11. Директивы определения идентификаторов и операции

11.1. Директивы определения идентификаторов

Директивы определения идентификаторов EQU (приравнять) и = (знак равенства) позволяют присвоить выражению символическое имя или идентификатор. Эти действия выполняет транслятор, заменяя в последующем тексте программы символические имена из левой части этих операторов строками из правой.

Формат

имя_идентификатора equ строка или числовое_выражение

имя_идентификатора = числовое_выражение

Выражение может быть

– целочисленной константой,

– адресом,

– другим символическим именем,

– меткой команды.

После присваивания имени им можно пользоваться всюду, где требуется указать это выражение.

Директива EQU и знак равенства сходны по назначению, но различаются следующим:

1. Определенные знаком «=» идентификаторы можно переопределять, а определенные директивой EQU – нельзя.

2. Директиву EQU можно использовать как с числовыми, так и с текстовыми выражениями, а знак «=» только с числовыми.

Директива EQU удобна для присваивания простых имен числам, сложным комбинациям адресов и другим подобным объектам, которые неоднократно используются в программе.

Пример.

```
K       EQU     1024            ; присвоить имя константе
TABLE   EQU     [EBP][ESI]      ; присвоить имя комбинации адресов
SPEED   EQU     RATE            ; определить синоним
COUNT   EQU     ECX             ; присвоить имя регистру
```

Ассемблер всегда пытается вычислить значение строки, воспринимая ее как выражение. Для того чтобы строка воспринималась именно как текстовая, необходимо заключить ее в угловые скобки:

<строка>

Угловые скобки являются операцией ассемблера, называемой *операцией выделения*. С ее помощью транслятору сообщается, что заключенная в угловые скобки строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы.

Директивы EQU используются в случае определения символов, которым не обязательно должно соответствовать целочисленное значение. Например, с помощью этой директивы можно определить вещественную константу:

PI EQU <3.1416>

Пример. Идентификатор можно легко связать с текстовой строкой, а затем на основе этого идентификатора создать переменную в программе:

```
pressKey EQU <"Для продолжения нажмите любую клавишу...",0>
. . . . . .
.data
prompt DB pressKey
```

Компилятор содержит директивы, значительно расширяющие возможности по работе с идентификаторами. Эти директивы аналогичны некоторым функциям обработки строк в языках высокого уровня. Под *строками* здесь понимается текст, описанный с помощью директивы EQU.

Директива слияния строк CATSTR. Формат:

идентификатор CATSTR строка_1, строка_2, ...

Результатом будет новая строка, состоящая из сцепленной слева направо последовательности строк строка_1, строка_2 и т.д. В качестве сцепляемых строк могут быть указаны имена ранее определенных идентификаторов.

Директива выделения подстроки в строке SUBSTR. Формат:

идентификатор SUBSTR строка, номер_позиции, размер

Результатом будет часть строки, заданной операндом строка, начинающаяся с позиции с номером номер_позиции и длиной, указанной операндом размер. Если требуется только остаток строки начиная с некоторой позиции, то достаточно указать номер позиции (без размера).

Директива определения вхождения одной строки в другую INSTR. Формат:

идентификатор INSTR [номер_нач_позиции,] строка_1, строка_2

В результате операнду идентификатор будет присвоено числовое значение, соответствующее номеру (первой) позиции, с которой в операнд строка_1 входит операнд строка_2. Если такого совпадения нет, то идентификатор получит значение 0. Номер начальной позиции необязателен.

Директива определения длины строки, задающей идентификатор, SIZESTR. Формат:

идентификатор SIZESTR строка

В результате операнду идентификатор будет присвоено числовое значение длины операнда строка.

11.2. Операции

Операция является модификатором, который используется в операторе языка ассемблера или в директиве.

Существует пять видов операций:

– арифметические,

– логические,

– отношения,

– операции, возвращающие значения,

– операции присваивания атрибута.

11.2.1. Арифметические операции

Арифметические операции производятся над числовыми операндами и дают числовой результат. Наиболее часто используются операции: сложение (+), вычитание (–), умножение (*) и деление (/). Они имеют формат

значение1 *операция* значение2

Пример.

```
CONST = 10
. . .
CONST = CONST +1
```

Операция деления (/) возвращает частное от деления нацело.

Операция MOD возвращает остаток от деления нацело.

Пример. Операторы

```
REM1 EQU 31416 / 10000
REM2 EQU 31416 MOD 10000
```

определят константы REM1 со значением 3 и REM2 со значением 1416.

Операции SHL и SHR имеют формат

значение *операция* выражение

Они сдвигают числовой операнд «значение» влево или вправо на число битов, равное значению операнда «выражение».

11.2.2. Логические операции

Логические операции используются для манипулирования отдельными двоичными битами байта, слова или двойного слова. Имеются логические операции AND, OR, XOR и NOT.

Напомним, что отличие данных операций от одноименных команд микропроцессора в том, что логические команды действуют на этапе исполнения программы (их выполняет микропроцессор), а логические операции – на этапе ее трансляции (под управлением ассемблера).

Операции AND, OR, XOR имеют формат

значение1 *операция* значение2

Формат операции NOT:

операция значение

11.2.3. Операции отношения

Операции отношения сравнивают два числовых значения или два адреса памяти и дают числовой результат. Формат операций:

операнд1 *операция* операнд2

Имеются следующие операции:

Название Признак истинности

EQ	значения операндов совпадают
NE	значения операндов не совпадают
LT	операнд1 меньше, чем операнд2
GT	операнд1 больше, чем операнд2
LE	операнд1 меньше, чем операнд2 или равен ему
GE	операнд1 больше, чем операнд2 или равен ему

Результатом может быть одно из двух чисел: 0, если отношение «ложно», и логическая единица (все биты равны 1), если оно «истинно».

Пример. Пусть CHOICE – ранее определенная константа. Тогда оператор

MOV AX, CHOICE LT 20

будет во время трансляции заменен на оператор

MOV AX, 0FFFFH

если значение CHOICE меньше 20, или на оператор

MOV AX, 0

если значение CHOICE больше или равно 20.

Так как операции отношения дают только два значения, то они редко используются сами по себе. Чаще всего они применяются в сочетании с другими операциями для формирования выражений «принятия решения».

Пример.

Пусть требуется загрузить в регистр AX значение 5, если значение CHOICE меньше 20, и 6 в противном случае. Такую задачу выполнит оператор

MOV AX, ((CHOICE LT 20) AND 5) OR ((CHOICE GE 20) AND 6)

Если значение CHOICE меньше 20, то выражение (CHOICE LT 20) «истинно», а выражение (CHOICE GE 20) «ложно». Поэтому промежуточная форма приведенного выше оператора будет иметь вид

MOV AX, (0FFFFH AND 5) OR (0 AND 6)

Ассемблер воспримет этот оператор как оператор

MOV AX, 5

С другой стороны, если значение CHOICE больше или равно 20, то выражение (CHOICE LT 20) «ложно», выражение (CHOICE GE 20) «истинно». Поэтому промежуточная форма примет вид

MOV AX,(0 AND 5) OR (0FFFFH AND 6)

и ассемблер воспримет ее как оператор

MOV AX, 6

11.2.4. Операции, возвращающие значения

Операции этой группы предоставляют информацию о переменных или метках программы.

Операция, обозначенная знаком доллара ($), возвращает адрес текущего оператора (текущее значение счетчика адреса).

Эта операция полезна, если мы хотим заставить ассемблер вычислить длины строк символов.

Пример. При трансляции операторов

```
MESSAGE     DB      "Ошибка"
MESSAGEL    EQU     $ – MESSAGE
```

ассемблер определит число символов в строке MESSAGE и присвоит его константе MESSAGEL. При выдаче сообщения на экран мы можем использовать значение константы MESSAGEL для задания числа выдаваемых символов.

Операция OFFSET возвращает адрес переменной или метки. Эту операцию мы уже встречали, когда рассматривали режимы адресации. Формат

OFFSET переменная или OFFSET метка

Операция TYPE возвращает размер в байтах переменной или элемента массива (объявленного с использованием операции дублирования DUP). Формат

TYPE переменная

Операции LENGTHOF (длина) и SIZEOF (размер) определяют характеристики массива, который определен в одной строке с меткой директивы определения данных. Формат операций

LENGTHOF переменная

SIZEOF переменная

Операция LENGTHOF возвращает число элементов массива.

