8

Профессиональные методики программирования

8.1. Введение

По моему первоначальному замыслу эта глава должна была быть посвящена методикам написания процедур на языке ассемблера. Однако со временем ее материал расширился и вышел за рамки задуманного. Возможно, это произошло из-за того, что основные концепции языков программирования стали во многом так схожи.

В настоящее время наметилась закономерная тенденция поиска универсальных подходов, облегчающих процесс обучения. Поэтому в данной главе я собираюсь показать вам различные методики программирования на примере низкоуровневого средства разработки программ — языка ассемблера. Другими словами, описанный здесь материал обычно излагается в курсе по программированию на языке C++ или Java, ориентированном на подготовленных учащихся, а также в одном из основных курсов информатики, называемом *языками программирования*. Ниже перечислены темы, рассмотренные в этой главе, которые можно отнести к основополагающим принципам программирования:

- создание и инициализация локальных переменных в стеке;
- область действия и время жизни переменных;
- передача параметров через стек;
- стековые фреймы;
- передача параметров по значению и по ссылке;
- типы параметров процедур: *входные*, *выходные* и *универсальные*;
- рекурсия.

Часть материала этой главы предназначена для дальнейшего изучения возможностей языка ассемблера:

- директивы `INVOKE`, `PROC` и `PROTO`;
- операторы `USES` и `ADDR`;
- модели памяти и описатели языка;
- использование косвенной адресации для доступа к параметрам, находящимся в стеке;
- создание программ, состоящих из нескольких модулей.

Мне хочется подчеркнуть, что полученные знания о языке ассемблера позволят вам понять логику проектировщика компилятора языка высокого уровня по части генерации машинного кода, т.е. того механизма, который, собственно, и заставляет программы работать.

8.2. Локальные переменные

Локальными называются переменные, которые создаются, используются и аннулируются в пределах одной процедуры. Если вам приходилось программировать на одном из языков высокого уровня, то вы уже должны иметь представление о локальных переменных.

При рассмотрении примеров в предыдущих главах мы объявляли все переменные в сегменте данных. Такие переменные называются *статическими глобальными*. Термин *статический* говорит о том, что такая переменная существует все время, пока выполняется программа, в которой она объявлена. Другими словами, время жизни статической переменной совпадает со временем выполнения программы. Термин *глобальный* определяет область действия переменной. Глобальную переменную можно использовать во всех процедурах, находящихся в текущем исходном файле.

В этой же главе мы научимся создавать и пользоваться локальными переменными в языке ассемблера. Они выгодно отличаются от глобальных переменных:

- ограниченная область действия локальной переменной позволяет быстрее выявить ошибку на этапе отладки, поскольку изменить ее значение может только ограниченное количество команд программы;
- применение локальных переменных позволяет более эффективно расходовать память компьютера, поскольку занимаемый ими участок оперативной памяти можно освободить и перераспределить для других переменных;
- одно и то же имя переменной может использоваться в нескольких процедурах, при этом не возникает конфликта имен.

Локальные переменные всегда создаются в области программного стека, поэтому им нельзя присвоить начальные значения еще на этапе компиляции программы. Локальные переменные можно инициализировать только во время выполнения программы.

8.2.1. Директива LOCAL

Директива `LOCAL` предназначена для объявления одной или нескольких локальных переменных внутри процедуры. В исходном коде она должна располагаться сразу за директивой `PROC`. Синтаксис директивы `LOCAL` следующий:

```
LOCAL Список_переменных
```

Здесь под списком переменных понимается перечень описаний переменных, разделенных запятой, который может занимать несколько строк. Описание каждой переменной задается в следующем виде:

```
имя:тип
```

В качестве имени переменной можно выбрать любой допустимый в языке ассемблера идентификатор. В качестве типа можно задать один из встроенных типов языка ассемблера, таких как `WORD`, `DWORD` и др., либо один из нестандартных типов, определенных программистом. (О структурах и других определяемых программистом типах данных речь пойдет в главе 10, “Структуры и макроопределения”.)

Пример 1. В процедуре **`MySub`** создается одна локальная переменная **`var1`**, которая имеет тип `BYTE`:

```
MySub PROC
   LOCAL var1:BYTE
```

Пример 2. В процедуре **`BubbleSort`** создаются две локальные переменные. Одна из них имеет имя **`temp`** и занимает двойное слово, а другая называется **`SwapFlag`** и занимает в памяти один байт:

```
BubbleSort PROC
   LOCAL temp:DWORD, SwapFlag:BYTE
```

Пример 3. В процедуре **`Merge`** создается одна локальная переменная **`pArray`**, в которой будет храниться указатель на слово, расположенное в памяти:

```
Merge PROC
   LOCAL pArray:PTR WORD
```

Пример 4. Переменная **`TempArray`** является массивом из десяти двойных слов. Обратите внимание, что размер массива указывается в квадратных скобках:

```
   LOCAL TempArray[10]:DWORD
```

Автоматическая генерация кода. Вполне вероятно, что вас может заинтересовать, какой код на самом деле сгенерирует компилятор ассемблера при использовании в программе локальных переменных. Чтобы ответить на этот вопрос, откройте окно Disassembly в отладчике Visual Studio. Давайте скомпилируем приведенный ниже фрагмент программы, в котором определяется заготовка процедуры с локальными переменными, и загрузим его в окно отладчика:

```
BubbleSort PROC
   LOCAL temp:DWORD, SwapFlag:DWORD
;
   ret
BubbleSort ENDP
```

Вот что вы увидите в окне дизассемблера отладчика (приведено с небольшими смысловыми изменениями):

```
BubbleSort:
   push ebp
   mov  ebp,esp
   add  esp,0FFFFFFF8h          ; Прибавляется -8 к регистру ESP
   mov  esp,ebp
   pop ebp
   ret
```

Команда `ADD` прибавляет число –8 к регистру `ESP`. В результате указатель стека смещается на 8 байтов вниз, что создает пространство для размещения в области стека двух локальных переменных. Адрес начала области локальных переменных процедуры загружается в регистр `EBP`, как показано на рис. 8.1.

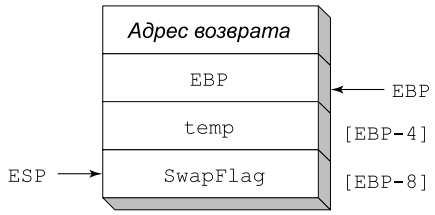

***Рис. 8.1.** Структура стека процедуры* **`BubbleSort`**

Пример процедуры *`SumOf`***.** В приведенном ниже фрагменте кода в процедуре **`SumOf`** используется локальная переменная **`tempSum`**, занимающая двойное слово:

```
SumOf PROC
   LOCAL  tempSum:DWORD
   mov    tempSum,eax
   add    tempSum,ebx
   add    tempSum,ecx           ; tempsum = eax + ebx + ecx
   mov eax,tempSum
   ret
SumOf ENDP
```

Резервирование памяти в стеке. Если вы планируете создавать в своей программе локальные переменные, являющиеся массивами переменной длины, необходимо позаботиться о том, чтобы при загрузке программы операционная система выделила достаточное количество памяти под стек. Например, во включаемом файле `Irvine32.inc` содержится приведенная ниже директива `STACK`, которая резервирует 4096 байтов под стек:

```
.stack 4096
```

Если в программе выполняются вложенные вызовы процедур, размер стека должен быть таким, чтобы в нем могли разместиться локальные переменные всех активных в произвольный момент времени выполнения программы процедур. Например, предположим, что в процедуре **`Sub1`** вызывается процедура **`Sub2`**, а в процедуре **`Sub2`** вызывается процедура **`Sub3`**. В каждой из этих процедур создается локальный массив:

```
Sub1 PROC
   LOCAL array1[50]:DWORD       ; 200 байтов
.
.
Sub2 PROC
   LOCAL array2[80]:WORD        ; 160 байтов
.
.
Sub3 PROC
   LOCAL array3[300]:BYTE       ; 300 байтов
```

При вызове процедуры **`Sub3`** в стеке будут находиться наборы локальных переменных процедур **`Sub1`**, **`Sub2`** и **`Sub3`**, под которые выделяется 660 байтов. К этому нужно прибавить два двойных слова (8 байтов), содержащих адреса возврата из процедур, а также зарезервировать дополнительную память для сохранения регистров в стеке, которое обычно выполняется в начале работы процедуры.

8.2.2. Контрольные вопросы раздела

1. Назовите три преимущества локальных переменных перед глобальными.
2. (*Да/Нет*). Локальным переменным можно присвоить начальные значения во время компиляции программы.
3. (*Да/Нет*). С помощью одной директивы `LOCAL` можно определить максимум четыре локальных переменных.
4. (*Да/Нет*). В двух разных процедурах может использоваться одно и то же имя локальной переменной.
5. Объявите локальную переменную **`pArray`**, которая является указателем на массив двойных слов.
6. Объявите локальную переменную **`buffer`**, которая является массивом из 20 байтов.
7. Объявите локальную переменную **`pwArray`**, которая является указателем на 16-разрядную переменную целого типа без знака.
8. Объявите локальную переменную **`myByte`**, которая является 8-разрядным целым числом со знаком.
9. Объявите локальную переменную **`myArray`**, которая является массивом из 20 двойных слов.

8.3. Стековые параметры

Существует два основных типа параметров процедуры — *регистровые* и *стековые*. В процедурах из библиотек `Irvine32` и `Irvine16` используются регистровые параметры. В этом разделе мы рассмотрим способы объявления и использования стековых параметров.

> Значения, которые передаются в процедуру перед ее вызовом, называются *аргументами*. Переменные процедуры, вместо которых подставляются переданные в процедуру значения, называются *параметрами*.

Регистровые параметры используются при выполнении оптимизации скорости работы программы. Однако часто это приводит к излишнему загромождению кода вызывающей программы. Кроме того, обычно при загрузке в регистры значений аргументов приходится сохранять в стеке их текущее состояние, как, например, при вызове процедуры **`DumpMem`**:

```
pushad
mov   esi,OFFSET array          ; Начальный адрес массива
mov   ecx,LENGTHOF array        ; Размер массива в блоках
mov   ebx,TYPE array            ; Определим формат вывода
call  DumpMem                   ; Отобразим содержимое памяти
popad
```

Альтернативой регистровым являются стековые параметры. При этом перед вызовом процедуры нужные параметры сначала нужно поместить в стек. Например, если бы в

процедуре **DumpMem** использовались стековые параметры, приведенный выше фрагмент кода выглядел бы так:

```
push   TYPE array
push   LENGTHOF array
push   OFFSET array
call   DumpMem
```

Для удобства программиста в компиляторе MASM предусмотрена специальная директива INVOKE, которая позволяет с помощью одного оператора поместить в стек аргументы и вызвать процедуру. Поэтому вместо приведенных выше четырех строк кода можно написать только одну:

```
INVOKE   DumpMem,OFFSET array,LENGTHOF array,TYPE array
```

Кроме того, есть еще одна причина, по которой вам нужно освоить стековый способ передачи параметров: он используется практически во всех компиляторах языков высокого уровня. Поэтому, если вы хотите, например, вызвать одну из библиотечных функций системы Windows, вы должны передать ей аргументы через стек.

8.3.1. Директива INVOKE

Директива INVOKE является очень гибким средством вызова процедур и по сути заменяет команду CALL процессоров Intel. Она позволяет передать в процедуру несколько аргументов. Синтаксис директивы INVOKE приведен ниже:

```
INVOKE Имя_процедуры [, Список_аргументов]
```

Аргументы, передаваемые процедуре в директиве INVOKE, перечисляются через запятую и могут отсутствовать вовсе. Уже сейчас нетрудно заметить основную разницу между директивой INVOKE и командой CALL: у последней нет списка аргументов.

В директиве INVOKE можно указать произвольное число аргументов, причем их список может занимать несколько строчек исходного кода. Возможные типы аргументов перечислены в табл. 8.1.

