Московский государственный технический университет
имени Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети

«УТВЕРЖДАЮ»
Заведующий кафедрой ИУ-6
_____ Сюзев В.В.

Г.С. Иванова, Т.Н. Ничушкина

МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ. СВЯЗЬ РАЗНОЯЗЫКОВЫХ МОДУЛЕЙ

Методические указания к лабораторным работам и домашним заданиям
по дисциплине Машинно-зависимые языки и основы компиляции

МОСКВА 2013

Содержание

ВВЕДЕНИЕ

Модульный принцип является неотъемлемой частью современной технологии программирования. Согласно этому принципу любая программа состоит из главной (основной) программы и совокупности *подпрограммам* или *процедур*. Главная программа по мере необходимости вызывает подпрограммы на выполнение, передавая им управление процессором. Достоинством указанной технологии является возможность разработки программ большого объема небольшими функционально законченными частями. При этом многие процедуры можно использовать в других местах программы или других программах, не прибегая к переписыванию частей программного кода. Дополнительные возможности предоставляет применение при разработке подпрограмм, написанных на различных языках программирования, как высокого, так и низкого уровней. При этом используются преимущества языка программирования, который дает наиболее эффективную реализацию алгоритма подпрограммы. Так, включение модулей, написанных на языке ассемблера, позволяет ускорить выполнение соответствующих частей программы и/или выполнить действия, программирование которых с использованием языков высокого уровня невозможно или затруднительно. С другой стороны, существует много библиотек подпрограмм на языках высокого уровня, которые с успехом можно использовать в ассемблерных программах.

Каждый язык программирования предусматривает свои способы представления данных, передачи управления и данных в подпрограммы, а также компоновки модулей. Поэтому, при связывании разноязыковых модулей должны быть даны ответы на следующие вопросы:

- как согласовать представление данных, описанных в различных языках;
- как организовать передачу управления в модуль и получение его обратно;
- как передать данные в модуль и получить обратно результаты его работы;
- как выполнить совместную компоновку программы, содержащей модули на разных языках программирования.

Для ответа на эти вопросы необходимо знать особенности реализации модульного принципа в различных языках программирования, а также системные соглашения о передаче управления и параметров в подпрограммы в конкретной операционной системе.

1 МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

1.1 Процедура ассемблера. Описание процедуры

Процедура в ассемблере – это относительно самостоятельный фрагмент, к которому возможно обращение из разных мест программы. На языках высокого уровня такие фрагменты оформляют соответствующим образом и называют подпрограммами: функциями или процедурами. Поддержка модульного принципа для ассемблера означает, что в языке существуют специальные машинные команды вызова подпрограммы и обратной передачи управления. Однако, в отличие от языков высокого уровня, ассемблер не требует специального оформления процедур. На любой адрес программы можно передать управление командой вызова процедуры, и оно вернется к вызвавшей процедуре, как только встретится команда возврата управления. Такая организация может привести к трудночитаемым программам, поэтому в язык ассемблера включены директивы логического оформления процедур. В *простейшем* случае с использованием директив процедуры описываются следующим образом:

<Имя> PROC [<Тип вызова>][<Язык>]
. . .
<Имя> ENDP

Тип вызова зависит от того, происходит ли вызов подпрограммы, находящейся в том же сегменте памяти или в другом. Вызов подпрограммы из того же сегмента называется ближним (**near**). Вызов подпрограммы из другого сегмента называется дальним (**far**). В модели FLAT, которая используется при работе по WINDOW’s, все процедуры являются ближними, что и подразумевается по умолчанию.

Язык – параметр, определяющий конвенцию о связи, т. е. способ передачи параметров и управления (см. далее).

1.2 Связь процедур ассемблера по управлению

Связь процедур ассемблера по управлению осуществляется с помощью специальных машинных команд: команды вызова процедуры CALL и команды возврата управления RET(рисунок 1.1).

Команда вызова процедуры имеет следующий формат:

CALL <Имя или адрес процедуры>

Если процедура специальным образом оформлена, то тип вызова (ближний или дальний) определяется автоматически по этому описанию. Тип вызова неоформленной процедуры необходимо уточнять, указывая перед именем или адресом **near ptr** для орга-

низации ближнего и **far ptr** – для организации дальнего вызова. Как уже упоминалось, по умолчанию в модели FLAT все вызовы – ближние.

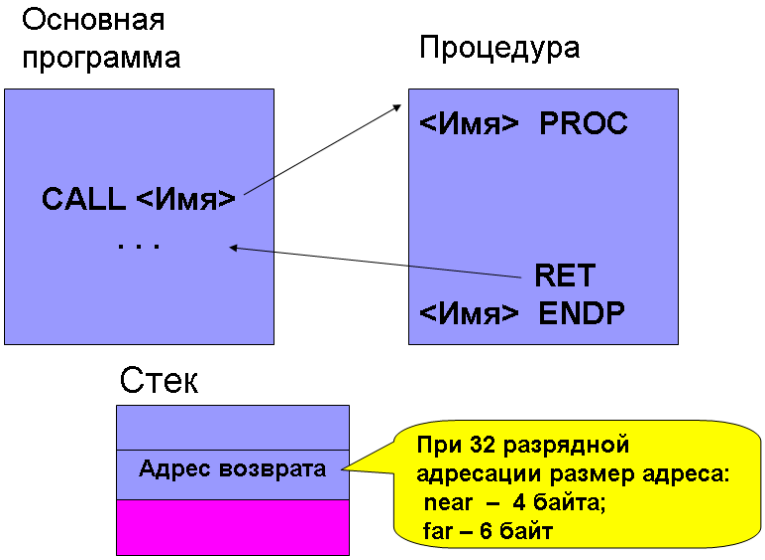


Рисунок 1.1 – Прямая и обратная передачи управления при вызовы процедур

Для обеспечения возврата управления из вызываемой процедуры команда CALL помещает в стек адрес возврата в вызывающую процедуру. Таким адресом является адрес следующей за CALL команды. Если осуществляется ближний вызов, то команда CALL помещает в стек смещение длиной 4 байта этой команды в сегменте.

