы GetMessageA извлекает сообщение из очереди сообщений приложения и на основе информации в нем инициализирует поля экземпляра структуры MSG. Таким образом, приложение получает полный доступ ко всей информации в сообщении, сформированном Windows.

```
MSG struc
hwnd      DD    ? ;дескриптор окна, которому адресовано сообщение
            ;возвращается функцией CreateWindowEx
mes       DD    ? ;идентификатор типа сообщения
            ;все константы начинаются с WM_
wParam    DD    ? ;дополнительная информация о сообщении
lParam    DD    ?
```

```

time      DD    ?    ; время помещения сообщения в очередь сообщений
pt        POINT <> ; координаты курсора мыши в момент помещения
MSG ends                                     ; сообщения в очередь

POINT     struc
x         DD    0
y         DD    0
POINT ends

```

hWnd — в поле передается дескриптор окна, сообщения для которого должны будут выбираться данной функцией. Параметр, позволяет создать своеобразный фильтр, заставляющий функцию GetMessage выбирать из очереди и передавать в цикл обработки сообщений сообщения лишь для определенного окна данного приложения. Если hWnd=0, то GetMessage будет выбирать из очереди сообщения для всех окон;

wMsgFilterMin и wMsgFilterMax — значения данных параметров также позволяют создавать фильтр для выбираемых функцией GetMessageA сообщений. Они задают диапазон номеров сообщений (поле mes структуры MSG), которые будут выбираться из очереди функцией GetMessage и передаваться в цикл обработки сообщений.

Цикл обработки сообщений состоит из двух функций:

```

BOOL TranslateMessage( const MSG *lpMsg);
LRESULT DispatchMessage( const MSG *lpmsg);

```

Эти функции имеют единственный параметр — указатель на экземпляр структуры MSG, предварительно заполненный информацией о сообщении функцией GetMessage.

Функция TranslateMessage предназначена для обнаружения сообщений от клавиатуры для данного приложения. Если приложение не будет самостоятельно обрабатывать ввод с клавиатуры, то эти сообщения передаются для обработки обратно Windows.

Функция DispatchMessage предназначена для передачи сообщения оконной функции. Такая передача производится не напрямую, так как сама DispatchMessage ничего не знает о месторасположении оконной функции, а косвенно — посредством системы Windows. При этом:

- функция DispatchMessage возвращает сообщение операционной системе;

- Windows, используя описание класса окна, передает сообщение нужной оконной функции приложения;
- после обработки сообщения оконной функцией управление возвращается операционной системе;
- Windows передает управление функции DispatchMessage;
- DispatchMessage завершает свое выполнение.

Так как вызов функции DispatchMessage является последним в цикле, то управление опять передается функции GetMessage. GetMessage выбирает следующее сообщение из очереди сообщений и, если оно удовлетворяет параметрам, заданным при вызове этой функции, то выполняется тело цикла. Цикл обработки сообщений выполняется до тех пор, пока не приходит сообщение, равное 0. Получение этого сообщения — единственное условие, при котором программа может выйти из цикла обработки сообщений.

Таким образом, главная функция будет иметь следующий вид:

```
.586
.MODEL FLAT, stdcall
include win.inc          ; файл с константами и структурами

; сегмент данных
.data
    HWND          DD 0          ; дескриптор главного окна
    Message       MSG <?>
    WC            WNDCLASS <?>
    HINST         DD 0          ; дескриптор приложения
    TITLENAME     DB 'Программа',0
    CLASSNAME     DB 'CLASS32',0
    ...

; сегмент кода
.code
START proc
; получить дескриптор приложения
    INVOKE GetModuleHandle, 0
    MOV     HINST, EAX
REG_CLASS:
; заполнить структуру окна стиль
    MOV WC.style, CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
; процедура обработки сообщений
    MOV WC.lpfWndProc, OFFSET WNDPROC
    MOV WC.cbClsExtra, 0
    MOV WC.cbWndExtra, 0
    MOV EAX, HINST
    MOV WC.hInstance, EAX
;----- иконка окна
```



```

        INVOKE LoadIcon, 0, IDI_APPLICATION
        MOV  WC.hIcon, EAX
;----- курсор окна
        INVOKE LoadCursor, 0, IDC_ARROW
        MOV  WC.hCursor, EAX
;-----
        MOV  WC.hbrBackground, GRAY_BRUSH ; цвет окна
        MOV  DWORD PTR WC.lpszMenuName, 0
        MOV  DWORD PTR WC.lpszClassName, OFFSET CLASSNAME
        INVOKE RegisterClass, OFFSET WC
; создать окно зарегистрированного класса
        INVOKE CreateWindowEx,
            0,
            OFFSET CLASSNAME,
            OFFSET TITLENAM,
            WS_CAPTION + WS_SYSMENU + WS_THICKFRAME + WS_GROUP + \
            WS_TABSTOP,
            100, ; X – координата левого верхнего угла
            100, ; Y – координата левого верхнего угла
            400, ; DX – ширина окна
            450, ; DY – высота окна
            0, 0, HINST, 0
; проверка на ошибку
        CMP  EAX, 0
        JZ   END_LOOP
        MOV  HWND, EAX ; дескриптор окна
; -----
        INVOKE ShowWindow, HWND, SW_SHOWNORMAL ; показать созданное окно
        INVOKE UpdateWindow, HWND ;перерисовать видимую часть окна

; цикл обработки сообщений
MSG_LOOP:
        INVOKE GetMessage,
            OFFSET Message, 0,0,0
        CMP  EAX, 0
        JE   END_LOOP
        INVOKE TranslateMessage, OFFSET Message
        INVOKE DispatchMessageA, OFFSET Message
        JMP  MSG_LOOP
END_LOOP:
; выход из программы (закрыть процесс)
        INVOKE ExitProcess, Message.wParam
START  endp
...

```

Для организации адекватной реакции со стороны Windows-приложения на действия пользователя и поддержания в актуальном состоянии того окна приложения, сообщения которого обрабатываются, используется *оконная функция*. Приложение может иметь несколько оконных функций. Их количество определяется количеством классов окон, зарегистрированных в системе функцией RegisterClass.

Когда для окна Windows-приложения появляется сообщение, операционная система Windows производит вызов соответствующей оконной функции. Сообщения, в зависимости от источника их появления в оконной функции, могут быть двух типов: синхронные и асинхронные. К *синхронным* сообщениям относятся те сообщения, которые помещаются в очередь сообщений приложения и терпеливо ждут момента, когда они будут выбраны функцией GetMessage. После этого поступившие сообщения попадают в оконную функцию, где и производится их обработка. *Асинхронные* сообщения попадают в оконную функцию в экстренном порядке, минуя при этом все очереди. Асинхронные сообщения, в частности, инициируются некоторыми функциями Win32 API, такими как CreateWindowEx или UpdateWindow. Координацию синхронных и асинхронных сообщений осуществляет Windows. Если рассматривать синхронное сообщение, то его извлечение производится функцией GetMessage с последующей передачей его обратно в Windows функцией DispatchMessage. Асинхронное сообщение, независимо от источника, который инициирует его появление, сначала попадает в Windows и затем — в нужную оконную функцию.

В схеме, реализованной в Windows, обработка сообщений приложением проводится в два этапа: на первом этапе приложение выбирает сообщение из очереди и отправляет его обратно во внутренние структуры Windows; на втором этапе Windows вызывает нужную оконную функцию приложения, передавая ей параметры сообщения. Преимущество этой схемы в том, что Windows самостоятельно решает все вопросы организации эффективной работы приложений.

Таким образом, при поступлении сообщения Windows вызывает оконную функцию и передает ей ряд параметров. Все они берутся из соответствующих полей сообщения.

Оконная функция имеет следующие параметры:

1. `hWnd` — дескриптор окна, которому предназначено сообщение;
2. `message` — идентификатор сообщения, характеризующий тип сообщения;
3. `wParam` и `lParam` — дополнительные параметры, являющиеся копиями соответствующих полей структуры поступившего сообщения.

Оставшиеся два поля структуры MSG: time и POINT используются достаточно редко, и при необходимости их значения можно извлечь непосредственно из экземпляра структуры сообщения.

Параметры оконной функции Windows помещает в стек.

Основные типы обрабатываемых идентификаторов сообщений

WM_CREATE	equ 1h	; создание окна
WM_DESTROY	equ 2h	; закрытие окна
WM_PAINT	equ 0Fh	; перерисовка окна
WM_COMMAND	equ 111h	; обработка команды Windows
WM_LBUTTONDOWN	equ 201h	; нажатие левой кнопки мыши
WM_LBUTTONUP	equ 202h	; отжатие левой кнопки мыши
WM_LBUTTONDOWNBLCLK	equ 203h	; двойное нажатие левой кнопки мыши
WM_RBUTTONDOWN	equ 204h	; нажатие правой кнопки мыши
WM_RBUTTONUP	equ 205h	; отжатие правой кнопки мыши
WM_RBUTTONDOWNBLCLK	equ 206h	; двойное нажатие правой кнопки мыши
WM_MBUTTONDOWN	equ 207h	; нажатие средней кнопки мыши
WM_MBUTTONUP	equ 208h	; отжатие средней кнопки мыши
WM_MBUTTONDOWNBLCLK	equ 209h	; двойное нажатие средней кнопки мыши

Структура процедуры окна строится по принципу switch-case, выбирая из списка тип события, которое необходимо обработать.

При перерисовке окна системой посылается сообщение WM_PAINT, именно при получении этого сообщения следует перерисовывать содержимое окна. Это сообщение генерируется системой при разворачивании окна, изменении его размеров, наложении на данное окно других окон с последующим их закрытием. Для того, чтобы осуществить вывод информации в окно, необходимо сначала получить контекст окна (контекст устройства). Это — некоторое число, посредством которого осуществляется связь между приложением и окном. Обычно контекст устройства определяется посредством функции

```
HDC GetDC( HWND hWnd );
```

 передается дескриптор окна

При получении сообщения WM_PAINT контекст устройства получается посредством функции

```
HDC BeginPaint(  
    HWND hwnd, // дескриптор окна  
    LPPAINTSTRUCT lpPaint); // указатель на структуру PAINTSTRUCT  
PAINTSTRUCT STRUCT  
    hdc    DWORD 0 ; дескриптор контекста окна  
    fErase DWORD 0 ; 1 - стирать фон, 0 - оставить фон окна  
    left   DWORD 0 ; левый  
    top    DWORD 0 ; верхний  
    right  DWORD 0 ; правый  
    bottom DWORD 0 ; нижний угол прямоугольника для перерисовки
```

```

    fRes    DWORD 0 ; резервный, используется системой
    fIncUp  DWORD 0 ; резервный, используется системой
    Reserv  DB 32 dup(0) ; резервный, используется системой
PAINTSTRUCT ENDS

```

Для вывода текста в окно используется функция

```

BOOL TextOut(
    HDC hdc,           // дескриптор контекста окна
    int nXStart,      // координата X начальной позиции
    int nYStart,      // координата Y начальной позиции
    LPCTSTR lpString, // указатель на строку символов
    int cbString);    // количество байт для вывода

```

Предварительно задать цвет фона и букв можно посредством функций

```

COLORREF SetBkColor(
    HDC hdc,           // дескриптор контекста окна
    COLORREF crColor); // цвет фона RGB
COLORREF SetTextColor(
    HDC hdc,           // дескриптор контекста окна
    COLORREF crColor); // цвет букв RGB

```

Центр системы координат для окна находится в левом верхнем углу, ось Y направлена вниз, ось X - вправо. Это общепринятый вариант для графических экранов.