Таблица 8.1. Типы аргументов директивы INVOKE

Аргумент	*Пример использования*
Непосредственно заданное значение	`10, 3000h, OFFSET myList, TYPE array`
Выражение целого типа	`(10 * 20), COUNT`
Имя переменной	`myList, array, myWord, myDword`
Адресное выражение	`[myList+2], [ebx + esi]`
Имя регистра	`eax, bl, edi`
`ADDR` *имя*	`ADDR myList`

Пример. В приведенном ниже фрагменте кода с помощью директивы INVOKE вызывается процедура **AddTwo**, которой передаются два 32-разрядных целых числа:

```
.data
 val1   DWORD   12345h
 val2   DWORD   23456h
.code
   INVOKE AddTwo,val1,val2
```

Без директивы INVOKE нам пришлось бы перед вызовом команды CALL поместить в стек значения переменных **val1** и **val2**, причем в обратном порядке, как принято в языке C++:

```
push   val2
push   val1
call   AddTwo
```

На рис. 8.2 показана структура стека непосредственно перед вызовом команды CALL.

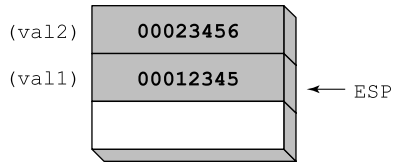

***Рис. 8.2.** Структура стека перед вызовом команды CALL, принятая в языке C/C++ и в библиотеке Irvine32.lib*

Учтите, что показанный здесь порядок передачи аргументов через стек не является единственным. Подробнее об этом мы поговорим в разделе 8.4.2.

8.3.1.1. Оператор ADDR

Оператор ADDR используется в директиве INVOKE для того, чтобы передать в процедуру *указатель* на переменную (т.е. адрес переменной), а не значение самой переменной. Такой способ передачи аргументов называется *передачей параметров по ссылке*. Например, в приведенном ниже фрагменте кода в процедуру **FillArray** передается адрес массива **myArray**:

```
INVOKE   FillArray, ADDR myArray
```

Оператор **ADDR** возвращает значение ближнего (near) или дальнего (far) указателя следующей за ним переменной в зависимости от выбранной программистом модели памяти программы. В защищенном режиме обычно используется линейная (flat) модель памяти, поэтому операторы ADDR и OFFSET возвращают одинаковые значения — 32-разрядное смещение переменной относительно начала сегмента памяти (т.е. адреса 00000000h). В учебных примерах, рассматриваемых в книге, линейная модель памяти определяется с помощью директивы .MODEL, которая находится в файле Irvine32.inc.

При создании программ для реального режима адресации чаще всего используется малая (small) модель памяти (с одним сегментом кода и одним сегментом данных), в которой операторы ADDR и OFFSET также возвращают одинаковые значения — 16-разрядное смещение переменной относительно начала сегмента данных. В учебных примерах, рассматриваемых в книге, малая модель памяти определяется во включаемом файле Irvine16.inc, который используется при создании программ для системы MS DOS.

В реальном режиме дальние указатели являются 32-разрядными числами, содержащими пару значений "сегмент-смещение". Они обычно используются при написании системных программ (таких как драйверов устройств), или же больших приложений, содержащих несколько сегментов кода и данных.

Пример 1. В приведенном ниже фрагменте кода вызывается процедура **`FillArray`**, которой передается адрес массива байтов. Обратите внимание, что мы разместили аргумент в отдельной строке кода вместе с комментарием:

```
.data
 myArray   BYTE   50 DUP(?)

.code
  INVOKE FillArray,
         ADDR myArray           ; Адрес массива
```

Пример 2. Ниже показано, как можно передать в процедуру **`Swap`** адреса двух первых элементов массива двойных слов:

```
.data
 Array   DWORD   20 DUP(?)


.code
 ...
  INVOKE Swap,
         ADDR Array,
         ADDR Array+4
```

8.3.2. Директива PROC

Директива `PROC` предназначена для описания имени процедуры и списка передаваемого ей параметров. Ее упрощенный синтаксис показан ниже:

```
Имя_процедуры PROC, Параметр_1,
                    Параметр_2,
                    .
                    .
                    Параметр_n
```

Список параметров можно также разместить в одной строке:

```
Имя_процедуры PROC, Параметр_1,Параметр_2,...,Параметр_n
```

Синтаксис описания одного параметра процедуры выглядит так:

```
Имя_Параметра:Тип
```

`Имя_Параметра` выбирается произвольно программистом и должно удовлетворять соглашению, принятому в языке ассемблера для имен меток. Область действия данного параметра ограничена текущей процедурой, поэтому она называется *локальной*. Это позволяет иметь одинаковые имена параметров в разных процедурах, однако учтите, что они не должны совпадать с именами глобальных переменных или меток кода. Параметр может иметь один из перечисленных ниже *`типов`*: `BYTE`, `SBYTE`, `WORD`, `SWORD`, `DWORD`,

SDWORD, FWORD, QWORD или TBYTE. Кроме того, параметр может являться указателем на переменную одного из стандартных типов. В этом случае говорят, что он имеет *уточняющий тип* (*qualified type*), примеры которого приведены ниже:

```
PTR BYTE            PTR SBYTE
PTR WORD            PTR SWORD
PTR DWORD           PTR SDWORD
PTR QWORD           PTR TBYTE
```

В этих выражениях перед оператором PTR могут использоваться атрибуты NEAR или FAR, однако они имеют особое значение только при создании специализированных приложений, имеющих нестандартную модель памяти. В качестве уточняющего можно использовать также один из собственных типов данных, созданных программистом с помощью директив TYPEDEF или STRUCT, как описано в главе 10, "Структуры и макроопределения".

8.3.2.1. Примеры

Давайте рассмотрим несколько примеров объявления процедур, в которых используются параметры разного типа. Некоторые из имен этих процедур мы будем использовать ниже в этой главе, однако пока что сама реализация их кода не имеет для нас никакого значения.

Пример 1. Приведенной ниже процедуре в качестве параметров передаются два двойных слова:

```
AddTwo   PROC,
              val1:DWORD,
              val2:DWORD
 . . .
AddTwo   ENDP
```

Пример 2. А этой процедуре передается указатель на байт:

```
FillArray   PROC,
                 pArray:PTR BYTE
  . . .
FillArray ENDP
```

Пример 3. А вот как можно передать процедуре два указателя на двойные слова:

```
Swap   PROC,
            pValX:PTR DWORD,
            pValY:PTR DWORD
  . . .
Swap ENDP
```

Пример 4. Приведенной ниже процедуре передается указатель на переменную типа байт, который подставляется вместо параметра **pBuffer**. Кроме того, в ней объявляется одна локальная переменная **fileHandle**, и размер ее соответствует двойному слову:

```
ReadFile   PROC,
                pBuffer:PTR BYTE
   LOCAL   fileHandle:DWORD
  . . .
ReadFile   ENDP
```

8.3.3. Директива PROTO

Директива PROTO создает *прототип* существующей процедуры. В прототипе описывается имя процедуры и список ее параметров. Прототипы позволяют вызывать процедуру до того, как она будет формально определена. Тем, кому приходилось программировать на C++, понятие прототипа должно быть уже знакомо, поскольку ни одно объявление классов не обходится без использования прототипов функций.

Компилятор MASM требует, чтобы для каждой процедуры, вызываемой с помощью оператора INVOKE, был описан прототип. Директива PROTO должна предшествовать в исходном файле оператору INVOKE. Другими словами, стандартный порядок этих директив и операторов должен быть такой:

```
MySub  PROTO                    ; Прототип процедуры
INVOKE   MySub                  ; Вызов процедуры
MySub  PROC                     ; Тело процедуры
.
.
MySub  ENDP
```

Возможен и другой вариант, когда тело процедуры находится в программе перед оператором INVOKE, который ее вызывает. В этом случае объявлением прототипа процедуры считается директива PROC:

```
MySub   PROC                    ; Тело процедуры
.
.
MySub   ENDP

INVOKE   MySub                  ; Вызов процедуры
```

Предположим, что вы уже написали текст самой процедуры. Тогда ее прототип легко создать на основе директивы PROC, внеся в нее следующие изменения:

- ключевое слово PROC нужно заменить на PROTO;
- удалить ключевое слово USES и следующий за ним список регистров, если они указаны при объявлении процедуры.

Например, предположим, что мы когда-то написали процедуру **ArraySum**:

```
ArraySum   PROC   USES esi ecx,
               ptrArray:PTR DWORD, ; Указатель на массив
                                   ;  двойных слов
               szArray:DWORD       ; Размер массива
; (Строки кода для ясности мы удалили
ArraySum ENDP
```

Прототип будет очень похож на оператор определения этой функции:

```
ArraySum   PROTO,
               ptrArray:PTR DWORD, ; Указатель на массив
                                   ;  двойных слов
               szArray:DWORD       ; Размер массива
```

Напомним, что оператор USES, описанный в разделе 5.5.5.1 главы 5, предназначен для автоматической генерации команд push и pop, с помощью которых сохраняются и восстанавливаются значения используемых в процедуре регистров.

8.3.3.1. Пример процедуры ArraySum

В качестве примера в этом разделе мы создадим новую версию процедуры **ArraySum**, описанную в предыдущих главах, которая вычисляет сумму элементов массива двойных слов. В первоначальной версии этой процедуры аргументы передавались через регистры. Теперь с помощью директивы PROC мы опишем стековые параметры, как показано ниже:

```
ArraySum   PROC   USES esi ecx,
                  ptrArray:PTR DWORD, ; Адрес массива
                   szArray:DWORD      ; Размер массива

  mov   eax,0                    ; Обнулим значение суммы
  mov   esi,ptrArray             ; Загрузим адрес массива
  mov   ecx,szArray              ; Загрузим длину массива
  cmp   ecx,0                    ; Массив нулевой длины?
  je    L2                       ; Если да, завершим работу

L1:
  add   eax,[esi]                ; Прибавим значение текущего
                                 ;  элемента массива
  add   esi,4                    ; Адрес следующего элемента
                                 ;  массива
  loop  L1                       ; Повторим цикл для всех
                                 ;   элементов массива
L2:
  ret                            ; Сумма находится в регистре EAX
ArraySum ENDP
```

Для вызова процедуры **ArraySum** и передачи ей адреса массива и числа элементов массива воспользуемся директивой INVOKE, как показано ниже:

```
.data
 array   DWORD   10000h,20000h,30000h,40000h,50000h
 theSum  DWORD ?

.code
 main PROC
    INVOKE ArraySum,
                   ADDR array, ; Адрес массива
                LENGTHOF array  ; Число элементов массива
    mov   theSum,eax           ; Сохраним значение суммы
```

Директива INVOKE существенно упрощает процесс передачи аргументов процедурам и уменьшает количество ошибок. Все дело в том, что намного легче в программах иметь дело с именованными параметрами, чем с названиями регистров. Имя параметра говорит само за себя, а регистры можно использовать для других целей.

8.3.4. Передача параметров по значению и по ссылке

Передача по значению. Если во время вызова процедуры ей в качестве аргументов передаются копии значений переменных, то в таком случае говорят, что параметры передаются *по значению*. Вообще говоря, программисты передают аргументы по значению в случае, если нужно защитить их значения от случайного изменения в процедуре. В этом случае значение переменной помещается в стек перед вызовом процедуры. В самой же вызываемой процедуре происходит выборка значения параметра из стека и его последующее использование. И даже если значение параметра будет изменено в процедуре, значение соответствующей переменной вызывающей программы, которая передана ей в качестве аргумента, останется без изменения. Дело в том, что при передаче аргументов по значению, из вызываемой процедуры нельзя получить доступ к переменным вызывающей программы.

В приведенном ниже фрагменте кода показан типичный случай, когда из процедуры **main** вызывается процедура **Sub1**, которой в качестве аргумента передается копия переменной **myData**. В процедуре **Sub1** входящему параметру будет соответствовать локальная переменная **someData**. Несмотря на то, что переменной **someData** в процедуре **Sub1** присваивается нулевое значение, это никак не влияет на значение переменной **myData**:

```
.data
 myData   WORD   1000h          ; Эта переменная не изменяется

.code
 main  PROC
    INVOKE   Sub1, myData
    exit
main  ENDP

Sub1  PROC  someData:WORD
    mov someData,0
    ret
Sub1 ENDP
```

Естественно, что из процедуры **Sub1** можно явно обратиться к переменной **someData** и изменить ее значение. Так или иначе, факт модификации переменной будет ограничен рамками процедуры **Sub1**. Поэтому при возникновении ошибки в программе, связанной с изменением значения переменной **someData**, ее можно будет достаточно легко локализовать и исправить.