Пример. Операторы

```
TABLE     DW        100 DUP(1)
. . . . . .
          MOV       ECX, LENGTHOF TABLE ; Загрузить в ECX число
                                        ; элементов в таблице TABLE
```

загрузят 100 в регистр ECX.

Операция SIZEOF возвращает число байтов, зарезервированных для массива. Иначе говоря, произведение операций LENGTHOF и TYPE.

Для определенной выше переменной TABLE оператор

```
MOV       ECX, SIZEOF TABLE   ; Загрузить в ECX число байтов
                              ; в таблице TABLE
```

загрузит 200 в регистр ECX.

11.2.5. Операции присваивания атрибутов

Операция указателя PTR нами уже рассматривалась. Она позволяет изменить у операнда атрибут типа (BYTE, WORD и др.).

Пример. Этой операцией можно воспользоваться для доступа к байтам в таблице слов. Если таблица определена следующим образом:

```
WORD_TABLE          DW    100 DUP (?)
```

то оператор

```
FIRST_BYTE          EQU   BYTE PTR WORD_TABLE
```

присвоит имя первому байту таблицы WORD_TABLE.

Напомним, что из-за обратного порядка следования байтов FIRST_BYTE будет младшим байтом в первом слове таблицы. Обращение к старшему байту будет FIRST_BYTE+1, обращение к младшему байту во втором слове таблицы FIRST_BYTE+2 и т.д.

Пример. Выше мы рассмотрели способ обращения к частям одной длинной переменной. Однако существует и обратная возможность: несколько коротких переменных можно загрузить в один длинный регистр. В приведенном ниже примере первое слово загружается в младшие 16 битов регистра EAX, а второе слово – в старшие 16 битов этого регистра. Такое возможно благодаря использованию операции DWORD PTR:

```
.DATA
WORDLIST WORD 5678H, 1234H
.CODE
MOV EAX, DWORD PTR WORDLIST ; EAX = 12345678H
```

Операция THIS создает адресный операнд с заданным типом (BYTE, WORD и др.) и определяет для него значение, равное адресу следующей доступной ячейки памяти.

Формат операции

THIS тип

Пример.

Последовательность операторов

```
FIRST_BYTE      EQU     THIS BYTE
WORD_TABLE      DW      100 DUP(?)
```

создает адресную константу FIRST_BYTE со значением, равным адресу переменной WORD_TABLE, и приписывает ей тип BYTE. Она выполняет ту же функцию, что и ранее рассмотренный оператор

```
FIRST_BYTE      EQU   BYTE PTR WORD_TABLE
```

11.3. Директива LABEL

Директива LABEL позволяет определить в программе метку и назначить ей нужный атрибут длины, не распределяя при этом физически память под переменную. После директивы LABEL можно указать любой стандартный атрибут длины, такой как BYTE, WORD, DWORD, QWORD или TBYTE.

Чаще всего директива LABEL используется для определения в программе дополнительных имен других размеров для переменных, размещенных в сегменте данных.

Пример.

```
.DATA
VAL16 LABEL WORD
VAL32 DWORD 12345678H
.CODE
MOV AX, VAL16 ; AX = 5678H
```

```
MOV DX, VAL16+2 ; DX = 1234H
```

Перед переменной VAL32 мы объявили метку VAL16 и присвоили ей атрибут длины WORD. Таким образом, мы просто назначили переменной VAL32 псевдоним VAL16.

Пример. Иногда возникает потребность создать одно длинное целое число на основе двух коротких целых чисел. Загрузим 32-разрядное значение, состоящее из двух 16-разрядных переменных, в регистр EAX:

```
.DATA
LONGVALUE LABEL DWORD
VAL1 WORD 5678H
VAL2 WORD 1234H
.CODE
MOV EAX, LONGVALUE ; EAX = 12345678H
```

12. Макросредства языка ассемблера

Макросредства – это основные инструменты модификации текста программы на этапе ее трансляции. Принцип работы макросредств основан на препроцессорной обработке, которая заключается в том, что текст, поступающий на вход транслятора, подвергается предварительному преобразованию.

12.1. Условные директивы

Условные директивы заставляют ассемблер либо транслировать, либо пропускать группу исходных команд и директив в зависимости от того, «истинно» или «ложно» в момент трансляции определенное условие.

Эта избирательная возможность «транслировать/не транслировать» позволяет:

– помещать в текст программы диагностические или специальные условия на случай тестовых прогонов,

– создавать специфические версии многоцелевой программы.

Для обеспечения условной трансляции порции текста программы необходимо перед ней поставить директиву IFx (x – модификатор, определяющий конкретную директиву), а после нее – директиву ENDIF.

Если условие в директиве IFx окажется «истинным», то операторы, расположенные между IFx и ENDIF, будут транслироваться.

Если условие окажется «ложным», то эти операторы будут пропущены и трансляция продолжится с оператора, следующего после ENDIF.

Сгруппируем директивы IFx в пары.

1. IFE и IF. Формат

```
IFE логическое выражение
IF логическое выражение
```

IFE дает значение «истинно», если логическое выражение ложно (равно 0); IF дает значение «истинно», если логическое выражение истинно (не равно 0).

2. IFDEF и IFNDEF. Формат

IFDEF идентификатор

IFNDEF идентификатор

IFDEF дает значение «истинно», если идентификатор определен или объявлен как внешний директивой EXTERN; IFNDEF дает значение «истинно» в противном случае.

3. IFIDN и IFDIF. Формат

IFIDN < строка1 >, < строка2 >

IFDIF < строка1 >, < строка2 >

Угловые скобки необходимы. Сравнение выполняется с учетом регистра символов.

IFIDN дает значение «истинно», если строки строка1 и строка2 идентичны; IFDIF дает значение «истинно», если они различаются.

4. IFIDNI и IFDIFI. Формат

IFIDNI < строка1 >, < строка2 >

IFDIFI < строка1 >, < строка2 >

Аналогичны предыдущим директивам, но сравнение выполняется без учета регистра символов.

Пример. Для включения диагностических процедур в тестовый прогон окаймите их директивами IF и ENDIF и определите константу DEBUG. Во время трансляции ассемблер проверит ее значение. Если это значение окажется нулевым, то диагностические процедуры будут пропущены.

Программа будет выглядеть следующим образом:

```
IF DEBUG
            ; Диагностические команды
    . . . . .
ENDIF
```

Если ранее в программу был помещен оператор

```
DEBUG EQU 1
```

то команды между директивами IF и ENDIF будут оттранслированы. А оператор

```
DEBUG EQU 0
```

предпишет ассемблеру пропустить команды, находящиеся между директивами IF и ENDIF.

Предположим, что условие оказалось «ложным». Альтернативную группу команд можно включить в программу с помощью признака альтернативы ELSE (иначе). В общем случае условные директивы имеют следующий формат:

```
IFx   [аргумент]
. . .          (Операторы для «истинного» значения условия)
ELSE
. . .          (Операторы для «ложного» значения условия)
```

```
ENDIF
```

Пример.

Признак альтернативы ELSE позволяет создать две версии программы, одна из которых будет выдавать приглашение к вводу и сообщения на английском языке, а другая – на русском.

Для этого можно определить константу LANGUAGE, которая позволит выбрать операторы, требуемые для соответствующего языка. Если ее значение равно 0, то ассемблер создает английскую версию; если равно 1 – русскую.

Связанный с текстом сообщений раздел программы может иметь следующий вид:

```
IFE   LANGUAGE
. . .          (Операторы английской версии)
ELSE
. . .          (Операторы русской версии)
ENDIF
```

Можно определить для ассемблера более двух вариантов трансляции с помощью вложения условных директив:

```
IFx
. . . . .
ELSE
      IFx
      . . . . .
      ENDIF
ENDIF
```

Эту последовательность условных директив можно заменить эквивалентной последовательностью дополнительных директив

```
IFx
. . . . .
ELSEIFx
. . . . .
ENDIF
```

Пример.