Для окончания перерисовки окна используется функция

```

BOOL EndPaint(
    HWND hWnd,           // дескриптор окна
    CONST PAINTSTRUCT *lpPaint // указатель на структуру PAINTSTRUCT
); // полученную функцией BeginPaint

```

Для завершения приложения используется сообщение WM_DESTROY, которое обрабатывается вызовом функции

```
void PostQuitMessage( int nExitCode);
```

Аргументом функции является код завершения приложения.

Для обработки нажатия на кнопку как элемент управления окна используется сообщение WM_COMMAND. При его получении программа сравнивает параметр lParam функции окна последовательно с дескрипторами кнопок и анализирует, с какой кнопкой произведено действие.

Если обработчик для данного типа сообщения не требуется, вызывается обработчик сообщения по умолчанию в виде функции

```

LRESULT DefWindowProc(
    HWND hWnd,           // дескриптор окна
    UINT Msg,           // тип сообщения
    WPARAM wParam,      // параметр сообщения
    LPARAM lParam);     // параметр сообщения

```

Пример оконной функции, выводящей в окно строку «Привет Всем».

```
.data
...
hBut          DD ? ; дескриптор кнопки
hdc           DD ? ; дескриптор контекста окна
ps            PAINTSTRUCT <?>
mess          db 'Привет всем!',0 ; надпись в окне
mess_len      equ $-mess-1

.code
...
; -----
; процедура окна
WNDPROC PROC hW:DWORD, Mes:DWORD, wParam:DWORD, lParam:DWORD
    CMP Mes, WM_DESTROY
    JE WMDESTROY
    CMP Mes, WM_CREATE
    JE WMCREATE
    CMP Mes, WM_PAINT
    JE WMPAINT
    JMP DEFWNDPROC
; создание окна
WMCREATE:
    MOV EAX, 0
    JMP FINISH
; перерисовка окна
WMPAINT:
    INVOKE BeginPaint, hW, offset ps
    mov hdc,eax
    INVOKE SetBkColor, hdc, RGBW

    mov eax, mess_len
    INVOKE TextOutA,
        hdc,
        10, 20, ;x, y
        offset mess,
        eax

    INVOKE EndPaint, hdc, offset ps
    MOV EAX, 0
    JMP FINISH
; обработка сообщения по умолчанию
DEFWNDPROC:
    INVOKE DefWindowProc,
        hW,
        Mes,
        wParam,
        lParam
    JMP FINISH
WMDESTROY:
    INVOKE PostQuitMessage, 0
    MOV EAX, 0
FINISH: ret
WNDPROC ENDP
END START
```

Пример программы, вычисляющей сумму двух чисел.

```
; файл win.inc
; директивы компилятору для подключения прототипов функций
include c:\masm32\include\kernel32.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\gdi32.inc
include c:\masm32\include\shlwapi.inc

; стандартные дескрипторы окна
IDI_APPLICATION = 32512
IDI_HAND        = 32513
IDI_QUESTION    = 32514
IDI_EXCLAMATION = 32515
IDI_ASTERISK    = 32516

; стандартные дескрипторы курсора
IDC_ARROW       = 32512
IDC_IBEAM       = 32513
IDC_WAIT        = 32514
IDC_CROSS       = 32515
IDC_UPARROW    = 32516
IDC_SIZE        = 32640
IDC_ICON        = 32641
IDC_SIZENWSE   = 32642
IDC_SIZENESW   = 32643
IDC_SIZEWE     = 32644
IDC_SIZENS     = 32645

; свойства окна
CS_VREDRAW      = 0001h
CS_HREDRAW      = 0002h
CS_KEYCVTWINDOW = 0004h
CS_DBLCLKS     = 0008h
CS_OWNDC       = 0020h
CS_CLASSDC     = 0040h
CS_PARENTDC    = 0080h
CS_NOKEYCVT    = 0100h
CS_SAVEBITS    = 0800h
CS_NOCLOSE     = 0200h
CS_BYTEALIGNCLIENT = 1000h
CS_BYTEALIGNWINDOW = 2000h
CS_GLOBALCLASS = 4000h

; цвет кисти
WHITE_BRUSH      = 0      ; прозрачное окно
LTGRAY_BRUSH    = 1
GRAY_BRUSH       = 2
DKGRAY_BRUSH    = 3
BLACK_BRUSH     = 4
NULL_BRUSH      = 5      ; привычный цвет окна
HOLLOW_BRUSH    = 5

; структура
WNDCLASS STRUCT
```

```

style          DD      ?
lpfnWndProc    DD      ?
cbClsExtra     DD      0
cbWndExtra     DD      0
hInstance     DD      ?
hIcon          DD      ?
hCursor        DD      ?
hbrBackground  DD      ?
lpszMenuName   DD      0
lpszClassName  DD      ?
WNDCLASS ENDS

```

```

; тип окна

```

```

WS_OVERLAPPED =00000000h
WS_ICONICPOPUP =0C000000h
WS_POPUP      =08000000h
WS_CHILD      =04000000h
WS_MINIMIZE   =02000000h
WS_VISIBLE    =01000000h
WS_DISABLED   =00800000h
WS_CLIPSIBLINGS =00400000h
WS_CLIPCHILDREN =00200000h
WS_MAXIMIZE   =00100000h
WS_CAPTION    =000C0000h
WS_BORDER     =00080000h
WS_DLGFRAME   =00040000h
WS_VSCROLL    =00020000h
WS_HSCROLL    =00010000h
WS_SYSMENU    =00008000h
WS_THICKFRAME =00004000h
WS_HREDRAW    =00002000h
WS_VREDRAW    =00001000h
WS_GROUP      =00002000h
WS_TABSTOP    =00001000h
WS_MINIMIZEBOX =00002000h
WS_MAXIMIZEBOX =00001000h

```

```

;WS_BORDER | WS_DLGFRAME

```

```

; отображение окна

```

```

SW_HIDE       =0
SW_SHOWNORMAL =1
SW_NORMAL     =1
SW_SHOWMINIMIZED =2
SW_SHOWMAXIMIZED =3
SW_MAXIMIZE   =3
SW_SHOWNOACTIVATE =4
SW_SHOW       =5
SW_MINIMIZE   =6
SW_SHOWMINNOACTIVE =7
SW_SHOWNA     =8
SW_RESTORE    =9

```

```

; Типы сообщений

```

```

WM_NULL       equ 0h
WM_CREATE     equ 1h
WM_DESTROY    equ 2h

```

```

WM_MOVE equ 3h
WM_SIZE equ 5h
WM_ACTIVATE equ 6h
WM_SETFOCUS equ 7h
WM_KILLFOCUS equ 08h
WM_ENABLE equ 0Ah
WM_SETREDRAW equ 0Bh
WM_SETTEXT equ 0Ch
WM_GETTEXT equ 0Dh
WM_GETTEXTLENGTH equ 0Eh
WM_PAINT equ 0Fh
WM_CLOSE equ 10h
WM_QUIT equ 12h
WM_KEYFIRST equ 100h
WM_KEYDOWN equ 100h
WM_KEYUP equ 101h
WM_CHAR equ 102h
WM_INITDIALOG equ 110h
WM_COMMAND equ 111h
WM_SYSCOMMAND equ 112h
WM_TIMER equ 113h
WM_HSCROLL equ 114h
WM_VSCROLL equ 115h
WM_MOUSEMOVE equ 200h
WM_LBUTTONDOWN equ 201h
WM_LBUTTONUP equ 202h
WM_LBUTTONDOWNBLCLK equ 203h
WM_RBUTTONDOWN equ 204h
WM_RBUTTONUP equ 205h
WM_RBUTTONDOWNBLCLK equ 206h
WM_MBUTTONDOWN equ 207h
WM_MBUTTONUP equ 208h
WM_MBUTTONDOWNBLCLK equ 209h
WM_WINDOWPOSCHANGING equ 46h
WM_WINDOWPOSCHANGED equ 47h
WM_SIZING equ 214h

```

; структура сообщения

```
POINT STRUCT
```

```
    x DD ?
```

```
    y DD ?
```

```
POINT ENDS
```

```
MSG STRUCT
```

```
    hwnd DD ?
```

```
    message DD ?
```

```
    wParam DD ?
```

```
    lParam DD ?
```

```
    time DD ?
```

```
    pt POINT <>
```

```
MSG ENDS
```

```
PAINTSTRUCT STRUCT
```

```
    hdc DWORD 0 ; дескриптор контекста окна
```



```

    fErase DWORD 0 ; 1 - стирать фон, 0 - оставить фон окна
    left   DWORD 0 ; левый
    top    DWORD 0 ; верхний
    right  DWORD 0 ; правый
    bottom DWORD 0 ; нижний угол прямоугольника для перерисовки
    fRes   DWORD 0 ; резервный, используется системой
    fIncUp DWORD 0 ; резервный, используется системой
    Reserv DB 32 dup(0) ; резервный, используется системой
PAINTSTRUCT ENDS

```

```

RECT STRUCT
    left   dd    0
    top    dd    0
    right  dd    0
    bottom dd    0
RECT ENDS

```

```

SRCCOPY          equ 0CC0020h
SRCPAINT         equ 0EE0086h
SRCAND           equ 8800C6h
SRCINVERT        equ 660046h
SRCERASE         equ 440328h

```

```

PATCOPY          equ 0F00021h
PATPAINT         equ 0FB0A09h
PATINVERT        equ 5A0049h
DSTINVERT        equ 550009h
BLACKNESS        equ 42h
WHITENESS        equ 0FF0062h

```

```

PS_SOLID         equ 0
PS_DASH          equ 1
PS_DOT           equ 2
PS_DASHDOT       equ 3
PS_DASHDOTDOT    equ 4
PS_NULL          equ 5
PS_INSIDEFRAME   equ 6
PS_USERSTYLE     equ 7
PS_ALTERNATE     equ 8
PS_STYLE_MASK    equ 0Fh
PS_ENDCAP_ROUND  equ 0h
PS_ENDCAP_SQUARE equ 100h
PS_ENDCAP_FLAT   equ 200h
PS_ENDCAP_MASK   equ 0F00h
PS_JOIN_ROUND    equ 0h
PS_JOIN_BEVEL    equ 1000h
PS_JOIN_MITER    equ 2000h
PS_JOIN_MASK     equ 0F000h
PS_COSMETIC      equ 0h
PS_GEOMETRIC     equ 10000h
PS_TYPE_MASK     equ 0F0000h