Передача по ссылке. Если во время вызова процедуры ей в качестве аргументов передаются адреса переменных, то в таком случае говорят, что параметры передаются *по ссылке*. При этом программист получает возможность изменить значение исходной переменной из вызываемой процедуры, воспользовавшись переданным адресом. Существует хорошее практическое правило, которое гласит, что параметры нужно передавать по ссылке только в том случае, если в процедуре их значение должно быть изменено. Хотя нужно отметить, что изменение в процедуре значения переданных ей в качестве параметров переменных нельзя отнести к хорошему стилю программирования.

В приведенном ниже примере в процедуру **Sub2** передается адрес переменной **myData**. После вызова процедуры этот адрес загружается в регистр ESI и затем используется в качестве базы при косвенном обращении к переменной **myData**, которой присваивается нулевое значение:

```
.data
 myData   WORD   1000h
 Sub2     PROTO  dataPtr:PTR WORD

.code
 main  PROC
    INVOKE  Sub2, ADDR myData   ; Аргумент передается по ссылке
    exit
 main ENDP

 Sub2  PROC   dataPtr:PTR WORD
    mov   esi,dataPtr           ; Загрузим адрес переменной
    mov   WORD PTR[esi],0       ; Присвоим ей нулевое значение,
                                ;  воспользовавшись косвенной
                                ;  адресацией
    ret
 Sub2 ENDP
```

Передача структур данных. У сформулированного нами выше практического правила о способах передачи аргументов в процедуру есть одно важное исключение. В языках программирования высокого уровня различные структуры данных (такие как массивы) всегда передаются по ссылке. В самом деле, ведь непрактично передавать большое количество данных по значению, поскольку перед вызовом процедуры их нужно целиком поместить в стек. В результате катастрофически снижается быстродействие программы и нерационально используется драгоценная (потому что ее объем обычно небольшой и всегда конечен) память, выделенная под стек. Единственный недостаток при передаче структур данных по ссылке состоит в том, что в процедуре появляется возможность изменить содержимое этой структуры. Для решения подобной проблемы в языке C++ предусмотрен описатель const, однако в языке ассемблера подобных средств нет.

8.3.5. Классификация параметров

Параметры процедуры обычно классифицируют в зависимости от направления движения потоков данных между вызывающей программой и процедурой, как описано ниже.

- ***Входные параметры*** передают данные из вызывающей программы в процедуру. При этом не предполагается, что вызывающая процедура должна изменять значение переменной, соответствующей параметру. И даже если это произойдет, изменение значения параметров не выйдет за рамки процедуры.
- ***Выходные параметры*** создаются путем передачи в процедуру указателей на переменные. В самой процедуре значения этих переменных не используются, но при возврате в вызывающую программу в них записывается некоторое значение. Например, в библиотеке поддержки терминальных приложений системы Win32 (Win32 Console Library) предусмотрена специальная функция **ReadConsole**, которая предназначена для чтения строки символов из стандартного устройства ввода в

массив байтов. Вызывающая программа передает в эту функцию указатель на переменную типа двойного слова, в которую будет записано целое число, показывающее, сколько байтов прочитано:

```
ReadConsole PROTO,
              handle:DWORD,     ; Дескриптор устройства для чтения
           lpBuffer:PTR BYTE,   ; Адрес буфера
 nNumberOfBytesToRead:DWORD,    ; Максимальное количество символов,
                                ;  которые нужно прочитать
                                ;  (длина буфера)
lpNumberOfBytesWritten:PTR DWORD,; Количество символов, которое
                                ;  было прочитано
          lpReserved:DWORD      ; Всегда нуль
```

В этом примере параметры `handle`, `lpReserved` и `nNumberOfBytesToRead` являются входными, а `lpBuffer` и `lpNumberOfBytesWritten` — выходными.

- ***Универсальные параметры*** предназначены для передачи в процедуру значений, которые она может изменить. В результате один и тот же параметр может использоваться как для передачи значения в процедуру, так и возврата значения в вызывающую программу. Следует заметить, что при передаче в процедуру адреса переменной, ее вполне можно использовать не только как выходной, но и как универсальный параметр.

8.3.6. Пример: обмен значений двух переменных

В приведенном ниже фрагменте программы используется процедура **Swap**, предназначенная для обмена значений двух 32-разрядных целых переменных. С ее помощью мы поменяем местами значения двух элементов массива. При этом содержимое массива выводится на экран дважды — до и после выполнения обмена:

```
TITLE Программа обмена двух целых чисел   (Swap.asm)

INCLUDE Irvine32.inc

Swap PROTO,                     ; Прототип процедуры
           pValX:PTR DWORD,
           pValY:PTR DWORD

.data
 Array   DWORD   10000h,20000h

.code
 main PROC
 ; Отобразим содержимое массива до обмена
   mov   esi,OFFSET Array       ; Адрес массива
   mov   ecx,2                  ; Число элементов
   mov   ebx,TYPE Array         ; Тип элементов
   call  DumpMem                ; Выведем массив на экран

   INVOKE  Swap, ADDR Array, ADDR [Array+4]
```

```
    ; Отобразим содержимое массива после обмена
      call DumpMem
      exit
    main ENDP

    ;-----------------------------------------------------------------
    Swap  PROC  USES eax esi edi,
                  pValX:PTR DWORD, ; Адрес первой переменной
                  pValY:PTR DWORD  ; Адрес второй переменной
    ;
    ; Процедура для обмена значения двух 32-разрядных целых чисел
    ; Возвращается: ничего
    ;-----------------------------------------------------------------
       mov   esi,pValX             ; Загрузим адреса переменных
       mov   edi,pValY
       mov   eax,[esi]             ; Загрузим значение первой
                                   ;  переменной
       xchg  eax,[edi]             ; Обменяем его со вторым
                                   ;  значением
       mov  [esi],eax              ; Заменим первое значение вторым
       ret
    Swap ENDP
END main
```

Процедура **Swap** имеет два универсальных параметра **pValX** и **pValY**. С их помощью в процедуру передаются исходные значения переменных, а возвращаются новые значения. Другими словами, эти параметры используются и как *входные*, и как *выходные*.

8.3.7. Методики поиска ошибок в программах

8.3.7.1. Сохранение и восстановление регистров

Вообще говоря, команды PUSH и POP выполняют очень важные действия. Они позволяют сохранить содержимое регистров общего назначения, а затем, после выполнения фрагмента программы, изменяющего их значение, восстановить их к первоначальному состоянию. Поскольку в процессорах Intel предусмотрено совсем немного регистров общего назначения, при программировании их всегда не хватает.

Предположим, что непосредственно перед выполнением цикла в регистр ECX было загружено какое-то важное значение. Но нам хорошо известно, что регистр ECX используется в качестве счетчика цикла. Поэтому перед тем, как загрузить в регистр ECX счетчик цикла, нам нужно сохранить его значение в стеке, а затем, после окончания цикла, восстановить его, как показано ниже:

```
      mov   ecx,importantVal
      push  ecx                   ; Сохраним ECX
      .
      mov   ecx,LoopCounter       ; Установим счетчик цикла
  L1:
      .
      .
      loop   L1
      pop ecx                     ; Восстановим ECX
```

В подобных случаях вы должны внимательно следить за тем, чтобы каждой команде PUSH в программе соответствовала своя команда POP. В приведенном ниже примере команда POP ошибочно помещена внутрь тела цикла, поэтому с большой долей вероятности можно сказать, что цикл будет выполняться бесконечно:

```
        mov   ecx,importantVal
        push  ecx                    ; Сохраним ECX
        .
        mov   ecx,LoopCounter        ; Установим счетчик цикла
    L1:
        .
        .
        pop   ecx                    ; Восстановим ECX ??!
        loop  L1
```

В данном случае мы только один раз поместили в стек значение регистра ECX, после чего в цикле несколько раз извлекли его из стека. Каждая команда POP неявно увеличивает значение указателя стека (ESP) на 4 байта. В результате указатель стека довольно быстро выйдет за пределы области, содержащей данные программы. Поскольку после выполнения цикла (если он все-таки когда-нибудь закончится!) в стеке вместо реальных данных будет находиться "мусор", при выполнении команды возврата из процедуры RET управление будет передано в произвольную область памяти, а не следующей после CALL команде вызывающей процедуры. В защищенном режиме это приведет к возникновению ситуации общего нарушения защиты (general protection fault).

8.3.7.2. Некорректные размеры операндов

При работе с массивами нужно всегда помнить, что адресация элементов массива зависит от их длины. Например, чтобы определить адрес второго элемента массива двойных слов, нужно прибавить число 4 к начальному адресу массива. Вспомните, как мы в разделе 8.3.6 передавали в процедуру **Swap** адреса первых двух элементов массива **DoubleArray**. Предположим, что при вызове процедуры **Swap** мы некорректно указали адрес второго элемента, как DoubleArray+1:

```
.data
 DoubleArray   DWORD   10000h,20000h

.code
  INVOKE   Swap, ADDR [DoubleArray + 0], ADDR [DoubleArray + 1]
```

Полученный после вызова процедуры **Swap** результат (шестнадцатеричные значения элементов массива **DoubleArray**) будет не таким, как вы того ожидали.

8.3.7.3. Передача некорректных типов указателей

При использовании директивы INVOKE необходимо иметь в виду, что компилятор ассемблера не выполняет проверку типов указателей, передаваемых в процедуру. Например, в процедуру **Swap**, описанную в разделе 8.3.6, нужно передать два указателя на двойные слова. Предположим, что в процедуру были переданы некорректные указатели на байты:

```
.data
 ByteArray   BYTE   10h,20h,30h,40h,50h,60h,70h,80h

.code
 INVOKE   Swap, ADDR [ByteArray + 0], ADDR [ByteArray + 1]
```

Компиляция и выполнение такой программы пройдет без ошибок. Однако в процедуре **Swap** в регистры ESI и EDI будут загружены адреса операндов и по ним будет выполнен обмен двух 32-разрядных значений. В результате массив **ByteArray** будет состоять из следующих значений: 20h, 30h, 40h, 50h, 40h, 60h, 70h и 80h.

8.3.7.4. Передача непосредственно заданных значений

Если в процедуру должны передаваться адреса переменных, вместо них нельзя указывать непосредственно заданные значения. Давайте рассмотрим приведенную ниже процедуру, которой в виде параметра должен передаваться один указатель:

```
Sub2   PROC   dataPtr:PTR WORD
   mov   esi,dataPtr            ; Загрузим адрес операнда
   mov   [esi],0                ; Обнулим значение
   ret
Sub2 ENDP
```

Приведенный ниже оператор INVOKE при компиляции не вызовет никаких ошибок, однако при выполнении программы возникнет ошибка. Предположим, что в процедуру **Sub2** было передано значение 1000h, которое интерпретируется как адрес переменной:

```
INVOKE   Sub2, 1000h
```

При запуске такой программы на выполнение произойдет прерывание из-за общего нарушения защиты, поскольку переменной с адресом 1000h нет в сегменте данных программы.

8.3.8. Контрольные вопросы раздела

1. (*Да/Нет*). В команде CALL нельзя указать аргументы, передаваемые процедуре.
2. (*Да/Нет*). В директиве INVOKE можно указать максимум три аргумента.
3. (*Да/Нет*). В директиве INVOKE можно указать только адреса переменных, но не имена регистров.
4. (*Да/Нет*). В директиве PROC можно указать оператор USES, а в директиве PROTO — нет.
5. (*Да/Нет*). В директиве PROC все параметры должны быть указаны в одной строке.
6. (*Да/Нет*). Лучше всего передавать массив в процедуру по ссылке, чтобы его содержимое не копировалось в стек.
7. (*Да/Нет*). Более безопасно передавать объект в процедуру по значению, а не по ссылке, поскольку в последнем случае значение этого объекта может быть изменено в процедуре.