Затем команда CALL загружает смещение вызываемой процедуры в том же сегменте в регистр команд EIP, и процессор переключается на выполнение процедуры.

При дальнем вызове – в стек помещается виртуальный адрес следующей за CALL команды: 4 байта смещение в сегменте и 2 байта – содержимое селектора. После чего команда CALL загружает номер дескриптора, содержащего базовый адрес сегмента, в котором находится процедура, в регистр CS, а смещение процедуры относительно начала ее сегмента – в регистр команд EIP, и процессор начинает выполнять процедуру.

Возврат из процедуры осуществляется по команде RET, которая должна быть последней выполняемой командой процедуры:

RET [<Число>].

Эта команда извлекает из стека ближний или дальний адрес возврата и загружает его в EIP или CS:EIP – в зависимости от типа вызова.

В команде RET может быть указан один операнд – число. Это число после извлечения из стека адреса возврата команда RET должна добавить к указателю стека ESP. Так можно удалить из стека передаваемые в процедуру параметры (см. раздел 1.3.4).

Кроме этого для гарантии нормального продолжения работы основной программы, получив управление, процедура должна сохранить в стеке содержимое регистров, которые она использует, а перед возвратом управления – восстановить его.

1.3 Организация передачи данных в процедурах на ассемблере

Процедуры на ассемблере могут получать или не получать данные из вызывающей процедуры и могут возвращать или не возвращать ей результаты своей работы. Существует несколько способов передачи параметров в процедуры. Рассмотрим их более подробно.

1.3.1 Передача параметров через регистры

Если данных передается немного, то самый быстрый и простой способ – передать параметры через регистры. Если же данных много, или они представляют сложные структуры типа массива или записи, то использовать регистры не рационально.

Пример 1.1. Написать программу, выполняющую сложение двух целых чисел. Основная программа записывает в регистры два параметра и адрес, по которому надо записать результат. Суммирование выполняет вызываемая процедура. Ниже приведен текст программы.

```
; Template for console application
          .586
          .MODEL  flat, stdcall
          OPTION CASEMAP:NONE

Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

          .CONST
MsgExit   DB    "Press Enter to Exit",0AH,0DH,0

          .DATA
A         DWORD  56
B         DWORD  34

          .DATA?
D         DWORD  ?
inbuf     DB    100 DUP (?)

          .CODE
Start:
```

```
        lea     EDX,D   ; занесение в регистр адреса результата
        mov     EAX,A   ; занесение в регистр первого числа
        mov     EBX,B    ; занесение в регистр первого числа
        call    SumDword  ; вызов подпрограммы

        Invoke StdOut,ADDR MsgExit
        Invoke StdIn,ADDR inbuf,LengthOf inbuf
        Invoke ExitProcess,0

SumDword PROC
        add   EAX,EBX        ; складываем числа
        mov   [EDX],EAX      ; отправляем результат на место
        ret
SumDword ENDP
        End    Start
```

1.3.2 Передача данных путем прямого обращения к памяти

При таком способе обмена данными вызываемая и вызывающая процедуры обращаются напрямую к данным по их символическим именам. Способ оформления такого обращения зависит от того, как организована программа: основная программа и процедура находятся в одном исходном модуле (файле) или в разных.

А. Совместная трансляция процедуры и основной программы. При совместной трансляции вся программа помещается в один файл, т.е. представляет собой один исходный модуль, который транслируется за один вызов транслятора. В этом случае формируется единое адресное пространство программы, и все имена данных программы видимы в процедуре.

Пример 1.2. Суммирование чисел. Процедура для доступа к данным использует их символические имена (рисунок 1.2).

```
; Template for console application
        .586
        .MODEL  flat, stdcall
        OPTION CASEMAP:NONE

Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib
```

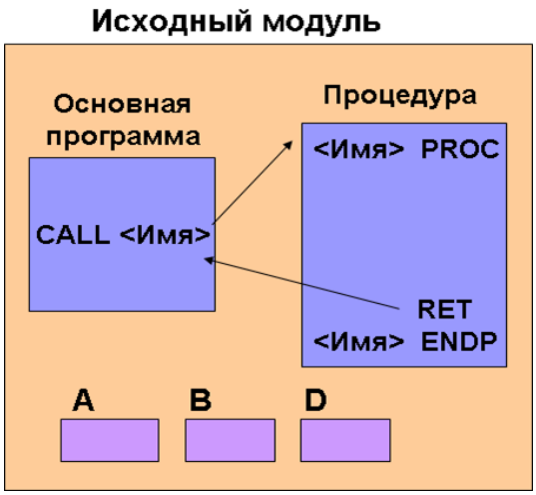


Рисунок 1.2 – Прямое обращение к данным при совместной трансляции основной программы и процедуры

```
            .CONST
MsgExit     DB     "Press Enter to Exit",0AH,0DH,0

            .DATA
A           DWORD  56
B           DWORD  34

            .DATA?
D           DWORD  ?
inbuf       DB     100 DUP (?)

            .CODE
Start:      call   SumDword
            Invoke StdOut,ADDR MsgExit
            Invoke StdIn,ADDR inbuf,LengthOf inbuf
            Invoke ExitProcess,0

SumDword    PROC
            push   EAX   ; сохранение содержимого регистра EAX
            mov    EAX,A
            add    EAX,B
            mov    D,EAX
            pop    EAX   ; восстановление содержимого регистра EAX
            ret
SumDword    ENDP
            End    Start
```

Содержимое регистра EAX сохраняется в стеке, поскольку основная программа может использовать его в своих целях, не предусматривающих его изменения процедурой.

Б. Раздельная трансляция процедур. При раздельной трансляции процедуры описываются в разных файлах, транслируются отдельно и объединяются в единую программу на этапе компоновки. Каждый файл в этом случае – отдельный модуль со своим адресным пространством. Поэтому необходимо указать компоновщику внутренние имена модуля, к которым будет происходить обращение из других модулей, и внешние имена, которые определены в других модулях, но к которым есть обращение из данного модуля. Для этого предусмотрены специальные директивы.

Директива PUBLIC описывает внутренние имена, к которым возможно обращение извне:

PUBLIC [<Язык>] <Имя> [, <Язык>] <Имя>...