Предположим, что русская и английская версии нашей программы заработали. Теперь нужно сделать другие версии, которые выдают сообщения и приглашения к вводу на французском и немецком языках.

Для этого модифицируем программу так, чтобы ассемблер выбирал язык по значениям 0, 1, 2 и 3 константы LANGUAGE (соответственно английский, русский, французский или немецкий). В этом случае раздел сообщений может иметь следующий вид:

```
IFE   LANGUAGE
. . . . .          (Операторы для выдачи сообщений на английском языке)
ELSE
      IFE LANGUAGE–1
      . . . . . .  (Операторы для выдачи сообщений на русском языке)
```

```
        ELSE
                IFE LANGUAGE–2
                . . . . . (Операторы для выдачи сообщений на французском языке)
                ELSE
                . . . . . (Операторы для выдачи сообщений на немецком языке)
                ENDIF
        ENDIF
ENDIF
```

12.2. Макросы

Макросы подобны подпрограммам и представляют собой «мини-программы», которые можно несколько раз вставлять в исходные программы. Макрос представляет собой специальным образом описанную последовательностью операторов на языке ассемблера (команд и директив). Макросу дается имя, а затем в нужных местах программы выписывается ссылка на этот макрос (указывается его имя). Когда транслятор просматривает текст программы и встречает такую ссылку, то он вместо нее подставляет в окончательный текст программы сам макрос – соответствующий фрагмент программы.

При использовании макросов применяется следующая терминология. Описание макроса называется макроопределением, ссылка на макрос – макрокомандой, процесс замены макрокоманды на макрос – макроподстановкой или макрогенерацией, а результат такой подстановки – макрорасширением.

12.2.1. Сравнение макросов и процедур

И макросы, и процедуры предоставляют возможность краткой ссылки на часто используемую последовательность команд. Однако между ними существует и различие.

Коды команд процедуры входят в программу однократно, и микропроцессор передает им управление по мере необходимости.

Коды команд макроса могут встречаться в программе неоднократно. При исполнении программы микропроцессор исполняет инструкции макроса «непосредственно», не передавая управление в другое место памяти. Таким образом, макрокоманда представляет собой директиву ассемблера; она служит инструкцией ассемблеру, а не микропроцессору.

По сравнению с процедурами макросы имеют три преимущества:

1. Макросы динамичны. За счет изменения входных параметров макроса можно изменить не только объекты, которыми он манипулирует, но и выполняемые над ними действия. Напротив, в случае процедуры можно изменять только передаваемые ей данные. Это делает процедуры гораздо менее гибкими.

2. Применение макросов вместо процедур несколько увеличивает скорость исполнения программ, так как микропроцессору не надо отвлекаться на выполнение команд вызова процедуры и возврата из нее.

Однако макросы имеют основной недостаток, которого лишены процедуры: при их применении объектные программы становятся длиннее, поскольку макросы расширяются при каждом их появлении и память заполняется повторяющимися последовательностями команд.

12.2.2. Определение и вызов макросов

Определение макроса (макроопределение) имеет три части.

1. Заголовок – директива MACRO. Формат:

имя MACRO [список формальных параметров]

Директива присваивает имя последовательности команд и директив языка ассемблера. В поле метки указано имя макроса, а в поле операнда – необязательный список формальных параметров. В списке формальных параметров указываются переменные – входные параметры, которые можно изменять при каждом вызове макроса.

2. Тело – последовательность команд и директив ассемблера, которые задают действия, выполняемые макросом.

3. Концевик – директива ENDM, которая отмечает конец макроопределения.

Для того чтобы использовать описанный макрос, его нужно «активизировать» с помощью макрокоманды. Для этого в нужном месте исходного кода программы на основе текста заголовка макроопределения указывается следующая синтаксическая конструкция:

имя [список фактических параметров]

Пример. Макрос для сложения значений размером в слово:

```
ADD_WORDS    MACRO    TERM1, TERM2, SUM
MOV          AX, TERM1
ADD          AX, TERM2
MOV          SUM, AX
ENDM
```

Для ассемблера безразлично, что будет указано в качестве операндов макрокоманды: имена регистров, ячейки памяти или непосредственные значения. Конечно, непосредственное значение нельзя использовать в качестве операнда SUM. Если окончательный результат имеет допустимую форму, то ассемблер выполняет подстановку операндов.

Пример. Можно сложить содержимое двух ячеек памяти с помощью макрокоманды

```
ADD_WORDS  PRICE, TAX, COST
```

Вместо макрокоманды ассемблер вставит в программу следующие команды:

```
MOV          AX, PRICE
```

```
ADD     AX, TAX
MOV     COST, AX
```

Можно сложить содержимое двух регистров, указав макрокоманду

```
ADD_WORDS   BX, CX, DX
```

На этот раз ассемблер вставит команды

```
MOV     AX, BX
ADD     AX, CX
MOV     DX, AX
```

При создании сложных макросов можно воспользоваться уже готовыми макросами, тем самым уменьшив себе объем работы. Макрос, который вызывается из другого макроса, называется вложенным. Использование вложенных макросов не приводит к каким-либо дополнительным накладным расходам, поскольку транслятор всегда заменит их фрагментом кода, точно так же, как если бы это был всего один макрос. Параметры, переданные во внешний макрос, можно непосредственно передать во внутренний макрос.

При создании макроса можно назначить его параметрам стандартные значения. Тогда, если при вызове макроса значение некоторого параметра будет пропущено, вместо него подставляется стандартное значение. Синтаксис инициализации параметров имеет следующий вид:

Имя_параметра := < значение >

Кроме того, можно указать, что требуется обязательное явное задание фактического параметра при вызове макроса, для этого формальный параметр задается в виде

Имя_параметра : REQ

12.2.3. Директива LOCAL

Если некоторый макрос вызывается в программе несколько раз, то при макрогенерации один и тот же идентификатор будет определен несколько раз, что, естественно, транслятор посчитает ошибкой.

Таким образом, если макрос содержит помеченные команды или директивы, то необходимо указать ассемблеру, чтобы он изменял метки в каждом макрорасширении. Сообщает ассемблеру, какие метки должны изменяться в каждом макрорасширении, директива LOCAL. Директива имеет формат

LOCAL [список формальных параметров]

Она заставляет ассемблер создать уникальное имя для каждой метки из списка формальных параметров и подставить это имя при каждом вхождении метки в макрорасширение. 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.

Пример.

Рассмотрим макрос, который заставляет микропроцессор ожидать, пока значение COUNT не уменьшится до нуля.

```
WAIT    MACRO   COUNT
        LOCAL   NEXT
        PUSH    ECX             ; Сохранить текущее значение ECX
        MOV     ECX, COUNT      ; Поместить счетчик в ECX
NEXT:   LOOP    NEXT    ; Повторять, пока счетчик не обратится в 0
        POP     ECX     ; Восстановить исходное значение ECX
ENDM
```

Указание метки NEXT в операторе LOCAL позволяет нам пользоваться этим макросом в программе более одного раза.

Оператор LOCAL должен следовать непосредственно за оператором MACRO. Если требуется несколько операторов LOCAL, то они должны быть первыми операторами в макроопределении. Они должны предшествовать любому другому оператору и даже комментариям.

Указание метки в операторе LOCAL также означает, что можно использовать такую же метку в других макросах. Ассемблер дает ей новое внутреннее имя при всяком макрорасширении; таким образом, дублирование меток исключается.

12.2.4. Условные директивы IFB и IFNB

В макросах могут использоваться все условные директивы, которые нами уже рассмотрены, а также две специальные директивы – IFB и IFNB.

В макрокоманде можно указать меньше параметров, чем имеется в макроопределении. В этом случае неуказанные параметры считаются пустыми.

Директива IFB (если пустой). Формат:

```
IFB < аргумент >
. . .
ENDIF
```

Действия выполняются, если аргумент пуст. Угловые скобки обязательны.

Директива IFB позволяет указать альтернативные способы обработки пустых параметров. Обычно она используется, чтобы заставить макрос завершиться раньше в том случае, если какие-либо необходимые ему параметры отсутствуют.

Директива IFNB (если не пустой). Формат:

```
IFNB < аргумент >
. . .
ENDIF
```

Действия выполняются, если аргумент не пуст. Угловые скобки обязательны.