8. (*Да/Нет*). При передаче адреса массива байтов в процедуру, в которую должен передаваться адрес массива слов, компилятор ассемблера не выведет сообщения об ошибке.
9. (*Да/Нет*). Передача непосредственно заданного значения в процедуру вместо адреса переменной приводит в защищенном режиме к возникновению прерывания из-за общего нарушения защиты.
10. Отличаются ли значения, возвращаемые операторами ADDR и OFFSET, при использовании в программе линейной модели памяти?
11. Опишите процедуру с именем **MultArray**, которой передается два указателя на массивы двойных слов и третий параметр, определяющий число элементов массивов.
12. Создайте директиву PROTO для процедуры из предыдущего упражнения.
13. Какие типы параметров (входные, выходные или универсальные) используются в процедуре **Swap**, описанной в разделе 8.3.6?
14. К какому типу (входному или выходному) относится параметр **lpBuffer** процедуры **ReadConsole**, описанной в разделе 8.3.5?
15. *Задача повышенной сложности.* Нарисуйте структуру стека и обозначьте в нем положение параметров, которая создается при выполнении приведенного ниже оператора INVOKE при использовании линейной модели памяти:

```
.data
 count = 10
 myArray   WORD   count DUP(?)

.code
 INVOKE   SumArray, ADDR myArray, count
```

8.4. Стековые фреймы

Выше мы уже говорили о том, что оператор INVOKE преобразовывается компилятором в последовательность команд PUSH, помещающих параметры в стек, которая завершается командой вызова процедуры CALL. Пользоваться оператором INVOKE очень удобно, поскольку при его обработке компилятор автоматически генерирует нужный ассемблерный код. Однако при этом основная цель изучения языка ассемблера (которую можно сформулировать как “изучение всех деталей”) отходит на второй план. Поэтому давайте разберемся, как можно напрямую поместить параметры в стек с помощью команд PUSH и вызвать процедуру с помощью команды CALL. В конечном итоге такой подход позволит вам с честью выйти из любой сложной ситуации. Для начала мы должны познакомиться с понятием модели памяти и описателей языка программирования высокого уровня.

Стековым фреймом (*stack frame*), или *записью активации* (*activation record*), называется область памяти в стеке, расположенная за адресом возврата из процедуры, в которой размещаются переданные ей параметры, сохраненные регистры и локальные переменные. Для создания стекового фрейма программа должна выполнить перечисленные ниже действия:

- поместить аргументы в стек;
- вызвать процедуру командой CALL, в результате чего адрес возврата помещается в стек;
- в начале выполнения процедуры сохранить в стеке регистр EBP;
- загрузить в регистр EBP текущий указатель стека из регистра ESP. С этого момента EBP выполняет функции базового регистра при обращении ко всем параметрам процедуры;
- для выделения из стека области памяти под размещение локальных переменных из регистра ESP нужно вычесть соответствующее значение.

На структуру стекового фрейма непосредственно оказывают влияние используемая в программе модель памяти и установленное соглашение о передаче параметров.

8.4.1. Модели памяти

Для определения ряда важных характеристик программы, таких как тип модели памяти, способ именования процедур и соглашение о передаче параметров, в компиляторе MASM используется директива .MODEL. Последние две характеристики особенно важны, когда язык ассемблера используется для связи программных модулей, написанных на разных языках высокого уровня. Ниже приведен синтаксис директивы .MODEL:

```
.MODEL  Модель_памяти [, Параметры_модели]
```

Типы моделей памяти. Первый параметр директивы .MODEL определяет модель памяти и может принимать одно из значений, указанных в табл. 8.2. Все модели памяти, за исключением линейной (flat), используются при создании 16-разрядных программ для реального режима адресации.

Таблица 8.2. Типы моделей памяти

Модель	*Название*	*Описание*
tiny	Крошечная	Один общий сегмент кода и данных размером не более 64 Кбайт. Используется для создания исполняемых файлов для системы MS DOS с расширением .COM
small	Малая	Один сегмент кода и один сегмент данных. По умолчанию используются ближние ссылки на код и данные
medium	Средняя	Несколько сегментов кода и один сегмент данных
compact	Компактная	Один сегмент кода и несколько сегментов данных
large	Большая	Несколько сегментов кода и данных
huge	Огромная	Аналогична модели large, за исключением того, что размер отдельных элементов данных может превышать один сегмент, т.е. быть больше чем 64 Кбайт
flat	Линейная	Используется при создании программ для защищенного режима. Для ссылок на код и на данные используются 32-разрядные указатели. Весь код и данные программы (включая системные ресурсы) размещаются в одном 32-разрядном сегменте, размером до 4 Гбайт

Во всех программах, рассматриваемых в данной книге и написанных для реального режима адресации, используется малая модель памяти, т.е. весь код и все данные в них (включая область стека) собраны в два отдельных сегмента. Вследствие этого в программе для ссылок на код и на данные используются только 16-разрядные смещения относительно соответствующего сегмента, а значения сегментных регистров никогда не изменяются.

В программах, написанных для защищенного режима, используется линейная модель памяти и 32-разрядные ссылки на код и на данные. В таких программах суммарный объем кода и данных не имеет каких-либо практических ограничений и может составлять максимум 4 Гбайт. Например, в файле `Irvine32.inc` используется приведенная ниже директива `.MODEL`:

```
.model flat,stdcall
```

Параметры модели памяти. Второй параметр директивы `.MODEL` может содержать как описатель языка программирования высокого уровня, так и тип стека (ближний или дальний). *Описатель языка* определяет порядок передачи параметров при вызове процедур, а также соглашение о присвоении именам процедурам и общедоступным символам. Подробнее об этом мы поговорим в разделе 8.4.3.1. Параметр, определяющий *тип стека*, может принимать одно из значений: `NEARSTACK` (по умолчанию) или `FARSTACK`. Эти описатели используются только в особых случаях, поэтому здесь мы их рассматривать не будем.

8.4.2. Описатели языка программирования высокого уровня

Теперь давайте рассмотрим описатели языка программирования высокого уровня, используемые в директиве `.MODEL`. Они позволяют программисту создавать на языке ассемблера процедуры, совместимые с теми, которые автоматически генерирует компилятор с соответствующего языка. Допускаются следующие описатели: `C`, `BASIC`, `FORTRAN`, `PASCAL`, `SYSCALL` и `STDCALL`. Первые четыре описателя определяют один из языков программирования, с которым должны быть совместимы процедуры, написанные на языке ассемблера. Два последних описателя по сути являются вариациями первых четырех.

В этой книге мы сосредоточимся на изучении трех самых популярных описателей: `C`, `PASCAL` и `STDCALL`. Ниже приведен пример использования каждого из них в директиве `.model`, определяющей линейную модель памяти:

- `.model flat, C`
- `.model flat, pascal`
- `.model flat, stdcall`

Во всех примерах программ, приведенных в этой книге, используется описатель типа языка `STDCALL`. Именно благодаря ему можно напрямую вызывать функции из библиотек системы MS Windows.

8.4.2.1. Описатель STDCALL

При использовании описателя `STDCALL`, компилятор ассемблера при вызове процедуры с помощью директивы `INVOKE` помещает ее аргументы в стек в обратном порядке

(т.е. первым в стек помещается аргумент, который указан в директиве INVOKE последним). Например, при компиляции директивы INVOKE

```
INVOKE   AddTwo,5,6
```

генерируются следующие ассемблерные команды:

```
push   6                    ; Второй аргумент
push   5                    ; Первый аргумент
call   AddTwo
```

Кроме порядка помещения аргументов в стек, существует еще одно важное соглашение о том, в каком месте программы они должны удаляться из стека. При использовании описателя STDCALL, аргументы из стека удаляются в момент возврата из процедуры с помощью особой формы команды RET с непосредственно заданным значением. Это значение просто прибавляется к регистру ESP после извлечения из стека адреса возврата из процедуры:

```
AddTwo PROC
.
.
ret 8                        ; К ESP прибавляется число 8
                             ;  после извлечения адреса возврата
                             ;  из процедуры
AddTwo ENDP
```

Если прибавить к указателю стека число 8, то мы тем самым вернем его состояние к тому, которое было до помещения аргументов в стек перед вызовом процедуры.

Кроме того, использование описателя STDCALL приводит к тому, что общедоступные имена процедур при экспортировании заменяются на другие, которые имеют следующий формат:

```
_имя@nn
```

Перед именем общей процедуры добавляется символ подчеркивания, а после него — символ @ и одна или несколько цифр, означающих количество байтов, которые занимают в стеке параметры процедуры, округленные в большую сторону до значения, кратного 4.

Например, предположим, что общая процедура **MySub** имеет два параметра типа двойного слова. Тогда компилятор ассемблера передаст компоновщику следующее имя общей процедуры: **_MySub@8**.

> Важно отметить, что по умолчанию компоновщик LINK32.EXE различает регистр символов. Это означает, что имена _MYSUB@8 и _MySub@8 считаются разными. Чтобы ознакомиться с полным списком имен объектного файла, воспользуйтесь утилитой DUMPBIN и укажите ей в командной строке параметр /SYMBOLS.

8.4.2.2. Описатель C

При использовании описателя C компилятор ассемблера помещает в стек параметры процедуры в обратном порядке, так же как и при STDCALL.

Что касается удаления аргументов из стека после вызова процедуры, то здесь подход несколько иной. Изменение значения регистра указателя стека ESP выполняется после возврата из процедуры уже в вызывающей программе. Поэтому при компиляции директивы INVOKE, рассмотренной в предыдущем примере, будет сгенерирован следующий код:

```
push   6                          ; Второй аргумент
push   5                          ; Первый аргумент
call   AddTwo
add   esp,8                       ; Удаление аргументов из стека
```

Кроме того, в отличие от STDCALL, при использовании описателя C перед именами общих процедур добавляется символ подчеркивания.

8.4.2.3. Описатель PASCAL

При использовании описателя PASCAL аргументы процедуры помещаются в стек в том порядке, в котором они указаны (т.е. слева направо). Например, при компиляции директивы INVOKE

```
INVOKE   AddTwo,5,6
```

генерируются следующие ассемблерные команды:

```
push   5                          ; Первый аргумент
push   6                          ; Второй аргумент
call   AddTwo
```

Что касается удаления аргументов из стека после вызова процедуры, то здесь применяется такой же подход, как и при использовании описателя STDCALL.

Если используется описатель PASCAL, то при передаче в объектном файле имени общей переменной компоновщику, все буквы в нем просто заменяются на прописные. Например, имя процедуры **AddTwo** будет преобразовано в **ADDTWO**.

8.4.3. Непосредственный доступ к параметрам в стеке

В разделе 8.3 мы уже рассматривали способ доступа к параметрам процедуры по имени. Кроме того, вы можете явно обратиться к параметрам процедуры, воспользовавшись формой записи, наподобие [ebp+8]. Однако при этом вы должны четко представлять себе структуру стека и понимать, что делает программа. При использовании такого способа доступа к параметрам их не нужно объявлять в заголовке процедуры. В результате вы не сможете воспользоваться директивой INVOKE, поскольку для этого нужно будет описать прототип процедуры с параметрами. В вызывающей программе нужно будет вручную поместить параметры в стек.

Например, перед вызовом процедуры **AddTwo** нужно поместить в стек два целых числа. Она возвращает в регистре EAX сумму этих чисел. Чтобы передать аргументы по методу STDCALL, мы должны поместить их в стек в обратном порядке:

```
.data
 sum   DWORD  ?
```

```
.code
   push   6                    ; Второй аргумент
   push   5                    ; Первый аргумент
   call   AddTwo               ; EAX = сумма
   mov    sum,eax              ; Сохраним сумму
```

Первое, что нужно сделать в процедуре **AddTwo**, — это сохранить в стеке значение регистра EBP. Этот момент нужно особенно отметить, поскольку в регистре EBP обычно хранится важное значение, которое используется в вызывающей программе. После этого в регистр EBP нужно загрузить текущее значение регистра ESP; оно будет использоваться в качестве базового при обращении к стековому фрейму:

```
AddTwo PROC
   push ebp
   mov  ebp,esp
```

Структура стекового фрейма после выполнения этих двух команд показана на рис. 8.3.