где <Язык> – параметр, определяющий конвенцию о связи, т.е. особенности формирования внутренних имен глобальных переменных и процедур (см. раздел 2.1.1)

<Имя> – символическое имя, которое должно быть доступно в других модулях.

Директива EXTERN описывает внешние имена – имена, определенные в других исходных модулях, к которым есть обращение из данного модуля:

EXTERN [<Язык>] <Имя> [(<Псевдоним>)]:<Тип>

[,[<Язык>] <Имя> [(<Псевдоним>)]:<Тип>...

где <Имя> – символическое имя, используемое в процедуре, но не описанное в ней;

<Тип> – определяется для различных типов имен следующим способом:

идентификатор: BYTE, WORD, DWORD;

имя процедуры, метка: NEAR, FAR;

константа, определенная посредством '=' или 'EQU': ABS.

При этом, если в одной процедуре имя описано как EXTERN, то в другой оно должно быть описано как PUBLIC.

Универсальная директива EXTERNDEF описывает любое имя, которое описано в одном модуле, а используется в других. В зависимости от обстоятельств может интерпретироваться как Public или Extern:

EXTERNDEF [<Язык>] <Имя>:<Тип>[[<Язык>] <Имя>:<Тип>]...

Пример 1.3. Рассмотрим пример 1.2, но разместим основную и вызываемую программы в разных модулях (см. рисунок 1.3).

Основная программа:

```
; Template for console application
```

```
.586
.MODEL  flat, stdcall
OPTION CASEMAP:NONE
```

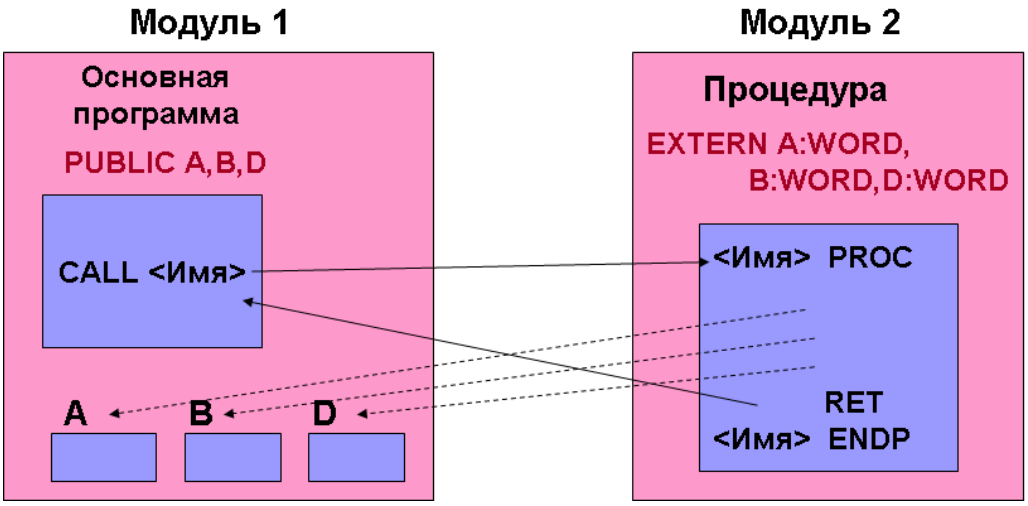


Рисунок 1.3 – Размещение программы и подпрограммы в разных исходных модулях

```
Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

        .CONST
MsgExit DB    "Press Enter to Exit",0AH,0DH,0

        .DATA
A       DWORD  56
B       DWORD  34

        .DATA?
D       DWORD  ?
inbuf   DB    100 DUP (?)

        PUBLIC A,B,D
        EXTERN SumDword:near

        .CODE
Start:  call   SumDword

        Invoke StdOut,ADDR MsgExit
        Invoke StdIn,ADDR inbuf,LengthOf inbuf
        Invoke ExitProcess,0
        End    Start
```

Вызываемая процедура:

```
          .586
          .MODEL  flat, stdcall
          OPTION CASEMAP:NONE

          .CODE

          EXTERN A:DWORD,B:DWORD,D:DWORD

SumDword PROC    c      ; указана конвенция языка «c»
          push    EAX
          mov     EAX,A
          add     EAX,B
          mov     D,EAX
          pop     EAX
          ret
SumDword ENDP
          END
```

Сохранение содержимого регистра EAX перед его использованием в процедуре с его восстановлением перед возвратом управления в основную программу позволяет гарантировать, что процедура не «испортит» содержимого этого регистра в основной программе.

Примечание – Создание многомодульных программ с использованием RADAsm выполняют следующим образом:

1) добавление модуля осуществляется с использованием пункта меню **Проект/Добавить новый/Модуль**;

2) ассемблирование модуля осуществляется при активизации пункта меню **Создать/Assemble Modules**. Для активизации этого пункта меню необходимо в меню **Проект/Настройка проекта,** пометить галочкой в списке меню таблицы выделить **Assemble Modules**

3) после получения объектного модуля до компоновки необходимо этот модуль также добавить к проекту, используя **Проект/Добавить существующие/Объектные модули**.

1.3.3 Передача параметров через таблицу адресов

В этом случае в памяти вызывающей программы создается специальная таблица адресов параметров. В таблицу перед вызовом процедуры записывают адреса передаваемых данных. Затем адрес самой таблицы заносится в один из регистров (например, EBX) и

управление передается вызываемой процедуре. Вызываемая процедура сохраняет в стеке содержимое всех регистров, которые собирается использовать, после чего выбирает адреса переданных данных из таблицы, выполняет требуемые действия и заносит результат по адресу, переданному в той же таблице.