Когда ассемблер обнаруживает директиву IFNB, то он транслирует связанные с ней команды только в том случае, если пользователь дал значение параметру; в противном случае он пропускает их.

Пример. Макрос, считывающий имя, отчество и фамилию, может быть рассчитан на то, что он будет вызван следующим образом:

GET_NAME NAME1, NAME2, NAME3

Макрос GET_NAME должен включать команды, которые получают соответственно имя, отчество, а затем фамилию. Однако отчество не всегда известно, и поэтому надо предусмотреть возможность его отсутствия.

Это можно сделать с помощью директивы IFNB. Она даст возможность включить команды получения отчества только в том случае, если пользователь задал соответствующий параметр. Следовательно, макроопределение GET_NAME должно иметь следующий общий вид:

```
GET_NAME MACRO NAME1, NAME2, NAME3
. . .                                  (Эти команды считывают имя)
IFNB    < NAME2 >
. . .                                  (Эти команды считывают отчество)
ENDIF
. . .                                  (Эти команды считывают фамилию)
ENDM
```

После этого можно воспользоваться формой оператора вида

GET_NAME Джон,, Браун

12.2.5. Задание макросов в исходных программах

Существуют два способа использования макросов: их можно задавать в начале программы или считывать в программу из отдельного файла.

Если макрос специфичен и требуется только для одной программы, то его можно задать в тексте программы, а затем вызывать по мере необходимости.

Чтобы макросы были доступны многим программам, макроопределения удобно помещать в отдельный файл. После того как такой файл создан, его содержимое можно включать в любую исходную программу. Тем самым все макросы становятся доступными для этой программы.

Для включения файла с макроопределениями (и, вообще, любого файла) в исходную программу необходимо использовать директиву INCLUDE.

Директива имеет формат:

INCLUDE файл

Она вставляет на время трансляции содержимое указанного файла в текущий файл исходной программы.

Пример.

INCLUDE MACRO.ASM

Пользование файла с макроопределениями имеет тот недостаток, что при указании его имени в директиве INCLUDE ассемблер считывает все заданные в нем макроопределения. Однако не все из них требуются в данной программе. В результате рабочая область заполняется ненужной информацией. Чтобы избежать этого, можно удалить из памяти ненужные макроопределения.

Для этого непосредственно за директивой INCLUDE надо поместить директиву PURGE, в которой перечислить имена подлежащих удалению макроопределений.

Пример.

```
INCLUDE MACRO.ASM
PURGE   MAC1, MAC2, MAC3
```

12.3. Блоки повторения

Иногда в некотором месте программы приходится выписывать несколько раз подряд один и тот же (или почти один и тот же) фрагмент, и хотелось бы, чтобы мы сами выписывали этот фрагмент только раз, а транслятор размножал его нужное число раз. Такая возможность предусмотрена в языке ассемблера, и реализуется она с помощью блоков повторения.

Блок повторения имеет такую же структуру, как макроопределение:

1) заголовок,

2) тело,

3) концевик ENDM.

Здесь тело – любое число любых инструкций (в частности, ими могут быть снова блоки повторения), а ENDM – директива, указывающая на конец тела и всего блока повторений.

При дублировании тело может выписываться без каких-либо изменений, а может копироваться и с модификациями. Как именно происходит дублирование, сколько копий создается – все это зависит от заголовка блока. Имеется несколько разновидностей заголовка, определяемых разными директивами повторения. Рассмотрим их.

Директивы повторения REPEAT и REPT эквивалентны и имеют формат:

```
REPEAT выражение
. . .
ENDM
```

Они получают свой счетчик числа повторений из выражения, указанного в поле операнда.

Пример. Следующее макроопределение зарезервирует LENGTH байтов памяти и присвоит им в качестве начальных значений числа от 1 до LENGTH:

```
ALLOCATE        MACRO   TLABEL, LENGTH
        TLABEL    EQU  THIS BYTE
        VALUE = 0
        REPEAT    LENGTH
                VALUE = VALUE + 1
                DB VALUE
        ENDM
ENDM
```

Обратите внимание на то, что здесь нам понадобились два оператора ENDM: одним отмечен конец действия директивы REPEAT, а другим – конец макроопределения.

После задания макроопределения ALLOCATE им можно воспользоваться для создания 40-байтовой таблицы TABLE1, указав такую последовательность операторов:

```
.DATA
ALLOCATE        TABLE1, 40
. . . . .
```

Директива WHILE также применяется для повторения определенного количества раз некоторой последовательности строк. Эта директива имеет формат

```
WHILE выражение
. . . . .
ENDM
```

При использовании директивы WHILE транслятор будет повторять тело блока до тех пор, пока значение операнда «выражение» не станет равным нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение выражения в процессе расширения блока повторения должно подвергаться изменению внутри последовательности строк).

Директивы повторения FOR и IRP эквивалентны и имеют формат:

```
FOR  параметр, < список аргументов >
. . .
ENDM
```

Они заставляют ассемблер повторять находящиеся между директивой FOR/IRP и ENDM команды и директивы по одному разу для каждого аргумента списка. При каждом повторении производится подстановка очередного аргумента вместо каждого вхождения параметра в тело блока повторения.

Заметим, что фактические параметры не должны содержать запятые, точки с запятой и угловые скобки (<, >).

Пример. Последовательность операторов

```
FOR  VALUE, < 1, 2, 3, 5, 7 >
DW VALUE*VALUE*VALUE
ENDM
```

создаст таблицу из 5 слов, содержащую кубы первых 5 простых чисел.

Директива FORC/IRPC похожа на директиву FOR/IRP, но ее аргументами являются не числа, а строковые переменные. Формат:

```
FORC параметр, строка
. . .
ENDM
```

Она заставляет ассемблер повторять находящиеся между директивами FORC/IRPC и ENDM команды и директивы по одному разу для каждого

символа строки. При каждом повторении производится подстановка очередного символа строки вместо каждого вхождения параметра в тело блока повторения.

Заметим, что во втором операнде нельзя указывать пробелы и точки с запятой.

Пример. Последовательность операторов

```
FORC    CHAR, 01234
DB      CHAR
ENDM
```

создает в памяти 5-байтовую таблицу, содержащую числа от 0 до 4.

12.4. Директивы EXITM и GOTO

Директива EXITM заставляет ассемблер завершить расширение макроса или блока повторения до их окончания. EXITM часто используется вместе с условной директивой.

Пример. Если макрос не может быть выполненным в ситуации, когда при его вызове не указан параметр param, то вставим в его начало последовательность

```
IFB < param >
EXITM
ENDIF
```

Директива

GOTO метка

переводит процесс расширения макроопределения или блока повторения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения или блока повторения. Метка, на которую передается управление, имеет специальный формат:

:метка

и является единственной инструкцией в строке.

12.5. Макрофункции

Заметим вначале, что употребляемый нами термин макрос по сути эквивалентен термину макропроцедура, используемому в документации к компилятору MASM для обозначения макросов, не возвращающих значения. Кроме макропроцедур, существуют также макрофункции, возвращающие значение.

Макрофункции практически ничем не отличаются от макропроцедур. Как и при обработке макропроцедуры, при обработке макрофункции вместо ее имени подставляется набор ассемблерных директив и команд. Единственное отличие макрофункции от макропроцедуры состоит в том, что она всегда

возвращает константу (целочисленную или строковую) в вызвавшую ее программу с помощью директивы EXITM.

В приведенном ниже примере макрокоманда IS_DEFINED возвращает истинное значение (число –1), если указанный в качестве параметра символ определен. В противном случае возвращается ложное значение (число 0):

```
IS_DEFINED MACRO SYMBOL
        IFDEF SYMBOL
                EXITM <–1>
        ELSE
                EXITM <0>
        ENDIF
ENDM
```

При вызове макрофункции список ее аргументов необходимо заключить в круглые скобки. Например, при вызове макрокоманды IS_DEFINED, ей передается имя символа WHITE_PEN, заключенное в скобки, причем символ может быть либо определен, либо нет:

```
MOV EAX, IS_DEFINED(WHITE_PEN)
```

В данном случае, если до вызова макрокоманды IS_DEFINED символ WHITE_PEN уже был каким-то образом определен в программе, компилятор ассемблера сгенерирует команду

```
MOV EAX, –1
```

Если же символ WHITE_PEN не был определен, компилятор ассемблера сгенерирует команду

```
MOV EAX, 0
```

12.6. Макрооперации

Макроассемблер предоставляет возможность пользоваться рядом операций в макроопределениях и блоках повторения.