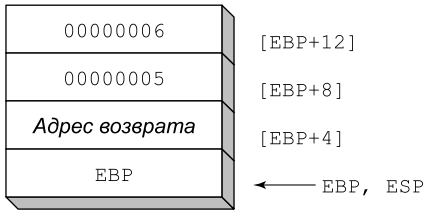

***Рис. 8.3.** Структура стекового фрейма процедуры AddTwo*

Два аргумента, числа 5 и 6, переданные процедуре, размещены по адресам EBP+8 и EBP+12, соответственно. Не забывайте, что число 6 было помещено в стек перед числом 5 и что стек "растет" от старших адресов к младшим. Учитывая все это в процедуре **AddTwo**, можно легко извлечь параметры из стека, сложить их и вернуть сумму в регистре EAX:

```
AddTwo PROC
   push  ebp
   mov   ebp,esp             ; Загрузим базу стекового фрейма
   mov   eax,[ebp + 12]      ; Загрузим второй аргумент
   add   eax,[ebp + 8]       ; Сложим его с первым аргументом
   pop   ebp
   ret 8                     ; При возврате удалим аргументы
                             ;  из стека
AddTwo ENDP
```

Обратите внимание, что для удаления аргументов по методу STDCALL, мы указали константу в качестве параметра команды RET. Этого можно было бы и не делать. Однако поступив так, мы полностью выполнили требования по вызову процедур, предусмотренные в методе STDCALL, и тем самым сделали процедуру **AddTwo** совместимой с другими процедурами, в которых используется аналогичный описатель.

В защищенном 32-разрядном режиме для процедуры, имеющей *n* параметров с именами P_i, где $i = \{1,2,...,n\}$, для обращения к каждому параметру в стеке можно воспользоваться выражением `[EBP+`k_i`]`, где $k_i = (i + 1) * 4$. Например, P_1 = `[ebp+8]`, P_2 = `[ebp+12]`, P_3 = `[ebp+16]`. Порядок нумерации параметров *I* зависит от используемой модели памяти. Например, в модели `STDCALL` параметры помещаются в стек в обратном порядке, т.е. от P_n до P_1. При использовании малой модели памяти в реальном режиме, формула для вычисления k_i будет иметь вид: $k_i = (i + 1) * 2$.

8.4.3.1. Сохранение и восстановление регистров

В процедурах часто после того, как в регистр `EBP` будет загружено содержимое регистра `ESP`, в стек помещаются значения других регистров, используемых в программе. В результате их значения могут быть восстановлены при возврате из процедуры. Напомним, в главе 5 мы уже говорили о том, что при возврате из процедуры должны быть восстановлены значения всех регистров, которые в ней были изменены. Это делается для того, чтобы в вызывающей программе можно было распоряжаться любыми регистрами на усмотрение программиста.

В приведенном ниже фрагменте кода после того, как в процедуре **AddTwo** в регистр `EBP` будет загружен адрес стекового фрейма, в стеке дополнительно сохраняется значение регистра `EDX`:

```
AddTwo PROC
  push  ebp
  mov   ebp,esp                 ; Загрузим адрес стекового фрейма
  push  edx                     ; Сохраним регистр EDX
  .
  .
  pop   edx                     ; Восстановим регистр EDX
  pop   ebp
  ret   8                       ; Удалим аргументы из стека
AddTwo ENDP
```

Помещение в стек регистра `EDX` не влияет на величину смещения параметров в стеке относительно регистра `EBP`, поскольку стек "растет" от старших адресов к младшим и значение в регистре `EBP` не меняется (рис. 8.4).

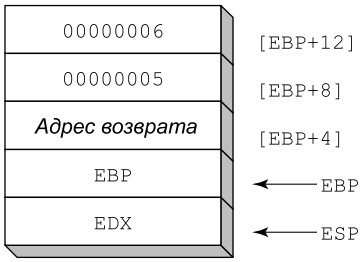

Рис. 8.4. *Структура стека процедуры* ***AddTwo*** *после сохранения регистра* `EDX`

8.4.4. Передача аргументов по ссылке

В предыдущих разделах во всех рассматриваемых нами примерах аргументы передавались в процедуры по значению. Однако иногда возникает необходимость передать в процедуру адрес переменной или массива. Напомним, что подобный способ называется *передачей аргументов по ссылке*.

8.4.4.1. Пример процедуры ArrayFill

Давайте напишем процедуру **ArrayFill**, которая инициализирует массив 16-разрядных случайных чисел. Ей должно передаваться два аргумента: адрес массива и количество его элементов. Очевидно, что первый аргумент нужно передавать по ссылке, а второй — по значению. Вызов такой процедуры не представляет особого труда. Нам нужно поместить в стек смещение массива и его размер, как показано ниже:

```
.data
 count = 100
 array   WORD   count DUP(?)

.code
 push   OFFSET array
 push   COUNT
 call   ArrayFill
```

Структура стекового фрейма процедуры **ArrayFill**, содержащего смещение массива **array** и число его элементов **count**, показана на рис. 8.5.

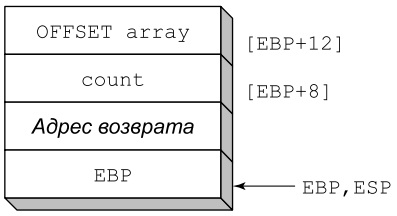

Рис. 8.5.** Структура стекового фрейма процедуры **ArrayFill

Чтобы внутри процедуры **ArrayFill** загрузить адрес массива **array** в регистр ESI, нужно воспользоваться приведенной ниже командой:

```
mov   esi,[ebp+12]              ; Загрузим смещение массива array
```

Вот полный текст процедуры **ArrayFill**:

```
ArrayFill PROC
  push  ebp
  mov   ebp,esp
  pushad
  mov   esi,[ebp+12]            ; Загрузим смещение массива array
  mov   ecx,[ebp+8]             ; Загрузим размер массива
  cmp   ecx,0                   ; ECX < 0?
  jle L2                        ; Да, выйдем из процедуры
```

```
L1:
  mov   eax,10000h              ; Сгенерируем случайное число
                                ;  в диапазоне 0 - FFFFh,
  call  RandomRange             ;  воспользовавшись библиотечной
                                ;  процедурой
  mov   [esi],ax                ; Сохраним его в текущем элементе
                                ;  массива
  add   esi,TYPE WORD           ; Вычислим адрес следующего
                                ;  элемента
  loop  L1

L2:
  popad
  pop   ebp
  ret   8                        ; Удалим аргументы из стека

ArrayFill ENDP
```

8.4.4.2. Команда LEA

Команда `LEA` (Load Effective Address, или загрузить текущий адрес) позволяет определить текущее смещение косвенного операнда любого типа. Поскольку при косвенной адресации может задействоваться один или два регистра общего назначения, нужно иметь средство для вычисления текущего смещения операнда во время выполнения программы. Рассмотренный выше оператор ассемблера `OFFSET` позволяет только определить смещение переменной на этапе компиляции.

Командой `LEA` удобно пользоваться для определения адреса параметра, находящегося в стеке. Например, если в процедуре определяется локальный массив, то для работы с ним часто нужно загрузить его смещение в индексный регистр. В приведенной ниже процедуре **`FillString`** как раз это и делается, после чего всем элементам байтового массива присваиваются случайные ASCII-цифры, значение которых находится в диапазоне 0–9:

```
FillString   PROC   USES eax esi
    LOCAL   string[20]:BYTE
; Создадим локальный 20-байтовый массив и запишем в него
; случайные ASCII-цифры, значение которых находится в диапазоне 0...9.
   lea   esi,string             ; Загрузим текущий адрес массива
   mov   ecx,20
L1:
   mov   eax,10
   call  RandomRange            ; AL = 0..9
   add   al,30h                 ; Преобразуем цифру в ASCII-код
   mov   [esi],al
   add   esi,1
   Loop  L1
   ret
FillString ENDP
```

Обратите внимание, что адрес массива **`string`** определяется не прямо, а косвенно (через регистр `EBP`), поэтому приведенная ниже команда вызовет сообщение компилятора об ошибке.

```
mov   eax,OFFSET string          ; Ошибка! Команду MOV..OFFSET
                                 ;  можно использовать только
                                 ;  для операндов, адреса которых
                                 ;  определяются непосредственно.
```

8.4.5. Создание локальных переменных

Выше мы уже говорили о преимуществе локальных переменных по сравнению с глобальными. Для создания локальных переменных необязательно пользоваться директивой компилятора LOCAL. Для тех, кому нравится все держать под контролем, это можно сделать и "вручную", выделив из стека участок памяти подходящего размера.

Пример на C++. В приведенном ниже фрагменте кода на C++ в функции **MySub** создается несколько локальных переменных: **X**, **Y**, **name** и **Z**:

```
void MySub()
{
   char X = 'X';
   int Y = 10;
   char name[20];
   name[0] = 'B';
   double Z = 1.2;
}
```

Этот фрагмент кода очень легко реализовать на языке ассемблера, если взять за основу соглашения, используемые в компиляторе Visual C++. По умолчанию каждый элемент стека имеет размер 32 бита. Поэтому размер памяти, выделяемый под каждую локальную переменную, округляется в большую сторону и всегда будет кратен 4. Общая длина памяти, занимаемой локальными переменными, равна 36 байтам (табл. 8.3).

Таблица 8.3. Распределение памяти под локальные переменные

Имя переменной	*Размер в байтах*	*Смещение в стеке*
X	4	EBP-4
Y	4	EBP-8
name	20	EBP-28
Z	8	EBP-36

В приведенной ниже реализации на языке ассемблера процедуры **MySub**, создаются четыре локальные переменные, которым присваиваются начальные значения. При выходе из процедуры локальные переменные аннулируются. Переменной **Z** назначается 64-разрядная константа, которая является закодированным числом с плавающей точкой:

```
MySub PROC
  push  ebp
  mov   ebp,esp
  sub   esp,36                   ; Создадим локальные переменные

  mov   BYTE PTR [ebp-4],'X'     ; X
  mov  DWORD PTR [ebp-8],10      ; Y
```

```
  mov   BYTE PTR [ebp-28],'Y'   ; name[0]
  mov DWORD PTR [ebp-32],3ff33333h ; Z(старшие 32 бита)
  mov DWORD PTR [ebp-36],33333333h ; Z(младшие 32 бита)

  mov   esp,ebp                    ; Аннулируем переменные
  pop ebp
  ret
MySub ENDP
```

8.4.6. Команды ENTER и LEAVE (*дополнительный материал*)

Команда ENTER предназначена для автоматического создания стекового фрейма в вызванной процедуре. Она позволяет выделить место под локальные переменные и сохранить в стеке регистр EBP. В частности, она выполняет три перечисленных ниже действия:

- сохраняет в стеке регистр EBP (выполняет команду push ebp);
- загружает в регистр EBP базовый адрес стекового фрейма (выполняет команду mov ebp,esp);
- выделяет память под локальные переменные (выполняет команду sub esp, *Размер_области*).

Команда ENTER имеет два операнда. Первый операнд является константой, которая указывает размер области в байтах, выделяемой для локальных переменных. Второй операнд также является константой. Он указывает лексический уровень вложенности процедуры:

```
ENTER Размер_области, Уровень_вложенности
```

Лексический уровень вложенности определяет глубину расположения процедуры в иерархии вложенных вызовов процедур. Это позволяет получить доступ из процедуры более низкого уровня вложенности к локальным переменным процедуры более высокого уровня вложенности. Поскольку подобная методика используется только в компиляторах языков высокого уровня, при программировании она практически не используется из-за сложности вручную отслеживать уровни вложенности процедур. Прояснить алгоритм работы команды ENTER поможет следующий псевдокод:

```
Уровень_вложенности = Уровень_вложенности MOD 32
   if Размер_Стека = 32 then
      Push(EBP);
      FrameTemp = ESP;
   else /* Размер_Стека = 16 */
      Push(BP);
      FrameTemp = SP;
   endif;

   if Уровень_вложенности = 0 then GOTO CONTINUE;

   if (Уровень_вложенности > 0) then
      for i = 1 to (Уровень_вложенности - 1) do
          if Размер_Операнда = 32 then
             if Размер_Стека = 32 then
```

```
                    EBP = EBP - 4;
                    Push(DWORD PTR [EBP]);
                 else /* Размер_Стека = 16 */
                    BP = BP - 4;
                    Push(DWORD PTR [BP]);
                 endif;

              else /* Размер_Операнда = 16 */
                 if Размер_Стека = 32 then
                    EBP = EBP - 2;
                    Push(WORD PTR[EBP]);
                 else /*Размер_Стека = 16 */
                    BP = BP - 2;
                    Push(WORD PTR [BP]);
                 endif;
              endif;
           enddo;

           if Размер_Операнда = 32 then
              Push(FrameTemp); /* Двойное слово */
           else /* Размер_Операнда = 16 */
              Push(FrameTemp); /* Слово */
           endif;
           GOTO CONTINUE;
        endif

     CONTINUE:
     if Размер_Стека = 32 then
        EBP = FrameTemp
        ESP = EBP - Размер_области_локальных_переменных;
     else /* Размер_стека = 16*/
        BP = FrameTemp
        SP = BP - Размер_области_локальных_переменных;
     endif;
```

Таким образом, если уровень вложенности процедуры не равен нулю, то команда `ENTER` после помещения в стек регистра `EBP` последовательно записывает в стек адреса стековых фреймов всех процедур предыдущего уровня, что позволяет при необходимости легко обратиться к их локальным переменным.