```

```

SM_CXSCREEN      equ 0
SM_CYSCREEN      equ 1

```

SM_CXVSCROLL	equ 2
SM_CYHSCROLL	equ 3
SM_CYCAPTION	equ 4
SM_CXBORDER	equ 5
SM_CYBORDER	equ 6
SM_CXDLGFRAME	equ 7
SM_CYDLGFRAME	equ 8
SM_CYVTHUMB	equ 9
SM_CXHTHUMB	equ 10
SM_CXICON	equ 11
SM_CYICON	equ 12
SM_CXCURSOR	equ 13
SM_CYCURSOR	equ 14
SM_CYMENU	equ 15
SM_CXFULLSCREEN	equ 16
SM_CYFULLSCREEN	equ 17
SM_CYKANJIWINDOW	equ 18
SM_MOUSEPRESENT	equ 19
SM_CYVSCROLL	equ 20
SM_CXHSCROLL	equ 21
SM_DEBUG	equ 22
SM_SWAPBUTTON	equ 23
SM_RESERVED1	equ 24
SM_RESERVED2	equ 25
SM_RESERVED3	equ 26
SM_RESERVED4	equ 27
SM_CXMIN	equ 28
SM_CYMIN	equ 29
SM_CXSIZE	equ 30
SM_CYSIZE	equ 31
SM_CXFRAME	equ 32
SM_CYFRAME	equ 33
SM_CXMINTRACK	equ 34
SM_CYMINTRACK	equ 35
SM_CXDOUBLECLK	equ 36
SM_CYDOUBLECLK	equ 37
SM_CXICONSPACING	equ 38
SM_CYICONSPACING	equ 39
SM_MENUDROPALIGNMENT	equ 40
SM_PENWINDOWS	equ 41
SM_DBCSENABLED	equ 42
SM_CMOUSEBUTTONS	equ 43
SM_CXFIXEDFRAME	equ SM_CXDLGFRAME
SM_CYFIXEDFRAME	equ SM_CYDLGFRAME
SM_CXSIZEFRAME	equ SM_CXFRAME
SM_CYSIZEFRAME	equ SM_CYFRAME
SM_SECURE	equ 44
SM_CXEDGE	equ 45
SM_CYEDGE	equ 46
SM_CXMINSPACING	equ 47
SM_CYMINSPACING	equ 48
SM_CXSMICON	equ 49
SM_CYSMICON	equ 50
SM_CYSMCAPTION	equ 51
SM_CXSMSIZE	equ 52

```

SM_CYSMSIZE          equ 53
SM_CXMENUSIZE       equ 54
SM_CYMENUSIZE       equ 55
SM_ARRANGE          equ 56
SM_CXMINIMIZED      equ 57
SM_CYMINIMIZED      equ 58
SM_CXMAXTRACK       equ 59
SM_CYMAXTRACK       equ 60
SM_CXMAXIMIZED      equ 61
SM_CYMAXIMIZED      equ 62
SM_NETWORK          equ 63
SM_CLEANBOOT        equ 67
SM_CXDRAG           equ 68
SM_CYDRAG           equ 69
SM_SHOWSOUNDS       equ 70
SM_CXMENUCHECK      equ 71
SM_CYMENUCHECK      equ 72
SM_SLOWMACHINE       equ 73
SM_MIDEASTENABLED   equ 74
SM_MOUSEWHEELPRESENT equ 75
SM_CMETRICS         equ 75
SM_XVIRTUALSCREEN   equ 76
SM_YVIRTUALSCREEN   equ 77
SM_CXVIRTUALSCREEN   equ 78
SM_CYVIRTUALSCREEN   equ 79
SM_CMONITORS        equ 80

; файл с текстом программы
.586
.MODEL FLAT, stdcall
RGBW          equ 00CCCCCh ; цвет фона окна
include win.inc
include console.inc1

.data
    Hwnd          DD 0 ; дескриптор главного окна
    HINST         DD 0 ; дескриптор приложения
    TITL          DB "Программа",0
    CLASSNAME     DB 'CLASS32',0
    Message       MSG <?>
    WC            WNDCLASS <?>

    CPBUT         db 'Рассчитать',0
    CLSBTN        db 'BUTTON',0
    CLSEDT        db 'EDIT',0
    CAP           db 'Сообщение',0
    TEXTA         db 20 dup(0) ; текст в полях редактирования
    TEXTB         db 20 dup(0)
    summa         dd 0

    hBut          DD ? ; дескриптор кнопки
    hedt1         DD ? ; дескриптор поля 1
    hedt2         DD ? ; дескриптор поля 2

```

¹ Текст файл console.inc был приведен ранее, в главе «Консольные приложения Windows»

```

hdc      DD ? ; дескриптор контекста окна
ps       PAINTSTRUCT    <?>
mess1    db    'Посчитать сумму двух чисел: ',0    ; надпись в окне
mess1_len equ    $-mess1-1
mess2    db    '=',10 dup(' '),0    ; результат суммы строковый
sum_len  equ    $-mess2-1

.code
START proc
; получить дескриптор приложения
    INVOKE GetModuleHandle, 0
    MOV     HINST, EAX
; заполнить структуру окна стиль
    MOV WC.style, CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
; процедура обработки сообщений
    MOV WC.lpfWndProc, OFFSET WNDPROC
    MOV EAX, HINST
    MOV WC.hInstance, EAX
    INVOKE LoadIcon, 0, IDI_APPLICATION
    MOV WC.hIcon, EAX
    INVOKE LoadCursor, 0, IDC_ARROW
    MOV WC.hCursor, EAX
    INVOKE CreateSolidBrush, RGBW
    MOV WC.hbrBackground, EAX
    MOV DWORD PTR WC.lpszMenuName, 0
    MOV DWORD PTR WC.lpszClassName, OFFSET CLASSNAME
    INVOKE RegisterClass, OFFSET WC
; создать окно зарегистрированного класса
    INVOKE CreateWindowEx, 0,
        OFFSET CLASSNAME,
        OFFSET TITL,
        WS_CAPTION + WS_SYSMENU + WS_THICKFRAME + WS_GROUP + WS_TABSTOP,
        100, ; X – координата левого верхнего угла
        100, ; Y – координата левого верхнего угла
        400, ; DX – ширина окна
        450, ; DY – высота окна
        0, 0, HINST,0
    CMP EAX, 0 ; проверка на ошибку
    JZ END_LOOP
    MOV HWND, EAX ; дескриптор окна

    INVOKE ShowWindow, HWND, SW_SHOWNORMAL ; показать созданное окно
    INVOKE UpdateWindow, HWND ;перерисовать видимую часть окна
;-----
; цикл обработки сообщений
MSG_LOOP:
    INVOKE GetMessage, OFFSET Message, 0,0,0
    CMP EAX, 0
    JE END_LOOP
    INVOKE TranslateMessage, OFFSET Message
    INVOKE DispatchMessageA, OFFSET Message
    JMP MSG_LOOP
END_LOOP:
    INVOKE ExitProcess, Message.wParam ; выход из программы
START endp

```

```

; -----
; процедура окна
WNDPROC PROC hW:DWORD, Mes:DWORD, wParam:DWORD, lParam:DWORD
    CMP Mes, WM_DESTROY
    JE WMDESTROY
    CMP Mes, WM_CREATE
    JE WMCREATE
    CMP Mes, WM_COMMAND
    JE WMCOMMAND
    CMP Mes, WM_PAINT
    JE WMPAINT
    JMP DEFWNDPROC
; -----
WMCREATE:
    INVOKE CreateWindowExA, 0, ; создание окна
        offset CLSEDT, ; поле редактирования 1
        offset TEXTA, ; имя класса окна
        WS_CHILD+WS_VISIBLE, ; надпись в поле
        10, 50, ; стиль окна
        60, 20, ; x, y
        hW, ; длина, ширина
        0, ; дескриптор окна
        HINST, ; дескриптор меню
        0 ; дескриптор приложения
    mov hedt1,eax ; сохранение дескриптора
    mov eax,0
    INVOKE ShowWindow, hedt1, SW_SHOWNORMAL
    INVOKE CreateWindowExA, 0, ; поле редактирования 2
        offset CLSEDT, ; имя класса окна
        offset TEXTB, ; надпись в поле
        WS_CHILD+WS_VISIBLE, ; стиль окна
        90, 50, ; x, y
        60, 20, ; длина, ширина
        hW, ; дескриптор окна
        0, ; дескриптор меню
        HINST, ; дескриптор приложения
        0 ; дескриптор приложения
    mov hedt2,eax ; сохранение дескриптора
    mov eax,0
    INVOKE ShowWindow, hedt2, SW_SHOWNORMAL
    INVOKE CreateWindowExA, 0, ; кнопка
        offset CLSBTN, ; имя класса окна
        offset CPBUT, ; надпись на кнопке
        WS_CHILD+WS_VISIBLE, ; стиль окна кнопки
        10, 90, ; x, y
        100, 20, ; длина, ширина
        hW, ; дескриптор окна
        0, ; дескриптор меню
        HINST, ; дескриптор приложения
        0 ; дескриптор приложения
    mov hBut,eax ; сохранение дескриптора
    mov eax,0
    INVOKE ShowWindow, hBut, SW_SHOWNORMAL
    MOV EAX, 0
    JMP FINISH

```

```

;-----
WMCOMMAND:                ; обработка нажатия кнопки
    mov eax, hBut
    cmp lParam, eax
    jne COM_END            ; команда не соответствует нажатию кнопки

    INVOKE SendMessage, hedt1, WM_GETTEXT, 20, offset TEXTA
    INVOKE SendMessage, hedt2, WM_GETTEXT, 20, offset TEXTB

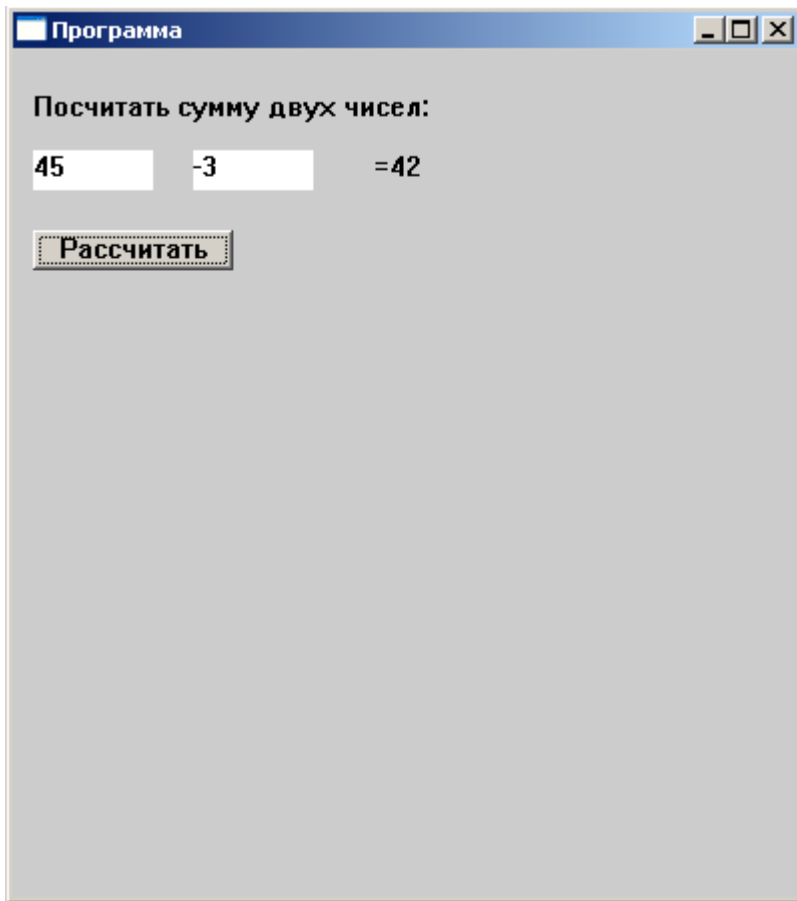
    INVOKE StrToInt, offset TEXTA
    mov summa, eax
    INVOKE StrToInt, offset TEXTB
    add summa, eax

    mov eax, sum_len
    INVOKE TextOutA,      ; стирание строки результата в окне
        hdc, 180, 50, offset mess2, eax
    INVOKE IntToStr, summa, offset mess2+1
    INVOKE LENSTR, offset mess2 ; определение длины результата
    push eax
    INVOKE TextOutA,      ; вывод результата
        hdc, 180, 50, offset mess2, eax
    pop ecx                ; очистка строки
    inc ecx
    mov al, ' '
    mov edi, offset mess2+1
CLR: mov [edi], al
    inc edi
    loop CLR
COM_END:
    MOV EAX, 0
    JMP FINISH
;-----
WMPAINT:                    ; перерисовка окна
    INVOKE BeginPaint, hW, offset ps
    mov  hdc, eax
    INVOKE SetBkColor, hdc, RGBW

    mov  eax, mess1_len
    INVOKE TextOutA, hdc, 10, 20, offset mess1, eax
    INVOKE EndPaint, hdc, offset ps
    MOV  EAX, 0
    JMP  FINISH
;-----
DEFWNDPROC:                ; обработка сообщения по умолчанию
    INVOKE DefWindowProc,
        hW, Mes, wParam, lParam
    JMP  FINISH
;-----
WMDESTROY:                 ; выход из цикла обработки сообщений
    INVOKE PostQuitMessage, 0
    MOV  EAX, 0
FINISH: ret
WNDPROC ENDP
END START

```

В результате работы программы получим окно:



Графика в оконных приложениях Windows

Система координат для вывода графических образов такая же, как и для ввода текстовой информации. Координаты измеряются в логических единицах, которые по умолчанию совпадают с пикселями. Начало отсчета – верхний левый угол окна.