Пример 1.4. Написать программу, подсчитывающую сумму элементов одномерного массива. Массив и его размер определяются в основной программе, суммирование элементов выполняет процедура. Результат сложения возвращается в основную программу. Данные передаем через таблицу адресов (см. рисунок 1.4).

```
; Template for console application
          .586
          .MODEL  flat, stdcall
          OPTION CASEMAP:NONE

Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

          .CONST
MsgExit   DB    "Press Enter to Exit",0AH,0DH,0

          .DATA
ary       SWORD  5,6,1,7,3,4 ; массив
count     DWORD  6           ;  размер массива

          .DATA?
inbuf     DB     100 DUP (?)
sum       SWORD  ?           ;  сумма элементов
tabl      DWORD  3 dup(?)    ; таблица адресов параметров

          EXTERN masculc:near

          .CODE
Start:
; формирование таблицы адресов параметров
          mov    tabl,offset ary
          mov    tabl+4,offset count
          mov    tabl+8,offset sum
          mov    EBX,offset tabl
```

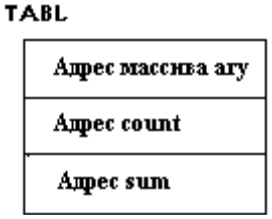


Рисунок 1.4 – Таблица адресов параметров

```
        call    masculc

        XOR     EAX,EAX
        Invoke StdOut,ADDR MsgExit
        Invoke StdIn,ADDR inbuf,LengthOf inbuf
        Invoke ExitProcess,0
        End     Start
```

Текст процедуры:

```
; Template for console application
        .586
        .MODEL  flat, stdcall
        OPTION CASEMAP:NONE
        .CODE
masculc proc    c
        push    AX      ; сохранение регистров
        push    ECX
        push    EDI
        push    ESI
; использование таблицы адресов параметров
        mov     ESI,[EBX]    ;загрузка адреса массива
        mov     EDI,[EBX+4] ; загрузка адреса размера массива
        mov     ECX,[EDI]    ; загрузка размера массива
        mov     EDI,[EBX+8]  ; загрузка адреса результата
        xor     AX,AX
; суммирование элементов массива
cycl:   add     AX,[ESI]
        add     ESI,2
        loop    cycl
; формирование результатов
        mov     [EDI],AX   ;загрузка результата по сохраненному адресу
        pop     ESI     ; восстановление регистров
        pop     EDI
        pop     ECX
        pop     AX
        ret
```

```
masculc endp
        END
```

1.3.4 Передача параметров в стеке

Наиболее распространенным способом передачи данных в практике программирования процессоров рассматриваемого типа является передача параметров в стеке. Именно этот способ принят в качестве базового и его используют языки высокого уровня.

Параметры помещают в стек командой PUSH, после чего управление передается вызываемой процедуре. Доступ к параметрам, хранящимся в стеке, из вызываемой процедуры осуществляют через регистр EBP. В этот регистр помещают адрес вершины стека, копируя его из регистра указателя стека ESP, а затем этот регистр используют как базовый при адресации параметров.

Для обеспечения корректного возврата в вызывающую процедуру старое значение регистра EBP помещают в стек первой командой процедуры. Параметры в стеке, адрес возврата и старое значение EBP вместе называют *фреймом активации процедуры*. Вызываемая процедура, зная структуру стека, извлекает параметры в соответствующие регистры, выполняет над ними операции и записывает результат, используя адрес, переданный в стеке.

Примечание – Следует помнить, что в стек можно поместить 2 или 4 байта. Если в стек надо поместить параметр размером 1 байт, то помещают 2 байта, но байт со старшим адресом не используется.

Пример 1.5. Программа суммирования элементов массива. Программа использует 3 параметра: два исходных значения и результат. Для первого параметра – массива в стек помещается адрес начала массива, второй параметр – адрес счетчика, содержащего количество элементов. В качестве третьего параметра в стек помещается адрес результата, куда процедура поместит полученное значение (см. рисунок 1.5).

```
; Template for console application
        .586
        .MODEL  flat, stdcall
        OPTION CASEMAP:NONE

Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

        .CONST
```

```
MsgExit  DB    "Press Enter to Exit",0AH,0DH,0

         .DATA
ary      SWORD  5,6,1,7,3,4 ; элементы массива
count    DWORD  6           ;  размер массива

         .DATA?
inbuf    DB     100 DUP (?)
sum      SWORD  ?   ;  сумма элементов

         EXTERN masculc:near

         .CODE
Start:
```

; запись параметров в стек

```
         push   offset ary
         push   offset count
         push   offset sum
```

; вызов процедуры

```
         call   masculc

         XOR    EAX,EAX
         Invoke StdOut, ADDR MsgExit
         Invoke StdIn,ADDR inbuf,LengthOf inbuf
         Invoke ExitProcess,0
         End    Start
```

Рисунок 1.5 – Состояние стека во время работы процедуры masculc

Модуль вычисления суммы элементов массива:

```
         .586
         .MODEL  flat, stdcall
         OPTION CASEMAP:NONE

         .CODE
masculc  proc   c
         push   EBP       ; сохранение регистра ebp на момент вызова п/п
         mov    EBP,ESP   ; запись в ebp адреса вершины стека
```

; Сохранение регистров, которые будут использоваться

```
         push   AX
         push   ECX
         push   EDI
```

```
            push    ESI
; Извлечение параметров из стека
            mov     ESI,[EBP+16]    ; адрес массива
            mov     EDI,[EBP+12]    ; адрес количества элементов
            mov     ECX,[EDI]       ; количество элементов
            mov     EDI,[EBP+8]     ; адрес результата
            xor     AX,AX           ; очистка регистра AX
cycl:       add     AX,[ESI]
            add     ESI,2
            loop    cycl

            mov     [EDI],AX        ; сохранение результата
 ; Восстановление регистров, которые были сохранены
            pop     ESI
            pop     EDI
            pop     ECX
            pop     AX
            pop     EBP     ; восстановление регистра ebp
            ret     12      ; удаление параметров из стека
masculc     ENDP
            END
```

При передаче параметров через стек возникает два вопроса:

– в каком порядке записывать параметры в стек;

– кто – вызывающая или вызываемая процедура – должен удалять параметры из стека.

В обоих вариантах есть свои плюсы и минусы. Например, если стек освобождает вызываемая процедура по команде **RET <Число байт>**, то код программы получается более коротким. Если за освобождение стека отвечает вызывающая программа, то становится возможным вызов нескольких процедур с одними и теми же значениями параметров просто последовательными командами CALL. Первый способ более строгий, он используется в языке Pascal. Второй, дающий больше возможностей для оптимизации, – в языках C и C++. Вопрос о порядке записи параметров в стек для ассемблера не столь важен, так как и записывают и извлекают параметры подпрограммы на ассемблере. Главное – чтобы этот вопрос был между ними согласован. А вот при взаимодействии ассемблера с языками высокого уровня, следует знать особенности передачи параметров этих языков.