Операция &. Формат:

&параметр&

Эта операция предписывает транслятору заменить имя параметра на его значение.

Назначение операции – указать границу формального параметра, выделить его из окружающего текста, при этом в окончательный текст программы она не попадает. Если знак & поставить не около параметра, то он будет просто опущен.

Операция позволяет задавать модифицируемые метки и операнды.

Пример. Следующее макроопределение выполнит команду PUSH EAX, если его вызвать как PUSHREG A:

```
PUSHREG   MACRO   LETTER
        PUSH    E&LETTER&X
ENDM
```

Пример. Следующее макроопределение образует таблицу байтов с заданным именем и заданной длиной:

```
DEF_TABLE      MACRO     SUFFIX, LENGTH
        TABLE&SUFFIX  DB  LENGTH DUP(?)
ENDM
```

Если в программе будет указан оператор

```
DEF_TABLE A, 5
```

то ассемблер превратит его в оператор

```
TABLEA    DB    5       DUP(?)
```

Операция & используется не только тогда, когда формальный параметр «сливается» с соседними именами и числами, но и когда его надо указать внутри строк. Дело в том, что транслятор игнорирует вхождения формального параметра в строки, и чтобы обратить его внимание на эти вхождения, перед параметром в строках надо ставить знак & (а если не ясна его правая граница, то & надо указывать и после параметра).

Пример. Блок

```
FORC A,"<
DB   'A,&A,&A&B'
ENDM
```

приведет к генерации следующих строк:

```
DB   'A,", "B'
DB   'A,<,<B'
```

Операция !. Формат:

!символ

Используется в аргументе для указания ассемблеру, что символ надо использовать как литерал, а не как имя, т.е. вне зависимости от любого специального значения, которое он может иметь в противном случае.

Пример. Блок

```
FOR X,<A!>B,Привет!, ПК!!>
DB '&X'
ENDM
```

приведет к генерации следующих строк:

```
DB   'A>B'
DB   'Привет,  ПК!'
```

Операция %. Формат:

%текст

Трактует текст как выражение, вычисляет его значение и заменяет текст полученным результатом. Текст может быть или числовым выражением, или уравнением.

Пример. Блок

```
K EQU 4
. . . . . .
FOR A,<K+1,%K+1,W%K+1>
DW A
```

```
ENDM
```

приведет к генерации следующих строк:

```
DW K+1
DW  5
DW W5
```

Операция ;;. Формат:

```
;; комментарий
```

Заставляет ассемблер опустить комментарии при расширении макроопределения или блока повторения. Без комментариев окончательная программа будет занимать меньше памяти, и поэтому будет транслироваться быстрее.

Также в макросах и блоках повторения широко используется операция < >, которую мы уже рассматривали.

При ее обработке транслятор воспринимает заключенный в угловые скобки текст как один элемент и не интерпретирует его содержимое, например, не разбивает список аргументов, разделенных запятой, на отдельные элементы. Данная операция используется, если в строке содержатся специальные символы, такие как запятая, знак процента, амперсанд, точка с запятой, которые не должны быть восприняты транслятором как символы-разделители или знаки операции.

Пример. Блок

```
FOR VAL,<<1,2>,3>
DB VAL
ENDM
```

приведет к генерации следующих строк:

```
DB  1,2
DB  3
```

13. Цепочечные команды

13.1. Общая характеристика цепочечных команд

Система команд микропроцессора имеет группу команд, позволяющих производить действия над блоками элементов (цепочками). Логически цепочки представляют собой последовательности элементов с любыми значениями (символы, числа и прочее), хранящихся в соседних ячейках памяти в виде двоичных кодов. Единственное ограничение состоит в том, что размеры элементов в цепочках фиксированы значениями байт, слово или двойное слово.

Цепочечные команды (называемые также командами обработки строк) обеспечивают возможность выполнения ряда операций-примитивов, обрабатывающих цепочки поэлементно.

Каждая операция-примитив реализуется в процессоре тремя командами без операндов. Каждая из этих команд работает с соответствующим размером элемента – байтом, словом или двойным словом. Кроме того, имеется одна

команда с явным указанием операндов, служащих только для определения типов операндов. После того как выяснен тип элементов цепочек по их описанию в сегменте данных, ассемблером генерируется одна из трех машинных команд.

Все цепочечные команды обрабатывают только один (текущий) элемент цепочки и автоматически передвигаются к следующему элементу.

Адреса текущих обрабатываемых элементов всегда находятся в регистрах ESI для строки, участвующей в команде в качестве источника, и EDI для строки, участвующей в команде в качестве приёмника.

Переход к следующему элементу производится увеличением или уменьшением содержимого регистров ESI, EDI.

Возможны два направления обработки цепочки:

– от начала цепочки к ее концу, то есть в направлении возрастания адресов;

– от конца цепочки к началу, то есть в направлении убывания адресов.

Направление определяется значением флага направления DF в регистре EFLAGS (FLAGS):

– если DF = 0, то значения индексных регистров ESI, EDI будут автоматически увеличиваться цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;

– если DF = 1, то значения индексных регистров ESI, EDI будут автоматически уменьшаться цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

Состоянием флага DF можно управлять с помощью двух команд, не имеющих операндов:

– CLD – очистить флаг направления (команда сбрасывает флаг направления DF в 0);

– STD – установить флаг направления (команда устанавливает флаг направления DF в 1).

Количество байтов, на которые изменяется содержимое регистров ESI, EDI определяется типом элементов строки или кодом цепочечной команды: на 1 при обработке байтов, на 2 при обработке слов и на 4 при обработке двойных слов.

Поскольку адреса текущих обрабатываемых элементов извлекаются командами из регистров ESI и EDI, перед выполнением команд необходимо нужные адреса в эти регистры поместить. Обычно перед началом обработки строк в указанные регистры заносятся адреса первых обрабатываемых элементов. Для этого могут использоваться команда LEA или операция OFFSET.

Пример. Если источником является строка SOURCE, а приёмником – строка DEST, то необходимы действия

```
LEA     ESI, SOURCE
LEA     EDI, DEST
```

Аналогичный результат получается при выполнении команд

```
MOV     ESI, OFFSET SOURCE
MOV     EDI, OFFSET DEST
```

13.2. Префиксы повторения

Логически к командам обработки строк нужно отнести и так называемые префиксы повторения.

Префиксы повторения указываются перед нужной цепочечной командой. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле. Число повторений заносится в регистр ECX. После очередного выполнения команды значение в регистре ECX уменьшается на единицу. Решение об очередном выполнении цепочечной команды принимается по состоянию регистра ECX и, возможно, по флагу нуля ZF.

Префикс повторения REP (REPeat) заставляет команды выполняться, пока содержимое в ECX не станет равным 0.

Префиксы повторения REPE (REPeat while Equal) и REPZ (REPeat while Zero) являются синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое ECX не равно 0 и флаг ZF равен 1. Как только одно из этих условий нарушается, управление передается следующей команде программы. Префиксы REPNE и REPNZ, также являющиеся синонимами, заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое ECX не равно 0 и флаг ZF равен 0. При нарушении одного из этих условий работа команды прекращается.

13.3. Команды пересылки цепочек

Команда MOVS копирует байт, слово или двойное слово из одной области памяти (цепочки) в другую. Она имеет формат

```
MOVS    адрес_приёмника, адрес_источника
```

Команда копирует элемент цепочки-источника (длиной в байт, слово или двойное слово), который располагается по адресу, соответствующему содержимому регистра ESI, в элемент цепочки-приёмника, который располагается по адресу, соответствующему содержимому регистра EDI.