Для создания графических образов используются 3 основных объекта: точка (пиксел), перо, кисть.

Цвет точки задается с помощью функции

```
COLORREF SetPixel(  
    HDC hdc,                // дескриптор контекста окна  
    int X,                  // x-координата точки  
    int Y,                  // y-координата точки  
    COLORREF crColor);     // цвет точки
```

Перо используется для рисования линий и контуров замкнутых фигур. Цвет пера задается при помощи функции

```
HPEN CreatePen(  
    int fnPenStyle,        // стиль пера  
                            // PS_SOLID    =0      _____  
                            // PS_DASH     =1      - - - - -  
                            // PS_DOT      =2      .....  
                            // PS_DASHDOT  =3      - . - . -  
                            // PS_DASHDOTDOT =4     - . . . .  
                            // PS_NULL     =5      невидимое перо  
    int nWidth,           // ширина пера  
    COLORREF crColor);   // цвет пера
```

Кисть используется для закрашивания замкнутых объектов. Цвет кисти задается с помощью функции

```
HBRUSH CreateSolidBrush( COLORREF crColor ); // цвет кисти
```

Можно заранее создать несколько кистей и перьев, а затем выбирать нужные с помощью функции

```
HGDIOBJ SelectObject(  
    HDC hdc,                // дескриптор контекста окна  
    HGDIOBJ hgdiobj );     // дескриптор объекта
```

Обновление информации в окне программы происходит по получению сообщения WM_PAINT. В реальных программах вывод информации в окно может происходить по различным событиям и из различных процедур. Кроме того, если информации в окне много, то непосредственный вывод при помощи функции TextOut достаточно медленный. Чтобы воспроизводить содержимое окна, необходимо где-то запомнить это содержимое. Возникает проблема сохранения информации (и не только текстовой), находящейся в окне.

В операционной системе Windows образуется виртуальное окно, и весь вывод информации производится туда. Затем по приходе сообщения WM_PAINT содержимое виртуального окна копируется на реальное окно. В целом общая схема такова:

1. При создании окна

- создается совместимый контекст устройства функцией

```
HDC CreateCompatibleDC( HDC hdc); // дескриптор контекста окна
```

Полученный контекст следует запомнить.

- создается карта бит, совместимая с данным контекстом функцией

```
HBITMAP CreateCompatibleBitmap(
    HDC hdc,          // дескриптор контекста окна
    int nWidth,      // ширина пиксельной картинке
    int nHeight );  // высота пиксельной картинке
```

- выбирается кисть цветом, совпадающим с цветом основного окна.

- создается битовый шаблон путем выполнения растровой операции с использованием выбранной кисти. Функция перерисовки заданного прямоугольника выбранной кистью

```
BOOL PatBlt(
    HDC hdc,          // дескриптор контекста окна
    int nXLeft,      // x- координата верхнего левого угла
    int nYLeft,      // y- координата верхнего левого угла
    int nWidth,      // ширина прямоугольника
    int nHeight,     // высота прямоугольника
    DWORD dwRop);   // код растровой операции:
;PATCOPY = 0F00021h копирует цвет кисти в окно.
;PATINVERT = 5A0049h комбинирует цвет кисти и цвет окна функцией XOR
;DSTINVERT = 550009h инвертирует цвет прямоугольника
;BLACKNESS = 42h закрашивает прямоугольник черным цветом
;WHITENESS = 0FF0062h закрашивает прямоугольник белым цветом
```

- ### 2. Вся информация выводится в виртуальное окно и дается команда перерисовки окна. Функция

```
BOOL InvalidateRect(
    HWND hWnd,       // дескриптор окна
    CONST RECT* lpRect, // координаты прямоугольника
    BOOL bErase);   // при 1 фон стирается, при 0 остается
    RECT STRUCT
        left    dd    ?
        top     dd    ?
        right   dd    ?
        bottom  dd    ?
    RECT ENDS
```

- ### 3. При получении сообщения WM_PAINT содержимое виртуального окна копируется на реальное окно. Функция

```

BOOL BitBlt(
    HDC hdcDest, // дескриптор совместимого контекста
    int nXDest, // x- координата левого верхнего угла совм. контекста
    int nYDest, // y- координата левого верхнего угла совм. контекста
    int nWidth, // ширина прямоугольника совместимого контекста
    int nHeight, // высота прямоугольника совместимого контекста
    HDC hdcSrc, // дескриптор контекста окна
    int nXSrc, // x- координата левого верхнего угла контекста окна
    int nYSrc, // y- координата левого верхнего угла контекста окна
    DWORD dwRop); // код растровой операции
    SRCCOPY equ 0CC0020h
    SRCPAINT equ 0EE0086h
    SRCAND equ 8800C6h
    SRCINVERT equ 660046h
    SRCERASE equ 440328h

```

Для рисования можно использовать следующие функции API:

```

BOOL LineTo( // провести прямую линию выбранным пером
             // от текущей точки до указанной точки
    HDC hdc, // дескриптор контекста окна
    int nXEnd, // x- координата конечной точки
    int nYEnd); // y- координата конечной точки
BOOL MoveToEx( // переместить текущую точку
    HDC hdc, // дескриптор контекста окна
    int X, // x- координата новой текущей точки
    int Y, // y- координата новой текущей точки
    LPPOINT lpPoint); // указатель на структуру POINT, куда
// поместятся координаты старой текущей точки, может быть равен 0
BOOL ArcTo( // рисование дуги
    HDC hdc, // дескриптор контекста окна
    int nLeftRect, // x-координата верхнего левого угла
    int nTopRect, // y- координата верхнего левого угла
    int nRightRect, // x- координата нижнего правого угла
    int nBottomRect, // y- координата нижнего правого угла
    int nXRadiall, // x- координата конца первого радиуса
    int nYRadiall, // y- координата конца первого радиуса
    int nXRadiall2, // x- координата конца второго радиуса
    int nYRadiall2); // y- координата конца второго радиуса
BOOL Rectangle( // рисование прямоугольника
    HDC hdc, // дескриптор контекста окна
    int nLeftRect, // x-координата верхнего левого угла
    int nTopRect, // y-координата верхнего левого угла
    int nRightRect, // x-координата нижнего правого угла
    int nBottomRect); // координата нижнего правого угла
BOOL Ellipse( // рисование эллипса
    HDC hdc, // дескриптор контекста окна
    int nLeftRect, // x- координата верхнего левого угла
    int nTopRect, // y- координата верхнего левого угла
    int nRightRect, // x- координата нижнего правого угла
    int nBottomRect); // y- координата нижнего правого угла

```

Для получения размера окна при его изменении используется функция

```

BOOL GetWindowRect( HWND hWnd, // дескриптор окна
                   LPRECT lpRect); // указатель на структуру RECT

```

Для освобождения контекста окна используется функция

```
int ReleaseDC(  
    HWND hWnd,    // дескриптор окна  
    HDC hDC);    // дескриптор контекста окна
```

Для удаления контекста окна используется функция

```
BOOL DeleteDC(    HDC hdc    ); // дескриптор контекста окна
```

Для получения сведений о размерах (в пикселях) элементов окна системы используется функция

```
int WINAPI GetSystemMetrics( __in int nIndex)  
nIndex – константа, определяющая необходимую величину для измерений  
SM_CXSCREEN=0 количество точек экрана по горизонтали  
SM_CYSCREEN=1 количество точек экрана по вертикали  
SM_CYCAPTION=4 высота строки заголовка  
SM_CXFRAME =32 толщина вертикальной рамки окна  
SM_CYFRAME =8 толщина горизонтальной рамки окна
```

Пример: вывести график функции $y=\sin(t)$ при $t \in [0, 360^0]$.

Вывод точек осуществляется функцией LineTo. Значение координат точек вычисляется по формулам:

$$X = \text{OffsetX} + t \cdot \text{ScaleX}, \quad Y = \text{OffsetY} + y \cdot \text{ScaleY}, \quad (*)$$

где OffsetX, OffsetY – смещения начала отсчета графика относительно начала отсчета окна;

ScaleX, ScaleY – масштабные коэффициенты осей.

Если принять, что данные, возвращаемые функцией GetWindowRect, позволяют вычислить ширину окна Width и высоту окна Height², то величины в формулах (*) можно определить как

$$\begin{aligned} \text{OffsetX} &= 0, & \text{OffsetY} &= \text{Height}/2, \\ \text{ScaleX} &= \text{Width}/360, & \text{ScaleY} &= 1/(\text{Height}/2) = 2/\text{Height} \end{aligned}$$

```
; текст программы  
.586  
.MODEL FLAT, stdcall  
RGBW equ 00D4D0C8h ; цвет фона в окне  
include win.inc3  
.data  
    HWND DD 0 ; дескриптор главного окна  
    HINST DD 0 ; дескриптор приложения  
    TITL DB "Программа", 0
```

² Размер окна, возвращаемый функцией GetWindowRect, соответствует окну с рамками и строкой заголовка. Для определения поля рисования окна необходимо вычесть из ширины окна 2 толщины рамки (чаще всего 8 пикселей), а из высоты – толщину рамки и толщину строки заголовка (чаще всего 27 пикселей). Более точно толщины окантовки окна можно получить функцией GetSystemMetrics

³ Файл win.inc см. выше

```

CLASSNAME      DB 'CLASS32',0
Message        MSG <?>
WC             WNDCLASS  <?>

hdc           DD 0 ; дескриптор контекста окна
hPen          DD 0 ; дескриптор пера
memdc         DD 0 ; дескриптор совместимого контекста окна
Height_       DD ? ; высота окна
Width_        DD ? ; ширина окна
rect_         RECT <>
ps            PAINTSTRUCT  <?>
messX         db  'y=sin(t)',0
mess_len      equ $-messX-1
OffsetX       DD 0
OffsetY       DD ?
ScaleX        DD ?
ScaleY        DD ?
X             DD 0
Y             DD 0
t             DD 0
cons          DD ? ; константа
deg_rad       DD 180 ; преобразование из градусов в радианы