Пример 1. В приведенном ниже фрагменте программы объявляется процедура, которая не имеет локальных переменных:

```
MySub PROC
   enter 0,0
```

Это эквивалентно следующим машинным командам:

```
MySub PROC
   push  ebp
   mov   ebp,esp
```

Пример 2. Приведенная ниже команда `ENTER` резервирует в стеке 8 байтов для локальных переменных:

```
MySub PROC
   enter 8,0
```

Она эквивалентна следующим командам:

```
MySub PROC
   push  ebp
   mov   ebp,esp
   sub   esp,8
```

> При использовании в начале процедуры команды ENTER мы настоятельно рекомендуем вам пользоваться в конце той же процедуры командой LEAVE. В противном случае пространство памяти, выделенное в стеке под локальные переменные, так и не будет освобождено. В результате при выполнении команды RET из стека будет извлечен некорректный адрес возврата.

***Команда** LEAVE*. Эта команда позволяет завершить использование стекового фрейма в процедуре. Она выполняет действия, противоположные ранее использовавшейся команде ENTER — восстанавливает содержимое регистров ESP и EBP к тому состоянию, которое было в момент вызова процедуры. Прояснить алгоритм работы команды LEAVE поможет следующий псевдокод:

```
if Размер_Стека = 32 then
   ESP = EBP;
else /* Размер_Стека = 16 */
   SP = BP;
endif;

if Размер_Операнда = 32 then
   EBP = Pop();
else /* Размер_Операнда = 16 */
   BP = Pop();
endif;
```

Снова возвращаясь к примеру процедуры **MySub**, его можно переписать следующим образом:

```
MySub PROC
  enter 8,0
  .
  .
  leave
  ret
MySub ENDP
```

Ниже приведена эквивалентная последовательность команд для резервирования в начале процедуры места в стеке под локальные переменные размером 8 байтов и его освобождения в конце процедуры:

```
MySub PROC
   push  ebp
   mov   ebp,esp
   sub   esp,8
```

```
    .
    .
    mov   esp,ebp
    pop   ebp
    ret
MySub ENDP
```

8.4.7. Контрольные вопросы раздела

1. (*Да/Нет*). Регистр EBP сохраняется в процедуре, использующей стековые параметры, всякий раз, как только происходит изменение его содержимого.
2. (*Да/Нет*). Для создания локальных переменных необходимо к указателю стека прибавить положительное целое число.
3. (*Да/Нет*). В процедурах, рассмотренных в этой главе, адрес последнего помещенного в стек аргумента был [ebp+8].
4. (*Да/Нет*). Передача аргументов по ссылке приводит к тому, что внутри вызванной процедуры смещение параметра выталкивается из стека.
5. Опишите параметры малой модели памяти.
6. Опишите параметры линейной модели памяти.
7. Чем отличаются имена внешних процедур, которые компилятор ассемблера передает компоновщику, при использовании параметров C и PASCAL директивы .MODEL?
8. Как удаляются аргументы процедуры из стека при использовании описателя языка STDCALL в директиве .MODEL?
9. Ниже приведена последовательность команд вызова процедуры **AddThree**, в которой складываются три двойных слова (будем считать, что в директиве .MODEL используется описатель языка STDCALL):

   ```
   push  10h
   push  20h
   push  30h
   call  AddThree
   ```

   Изобразите графически структуру стекового фрейма процедуры **AddThree** сразу после сохранения в стеке регистра EBP.
10. Напишите последовательность команд для процедуры **AddThree**, о которой шла речь в предыдущем упражнении, которые вычисляют сумму трех стековых параметров.
11. В чем состоит принципиальное отличие команды LEA от MOV...OFFSET?
12. Какое количество памяти выделяется для переменной типа char при выполнении процедуры **MySub**, написанной на языке C++ и рассмотренной в разделе 8.4.5?
13. *Обсуждение проблемы.* Какие преимущества имеет соглашение о вызовах процедур, принятое в языке C, по сравнению с языком Pascal?

8.5. Рекурсия

Рекурсивной называется такая процедура, которая явно или неявно вызывает сама себя. *Рекурсия*, или практика вызова рекурсивных процедур, является очень мощным средством при работе со структурами данных, которые имеют периодический характер. В качестве примера здесь уместно привести связанные списки и различные типы связных графов, при обработке которых программа должна последовательно "обойти" все их узлы.

Бесконечная рекурсия. Самый очевидный тип рекурсии возникает в случае, когда процедура явно вызывает сама себя. Например, в приведенной ниже программе существует процедура **Endless**, которая без всяких условий постоянно вызывает сама себя:

```
TITLE   Бесконечная рекурсия        (Endless.asm)

 INCLUDE Irvine32.inc
.data
 endlessStr   BYTE   "Эта рекурсия никогда не закончится",0

.code
 main PROC
      call  Endless
      exit
main ENDP

Endless PROC
      mov   edx,OFFSET endlessStr
      call  WriteString
      call  Endless
      ret                       ; Эта команда никогда
                                ;  не будет выполнена
Endless ENDP
END main
```

Очевидно, что этот пример не имеет какой-либо практической ценности. При каждом вызове рекурсивной процедуры **Endless** из стека будет "забираться" 4 байта, поскольку команда `CALL` помещает в него адрес возврата из процедуры. Кроме того, до команды `RET` управление так никогда и не дойдет; рано или поздно, выполнение программы завершится аварийно из-за того, что место в стеке будет исчерпано.

> Если вы работаете в среде Windows 2000/XP, для наблюдения за работой описанной выше программы лучше всего воспользоваться системным монитором. Откройте диспетчер задач, нажав комбинацию клавиш <Ctrl+Alt+Del>, перейдите на вкладку Performance (Быстродействие) и запустите программу `Endless.exe`, находящуюся в каталоге примеров к этой главе. Для этого воспользуйтесь командой меню File⇨New Task (Run…) [Файл⇨Новая задача (Выполнить…)]. Вы увидите, что количество свободной памяти, выделенное для нашей задачи, будет медленно уменьшаться, а ресурсы процессора будут расходоваться на все 100%. По истечении некоторого времени стек программы переполнится, что вызовет прерывание в работе процессора. В результате работа программы `Endless.exe` будет аварийно завершена.

8.5.1. Рекурсивное вычисление суммы

Рекурсивный вызов процедуры приобретает практический смысл только тогда, когда внутри такой процедуры предусмотрены условия для завершения ее работы. Как только условие завершения рекурсивной процедуры становится истинным, цепочка вызовов такой процедуры начинает "разматываться" в обратном направлении. При этом выполняются все "зависшие" до этого команды RET. В качестве иллюстрации давайте создадим рекурсивную процедуру **CalcSum**, которая вычисляет сумму целых чисел от 1 до *n*, где *n* — входной параметр, передаваемый в регистре ECX. Сумма чисел возвращается в регистре EAX:

```
TITLE   Вычисление суммы целых чисел   (CSum.asm)

INCLUDE Irvine32.inc

.code
 main PROC
     mov   ecx,5               ; Значение счетчика = 5
     mov   eax,0               ; Значение суммы = 0
     call  CalcSum             ; Вычислим сумму
L1:
     call  WriteDec            ; Отобразим регистр eax
     call  CrLf                ; Переведем строку
     exit
 main ENDP

 ;-----------------------------------------------------------------
 CalcSum PROC
 ; Вычисляет сумму последовательных целых чисел
 ; Передается:   ecx = счетчик чисел
 ; Возвращается: eax = сумма чисел
 ;-----------------------------------------------------------------
     cmp   ecx,0               ; Счетчик равен нулю?
     jz L2                     ; Да, выйдем из программы
     add   eax,ecx             ; Нет, прибавим текущее значение
                               ;  счетчика к сумме
     dec   ecx                 ; Уменьшим на 1 значение счетчика
     call  CalcSum             ; Рекурсивный вызов процедуры
 L2:
     ret
 CalcSum ENDP
 end Main
```

В первых двух строках процедуры **CalcSum** проверяется условие завершения работы, которое истинно при условии ECX = 0. При этом дальнейшие рекурсивные вызовы не выполняются и управление передается команде RET. При первом выполнении этой команды управление возвращается в предыдущий вызов процедуры **CalcSum**, команда RET которой снова возвращает управление предыдущему вызову процедуры **CalcSum** и т.д. до самого верхнего уровня вызовов. В табл. 8.4 показаны (в виде меток программы) адреса возврата из процедуры, которые помещает в стек команда CALL, а также текущие значения регистра ECX (счетчика) и EAX (суммы).

Таблица 8.4. Стековые фреймы процедуры CalcSum

Адрес возврата	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

Из приведенного выше примера видно, что даже для вызова простейшей рекурсивной процедуры требуется довольно много стекового пространства. При каждом вызове такой процедуры из стека выделяется минимум 4 байта, в которые командой `CALL` заносится адрес возврата из процедуры.

8.5.2. Вычисление факториала

В большинстве рекурсивных процедур используются стековые параметры, поскольку стек идеально подходит для сохранения временных данных во время рекурсивного процесса. Эти данные используются для завершения процесса рекурсии и возврата в вызвавшую подпрограмму.

Следующий пример, который мы должны рассмотреть, является своего рода "классикой жанра" рекурсивных процедур и связан с вычислением факториала целого беззнакового числа *n*. Ниже приведен исходный код функции **`factorial`**, записанный на языке высокого уровня C/C++/Java. Ей в качестве параметра передается исходное число *n*, факториал которого нужно вычислить:

```
int function factorial(int n)
{
    if(n == 0)
         return 1;
    else
         return n * factorial(n-1);
}
```

Рекурсивный алгоритм вычисления факториала числа *n* основан на том предположении, что для любого неотрицательного *n* мы можем вычислить факториал числа $n - 1$. Тогда процесс вычисления *n*! будет продолжаться до тех пор, пока *n* не станет равным нулю. По определению 0! равен 1. Собственно значение выражения *n*! вычисляется во время обратного хода алгоритма, когда происходит накопление результатов каждого умножения. Например, на рис. 8.6 показан процесс вычисления 5!. В левом столбце изображена нисходящая часть алгоритма, а в правом — восходящая.