После пересылки элемента команда MOVS изменяет указатели строки-источника ESI и строки-приёмника EDI так, чтобы они указывали на следующие элементы цепочки. Если флаг DF равен 0, то микропроцессор увеличивает значения регистров ESI и EDI после пересылки и тем самым адресуется к следующим элементам памяти. Если флаг DF равен 1, то микропроцессор уменьшает значения регистров ESI и EDI после пересылки и тем самым адресуется к предыдущему элементу памяти.

Команды MOVSB, MOVSW и MOVSD выполняют аналогичные действия. При этом первая команда оперирует элементами длиной в байт,

вторая – элементами длиной в слово, третья – элементами длиной в двойное слово.

Последовательность действий, которые нужно выполнить в программе для того, чтобы переслать цепочку элементов из одной области памяти в другую с помощью команды MOVS следующая.

1. Установить значение флага DF в зависимости от того, в каком направлении будут обрабатываться элементы цепочки – в направлении возрастания или убывания адресов.

2. Загрузить адреса первых обрабатываемых элементов цепочек в регистры ESI и EDI.

3. Загрузить в регистр ECX количество обрабатываемых элементов.

4. Записать команду MOVS с префиксом REP.

Пример. Пусть необходимо скопировать 100 элементов из строки SOURCE в строку DEST, обе строки описаны в сегменте данных как массивы байтов.

```
CLD
LEA     ESI, SOURCE
LEA     EDI, DEST
MOV     ECX, 100
REP     MOVSB
```

13.4. Команды сравнения цепочек

Команды, реализующие операцию-примитив сравнения цепочек, производят сравнение элемента цепочки-источника с элементом цепочки-приёмника. Синтаксис команды CMPS:

```
CMPS    адрес_приёмника, адрес_источника
```

Алгоритм работы команды CMPS заключается в вычитании элемента цепочки-приёмника, который располагается по адресу, соответствующему содержимому регистра EDI, из элемента цепочки-источника, который располагается по адресу, соответствующему содержимому регистра ESI. Команда производит вычитание элементов, не записывая при этом результата, и устанавливает флаги ZF, SF и OF. Так, если флаг нуля ZF равен 1, то элементы оказались равны, а если ZF равен 0, то, наоборот, не равны. После выполнения команды регистры ESI и EDI изменяются так, чтобы указывать на следующие элементы цепочки.

Заметим, что в отличие от команды CMPS команда CMP вычитает операнд-источник из операнда-приёмника. Это означает, что команды условной передачи управления, указываемые после команды CMPS, должны отличаться от тех, что в аналогичной ситуации следовали бы за командой CMP. А именно, в таблицах, приведенных для команды CMP, слова «источник» и «приёмник» необходимо поменять местами.

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

командой CMPS определить префикс повторения. В данном случае префикс REP не имеет смысла, поскольку при его применении во флагах будет возвращен лишь результат сравнения двух последних элементов. С командой CMPS удобно использовать префиксы повторения REPE/REPZ или REPNE/REPNZ для поиска первой пары несовпадающих или, наоборот, совпадающих элементов.

Повторение операций сравнения может завершиться в двух случаях:

1) значение регистра ECX стало равным 0,

2) флаг ZF стал равен 0 (префикс REPE), либо 1 (префикс REPNE).

При этом может понадобиться узнать, какой из случаев имел место. Выяснить причину прекращения сравнений можно путем использования после команды CMPS одной из команд условной передачи управления (JECXZ, JE, JNE и т.д.).

Поскольку при выполнении команды CMPS значения регистров ESI и EDI изменяются, по завершении цикла они указывают не на искомые элементы в цепочке, а на следующие, которые должны были бы сравниваться.

Пример. Команды

```
CLD
MOV ECX, 100
REPE CMPS DEST, SOURCE
JA L1
```

будут сравнивать до 100 пар элементов строк SOURCE и DEST с целью найти два несовпадающих элемента. Если из этих элементов больше тот, что находится в строке SOURCE, произойдет переход на метку L1.

А команды

```
CLD
MOV     ECX, 100
REPNE   CMPS    DEST, SOURCE
```

будут сравнивать до 100 пар элементов строк SOURCE и DEST с целью найти два совпадающих элемента.

Ассемблер преобразует команду CMPS в команду CMPSB (при сравнении байтов), в команду CMPSW (при сравнении слов) или в команду CMPSD (при сравнении двойных слов) в зависимости от описания операндов.

13.5. Команды сканирования цепочек

Команды, реализующие операцию-примитив сканирования цепочек, производят поиск некоторого значения в цепочке. Искомое значение предварительно должно быть помещено в один из регистров AL, AX или EAX (в зависимости от размера элемента цепочки).

Синтаксис команды SCAS:

```
SCAS    адрес_приёмника
```

Принцип поиска здесь тот же, что и в команде сравнения CMPS, то есть вычитание элемента цепочки-приёмника, который располагается по адресу, соответствующему содержимому регистра EDI, из содержимого регистра

AL/AX/EAX. В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются. После выполнения команды регистр EDI изменяется так, чтобы указывать на следующий элемент цепочки.

Так же как и в случае команды CMPS, с командой SCAS удобно использовать префиксы REPE/REPZ или REPNE/REPNZ. Если искомый элемент обнаружен, то смещение адреса следующего за ним элемента содержится в регистре EDI. С помощью команд условного перехода можно определить, найден такой элемент или не найден.

Ассемблер преобразует команду SCAS в команду SCASB (при поиске байта), в команду SCASW (при поиске слова) или в команду SCASD (при поиске двойного слова).

Пример.

```
.DATA
ALPHA DB "ABCDEFGH"
. CODE
MOV EDI, OFFSET ALPHA
MOV AL,'F'
MOV ECX, 8
CLD
REPNE SCASB
JNZ QUIT
DEC EDI
```

Здесь при нахождении в строке ALPHA символа F в регистре EDI будет содержаться его адрес (после выполнения команды декремента). Если же искомого символа нет в исходной строке, то будет осуществлен переход на метку QUIT.

13.6. Команды загрузки и сохранения цепочек

Данные команды служат для того, чтобы загрузить элемент цепочки в регистр или изменить его.

Операция-примитив загрузки элемента цепочки в регистр позволяет скопировать элемент цепочки в регистр AL, AX, EAX (в зависимости от размера элемента цепочки).

Синтаксис команды LODS:

```
LODS        адрес_источника
```

Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому регистра ESI, и поместить его в регистр AL/AX/EAX. После выполнения команды регистр ESI изменяется так, чтобы указывать на следующий элемент цепочки. Префикс повторения с этой командой обычно не используется.

Ассемблер преобразует команду LODS в команду LODSB (при работе с цепочкой байтов), в команду LODSW (при работе с цепочкой слов) или в команду LODSD (при работе с цепочкой двойных слов).

Операция-примитив сохранения содержимого регистра в цепочке позволяет произвести действие, обратное действию команды LODS, то есть перенести значение из регистра в элемент цепочки. Синтаксис команды STOS:

STOS адрес_приёмника

Команда пересылает элемент из регистра AL/AX/EAX в элемент цепочки по адресу, соответствующему содержимому регистра EDI. После выполнения команды регистр EDI изменяется так, чтобы указывать на следующий элемент цепочки.

Ассемблер преобразует команду STOS в команду STOSB (при работе с цепочкой байтов), в команду STOSW (при работе с цепочкой слов) или в команду STOSD (при работе с цепочкой двойных слов).

Будучи повторяемой, команда STOS удобна для заполнения строки заданным значением.

Пример. В приведенном ниже фрагменте кода выполняется инициализация строки STRING значением 0FFH.

```
.DATA
STRING DB 100 DUP(?)
.CODE
MOV     AL, 0FFH
MOV     EDI, OFFSET STRING
MOV     ECX, 100
CLD
REP     STOSB
```

14. Сложные структуры данных

14.1. Структуры

Структуры в языках программирования высокого уровня используются очень давно, практически с самого момента их появления. Структуры в языке ассемблера практически аналогичны структурам, используемым в языках высокого уровня C или C++. Поэтому можно легко преобразовать структуры, описывающие параметры библиотечных функций Windows API, и сделать так, чтобы с ними мог работать компилятор ассемблера. При использовании для отладки программ отладчика Microsoft Visual Studio во время выполнения программы можно легко контролировать значения полей структуры по их именам, т.е. почти так же, как и в языке высокого уровня.