1.4 Особенности реализации рекурсивных программ в ассемблере

Рекурсивные алгоритмы предполагают реализацию в виде процедуры, которая сама себя вызывает. При этом необходимо обеспечить, чтобы каждый последовательный вызов процедуры не разрушал данных, полученных в результате предыдущего вызова. Для этого, каждый вызов должен иметь свой набор параметров, регистры и все промежуточные результаты. Средства модульного программирования ассемблера позволяют выполнить это требование и реализовать рекурсивный алгоритм. Для сохранения данных очередного вызова и передачи параметров следующей активации процедуры лучше использовать стек. А удобную организацию стека позволяют организовать *структуры*.

Структура в ассемблере аналогична структурам (записям) в языках высокого уровня. Структура представляет собой шаблон с описаниями форматов данных, который можно накладывать на различные участки памяти, чтобы затем обращаться к полям этих участков памяти с помощью имен, определенных в описании структуры. Особенно удобны структуры при обращении к областям памяти, не входящим в сегменты данных и кода программы, т.е. полям, которые нельзя описать с помощью символических имен.

Формат описания структуры:

<Имя структуры> STRUCT

<Описание полей>

<Имя структуры> ENDS

где <Имя структуры> – символьное имя структуры,

<Описание полей> – любой набор псевдокоманд определения переменных или вложенных структур.

Эта последовательность директив описывает, но не размещает в памяти структуру данных. Для чтения или записи элемента структуры применяется точечная нотация:

<Имя структуры>. <Имя поля>.

Кроме того, структуры используются, когда в программе многократно повторяются сложные коллекции данных с единым строением, но с различными значениями полей. В этом случае, для создания такой структуры в памяти достаточно использовать имя структуры как псевдокоманды по шаблону:

<Имя переменной> <Имя структуры>

<<Значение поля 1>, <Значение поля 2>,…<Значение поля n>>

Например, пусть в программе, обрабатывающей данные о студентах, необходимо объявить несколько блоков данных с однородными сведениями о нескольких студентах. Такие данные удобно оформить в виде структуры с именем Student:

```
Student  struct
```

```
        Family    db 20 dup (' ' )   ; Фамилия студента
        Name      db 15 dup(' ')     ; Имя
        Birthdata db '  /  /  '      ; Дата  рождения
Student  ends
```

Определить с помощью этой структуры в программе две переменные с именами stud1 и stud2 можно следующим обращением:

```
stud1 Student <'Иванов','Петр','23/12/72'>
stud2 Student <'Сидоров','Павел','12/05/84'> .
```

При создании рекурсивных процедур STRUCT используется для описания шаблона данных очередного вызова (фрейма активации). При обращении к данным фрейма или сохранении фрейма очередной активации обращение происходит с помощью полей структуры, что значительно упрощает процессы чтения и записи в стек данных активации.

Пример 1.6. Написать рекурсивную процедуру вычисления факториала числа.

В процедуре определения факториала числа воспользуемся следующими утверждениями:

$$N! = \begin{cases} N*(N-1)!\text{ , при } N\neq 0 \text{ – рекурсивное утверждение;} \\ 1 \text{ , при } N=0 \text{ – базисное утверждение.} \end{cases}$$

Таким образом, каждая активация должна иметь доступ и сохранять результат вычисления и текущее значение числа для расчета N*(N-1). Кроме того, при очередном вызове процедура должна сохранять регистр базы параметров (EBP) и адрес возврата. Поэтому фрейм активации включает:

- значение регистра EBP – 4 байта,
- адрес возврата для случая ближнего вызова – 4 байта,
- число N на данном уровне рекурсии – 2 байта,
- адрес результата – 4 байта.

Для работы с фреймом опишем структуру, перечисляя поля в том порядке, в котором они будут размещаться в стеке (рисунок 1.6).

```
FRAME    STRUCT
   SAVE_EBP     DD    ?
   SAVE_EIP     DD    ?

   N            DW    ?
   RESULT_ADDR  DD    ?
FRAME    ENDS
```

Рисунок 1.6 – Структура Frame

Текст программы:

```
          .586
          .MODEL  flat, stdcall
          OPTION CASEMAP:NONE

Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

FRAME     STRUCT
   SAVE_EBP     DD   ?
   SAVE_EIP     DD   ?
   n            DW   ?
   result_addr DD   ?
FRAME     ENDS

          .CONST
MsgExit   DB     "Press Enter to Exit",0AH,0DH,0

          .DATA
n         DW     5   ;  N – исходное число

          .DATA?
inbuf     DB     100 DUP (?)
result    DD     ?       ;  резервирование памяти под результат
          .CODE
Start:    ; формирование стека
          push   offset result ; запись в стек адреса результата
          push   n          ;  запись в стек исходного числа
          call   fact

          XOR    EAX,EAX
          Invoke StdOut,ADDR MsgExit
          Invoke StdIn,ADDR inbuf,LengthOf inbuf
          Invoke ExitProcess,0

fact      PROC

          ;  доформирование стека
          push   EBP
```

```
            mov     EBP,ESP
            push    EBX
            push    AX
; извлечение из стека адреса результата
            mov     EBX,FRAME.result_addr[EBP]
            mov     AX,FRAME.n[EBP]  ; извлечение из стека N
            cmp     AX,0
            je      done     ;  выход из рекурсии
            push    EBX      ; сохранение в стеке адреса результата
            dec     AX       ;  N=N-1
            push    AX       ;  сохранение в стеке очередного N
            call    fact     ;  рекурсивный вызов
; извлечение из стека адреса результата
            mov     EBX,FRAME.result_addr[EBP]
; вычисление результата очередной активации
            mov     AX,[EBX]
            mul     FRAME.n[EBP]
            jmp     short return
done:       mov     EAX,1       ; если  ax=0 , то факториал равен 1
; запоминаем результат активации
return:     mov     [EBX],AX
            pop     AX
            pop     EBX
            pop     EBP
            ret     6
fact        ENDP
            End     Start
```

На рисунке 1.7 показано содержимое стека во время выполнения рекурсивной процедуры (3 активации).