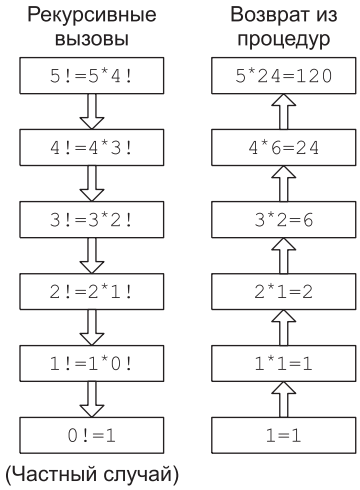

Рис. 8.6. *Иллюстрация рекурсивного алгоритма вычисления 5!*

Ниже приведена реализация рекурсивного алгоритма вычисления факториала на языке ассемблера. Перед вызовом процедуры **`Factorial`** в стек помещается число *n* (целое беззнаковое число от 0 до 12). Значение факториала возвращается в регистре `EAX`. Поскольку используется один 32-разрядный регистр общего назначения `EAX`, то максимальное значение факториала, которое может в него поместиться, равно 479 001 600, или 12!:

```
TITLE   Вычисление факториала     (Fact.asm)

INCLUDE Irvine32.inc
.code
 main PROC
   push  12                    ; Вычислим 12!
   call  Factorial             ; Результат в EAX
ReturnMain:
   call  WriteDec              ; Отобразим результат
   call  CrLf
   exit
main ENDP

;-----------------------------------------------------------
Factorial PROC
; Процедуры вычисления факториала.
; Передается: [ebp+8] = n, исходное число, факториал
;   которого нужно вычислить
; Возвращается: eax = факториал числа n
;-----------------------------------------------------------
   push  ebp
   mov   ebp,esp
   mov   eax,[ebp+8]           ; Загрузим число n
   cmp   eax,0                 ; n > 0?
   ja    L1                    ; Да, продолжим вычисление
```

```
    mov   eax,1                   ; Нет, вернем 1
    jmp   L2

 L1:
    dec   eax
    push  eax                     ; Вычислим (n-1)!
    call  Factorial
 ; Команды, расположенные в этом месте программы,
 ;  выполняются после возврата из рекурсивной процедуры.
ReturnFact:
    mov   ebx,[ebp+8]             ; Загрузим n
    mul   ebx                     ; edx:eax = eax * ebx
L2:
    pop ebp                       ; Выйдем из процедуры
                                  ;  и возвратим результат в EAX
    ret 4                         ; Удалим аргумент из стека
 Factorial ENDP
 END main
```

При вызове процедуры **`Factorial`** в стек помещается адрес следующей за ней команды. В процедуре **`main`** это будет адрес метки **`ReturnMain`**, а в процедуре **`Factorial`** — адрес метки **`ReturnFact`**. На рис. 8.7 показана структура стека программы `Fact.asm` после выполнения нескольких рекурсивных вызовов. Нетрудно заметить, что при каждом рекурсивном вызове программы **`Factorial`** в стек, кроме адреса возврата, помещается новое значение числа *n* и регистр `EBP`.

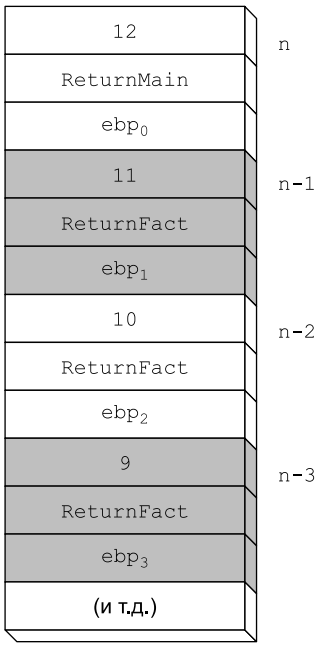

Рис. 8.7. *Использование стека в программе вычисления факториала*

В нашем примере при каждом вызове процедуры **Factorial** из стека выделяется 12 байтов памяти. При каждом рекурсивном вызове этой процедуры в стек помещается значение входного параметра, равного $n - 1$. Процедура возвращает в регистре EAX вычисленное значение факториала, которое затем умножается на число, которое было помещено в стек перед вызовом процедуры **Factorial**.

8.5.3. Контрольные вопросы раздела

1. (*Да/Нет*). При выполнении одних и тех же действий рекурсивная процедура требует выделения меньшего количества памяти из стека, чем нерекурсивная.
2. При выполнении какого из условий в процедуре **Factorial** прекращается рекурсивный вызов этой процедуры?
3. Какая команда выполняется в процедуре **Factorial** после завершения команды рекурсивного вызова?
4. Что произойдет, если с помощью процедуры **Factorial** попытаться вычислить значение выражения 13!?
5. *Задача повышенной сложности*. Какое количество стековой памяти требуется для работы программы **Factorial** при вычислении 12!?
6. *Задача повышенной сложности*. Запишите на псевдокоде рекурсивный алгоритм, который генерирует первые 20 чисел последовательности Фибоначчи (1, 1, 2, 3, 5, 8, 13, 21,...). Объясните, почему этот алгоритм не является эффективным.

8.6. Создание многомодульных программ

При разработке крупных проектов, с программой очень неудобно работать, когда весь ее исходный код находится в одном файле. Поэтому для удобства имеет смысл разбить весь проект на несколько файлов с исходным кодом, которые называются *модулями*. Тем самым вы облегчите себе задачу последующего анализа такого кода и внесения в него изменений. Если изменения будут внесены только в один модуль, то потребуется перекомпилировать только его и затем выполнить повторную сборку всей программы компоновщиком. В целом, можно сказать, что перекомпоновка объектных модулей программы выполняется гораздо быстрее, чем ассемблирование большого исходного файла.

При создании многомодульной программы обычно выполняют несколько стандартных действия, описанных ниже.

- Создают основной исходный модуль (ASM-файл) проекта. В него помещают процедуру начального запуска `main` и ряд других вспомогательных процедур.
- Для каждой большой процедуры проекта создают отдельный модуль. При использовании небольших процедур имеет смысл собрать их в одном модуле.
- В основном модуле необходимо описать с помощью директивы PROTO имена и параметры всех вызываемых процедур.
- При необходимости включить директивы PROTO во все модули программы. Строго говоря, в каждом модуле нужно описать с помощью директив PROTO только вызываемые процедуры, поскольку неиспользуемые прототипы попросту игнорируются компилятором ассемблера.

Легче всего хранить файлы многомодульной программы в отдельном каталоге на диске. Этим мы и займемся при рассмотрении в следующем разделе программы ArraySum.

8.6.1. Пример: программа ArraySum

Программу ArraySum, которую мы рассматривали в главе 5, разбить на модули совсем несложно. Однако в этом разделе мы добавим в нее описанную выше возможность передачи параметров с помощью директив PROTO и INVOKE. Чтобы напомнить вам структуру программы, мы повторили здесь известный вам рисунок из главы 5. На нем выделены те процедуры, которые входят в библиотеку объектных модулей автора книги (рис. 8.8).

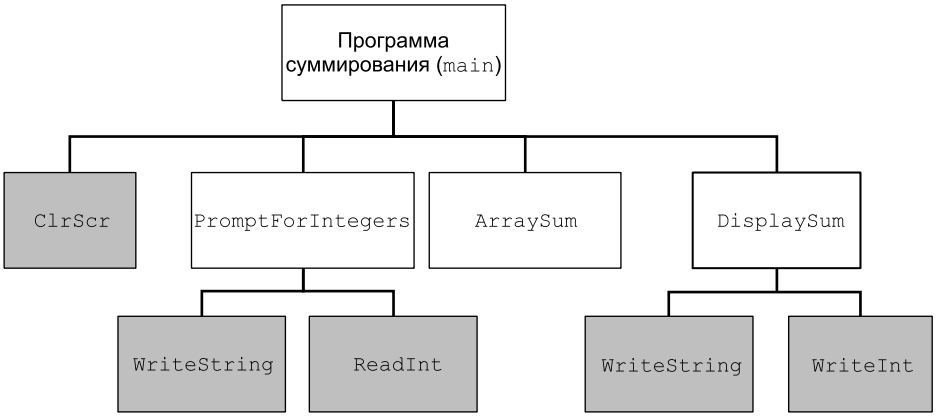

***Рис. 8.8.** Структурная схема программы* **ArraySum**

На этой структурной схеме показано дерево вызовов процедур программы ArraySum. Например, из процедуры **main** вызывается процедура **PromptForIntegers**, из которой в свою очередь вызываются процедуры **WriteString** и **ReadInt**.

8.6.1.1. Определение прототипов процедур

Для удобства мы поместим прототипы всех процедур в отдельном включаемом файле sum.inc. Этот файл имеет текстовый формат. В него включается другой файл Irvine32.inc, а также три прототипа процедур, используемых в программе:

```
; Включаемый файл программы ArraySum Program    (sum.inc)

INCLUDE Irvine32.inc

PromptForIntegers PROTO,
           ptrPrompt:PTR BYTE,  ; Адрес строки приглашения
            ptrArray:PTR DWORD, ; Адрес массива
           arraySize:DWORD      ; Число элементов массива

ArraySum PROTO,
```

```
              ptrArray:PTR DWORD, ; Адрес массива
              arraySize:DWORD     ; Число элементов массива

DisplaySum PROTO,
              ptrPrompt:PTR BYTE,  ; Адрес строки приглашения
                 theSum:DWORD      ; Сумма элементов массива
```

8.6.1.2. Основной модуль программы

Прежде всего, мы должны рассмотреть структуру основного модуля программы, который будет называться `Sum_main.asm`. В него мы поместили данные программы и основную процедуру. Кроме того, в этом модуле включается файл `sum.inc`, в котором описаны прототипы всех процедур, используемых в программе:

```
   TITLE Программа суммирования целых чисел     (Sum_main.asm)

; Эта программа запрашивает у пользователя несколько целых чисел,
   ; сохраняет их в массиве, вычисляет сумму элементов этого массива
   ; и отображает полученный результат на экране компьютера.

 INCLUDE Irvine32.inc

   INCLUDE sum.inc                 ; Подключим прототипы процедур

   ; Для изменения размера массива измените переменную Count:
   Count = 3

   .data
; Запрашивает у пользователя несколько целых чисел и
   ;  записывает их в массив.
   ; Передается: ESI = адрес массива двойных слов,
   ;             ECX = размер массива.
   ; Возвращается: ничего
   ; Вызывает: ReadInt, WriteString

   array   DWORD   Count   DUP(?)
   sum     DWORD ?

   .code
    main PROC
    call  ClrScr
    INVOKE PromptForIntegers,      ; Введем массив чисел
                ADDR prompt1,
                ADDR array,
                Count

    INVOKE ArraySum,               ; Просуммируем элементы массива
                ADDR array,        ; Сумма возвращается в EAX
                Count

    mov   sum,eax                  ; Сохраним значение суммы
                                   ;  в переменной
    INVOKE DisplaySum,             ; Отобразим сумму на экране
                ADDR prompt2,
```

```
                sum
    call  CrLf
    exit
main ENDP
END main
```

8.6.1.3. Модуль PromptForIntegers

Процедуру **PromptForIntegers** мы поместим в отдельный модуль, который назовем _prompt.asm. Начинать имя файла с символа подчеркивания вовсе не обязательно, но тем самым мы хотели подчеркнуть, что этот модуль является частью большого проекта. В этом модуле с помощью директивы INCLUDE также включается файл sum.inc:

```
TITLE    Процедура ввода целых чисел   ( модуль _prompt.asm)

 INCLUDE sum.inc
.code

;-----------------------------------------------------------------
   PromptForIntegers PROC,
            ptrPrompt:PTR BYTE,    ; Адрес строки приглашения
             ptrArray:PTR DWORD,   ; Адрес массива
            arraySize:DWORD        ; Число элементов массива
   ;
   ; Запрашивает у пользователя несколько целых чисел и
   ;  записывает их в массив.
   ; Возвращается: ничего
   ;-----------------------------------------------------------------
     pushad                        ; Сохраним все регистры
     mov   ecx,arraySize           ; Загрузим размер массива
     cmp   ecx,0                   ; Размер массива <= 0?
     jle   L2                      ; Да, завершим работу
     mov   edx,ptrPrompt           ; Загрузим адрес приглашения
     mov   esi,ptrArray            ; Загрузим адрес массива
  L1:
     call  WriteString             ; Выведем приглашение
     call  ReadInt                 ; Прочитаем число (оно в EAX)
     mov   [esi],eax               ; Запишем число в массив
     call  CrLf                    ; Перейдем на новую строку
                                   ;  на экране
     add   esi,4                   ; Скорректируем указатель
                                   ;  на следующий элемент массива
     loop  L1                      ; Повторим цикл для ввода всех
                                   ;   элементов массива
  L2:
     popad                         ; Восстановим все регистры
     ret
   PromptForIntegers ENDP
   END
```

В этой процедуре мы позаботились о сохранении и восстановлении всех регистров общего назначения с помощью команд PUSHAD и POPAD. В процедурах, входящих в библиотеки Irvine32.lib и Irvine16.lib, значения регистров также сохраняются,

поэтому в вызывающих их программах можно быть уверенным в том, что значения регистров не будут неожиданно изменены.