Для использования структуры в программе необходимо выполнить три действия:

1) определить структуру,

2) объявить переменные структурного типа, которые будем называть структурными переменными,

3) написать команды, которые обращаются к структурным переменным или к полям структуры (что одно и то же).

14.1.1. Определение структуры

Структура определяется с помощью директив STRUCT и ENDS. Внутри структуры ее поля определяются точно так же, как и обычные переменные в программе. Общий синтаксис определения структуры следующий:

имя STRUCT
; объявление полей
имя ENDS

При объявлении полей структуры для них можно указать один из инициализаторов, который определяет их значение по умолчанию. Возможные типы инициализаторов:

1. Неопределенное значение. Если значение поля заранее не определяется, то структурная переменная описывается с помощью спецификатора «?».

2. Строковое значение. Поле структуры может содержать строку символов, которая заключается в кавычки.

3. Целочисленное значение. Чтобы присвоить полю структуры целочисленное значение, воспользуйтесь целочисленной константой или выражением.

4. Массивы. Если поле структуры является массивом, то для его инициализации используется операция DUP.

Пример. Рассмотрим структуру COORD, которая используется в библиотечных функциях Windows API для представления координат элемента X и Y на экране. Поле структуры под названием X, располагается со смещением 0 относительно начала структуры, а поле Y – со смещением 2, как показано ниже.

```
COORD STRUCT
        X WORD ? ; Смещение +00H
        Y WORD ? ; Смещение +02H
COORD ENDS
```

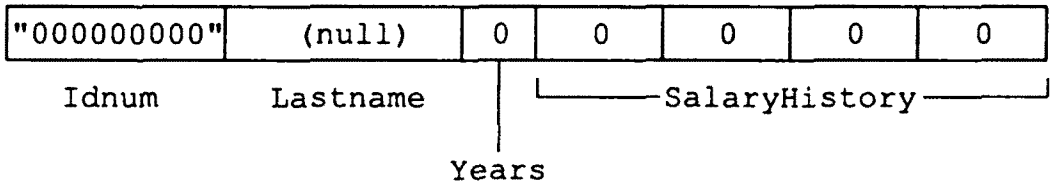
Пример. Рассмотрим определение структуры EMPLOYEE, которая описывает информацию о сотруднике и содержит такие поля: идентификатор сотрудника, фамилию, дату найма (год) и средний размер заработка по годам за последние 4 года.

Ниже приведено определение структуры.

```
EMPLOYEE STRUCT
        IDNUM           BYTE        "00000000"
        LASTNAME        BYTE        30 DUP(0)
```

```
    YEARS               WORD    0
    SALARYHISTORY       DWORD   0,0,0,0
EMPLOYEE ENDS
```

На рисунке показано, как эта структура размещается в оперативной памяти компьютера.



14.1.2. Объявление структурных переменных

После определения структуры в программе нужно создать один или несколько ее экземпляров – структурных переменных, и при необходимости присвоить им начальные значения. Если при определении структурной переменной используются пустые угловые скобки <>, ассемблер присваивает ее полям значения, заданные по умолчанию с помощью инициализаторов во время определения самой структуры. Чтобы задать элементам структуры новые исходные значения, укажите их в угловых скобках. Ниже приведены примеры определения экземпляров структур COORD и EMPLOYEE, в которых применена как явная, так и неявная инициализация их элементов.

```
.DATA
POINT1 COORD <5,10>
POINT2 COORD <>
WORKER EMPLOYEE <>
```

Существует также возможность при создании структурной переменной переопределить стандартное значение только некоторых полей. В приведенном ниже примере переопределяется поле IDNUM структурной переменной PERSON1, имеющей тип EMPLOYEE.

```
PERSON1 EMPLOYEE <"555223333">
```

При инициализации полей структуры вместо угловых скобок можно также использовать фигурные скобки, как показано ниже.

```
PERSON2 EMPLOYEE {"555223333"}
```

Если значение инициализатора для строкового поля короче самого поля, оставшиеся позиции заполняются пробелами.

Если при инициализации структурной переменной нужно пропустить несколько полей, укажите вместо них запятую. Например, в приведенном ниже фрагменте кода инициализация поля IDNUM пропускается, а полю LASTNAME присваивается значение "ИВАНОВ".

```
PERSON3 EMPLOYEE <, "ИВАНОВ">
```

Если поле структурной переменной является массивом, то для инициализации части или всех элементов массива используется оператор DUP.

Если значение инициализатора короче размера поля структуры, оставшиеся его позиции заполняются нулями. Ниже в качестве примера мы проинициализировали первые два элемента поля SALARYHISTORY, а оставшиеся два элемента заполнили нулями.

```
PERSON4 EMPLOYEE <,,, 2 DUP(200000)>
```

Можно создать массив структур, т.е. такой массив, каждый элемент которого является структурой. В приведенном ниже фрагменте кода каждому элементу массива ALLPOINTS присваивается значение<0, 0>:

```
NUMPOINTS = 3
ALLPOINTS COORD NUMPOINTS DUP(<0,0>)
```

14.1.3. Обращение к структурным переменным

При непосредственном обращении к отдельным полям структуры в качестве префикса должно быть указано имя структурной переменной или самой структуры.

Пример. Обращение к полям структурной переменной WORKER.

```
.DATA
WORKER EMPLOYEE <>
.CODE
MOV DX, WORKER.YEARS
MOV WORKER.SALARYHISTORY, 200000 ; сумма заработка за 1-й год
MOV WORKER.SALARYHISTORY+4, 300000;сумма заработка за 2-й год
MOV EDX, OFFSET WORKER.LASTNAME
```

Для определения размеров структурной переменной и ее отдельных полей используются операторы TYPE и SIZEOF.

Пример. Рассмотрим структуру EMPLOYEE (в комментариях указан размер в байтах).

```
EMPLOYEE STRUCT
    IDNUM           BYTE    "00000000"      ; 9
    LASTNAME        BYTE    30 DUP(0)       ; 30
    YEARS           WORD    0               ; 2
    SALARYHISTORY   DWORD   0,0,0,0         ; 16
EMPLOYEE ENDS                               ; итого 57 байтов
```

При использовании оператора определения данных

```
.DATA
WORKER EMPLOYEE <>
```

каждое из приведенных ниже выражений вернет одно и то же значение:

```
TYPE EMPLOYEE       ; 57 (длина типа)
SIZEOF EMPLOYEE     ; 57 (число байтов)
SIZEOF WORKER       ; 57
```

Для отдельных полей можно записать следующие выражения:

```
TYPE EMPLOYEE.SALARYHISTORY         ; 4
LENGTHOF EMPLOYEE.SALARYHISTORY     ; 4
```

```
TYPE EMPLOYEE.YEARS                    ; 2
```

Для обращения к полям структурной переменной удобно применять косвенную адресацию, если в один из регистров общего назначения (например в ESI) загрузить ее адрес. Благодаря этому можно без проблем передать адрес структурной переменной в процедуру или обрабатывать элементы массива структур. При косвенном обращении к полям структурной переменной используется оператор PTR, как показано ниже:

```
MOV ESI, OFFSET WORKER
MOV AX, (EMPLOYEE PTR [ESI]).YEARS
```

14.1.4. Вложенные структуры

В языке ассемблера можно создавать сложные определения, состоящие из вложенных друг в друга структур. При этом поля внешней структуры сами являются структурами.