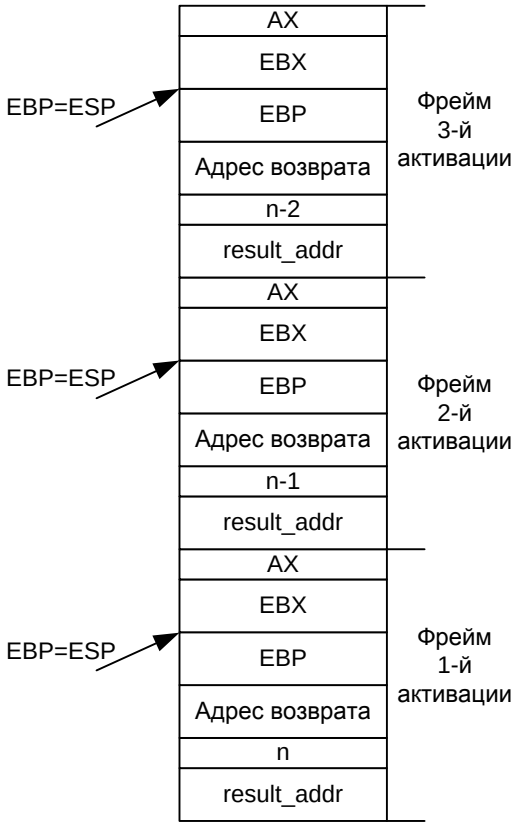


Рисунок 1.7 – Содержимое стека в процессе рекурсивного спуска

1.5 Директивы описания процедур

При работе с процедурами в ассемблере необходимо соблюдать большое количество различных правил, например, правил передачи параметров, формирования внутренних имен, сохранения регистров и т.п. Использование директив описания процедур позволяет существенно упростить эти операции.

1.5.1 Директива заголовка процедуры

Директива заголовка процедуры позволяет объявить основные характеристики процедуры. Формат директивы:

<Имя процедуры> PROC [<Тип вызова>] [<Конвенция о связи>]
[<Доступность>]
[USES <Список используемых регистров>]
[,<Параметр> [:<Тип>]]...

где <Тип вызова>:

far – межсегментный – программа и процедура находятся в различных сегментах,

near – внутрисегментный – программа и процедура находятся в одном сегменте (используется по умолчанию);

<Конвенция о связи> – имя конвенции о связях (см. раздел 2.1.1), которая определяет способ передачи параметров, формирование внутренних имен и т.п., по умолчанию используется конвенция, указанная в **.MODEL:**

STDCALL – стандартные соглашения, используемые в Windows;

C – соглашения, принятые в языке C,

PASCAL – соглашения, принятые в языке Pascal, и др.

<Доступность> – видимость процедуры из других модулей:

public – общедоступная (используется по умолчанию);

private – внутренняя;

export – межсегментная и общедоступная.

<Список используемых регистров> – содержит регистры, используемые в процедуре, используется для их автоматического сохранения и восстановления.

<Параметр> – имя параметра процедуры.

<Тип> – тип параметра или VARARG. Если тип не указан, то по умолчанию для 32-х разрядной адресации берется DWORD. Если указано VARARG, то разрешается использовать список аргументов через запятую.

Пример:

```
ABC    PROC NEAR STDCALL PUBLIC USES EAX, X:DWORD,
                                    Y:BYTE,H:DWORD PTR
```

1.5.2 Директива описания локальных переменных

Директива описания локальных переменных используется для объявления локальных переменных процедуры, память под которые отводится в стеке при вызове процедуры. При завершении процедуры эта память освобождается. Директива помещается сразу после PROC.

Формат директивы:

LOCAL <Имя>[[<Количество>]][:<Тип>] [,<Имя>[[<Количество>]][:<Тип>]]...

Пример:

```
ABC    PROC    USES EAX,X:VARARG
       LOCAL   ARRAY[20]:BYTE
   . . .
```

1.5.3 Директива объявления прототипа процедуры

Директива предварительно объявляет процедуру, указывая ее имя, список параметров и некоторые описатели (прототип). Использование этой директивы позволяет описывать уже объявленные процедуры в любом порядке, а не обязательно до первого вызова.

Формат директивы:

<Имя процедуры> PROTO [<Тип вызова>]
[<Соглашения о связи>]
[<Доступность>]
[,<Параметр> [:<Тип>]]...

Описание совпадает с описанием параметров директивы PROC. Используется для указания списка и типов параметров для вызова процедуры директивой INVOKE.

Пример:

```
MaxDword PROTO  NEAR STDCALL PUBLIC
                                X:DWORD,Y:DWORD,ptrZ:PTR DWORD
```

или с учетом умолчаний:

```
MaxDword PROTO  X:DWORD,Y:DWORD,ptrZ:PTR DWORD
```

1.5.4 Директива вызова процедуры

Директива (макрокоманда) вызова процедуры существенно упрощает вызов процедуры, поскольку позволяет сразу указать параметры и другую информацию.

Формат директивы:

INVOKE <Имя процедуры или ее адрес> [, <Список аргументов>]

Аргументы должны совпадать по порядку и типу с параметрами, указанными при описании процедуры в директиве PROC.

Типы аргументов директивы INVOKE:

- целое значение, например: 27h, -128;
- выражение целого типа, в том числе использующее операторы получения атрибутов полей данных (см. далее):
  (10*20), TYPE mas, SYZEOF mas+2, OFFSET AR;
- регистр, например: EAX, BH;
- символический адрес переменной, например: Ada1, var2_2;
- адресное выражение, например: 4[EDI+EBX], Ada+24, ADDR AR.