И последнее, поскольку модуль _prompt.asm не является стартовым по отношению ко всей программе в целом, в нем используется директива END без параметров. Если вы помните, точку входа мы указали в основном модуле.

8.6.1.4. Модуль ArraySum

Процедуру **ArraySum** поместим в модуль _arrysum.asm:

```
TITLE  Процедура ArraySum      (Модуль _arrysum.asm)

 INCLUDE sum.inc
.code
;-------------------------------------------------------------
ArraySum PROC,
         ptrArray:PTR DWORD,    ; Адрес массива
        arraySize:DWORD         ; Число элементов массива
;
; Вычисляет сумму элементов массива 32-разрядных целых чисел
; Возвращается: EAX = сумма элементов массива
;-------------------------------------------------------------
  push  ecx                     ; EAX сохранять не нужно!
  push  esi

  mov   eax,0                   ; Обнулим значение суммы
  mov   esi,ptrArray
  mov   ecx,arraySize
  cmp   ecx,0                   ; Размер массива <= 0?
  jle   L2                      ; Да, завершим работу
L1:
  add   eax,[esi]               ; Прибавим очередной элемент
                                ;  массива
  add   esi,4                   ; Вычислим адрес следующего
                                ;  элемента массива
  loop L1                       ; Повторим цикл для всех
                                ;  элементов массива
L2:
  pop   esi
  pop   ecx
  ret                           ; Вернем сумму в регистре EAX
ArraySum ENDP
END
```

В процедуре **ArraySum** используются регистры EAX, ECX и ESI, причем их содержимое изменяется. Поэтому вначале работы процедуры регистры ECX и ESI помещаются в стек, а при возврате — восстанавливаются из стека. В то же время, в регистре EAX возвращается вычисленное значение суммы, поэтому его содержимое в процедуре сохранять не нужно.

8.6.1.5. Модуль DisplaySum

Процедуру **DisplaySum** разместим в модуле _display.asm:

```
TITLE Процедура DisplaySum        ( Модуль _display.asm)

 INCLUDE sum.inc
.code
;-----------------------------------------------------------------
DisplaySum PROC,
       ptrPrompt:PTR BYTE,      ; Адрес строки приглашения
          theSum:DWORD          ; Сумма элементов массива
;
; Отображает сумму элементов массива на экране.
; Возвращается: ничего
;-----------------------------------------------------------------
  push  eax
  push  edx
  mov   edx,ptrPrompt           ; Загрузим адрес строки
                                ;  приглашения
  call  WriteString
  mov   eax,theSum
  call  WriteInt                ; Отобразим EAX на экране
  call  CrLf
  pop   edx
  pop   eax
  ret
DisplaySum ENDP
END
```

8.6.1.6. Командный файл для автоматического ассемблирования и компоновки

Для выполнения автоматического ассемблирования и компоновки нашей программы создадим специальный командный файл. В нем мы укажем имена всех исходных файлов, которые нужно откомпилировать, и имена объектных файлов и библиотек, из которых будет скомпонован исполняемый файл. Текст командного файла приведен ниже:

```
PATH c:\Masm615
   SET  INCLUDE=c:\Masm615\include
   SET  LIB=c:\Masm615\lib

   ML -Zi -c -Fl -coff Sum_main.asm _display.asm _arrysum.asm
_prompt.asm
   if errorlevel 1 goto terminate

   LINK32 Sum_main.obj _display.obj _arrysum.obj _prompt.obj
          irvine32.lib kernel32.lib /SUBSYSTEM:CONSOLE /DEBUG[1]
   if errorLevel 1 goto terminate
   dir
  :terminate
   pause
```

После запуска этого командного файла вы увидите на экране такой текст:

[1] Хотя команда вызова компоновщика приведена на страницах этой книги в двух строках, в командном файле все имена файлов должны быть перечислены в одной строке.

```
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.
Assembling: Sum_main.asm
Assembling: _display.asm
Assembling: _arrysum.asm
Assembling: _prompt.asm
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

8.6.2. Контрольные вопросы раздела

1. (*Да/Нет*). Процесс компоновки объектных модулей намного быстрее, чем компилирование исходных ASM-файлов.
2. (*Да/Нет*). Разделение большой программы на короткие модули усложняет ее дальнейшее сопровождение.
3. (*Да/Нет*). В многомодульной программе после оператора `END` нужно указывать параметр (имя стартовой процедуры) только в одном (основном) модуле.
4. (*Да/Нет*). Использование директивы `PROTO` приводит к повышенному расходу оперативной памяти, поэтому вы должны следить за тем, чтобы с помощью этих директив описывались только те процедуры, которые на самом деле будут вызываться, а не все подряд.

8.7. Резюме

Директива `LOCAL` предназначена для объявления одной или нескольких локальных переменных внутри процедуры. В исходном коде она должна располагаться сразу за директивой `PROC`. Локальные переменные имеют ряд преимуществ перед глобальными:

- ограниченная область действия локальной переменной позволяет быстрее выявить ошибку на этапе отладки, поскольку изменить ее значение может только ограниченное количество команд программы;
- применение локальных переменных позволяет более эффективно расходовать память компьютера, поскольку занимаемый ими участок оперативной памяти можно освободить и перераспределить для других переменных;
- одно и то же имя переменной может использоваться в нескольких процедурах и при этом конфликт имен не возникает.

Существует два основных типа параметров процедуры — регистровые и стековые. В процедурах из библиотек `Irvine32` и `Irvine16` используются регистровые параметры, поскольку они оптимизированы на максимальную скорость работы. Однако часто использование регистровых параметров приводит к излишнему загромождению кода вызывающей программы. Альтернативой регистровым являются стековые параметры. При этом перед вызовом процедуры нужные параметры сначала необходимо поместить в стек.

Директива INVOKE является очень гибким средством вызова процедур и по сути заменяет команду CALL процессоров Intel. Она позволяет передать в процедуру несколько аргументов. Оператор ADDR используется в директиве INVOKE для того, чтобы передать в процедуру *указатель* на переменную (т.е. адрес переменной), а не значение самой переменной.

Стековым фреймом, или записью активации, называется область памяти в стеке, расположенная за адресом возврата из процедуры, в которой размещаются переданные ей параметры, сохраненные регистры и локальные переменные. Он создается в момент начала выполнения процедуры.

Директива PROC предназначена для описания имени процедуры и списка передаваемых ей параметров. Директива PROTO создает прототип существующей процедуры. В прототипе описывается имя процедуры и список ее параметров.

Если во время вызова процедуры ей в качестве аргументов передаются копии значений переменных, то в таком случае говорят, что параметры передаются по значению. Если во время вызова процедуры ей в качестве аргументов передаются адреса переменных, то в таком случае говорят, что параметры передаются по ссылке. При этом программист получает возможность изменить значение исходной переменной из вызываемой процедуры, воспользовавшись переданным адресом. В языках программирования высокого уровня различные структуры данных (такие как массивы) всегда передаются по ссылке. Это же утверждение верно и для языка ассемблера.

Ниже перечислены несколько полезных методик, используемых при поиске и локализации ошибок в программах.

1. Команды PUSH и POP выполняют очень важные действия. Они позволяют сохранить содержимое регистров общего назначения, а затем, после выполнения фрагмента программы, изменяющего их значение, восстановить их к первоначальному состоянию. Их несогласованное использование часто является источником ошибок.
2. При работе с массивами нужно всегда помнить, что адресация элементов массива зависит от их длины.
3. При использовании директивы INVOKE необходимо иметь в виду, что компилятор ассемблера не выполняет проверку типов указателей, передаваемых в процедуру.
4. Если в процедуру должны передаваться адреса переменных, вместо них нельзя указывать непосредственно заданные значения.

Для определения ряда важных характеристик программы, таких как тип модели памяти, способ именования процедур и соглашение о передаче параметров, в компиляторе MASM используется директива .MODEL. Во всех программах, рассматриваемых в данной книге и написанных для реального режима адресации, используется малая модель памяти, т.е. весь код и все данные в них (включая область стека) собраны в два отдельных сегмента. В программах, написанных для защищенного режима, используется линейная модель памяти и 32-разрядные ссылки на код и на данные. В таких программах суммарный объем кода и данных не имеет каких-либо практических ограничений и может составлять максимум 4 Гбайт.

В директиве .MODEL допускаются следующие описатели языка: C, BASIC, FORTRAN, PASCAL, SYSCALL и STDCALL.

Для обращения к параметрам процедур используется косвенная адресация через регистр EBP. Воспользовавшись формой записи наподобие [ebp+8], программист может получить полный контроль над адресаций параметров в стеке.

Команда LEA (Load Effective Address, или Загрузить текущий адрес) позволяет определить текущее смещение косвенного операнда любого типа. Ею удобно пользоваться для определения адреса параметра, находящегося в стеке.

Команда ENTER предназначена для автоматического создания стекового фрейма в вызванной процедуре. Она позволяет выделить место под локальные переменные и сохранить в стеке регистр EBP. Команда LEAVE позволяет завершить использование стекового фрейма в процедуре. Она выполняет действия, противоположные ранее использовавшейся команде ENTER — восстанавливает содержимое регистров ESP и EBP к тому состоянию, которое было в момент вызова процедуры.

Рекурсивной называется такая процедура, которая явно или неявно вызывает сама себя. Рекурсия, или практика вызова рекурсивных процедур, является очень мощным средством при работе со структурами данных, которые имеют периодический характер.

При разработке крупных проектов, с программой очень неудобно работать, когда весь ее исходный код находится в одном файле. Поэтому для удобства имеет смысл разбить весь проект на несколько файлов с исходным кодом, которые называются модулями. Тем самым вы облегчите себе задачу последующего анализа такого кода и внесения в него изменений.

8.8. Упражнения по программированию

Предложенные ниже упражнения по программированию можно выполнить как в виде 32-разрядных приложений для защищенного режима, так и в виде 16-разрядных приложений для реального режима работы процессора.

8.8.1. Обмен целых чисел

Создайте массив неупорядоченных целых чисел. Воспользовавшись в цикле процедурой **Swap**, описанной в разделе 8.3.6, поменяйте местами значения соседних элементов массива.

8.8.2. Процедура DumpMem

Напишите программу-оболочку для библиотечной процедуры **DumpMem**, которой можно было бы передавать параметры через стек. Выберите для нее подходящее имя, которое немного отличается от оригинала, например **DumpMemory**. Ниже приведен пример вызова такой процедуры:

```
INVOKE DumpMemory,OFFSET array,LENGTHOF array,TYPE array
```

Напишите тестовую программу, в которой эта процедура вызывается несколько раз с разными значениями аргументов.

8.8.3. Нерекурсивное вычисление факториала

Напишите нерекурсивную версию процедуры **`Factorial`**, рассмотренной в разделе 8.5.2, в которой используется цикл. Создайте небольшую тестовую программу, в которой значение *n* для вычисления факториала должен ввести пользователь. Результат вычисления факториала отобразите на экране.

8.8.4. Сравнение программ вычисления факториала

Напишите программу, с помощью которой можно было бы сравнить время выполнения обеих версий процедур вычисления факториала: рекурсивной, описанной в разделе 8.5.2, и нерекурсивной, которая была создана в результате выполнения предыдущего упражнения. Для определения времени выполнения процедур воспользуйтесь библиотечной процедурой **`GetMseconds`**. Отобразите результаты измерений в миллисекундах на экране. Для повышения точности измерений, определите время выполнения цикла, в котором процедура `Factorial` вызывается несколько тысяч раз.

8.8.5. Наибольший общий делитель (НОД)

Напишите программу нахождения наибольшего общего делителя (НОД) двух целых чисел, в которой был бы реализован рекурсивный алгоритм Евклида. Описание этого алгоритма можно найти в любом учебнике по алгебре либо в Web. Напомним, что нерекурсивную версию этой программы вы должны были создать во время решения упражнения по программированию 7.8.6.