```

```
.code
```

```
START proc
```

```
; получить дескриптор приложения
```

```
    INVOKE GetModuleHandle, 0
```

```
    MOV     HINST, EAX
```

```
; заполнить структуру окна
```

```
    MOV WC.style, CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
```

```
; процедура обработки сообщений
```

```
    MOV WC.lpfWndProc, OFFSET WNDPROC
```

```
    MOV EAX, HINST
```

```
MOV WC.hInstance, EAX
```

```
INVOKE LoadIcon, 0, IDI_APPLICATION
```

```
    MOV WC.hIcon, EAX
```

```
INVOKE LoadCursor, 0, IDC_ARROW
```

```
MOV WC.hCursor, EAX
```

```
INVOKE CreateSolidBrush, RGBW
```

```
MOV WC.hbrBackground, EAX
```

```
    MOV WC.lpszMenuName, 0
```

```
    MOV WC.lpszClassName, OFFSET CLASSNAME
```

```
INVOKE RegisterClass, OFFSET WC
```

```
INVOKE CreateWindowEx,0, ; создать окно зарегистрированного класса
```

```
    OFFSET CLASSNAME,
```

```
    OFFSET TITL,
```

```
    WS_CAPTION + WS_SYSMENU + WS_THICKFRAME + WS_GROUP +
```

```
WS_TABSTOP,
```

```
    100, ; X — координата левого верхнего угла
```

```
    100, ; Y — координата левого верхнего угла
```

```
    400, ; DX — ширина окна
```

```
    450, ; DY — высота окна
```

```
    0, 0, HINST,0
```

```
; проверка на ошибку
```

```

        CMP EAX, 0
        JZ  END_LOOP
        MOV HWND, EAX ; дескриптор окна
        INVOKE ShowWindow, HWND, SW_SHOWNORMAL ; показать созданное
окно
        INVOKE UpdateWindow, HWND ;перерисовать видимую часть окна

; цикл обработки сообщений
MSG_LOOP:
        INVOKE GetMessage, OFFSET Message, 0,0,0
        CMP EAX, 0
        JE  END_LOOP
        INVOKE TranslateMessage, OFFSET Message
        INVOKE DispatchMessageA, OFFSET Message
        JMP MSG_LOOP
END_LOOP:
        INVOKE ExitProcess, Message.wParam ; выход из программы (закреть
процесс)
        START endp

; -----
; процедура окна
WNDPROC PROC hW:DWORD, Mes:DWORD, wParam:DWORD, lParam:DWORD
        CMP Mes, WM_DESTROY
        JE  WMDESTROY
        CMP Mes, WM_CREATE
        JE  WMCREATE
        CMP Mes, WM_PAINT
        JE  WMPAINT
        CMP Mes, WM_SIZE
        JE  WMSIZE
        JMP DEFWNDPROC

WMSIZE:
        cmp memdc,0
        jne WMCREATE
        INVOKE DeleteDC, memdc
; создание окна
WMCREATE:
        INVOKE GetDC, hW
        mov hdc, eax
        INVOKE CreateCompatibleDC, hdc
        mov memdc, eax
        INVOKE GetWindowRect, hW, offset rect_
        mov eax, rect_.bottom
        sub eax, rect_.top
        mov Height_, eax
        mov eax, rect_.right
        sub eax, rect_.left
        mov Width_, eax

        INVOKE CreateCompatibleBitmap, hdc, Width_, Height_
        INVOKE SelectObject, memdc, eax
        INVOKE CreateSolidBrush, RGBW
        INVOKE SelectObject, memdc, eax

```

```

INVOKE PatBlt, memdc, 0, 0, Width_, Height_, PATCOPY
INVOKE ReleaseDC, hW, hdc

INVOKE CreatePen, PS_SOLID, 0,0    ; черное перо
mov hPen, eax
INVOKE SelectObject, memdc, hPen
mov eax, Height_
sub eax, 27
shr eax, 1
mov OffsetY, eax
INVOKE MoveToEx, memdc, 0, OffsetY, 0 ; горизонтальная ось
INVOKE LineTo, memdc, Width_, OffsetY

INVOKE CreatePen, PS_SOLID, 2, 000000FFh ; красное перо
mov hPen, eax
INVOKE SelectObject, memdc, hPen
INVOKE MoveToEx, memdc, 0, OffsetY, 0

mov eax, 0
mov t, eax
mov eax, 8 ; GetSystemMetrics(SM_CXFRAME)*2
mov cons, eax
fild Width_
fisub cons
fidiv deg_rad
mov eax, 2
mov cons, eax
fidiv cons
fstp ScaleX

mov eax, 27 ; GetSystemMetrics(SM_CYFRAME)*2 +
mov cons, eax ; GetSystemMetrics(SM_CXFRAME)*2
fild Height_
fisub cons
mov eax, 2
mov cons, eax
fidiv cons
fstp ScaleY

mov ecx, 360
L: push ecx
fild t
fmul ScaleX
fiadd OffsetX
fistp X
fild t
fldpi
fmulp
fidiv deg_rad
fsin
fchs
fmul ScaleY
fiadd OffsetY
fistp Y
inc t

```

```

    INVOKE LineTo, memdc, X, Y
    pop ecx
    loop L
    INVOKE InvalidateRect, hW, offset rect_,0
    INVOKE SetBkColor, memdc, RGBW

    mov eax, mess_len
    INVOKE TextOutA, memdc, 10, 20,
        offset messX, eax ; вывод текста y=sin(t)
    MOV EAX, 0
    JMP FINISH

; перерисовка окна
WMPAINT:
    INVOKE BeginPaint, hW, offset ps
    mov hdc, eax
    INVOKE BitBlt,
        hdc,
        0, 0,
        Width_, Height_,
        memdc,
        0, 0,
        SRCCOPY

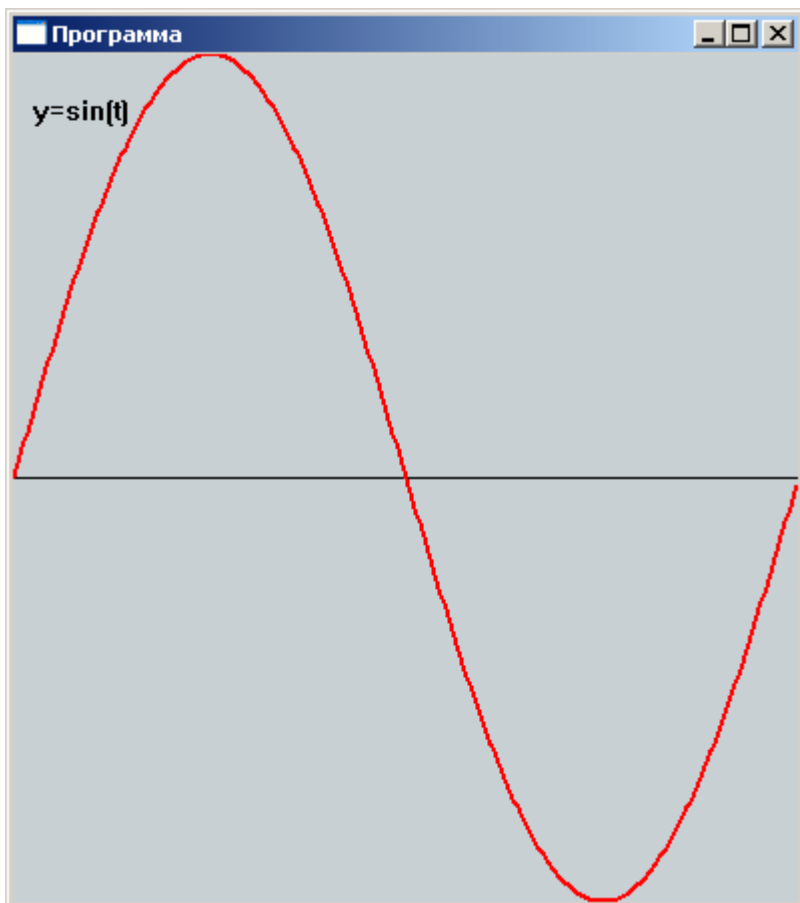
    INVOKE EndPaint, hdc, offset ps
    MOV EAX, 0
    JMP FINISH

; обработка сообщения по умолчанию
DEFWNDPROC:
    INVOKE DefWindowProc,
        hW,
        Mes,
        wParam,
        lParam
    JMP FINISH

WMDESTROY:
    INVOKE DeleteDC, hPen
    INVOKE DeleteDC, memdc
    INVOKE PostQuitMessage, 0
    MOV EAX, 0
FINISH:
    ret
WNDPROC ENDP
END START

```

Результат работы программы:



Ресурсы в Windows-приложениях

Ресурс — это специальный объект, используемый программой, но не определяемый в ее теле. Наиболее употребляемыми ресурсами являются

- иконки;
- курсоры;
- битовые картинки;
- строки;
- диалоговые окна;
- меню;
- акселераторы.

Преимущества ресурсов:

- Ресурсы загружаются в память лишь при обращении к ним, т.е. реализуется экономия памяти.
- Свойства ресурсов поддерживаются системой автоматически, не требуя от программиста написания дополнительного кода.

Описание ресурсов производится в отдельном текстовом файле с расширением `.rc`, в котором для описания каждого типа ресурса используются специальные операторы. Подготовку этого файла можно вести двумя способами: ручным и автоматизированным.

Ручной способ предполагает следующее:

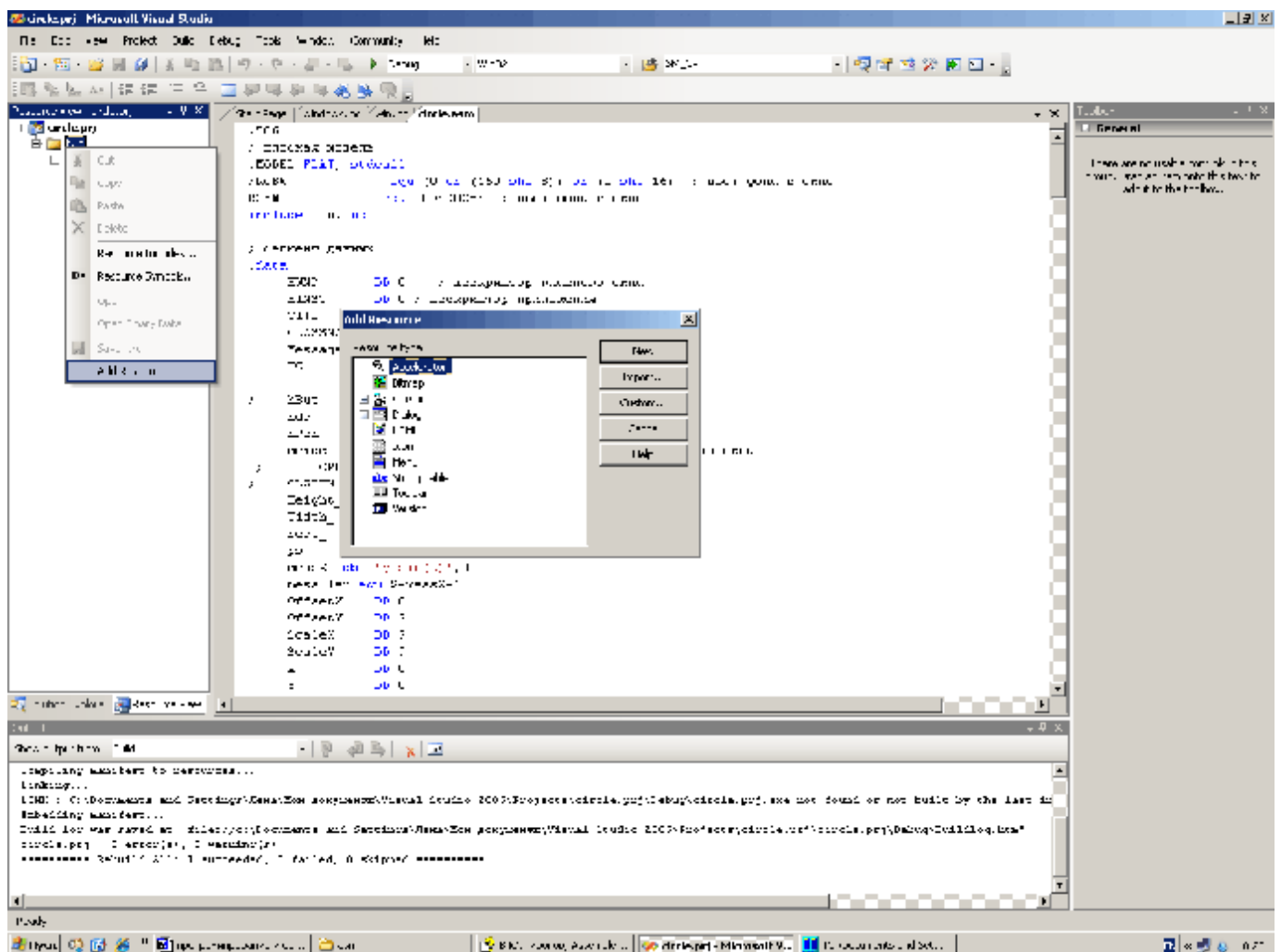
- разработчик ресурса хорошо знает операторы, необходимые для описания конкретного ресурса;
- ввод текста ресурсного файла выполняется с помощью редактора, который не вставляет в вводимый текст элементы форматирования. Наиболее доступный редактор для этих целей входит в состав программного обеспечения Windows — это `notepad.exe`.

Автоматический способ создания ресурсного файла предполагает использование специальной программы — редактора ресурсов, который позволяет визуализировать процесс создания ресурса. Конечные результаты работы этой программы могут быть двух видов: в виде текстового файла с расширением `.rc`, который впоследствии можно подвергнуть ручному редактированию, либо в виде двоичного файла, уже пригодного к включению в исполняемый файл приложения.

После того как ресурсы описаны в файле с расширением `.rc`, они должны быть преобразованы в вид, пригодный для включения их в общий исполняемый файл приложения. Для этого необходимо:

1. Откомпилировать ресурсный файл. На этом шаге выполняется преобразование текстового представления ресурсного файла с расширением `.rc` в двоичное представление с расширением `.res`. Для реализации этого действия в пакете MASM есть специальная программа `rc.exe` — компилятор ресурсов.
2. Включить ресурсы в исполняемый файл приложения. Это действие выполняет компоновщик `link.exe`, которому в качестве последнего параметра должно быть передано имя ресурсного файла (`.res`).

Для создания ресурсов в Visual Studio достаточно в дереве проекта добавить новый файл ресурса (Resource Files → Add → New Item, Resource File (.rc)) и задать имя файла ресурсов. Для добавления в файл ресурсов новых описаний нужно кликнуть дважды на файле ресурсов и в открывшемся окне ресурсов дерева проектов выбрать по правой кнопке Add Resource.



Иконки, курсоры могут быть описаны в самом файле ресурсов, либо храниться в отдельном файле *.ico, *.cur. Компилятор ресурсов MASM поддерживает последний случай.

При добавлении некоторого ресурса в файле resource.h, подключаемом к проекту по умолчанию, появляются строки с именами пользовательских констант, значения которых присвоены тем или иным ресурсам:

```
#define IDI_ICON1 101
#define IDC_CURSOR1 101
```

В файле *.rc появляются строки, ставящие в соответствие каждой пользовательской константе ресурса тип ресурса и файл с описанием этого ресурса:

```
IDI_ICON1 ICON "m.ico"
IDC_CURSOR1 CURSOR "mouse.cur"
```

Для использования ресурсов в программе заменяем соответствующие участки кода (вызов функций LoadIcon, LoadCursor при регистрации класса окна):

```
INVOKE LoadIcon, HINST, 101
MOV WC.hIcon, EAX
INVOKE LoadCursor, HINST, 101
MOV WC.hCursor, EAX
```

При компоновке программы из командной строки компилируется вначале файл ресурсов:

```
rc.exe myres.rc
```

а затем полученный файл myres.res подключается к исполняемому модулю:

```
link.exe /subsystem:windows myfile.obj myres.res
```

II

Меню. Меню также может быть задано в файле ресурсов. В программе оно определяется по имени (строке). Для обычного окна при регистрации класса следует указать ссылку на меню окна. Описание простейшего меню имеет вид (Файл → Выход)

```
IDR_MENU1 MENU
BEGIN
    POPUP "Файл"
    BEGIN
        MENUITEM "Выход", ID_40001
    END
END
```

Каждый пункт меню описывается своей константой в файле resource.h:

```
#define IDR_MENU1 101
#define ID_40001 40001
```

Для использования меню в программе включаем в регистрацию класса окна указатель меню:

Для обработки выбора одного из элементов меню используется отдельная программа обработчика меню. В качестве параметров в функцию обработки меню передаются дескриптор окна и параметр, несущий информацию об активном пункте меню. Для вызова функции меню в оконной функции используется сообщение WM_COMMAND.

Ниже приведен текст программы рисования функции $y=\sin(t)$, использующей ресурсы: иконку, курсор, меню.

Файл ресурсов: myres.rc

```
#define IDC_CURSOR1          101
#define IDI_ICON1           101
#define IDR_MENU1           101
#define ID_40001            40001

IDC_CURSOR1          CURSOR          "mouse.cur"
IDI_ICON1            ICON            "icon1.ico"

IDR_MENU1 MENU
BEGIN
    POPUP "Файл"
    BEGIN
        MENUITEM "Выход",          ID_40001
    END
END

; текст программы
.586
.MODEL FLAT, stdcall
RGBW          equ 00D4D0C8h          ; цвет фона в окне
include win.inc
.data
    HWND          DD 0 ; дескриптор главного окна
    HINST          DD 0 ; дескриптор приложения
    TITL          DB "Программа",0
    CLASSNAME     DB 'CLASS32',0
    Message       MSG <?>
    WC            WNDCLASS <?>

    hdc          DD 0 ; дескриптор контекста окна
    hPen         DD 0 ; дескриптор пера
    memdc        DD 0 ; дескриптор совместимого контекста окна
    CPBUT        DB 'Выход',0 ; надпись на кнопке
    CLSBTN       DB 'BUTTON',0 ; класс окна "КНОПКА"
    Height_      DD ? ; высота окна
    Width_       DD ? ; ширина окна
    rect_        RECT <>
```

```

ps                PAINTSTRUCT    <?>
messX             db    'y=sin(t)',0
mess_len          equ    $-messX-1
OffsetX           DD    0
OffsetY           DD    ?
ScaleX            DD    ?
ScaleY            DD    ?
X                 DD    0
Y                 DD    0
t                 DD    0
cons              DD    1
deg_rad           DD    180

```

.code

```

START proc
    INVOKE GetModuleHandle, 0    ; получить дескриптор приложения
    MOV     HINST, EAX
    MOV WC.style, CS_HREDRAW+CS_VREDRAW+CS_GLOBALCLASS
; процедура обработки сообщений
    MOV WC.lpfWndProc, OFFSET WNDPROC
    MOV EAX, HINST
    MOV WC.hInstance, EAX
    INVOKE LoadIcon, HINST, 101
    MOV WC.hIcon, EAX
    INVOKE LoadCursor, HINST, 101
    MOV WC.hCursor, EAX
    INVOKE CreateSolidBrush, RGBW
    MOV WC.hbrBackground, EAX
    MOV WC.lpszMenuName, 101
    MOV WC.lpszClassName, OFFSET CLASSNAME
    INVOKE RegisterClass, OFFSET WC
    INVOKE CreateWindowEx, ; создать окно зарегистрированного класса
        0,
        OFFSET CLASSNAME,
        OFFSET TITL,
        WS_CAPTION + WS_SYSMENU + WS_THICKFRAME + WS_GROUP + WS_TABSTOP,
        100, ; X - координата левого верхнего угла
        100, ; Y - координата левого верхнего угла
        400, ; DX - ширина окна
        450, ; DY - высота окна
        0, 0, HINST, 0
    CMP EAX, 0    ; проверка на ошибку
    JZ END_LOOP
    MOV HWND, EAX ; дескриптор окна
; -----
    INVOKE ShowWindow, HWND, SW_SHOWNORMAL ; показать созданное окно
    INVOKE UpdateWindow, HWND ; перерисовать видимую часть окна
; цикл обработки сообщений
MSG_LOOP:
    INVOKE GetMessage,
        OFFSET Message, 0, 0, 0
    CMP EAX, 0
    JE END_LOOP
    INVOKE TranslateMessage, OFFSET Message
    INVOKE DispatchMessageA, OFFSET Message

```

```

        JMP MSG_LOOP
END_LOOP:
; выход из программы (закрыть процесс)
        INVOKE ExitProcess, Message.wParam
START endp

; Процедура обработки элементов меню
MenuProc proc hW:DWORD, wParam:DWORD
        push eax
        mov eax, wParam
        cmp eax, 40001
        je MEXIT
        jmp MEXIT1
MEXIT:
        INVOKE PostQuitMessage, 0
MEXIT1:
        pop eax
        ret
MenuProc endp

; -----
; процедура окна
; расположение параметров в стеке
; [EBP+014H] LPARAM
; [EBP+10H] WAPARAM
; [EBP+0CH] MES
; [EBP+8] HWND
WNDPROC PROC hW:DWORD, Mes:DWORD, wParam:DWORD, lParam:DWORD
        CMP Mes, WM_DESTROY
        JE WMDESTROY
        CMP Mes, WM_CREATE
        JE WMCREATE
        CMP Mes, WM_COMMAND
        JE WMCOMMAND
        CMP Mes, WM_PAINT
        JE WMPAINT
        CMP Mes, WM_SIZE
        JE WMSIZE
        JMP DEFWNDPROC
; создание окна
WMCREATE:
WMSIZE:
        INVOKE GetDC, hW
        mov hdc, eax
        INVOKE CreateCompatibleDC, hdc
        mov memdc, eax
        INVOKE GetWindowRect, hW, offset rect_
        mov eax, rect_.bottom
        sub eax, rect_.top
        mov Height_, eax
        mov eax, rect_.right
        sub eax, rect_.left
        mov Width_, eax

        INVOKE CreateCompatibleBitmap,