Операторы получения атрибутов полей данных. Могут включаться в адресные выражения и выражения целого типа:

ADDR <Имя поля данных> – возвращает ближний или дальний адрес переменной в зависимости от модели памяти – для Flat ближний;

OFFSET <Имя поля данных> – возвращает смещение переменной относительно начала сегмента – для Flat совпадает с ADDR;

TYPE <Имя поля данных> – возвращает размер в байтах элемента описанных данных;

LENGTHOF <Имя поля данных> – возвращает количество элементов, заданных при определении данных;

SIZEOF <Имя поля данных> – возвращает размер поля данных в байтах;

<Тип> **PTR** <Имя поля данных> – изменяет тип поля данных на время выполнения команды.

Пример 1.7. Определение максимального из двух целых чисел. При составлении программы использованы директивы и атрибуты полей данных.

```
; Template for console application
          .586
          .MODEL  flat, stdcall
          OPTION CASEMAP:NONE
Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib

MaxDword PROTO  X:DWORD,Y:DWORD,ptrZ:PTR DWORD

          .CONST
MsgExit   DB    "Press Enter to Exit",0AH,0DH,0

          .DATA
A         DWORD  56
B         DWORD  34

          .DATA?
D         DWORD  ?
inbuf     DB     100 DUP (?)

          .CODE
Start:

          INVOKE MaxDword,A,B,ADDR D

          XOR    EAX,EAX
          Invoke StdOut,ADDR MsgExit
          Invoke StdIn,ADDR inbuf,LengthOf inbuf
          Invoke ExitProcess,0
```

```
MaxDword PROC   USES EAX EBX,X:DWORD,Y:DWORD,ptrZ:PTR DWORD
          mov    EBX,ptrZ
          mov    EAX,X
          cmp    EAX,Y
          jg     con
          mov    EAX,Y
 con:     mov    [EBX],EAX
          ret
 MaxDword ENDP

          End    Start
```

2 СВЯЗЬ РАЗНОЯЗЫКОВЫХ МОДУЛЕЙ В WINDOWS

Основные проблемы связи разноязыковых модулей:

- осуществление совместной компоновки модулей;
- организация передачи и возврата управления;
- передача параметров:
  - с использованием глобальных переменных,
  - с использованием стека (по значению и по ссылке),
- обеспечение возврата результата функции;
- обеспечение корректного использования регистров процессора.

2.1 Основные правила организации связи разноязыковых модулей

Корректное обращения к процедурам, написанным на ассемблере, из приложений Windows, и наоборот, предполагает соблюдение определенных правил. Эти правила определяют способ передачи параметров, закономерности формирования внутренних имен подпрограмм и глобальных данных и применяемую модель памяти.

2.2 Конвенции о связи модулей. Правила передачи параметров

Правила, декларирующие способы передачи параметров при организации связи модулей, получили название «конвенции». Поскольку первоначально основные правила передачи управления и параметров определялись языком программирования, названия основных конвенций связано с именами двух основных универсальных языков программирования: Паскаль и Си. Остальные получили свои имена в соответствии с основными свойствами: стандартная Windows, защищенная и регистровая.

Конвенция **Паскаль** предполагает, что параметры помещаются в стек в том порядке, в котором они встречаются в списке формальных параметров подпрограммы. Причем все параметры передаются через стек, регистры для передачи параметров не используются. Завершаясь, подпрограмма удаляет параметры из стека, а потом возвращает управление.

Конвенция **Си** предполагает обратный порядок помещения параметров в стек, регистры также не используются, и параметры из стека удаляет вызывающая программа.

Стандартная и **Защищенные** конвенции используют обратный порядок занесения параметров в стек, но очистку стека вызываемой процедурой. Эти конвенции очень похожи. Отличие только в том, что Защищенная конвенция формирует исключение при обнаружении ошибок, связанных с передачей параметров.

Регистровая конвенция означает передачу до трех параметров в регистрах. Обычно этого хватает, но если параметров больше, то остальные передаются через стек.

Конвенции реализованы в основных средах программирования (таблица 2.1).

Таблица 2.1 – Конвенции по передаче параметров

	Конвенция	**Delphi**	**C++ Builder и Visual C++**	**Порядок параметров в стеке**	**Очистка стека**	**Использование регистров**
1	Паскаль	pascal	_ _pascal	Слева направо	Вызываемая процедура	Нет
2	Си	cdecl	_ _cdecl	Справа налево	Вызывающая программа	Нет
3	Стандартная	stdcall	_ _ stdcall	Справа налево	Вызываемая процедура	Нет
4	Защищенная	savecall	–	Справа налево	Вызываемая процедура	Нет
5	Регистровая	register	_ _fastcall	Справа налево	Вызываемая процедура	Три регистра EAX, EDX, ECX (**VC** – до 2-х), остальные параметры – в стеке

2.3 Правила формирования внутренних имен подпрограмм и глобальных данных

Компилятор Delphi Pascal изменяет внутренние имена подпрограмм и глобальных переменных, заменяя строчные буквы на прописные. Это позволяет не учитывать регистр при записи программы на этом языке.

Компиляторы С изменяют имена всех глобальных («extern») переменных программы, добавляя перед ними символ подчеркивания «_». При этом строчные и прописные буквы в С различаются. Компиляторы С++ дописывают к именам функций специальные комбинации символов, отражающие используемый способ передачи параметров и их тип. В таблице 2.2 приведены основные особенности согласования имен в перечисленных средах.

Таблица 2.2 – Особенности формирования внутренних имен

	Delphi Pascal	**Borland C++**	**Visual C++**
Чувствительность к регистру клавиатуры	Не различает строчных и прописных букв	Различает строчные и прописные буквы	Различает строчные и прописные буквы
Преобразование внешних имен	Преобразует все строчные буквы имен в прописные	Помещает символ «_» перед внешними именами	Помещает символ «_» перед внешними именами
Преобразование	Преобразует все	Изменяет внутреннее	Изменяет внутреннее

имен подпрограмм	строчные буквы имен в прописные	имя подпрограммы: **@<Имя>$q<описание параметров>**	имя подпрограммы: **@_<имя>@<число параметров* 4>**

2.4 Сохранение регистров и модель памяти

Во всех рассматриваемых средах необходимо сохранять регистры: EBX, EBP, ESI, EDI, регистры EAX, EDX, ECX нигде сохранять не надо.