Пример. Структуру, определяющую координаты левого верхнего и правого нижнего углов прямоугольника (назовем ее RECTANGLE) можно определить в виде совокупности двух структур типа COORD, как показано ниже.

```
RECTANGLE STRUCT
        UPPERLEFT COORD <>
        LOWERRIGHT COORD <>
RECTANGLE ENDS
```

После этого можно объявлять структурные переменные типа RECTANGLE, причем значение полей вложенных структур типа COORD можно оставить либо неопределенными, либо явно задать значения координат, как показано ниже. В следующем примере используются все возможные формы записи.

```
RECT1     RECTANGLE    < >
RECT2     RECTANGLE    { }
RECT3     RECTANGLE    { {10,10}, {50,20} }
RECT4     RECTANGLE    < <10,10>, <50,20> >
```

Ниже приведен пример команды, в которой выполняется непосредственное обращение к одному из полей вложенной структуры.

```
MOV RECT1.UPPERLEFT.X, 10
```

Для обращения к вложенному полю структуры можно также воспользоваться косвенной адресацией. В приведенном ниже примере мы присваиваем значение 40 координате Y верхнего левого угла прямоугольника, адрес структурной переменной которого находится в регистре ESI.

```
MOV ESI, OFFSET RECT1
MOV (RECTANGLE PTR [ESI]).UPPERLEFT.Y, 10
```

Для определения адреса отдельных полей структуры, включая ее вложенные поля, используется оператор OFFSET:

```
MOV EDI, OFFSET RECT2.LOWERRIGHT
```

```
MOV (COORD PTR [EDI]).X, 50
MOV EDI, OFFSET RECT2.LOWERRIGHT.X
MOV WORD PTR [EDI], 50
```

14.2. Объединения

Объединения (union) отличаются от структур тем, что все поля структуры имеют разное смещение относительно ее начала, тогда как все поля объединения имеют одно и тоже смешение. Длина объединения равна длине его наибольшего поля. Если объединение не является частью структуры, то оно объявляется с помощью директив UNION и ENDS:

```
имя UNION
        ; объявление полей
имя ENDS
```

Если объединение является частью структуры, синтаксис будет немного отличаться:

```
имя_структуры STRUCT
        ; объявление полей структуры
        UNION имя_объединения
                ; объявление полей объединения
        ENDS
имя_структуры ENDS
```

Правила объявления полей в объединении в общем-то такие же, как и при объявлении полей структур, за исключением того, что поля в объединении могут иметь только один инициализатор.

Пример. В объединении INTEGER для одного и того же участка данных объявляются три разных атрибута размера.

```
INTEGER UNION
        D DWORD 0
        W WORD 0
        B BYTE 0
INTEGER ENDS
```

Если внутри структуры содержатся объединения, то после имени поля структуры указывается имя объединения, как это сделано при объявлении поля FILEID внутри структуры FILEINFO:

```
FILELNFO STRUCT
        FILEID INTEGER <>
        FILENAME BYTE 64 DUP(?)
FILELNFO ENDS
```

Другой вариант – объявить объединение непосредственно внутри структуры в соответствии с описанным выше синтаксисом:

```
FILEINFO STRUCT
        UNION FILEID
                D DWORD 0
```

```
        W WORD 0
        B BYTE 0
    ENDS
    FILENAME BYTE 64 DUP(?)
FILEINFO ENDS
```

Экземпляр объединения – «объединенная переменная» – объявляется и инициализируется почти так же, как и структурная переменная. Тем не менее, при этом существует одно важное отличие: для объединенной переменной допускается только один инициализатор. Ниже приведен пример объявления объединенных переменных типа INTEGER.

```
VAL1 INTEGER <12345678H>
VAL2 INTEGER <100H>
VAL3 INTEGER <>
```

При использовании объединенной переменной в коде программы, нужно указать ее имя и через точку имя одного из ее полей. В приведенном ниже примере значения регистров общего назначения сохраняются в полях объединенной переменной типа INTEGER. Обратите внимание на то, насколько удобно пользоваться в программе операндами разного типа.

```
MOV VAL3.B, AL
MOV VAL3.W, AX
MOV VAL3.D, EAX
```

14.3. Работа с динамической памятью

Любые данные, обрабатываемые программой, располагаются в памяти. Традиционно в любом языке программирования высокого уровня существуют два способа распределения памяти для переменных и постоянных данных – статический и динамический.

В ассемблере в силу его специфики явно присутствует возможность только статического распределения памяти с помощью директив резервирования и инициализации памяти в сегменте данных. Статические объекты данных занимают отведенную им память на все время работы программы.

При написании ассемблерных программ мы всегда использовали именно статическое распределение памяти. Недостаток этого очевиден – если объект данных занимает большую область памяти и имеет малый период жизни, то выделять память на все время работы программы неразумно. Гораздо эффективнее в таких случаях иметь возможность для временного выделения памяти объекту данных на период его жизни.

Динамическими объектами данных называются объекты данных, обладающие двумя особенностями:

– распределение/освобождение памяти для объекта данных производится по явному запросу программы;

– доступ к динамическим объектам данных производится не по их именам, а посредством указателей, содержащих ссылку (адрес) на созданный динамический объект.

Существует и третья особенность, о которой, возможно, не подозревают программирующие на языках высокого уровня – технология выделения памяти для динамических объектов данных. Эта технология напрямую зависит от операционной системы, для которой разрабатывается программа. Так, в MS-DOS динамическое выделение памяти во время работы приложения осуществляется с помощью двух функций прерывания 21H: 48H (выделение блока памяти) и 49H (освобождение блока памяти). Единицей выделения памяти при этом является параграф (16-байтовый блок памяти).

Гораздо большими возможностями обладает операционная система Windows. В Windows существуют несколько механизмов динамического выделения памяти: виртуальная память, кучи (пулы), отображаемые в память файлы. Мы рассмотрим только механизм работы с кучами Windows.

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

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

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

Во время инициализации процесса система выделяет ему стандартную кучу (или кучу по умолчанию), размер которой составляет 1 Мбайт. При желании компоновщику можно задать другой размер стандартной кучи в командной строке или настройках интегрированной среды разработки. Создание и уничтожение стандартной кучи производится системой, поэтому в API не существует функций, управляющих этим процессом. Только одна функция должна вызываться перед началом работы со стандартной кучей – GetProcessHeap:

HANDLE GetProcessHeap (VOID)

Функция GetProcessHeap возвращает дескриптор (идентификатор), используемый далее другими функциями работы с кучей.

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

HANDLE HeapCreate (DWORD dwOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize)

Размер создаваемой этой функцией кучи определяется параметрами dwInitialSize (начальный размер) и dwMaximumSize (максимальный размер),

параметр dwOptions определяет необязательные атрибуты новой кучи. Возвращаемое функцией HeapCreate значение – дескриптор кучи, который используется затем другими функциями, работающими с данной кучей. Уничтожение дополнительной кучи осуществляется вызовом функции HeapDestroy, которой в качестве параметра передается дескриптор уничтожаемой кучи:

BOOL HeapDestroy (HANDLE hHeap)

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

Выделение физической памяти из кучи производится по запросу HeapALLoc:

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

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

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

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

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

Изменяя размеры блоков, вы можете совсем запутаться. Функция HeapSize поможет определить текущий размер блока по адресу lpMem в куче hHeap:

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

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

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

Пример. Выделим память под массив из 100 однобайтовых элементов и заполним этот массив значениями от 1 до 100.

```
.686
.MODEL FLAT, STDCALL
EXTERN   GetProcessHeap@0:PROC
EXTERN   HeapAlloc@12:PROC
EXTERN   HeapFree@12:PROC
.DATA
HANDLE_HEAP DD   0       ; дескриптор кучи
```

```
SIZE_MASS DD 100              ; размер массива
ADDR_MASS DD 0                ; адрес массива
.CODE
MAIN:
; получение дескриптора стандартной кучи
CALL        GetProcessHeap@0
MOV HANDLE_HEAP, EAX
// выделение памяти под массив
PUSH        SIZE_MASS ; размер массива в байтах
PUSH        0       ; флаги не задаем
PUSH        HANDLE_HEAP  ; дескриптор кучи
CALL        HeapAlloc@12
MOV ADDR_MASS, EAX     ; адрес массива в куче
; работа с массивом
MOV EBX, ADDR_MASS
MOV ECX, 100
MOV EDI, 0
MOV AL, 0
START:
        INC AL
        MOV [EBX][EDI], AL
        INC EDI
LOOP START
. . . . . .
; удаление массива
PUSH        ADDR_MASS ; адрес массива в куче
PUSH        0
PUSH        HANDLE_HEAP ; дескриптор кучи
CALL        HeapFree@12
RET
END MAIN
```

В заключение заметим, что при программировании под систему Windows команды языка C++ new/delete и функции языка C malloc/free часто представляют собой лишь многослойные оболочки функций API, предназначенных для работы со стандартными кучами. Например, именно такая реализация используется в Visual C++.