```

```

    hdc, Width_, Height_
    INVOKE SelectObject, memdc, eax
    INVOKE CreateSolidBrush, RGBW
    INVOKE SelectObject, memdc, eax
    INVOKE PatBlt, memdc, 0, 0, Width_, Height_, PATCOPY
    INVOKE ReleaseDC, hW, hdc

    INVOKE CreatePen, PS_SOLID, 0,0
    mov hPen, eax
    INVOKE SelectObject, memdc, hPen
    mov eax, Height_
    sub eax, 27
    shr eax, 1
    mov OffsetY, eax
    INVOKE MoveToEx, memdc, 0, OffsetY, 0
    INVOKE LineTo, memdc, Width_, OffsetY

    INVOKE CreatePen, PS_SOLID, 2, 000000FFh
    mov hPen, eax
    INVOKE SelectObject, memdc, hPen
    INVOKE MoveToEx, memdc, 0, OffsetY, 0

    mov eax, 0
    mov t, eax
    mov eax, 8                ; GetSystemMetrics(SM_CXFRAME)*2
    mov cons, eax
    fld Width_
    fisub cons
    fidiv deg_rad
    mov eax, 2
    mov cons, eax
    fidiv cons
    fstp ScaleX

    mov eax, 27            ; GetSystemMetrics(SM_CYFRAME)*2 +
                          ;GetSystemMetrics(SM_CXFRAME)*2
    mov cons, eax
    fld Height_
    fisub cons
    mov eax, 2
    mov cons, eax
    fidiv cons
    fstp ScaleY

    mov ecx, 360
L:  push ecx
    fld t
    fmul ScaleX
    fiadd OffsetX
    fistp X
    fld t
    fldpi
    fmulp
    fidiv deg_rad
    fsin

```

```

    fchs
    fmul ScaleY
    fiadd OffsetY
    fistp Y
    inc t
    INVOKE LineTo, memdc, X, Y
    pop ecx
    loop L

    INVOKE InvalidateRect, hW, offset rect_,0
    INVOKE SetBkColor, memdc, RGBW

mov eax, mess_len
INVOKE TextOutA,
    memdc,
    10, 20,
    offset messX,
    eax

    MOV EAX, 0
    JMP FINISH
; перерисовка окна
WMPAINT:
    INVOKE BeginPaint, hW, offset ps
    mov hdc, eax
    INVOKE BitBlt, hdc,
        0, 0, Width_, Height_,
        memdc, 0, 0, SRCCOPY
    INVOKE EndPaint, hdc, offset ps
    MOV EAX, 0
    JMP FINISH
; обработка сообщения по умолчанию
DEFWNDPROC:
    INVOKE DefWindowProc,
        hW,
        Mes,
        wParam,
        lParam
    JMP FINISH
; обработка событий меню
WMCOMMAND:
    INVOKE MenuProc, hW, wParam
    mov eax, 0
    jmp FINISH

WMDESTROY:
    INVOKE DeleteDC, hPen
    INVOKE DeleteDC, memdc
    INVOKE PostQuitMessage, 0
    MOV EAX, 0
FINISH:
    ret
WNDPROC ENDP
END START

```


Диалоговые окна являются наиболее сложными элементами ресурсов. Для диалога не задается идентификатор, обращение к нему происходит по его имени (строке).

```
#define WS_SYSMENU          0x00080000L
#define WS_MINIMIZEBOX     0x00020000L
#define WS_MAXIMIZEBOX     0x00010000L
DIALOG DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
}
```

Как видим, определение диалога начинается со строки, содержащей ключевое слово `DIALOG`. В этой же строке далее указывается положение и размер диалогового окна. Далее идут строки, содержащие другие свойства окна. Наконец идут фигурные скобки. В данном случае они пусты. Это означает, что на окне нет никаких управляющих элементов. Тип окна, а также других элементов определяется константами, которые мы поместили в начале файла. Эти константы стандартны, и для языка Си хранятся в файле `RESOURCE.H`. Мы все константы будем определять непосредственно в файле ресурсов. Константы определяются согласно нотации языка Си.

Прежде чем разбирать пример, рассмотрим особенности работы с диалоговыми окнами. Диалоговое окно очень похоже на обычное окно. Так же как обычное окно, оно имеет свою процедуру. Процедура диалогового окна имеет те же параметры, что и процедура обычного окна. Сообщений, которые приходят на процедуру диалогового окна, гораздо меньше. Но те, которые у диалогового окна имеются, в основном совпадают с аналогичными сообщениями для обычного окна. Только вместо сообщения `WM_CREATE` приходит сообщение `WM_INITDIALOG`, а вместо сообщения `WM_DESTROY` – `WM_CLOSE`. Процедура диалогового окна может возвращать либо нулевое, либо ненулевое значение. Ненулевое значение должно возвращаться в том случае, если процедура обрабатывает (берет на себя обработку) данное сообщение, и ноль - если предоставляет обработку системе.

При создании обычного окна все его свойства определяются тремя факторами: свойствами класса, свойствами, определяемыми при создании окна, реакцией процедуры окна на определенные сообщения. При создании диалогового окна все свойства заданы в ресурсах. Часть этих свойств задается, когда при вызове функции создания диалогового окна (DialogBoxParamA) неявно вызывается функция CreateWindow. Остальная же часть свойств определяется поведением внутренней функции, которую создает система при создании диалогового окна. Если с диалоговым окном что-то происходит, то сообщение сначала приходит на внутреннюю процедуру, а затем вызывается процедура диалогового окна, которую мы создаем в программе. Если процедура возвращает 0, то внутренняя процедура продолжает обработку данного сообщения, если же возвращается ненулевое значение, внутренняя процедура не обрабатывает сообщение.

Функция создания диалогового окна:

```

INT_PTR DialogBoxParam(
    HINSTANCE hInstance,           // дескриптор приложения
    LPCTSTR lpTemplateName,       // указатель шаблона диалогового окна
    HWND hWndParent,              // дескриптор родительского окна
    DLGPROC lpDialogFunc,         // указатель процедуры обработки
                                  // сообщений диалогового окна
    LPARAM dwInitParam);         // значение lParam, передаваемое
                                  // диалоговому окну при создании

// файл dial.rc
// определение констант
#define WS_SYSMENU      0x00080000L
#define WS_MINIMIZEBOX 0x00020000L
#define WS_MAXIMIZEBOX 0x00010000L

// идентификаторы
#define STR1 1
#define STR2 2
#define IDI_ICON1 3

// определили иконку
IDI_ICON1 ICON "icon1.ico"

// определение диалогового окна
DIALOG1 DIALOG 0, 0, 240, 120
STYLE WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "Пример диалогового окна"
FONT 8, "Arial"
{
}

```

```

;файл dial.asm
.386P
.MODEL FLAT, stdcall
include win.inc

_DATA SEGMENT DWORD PUBLIC USE32 'DATA'
MSG MSGSTRUCT <?>
HINST DD 0 ; дескриптор приложения
PA DB "DIAL1",0
BUF1 DB 40 dup(0)
BUF2 DB 40 dup(0)
_DATA ENDS

; сегмент кода
_TEXT SEGMENT DWORD PUBLIC USE32 'CODE'
WNDPROC PROC HWND:DWORD, MES:DWORD, WPARAM:DWORD, LPARAM:DWORD
PUSH EBX
PUSH ESI
PUSH EDI
CMP MES, WM_CLOSE
JNE L1
INVOKE EndDialog, HWND, 0
JMP FINISH
L1:
CMP MES, WM_INITDIALOG
JNE FINISH
; загрузить иконку
INVOKE LoadIconA, HINST, 3
; установить иконку
INVOKE SendMessageA, HWND, WM_SETICON, 0, EAX
FINISH:
POP EDI
POP ESI
POP EBX
MOV EAX, 0
RET
WNDPROC ENDP

START proc
; получить дескриптор приложения
INVOKE GetModuleHandleA, 0
MOV [HINST],EAX
; создать диалоговое окно
INVOKE DialogBoxParamA, HINST, offset PA, 0, offset WNDPROC, 0
START endp
_TEXT ENDS
END START

```

Акселераторы позволяют выбирать пункты меню просто сочетанием клавиш. Таблица акселераторов является ресурсом, имя которого должно совпадать с именем того меню (ресурса), пункты которого она определяет. Например, определяется один акселератор на пункт меню `MENUP`, имеющий идентификатор 4.

```

MENUP ACCELERATORS
{
VK_F5, 4, VIRTKEY
}

```

Общий вид таблицы акселераторов.

```

Имя ACCELERATORS
{
Клавиша 1, Идентификатор пункта меню (1) [,тип] [,параметр]
Клавиша 2, Идентификатор пункта меню (2) [,тип] [,параметр]
Клавиша 3, Идентификатор пункта меню (3) [,тип] [,параметр]
...
Клавиша N, Идентификатор пункта меню (N) [,тип] [,параметр]
}

```

Клавиша – это либо символ в кавычках, либо код ASCII символа, либо виртуальная клавиша. Если вначале стоит код символа, то тип задается как ASCII. Если используется виртуальная клавиша, то тип определяется как VIRTUAL. Все названия (макроимена) виртуальных клавиш можно найти в include- файлах (windows.h).

Параметр может принимать одно из следующих значений: NOINVERT, ALT, CONTROL, SHIFT. Значение NOINVERT означает, что не подсвечивается выбранный при помощи акселератора пункт меню. Значения ALT, CONTROL, SHIFT означают, что, кроме клавиши, определенной в акселераторе, должна быть нажата одна из управляющих клавиш. Кроме этого, если клавиша определяется в кавычках, то нажатие при этом клавиши CONTROL определяется знаком "^": "^A".

Механизм работы акселераторов. Для того чтобы акселераторы работали, необходимо выполнить два условия:

- Должна быть загружена таблица акселераторов. Для этого используется функция

```

HACCEL LoadAccelerators(
    HINSTANCE hInstance,    // дескриптор приложения
    LPCTSTR lpTableName);  // таблица акселераторов

```

- Сообщения, пришедшие от акселератора, следует преобразовать в сообщение WM_COMMAND. Для этого используется функция

```

int TranslateAccelerator(
    HWND hWnd,              // дескриптор окна
    HACCEL hAccTable,       // таблица акселераторов
    LPMSG lpMsg);           // указатель на структуру MSG

```

Функция TranslateAccelerator преобразует сообщения WM_KEYDOWN и WM_SYSKEYDOWN в сообщения WM_COMMAND и WM_SYSCOMMAND соответственно. При этом в старшем слове параметра WPARAM помещается 1, как отличие для акселератора. В младшем слове содержится идентификатор пункта меню.

Сообщение WM_SYSCOMMAND генерируется для пунктов системного меню или меню окна.

Функция TranslateAccelerator возвращает ненулевое значение, если было произведено преобразование сообщения акселератора, в противном случае возвращается 0. Необходимо включить вызов этой функции в цикл сообщений.

```
// файл menu1.rc
// определение констант
#define WS_SYSMENU      0x00080000L
#define WS_MINIMIZEBOX  0x00020000L
#define WS_MAXIMIZEBOX  0x00010000L
#define WS_POPUP        0x80000000L
#define VK_F5           0x74
#define st WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX

MENUP MENU
{
  POPUP "&Первый пункт"
  {
    MENUITEM "&Первый",1
    MENUITEM "В&торой",2,HELP
    MENUITEM "Что-то?",8
  }

  POPUP "&Второй пункт"
  {
    MENUITEM "Трети&й",3
    MENUITEM "Четверт&ый",4
    MENUITEM SEPARATOR
    POPUP "Еще подмен&ю"
    {
      MENUITEM "Десятый пунк&т",6
    }
  }
  MENUITEM "Вы&ход",5
}

// идентификаторы
#define IDI_ICON1 100

// определили иконку
IDI_ICON1 ICON "icon1.ico"

// определение диалогового окна
DIAL1 DIALOG 0, 0, 240, 120
STYLE WS_POPUP | st
CAPTION "Пример немодального диалогового окна"
FONT 8, "Arial"
{
}

MENUP ACCELERATORS
{
  VK_F5, 4, VIRTKEY, ALT
}

;файл menu1.asm
.386P
.MODEL FLAT, stdcall
WM_CLOSE      equ 10h
```

```

WM_INITDIALOG equ 110h
WM_SETICON    equ 80h
WM_COMMAND    equ 111h

```

```

; прототипы внешних процедур
EXTERN ShowWindow@8:NEAR
EXTERN MessageBoxA@16:NEAR
EXTERN ExitProcess@4:NEAR
EXTERN GetModuleHandleA@4:NEAR
EXTERN LoadIconA@8:NEAR
EXTERN LoadMenuA@8:NEAR
EXTERN SendMessageA@16:NEAR
EXTERN SetMenu@8:NEAR
EXTERN LoadAcceleratorsA@8:NEAR
EXTERN TranslateAcceleratorA@12:NEAR
EXTERN GetMessageA@16:NEAR
EXTERN DispatchMessageA@4:NEAR
EXTERN PostQuitMessage@4:NEAR
EXTERN CreateDialogParamA@20:NEAR
EXTERN DestroyWindow@4:NEAR
EXTERN TranslateMessage@4:NEAR

```

```
MSGSTRUCT STRUC
```

```

    MSHWND DD ?
    MSMESSAGE DD ?
    MSWPARAM DD ?
    MSLPARAM DD ?
    MSTIME DD ?
    MSPT DD ?

```

```
MSGSTRUCT ENDS
```

```
; сегмент данных
```

```

_DATA SEGMENT DWORD PUBLIC USE32 'DATA'
    NEWHWND DD 0
    MSG MSGSTRUCT <?>
    HINST DD 0 ; дескриптор приложения
    PA DB "DIAL1",0
    PMENU DB "MENUP",0
    STR1 DB "Выход из программы",0
    STR2 DB "Сообщение",0
    STR3 DB "Выбран четвертый",0
    ACC DD ?
_DATA ENDS

```

```
; сегмент кода
```

```
_TEXT SEGMENT DWORD PUBLIC USE32 'CODE'
```

```

; процедура окна
; расположение параметров в стеке
; [EBP+014H] LPARAM
; [EBP+10H] WAPARAM
; [EBP+0CH] MES
; [EBP+8] HWND
WNDPROC PROC

```

```

    PUSH EBP
    MOV EBP,ESP
    PUSH EBX
    PUSH ESI
    PUSH EDI

```

```

;-----
    CMP DWORD PTR [EBP+0CH],WM_CLOSE
    JNE L1

```

```

; закрыть диалоговое окно
    JMP L5

```

```

L1:
    CMP DWORD PTR [EBP+0CH],WM_INITDIALOG
    JNE L3

```

```

; загрузить иконку
    PUSH 100 ; идентификатор иконки
    PUSH [HINST] ; идентификатор процесса
    CALL LoadIconA@8
; установить иконку
    PUSH EAX
    PUSH 0 ; тип иконки (маленькая)
    PUSH WM_SETICON
    PUSH DWORD PTR [EBP+08H]
    CALL SendMessageA@16
; загрузить меню
    PUSH OFFSET PMENU
    PUSH [HINST]
    CALL LoadMenuA@8
; установить меню
    PUSH EAX
    PUSH DWORD PTR [EBP+08H]
    CALL SetMenu@8
;-----
    MOV EAX, 1 ; вернуть не нулевое значение
    JMP FIN
; проверяем, не случилось ли чего с управляющими
; элементами на диалоговом окне
L3:
    CMP DWORD PTR [EBP+0CH], WM_COMMAND
    JE L6
    JMP FINISH
; здесь определяем идентификатор, в данном случае
; это идентификатор пункта меню сообщения
L6:
    CMP WORD PTR [EBP+10H], 4
    JNE L4
    PUSH 0 ; MB_OK
    PUSH OFFSET STR2
    PUSH OFFSET STR3
    PUSH 0
    CALL MessageBoxA@16
    JMP FINISH
L4:
    CMP WORD PTR [EBP+10H], 5
    JNE FINISH
; сообщение
    PUSH 0 ; MB_OK
    PUSH OFFSET STR2
    PUSH OFFSET STR1
    PUSH 0
    CALL MessageBoxA@16
; закрыть диалоговое немодальное окно
L5:
    PUSH DWORD PTR [EBP+08H]
    CALL DestroyWindow@4
; послать сообщение для выхода из кольца
; обработки сообщений
    PUSH 0
    CALL PostQuitMessage@4 ; сообщение WM_QUIT
FINISH:
    MOV EAX, 0
FIN:
    POP EDI
    POP ESI
    POP EBX
    POP EBP
    RET 16
WNDPROC ENDP
START proc
; получить дескриптор приложения

```

```

    PUSH 0
    CALL GetModuleHandleA@4
    MOV [HINST], EAX
; загрузить акселераторы
    PUSH OFFSET PMENU
    PUSH [HINST]
    CALL LoadAcceleratorsA@8
    MOV ACC, EAX ; запомнить дескриптор таблицы
; создать немодальный диалог
    PUSH 0
    PUSH OFFSET WNDPROC
    PUSH 0
    PUSH OFFSET PA
    PUSH [HINST]
    CALL CreateDialogParamA@20
; визуализировать немодальный диалог
    MOV NEWHWND, EAX
    PUSH 1 ; SW_SHOWNORMAL
    PUSH [NEWHWND]
    CALL ShowWindow@8 ; показать созданное окно
; кольцо обработки сообщений
MSG_LOOP:
    PUSH 0
    PUSH 0
    PUSH 0
    PUSH OFFSET MSG
    CALL GetMessageA@16
    CMP EAX, 0
    JE END_LOOP
; транслировать сообщение акселератора
    PUSH OFFSET MSG
    PUSH [ACC]
    PUSH [NEWHWND]
    CALL TranslateAcceleratorA@12
    CMP EAX, 0
    JNE MSG_LOOP
    PUSH OFFSET MSG
    CALL TranslateMessage@4
    PUSH OFFSET MSG
    CALL DispatchMessageA@4
    JMP MSG_LOOP
END_LOOP:
    PUSH 0
    CALL ExitProcess@4
START endp

_TEXT ENDS
END START

```


ПРИЛОЖЕНИЕ 1

Таблица кодов ASCII

Коды управляющих символов

Символ	Код	Клавиши	Значение
nul	0	Ctrl + @	Нуль
soh	1	Ctrl + A	Начало заголовка
stx	2	Ctrl + B	Начало текста
etx	3	Ctrl + C	Конец текста
eot	4	Ctrl + D	Конец передачи
enq	5	Ctrl + E	Запрос
ack	6	Ctrl + F	Подтверждение
bel	7	Ctrl + G	Сигнал (звонок)
bs	8	Ctrl + H	Забой (шаг назад)
ht	9	Ctrl + I	Горизонтальная табуляция
lf	10	Ctrl + J	Перевод строки
vt	11	Ctrl + K	Вертикальная табуляция
ff	12	Ctrl + L	Новая страница
cr	13	Ctrl + M	Возврат каретки
so	14	Ctrl + N	Выключить сдвиг
si	15	Ctrl + O	Включить сдвиг
dle	16	Ctrl + P	Ключ связи данных
dc1	17	Ctrl + Q	Управление устройством 1
dc2	18	Ctrl + R	Управление устройством 2
dc3	19	Ctrl + S	Управление устройством 3
dc4	20	Ctrl + T	Управление устройством 4
nak	21	Ctrl + U	Отрицательное подтверждение
syn	22	Ctrl + V	Синхронизация
etb	23	Ctrl + W	Конец передаваемого блока
can	24	Ctrl + X	Отказ
em	25	Ctrl + Y	Конец среды
sub	26	Ctrl + Z	Замена
esc	27	Ctrl + [Ключ
fs	28	Ctrl + \	Разделитель файлов
gs	29	Ctrl +]	Разделитель группы
rs	30	Ctrl + ^	Разделитель записей
us	31	Ctrl + _	Разделитель модулей























Базовая таблица кодировки ASCII

32	пробел	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	“	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	‘	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	

Кодовая таблица 1251-MS-Windows

128	Ђ	144	Ђ	160		176	°	192	А	208	Р	224	а	240	р
129	Ѓ	145	‘	161	Ў	177	±	193	Б	209	С	225	б	241	с
130	,	146	’	162	ѣ	178	І	194	В	210	Т	226	в	242	т
131	ѓ	147	“	163	Ј	179	і	195	Г	211	У	227	г	243	у
132	„	148	”	164	ѡ	180	г	196	Д	212	Ф	228	д	244	ф
133	…	149	•	165	Љ	181	μ	197	Е	213	Х	229	е	245	х
134	†	150	–	166	і	182	¶	198	Ж	214	Ц	230	ж	246	ц
135	‡	151	—	167	§	183	·	199	З	215	Ч	231	з	247	ч
136	€	152	□	168	Ѓ	184	ё	200	И	216	Ш	232	и	248	ш
137	‰	153	™	169	©	185	№	201	Й	217	Щ	233	й	249	щ
138	Љ	154	љ	170	Є	186	є	202	К	218	Ђ	234	к	250	ь
139	<	155	>	171	«	187	»	203	Л	219	Ы	235	л	251	ы
140	Њ	156	њ	172	¬	188	ј	204	М	220	Ь	236	м	252	ь
141	Ќ	157	ќ	173	-	189	ѕ	205	Н	221	Э	237	н	253	э
142	Ћ	158	ћ	174	®	190	ѕ	206	О	222	Ю	238	о	254	ю
143	Ц	159	ц	175	İ	191	і	207	П	223	Я	239	п	255	я

Кодовая таблица 866-MS-DOS

128	А	144	Р	160	а	176		192	Л	208		224	р	240	≡Ë
129	Б	145	С	161	б	177		193	┘	209	≡	225	с	241	±ë
130	В	146	Т	162	в	178		194	┘	210	┘	226	т	242	≥
131	Г	147	У	163	г	179		195	┘	211	┘	227	у	243	≤
132	Д	148	Ф	164	д	180		196	—	212	≡	228	ф	244	┘
133	Е	149	Х	165	е	181		197	┘	213	≡	229	х	245	┘
134	Ж	150	Ц	166	ж	182		198	┘	214	┘	230	ц	246	÷
135	З	151	Ч	167	з	183		199	┘	215	┘	231	ч	247	≈
136	И	152	Ш	168	и	184		200	┘	216	≡	232	ш	248	°
137	Й	153	Щ	169	й	185		201	┘	217	┘	233	щ	249	·
138	К	154	Ъ	170	к	186		202	┘	218	┘	234	ъ	250	·
139	Л	155	Ы	171	л	187		203	┘	219		235	ы	251	√
140	М	156	Ь	172	м	188		204	┘	220		236	ь	252	n
141	Н	157	Э	173	н	189		205	≡	221		237	э	253	²
142	О	158	Ю	174	о	190		206	┘	222		238	ю	254	■
143	П	159	Я	175	п	191		207	≡	223		239	я	255	