Согласование типа вызова не выполняется, поскольку во всех случаях используется модель памяти Flat, для которой все вызовы ближние near, но смещение имеет размер 32 бита.

3 ОСНОВНЫЕ ПРИНЦИПЫ ВЗАИМОДЕЙСТВИЯ МОДУЛЕЙ НА DELPHI PASCAL И ЯЗЫКЕ АССЕМБЛЕРА

3.1 Соглашения о передаче управления между модулями

Программы на Delphi Pascal используют модель FLAT, а потому модули ассемблера должны быть разработаны применительно к той же модели. Следовательно, все адреса в ассемблере должны быть ближними и состоять только из смещения в сегменте (4 байта).

Параметры передаются в вызываемую подпрограмму через стек, и там же размещаются локальные переменные. Вызов подпрограммы реализуется по варианту:

```
push   <Параметр 1>   ; занесение параметров в стек
...
push   <Параметр n>
call   <Имя подпрограммы>      ; вызов подпрограммы
```

Вызываемые подпрограммы должны иметь стандартно оформленные вход – *пролог* и выход – *эпилог*.

Пролог:

```
<Имя>  proc  near
       push  EBP       ;сохранить старое EBP в стеке
       mov   EBP,ESP   ;установить базу для параметров в стеке
       sub   ESP,<Объем памяти локальных переменных>
       <Сохранение используемых регистров>
       ...
```

Эпилог:

```
       ...
       <Восстановление используемых регистров>
       mov   ESP,EBP ; удалить область локальных переменных
       pop   EBP       ; восстановить значение EBP
       ret   <Размер области параметров>
```

В момент получения управления подпрограммой в стеке находятся параметры в виде значений или адресов и 4-х байтовый адрес возврата в вызывающую программу (рисунок 3.1, *а*). Затем вызываемая подпрограмма размещает в стеке старое значение EBP, область локальных переменных и использует стек для своих надобностей (рисунок 3.1, *б*).

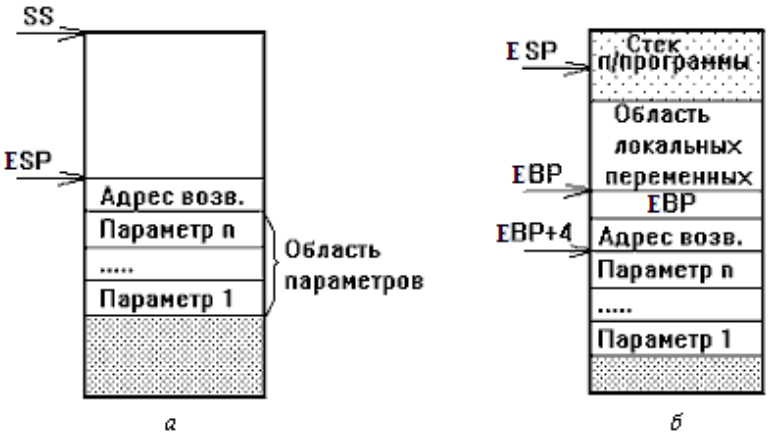


Рисунок 3.1 – Содержимое стека:

а – в момент передачи управления подпрограмме; *б* – во время работы подпрограммы

Адрес области параметров в этом случае определяется относительно содержимого регистра EBP. Так [EBP+8] – адрес последнего параметра. Адреса остальных параметров определяются аналогично с учетом длины каждого параметра в стеке (см. далее).

При выходе из подпрограммы команда **ret** должна удалить из стека всю область параметров, в противном случае произойдет нарушение работы вызывающей программы.

3.2 Соответствие форматов данных

Delphi Pascal использует следующие внутренние представления данных.

Целое –

shortint: -128..127 – байт со знаком;

byte: 0..255 – байт без знака;

smallint: -32768..32767 – слово со знаком;

word: 0..65535 – слово без знака;

integer, **longint**: – двойное слово со знаком.

Символ – **char –** код ANSI байт без знака.

Булевский тип – **boolean** – 0(false) и 1(true) – байт без знака.

Указатель – **pointer –** 32-х разрядное смещение.

Строка – **shortstring –** символьный вектор указанной при определении длины, содержащий текущую длину в первом байте.

Массив – **array –** последовательность элементов указанного типа, расположенных в памяти таким образом, что правый индекс возрастает быстрее левого (для матрицы - построчно).

Для обращения к данным этих типов в программе на ассемблере необходимо использовать вполне определенные типы переменных. Соответствие типов представлено в таблице 3.1

Таблица 3.1 – Соответствие типов ассемблера и Delphi Pascal

№	Тип данных ассемблера	Размер памяти	Соответствующий тип данных Delphi Pascal
1	BYTE	1 байт	**Shortint**, **byte, char, boolean**
2	WORD	2 байта	**SmallInt**, **word**
3	FWORD	6 байт	**Real**
4	DWORD	4 байта	**Single, указатель, integer**
5	QWORD	8 байт	**Double, comp**
6	TBYTE	10 байт	**Extended**

Сложные структурные типы описывают, указав тип только первого элемента и используя затем его для обработки всей структуры.

3.3 Передача параметров по значению и ссылке. Возврат результатов функций

В Delphi Pascal параметры могут передаваться двумя способами: по значению и по ссылке (с указанием **var** или **const)**. В первом случае подпрограмме передаются копии значений параметров, и, соответственно, она не имеет возможности менять значения передаваемых параметров в вызывающей программе. Во втором случае подпрограмма получает адреса передаваемых значений и может не только читать значения, но и менять их. И в том и в другом случае параметры или их адреса заносятся в стек. Причем, параметры всегда помещаются в стек в том порядке, в котором они описаны при объявлении процедуры, то есть слева направо. Исключение составляет только конвенция register, при которой до трех параметров помещается в регистры, а остальные помещаются в стек в порядке, обратном их описанию в списке параметров.

Параметры – значения. Параметры – значения скалярного типа (char, Boolean, smallint, word, shortint, byte, integer и перечисляемые типы) непосредственно помещаются в стек. Если размер параметра составляет 1 байт, то он помещается в стек в виде целого слова. Сам параметр располагается в первом (младшем) байте этого слова, старший байт при этом не инициализируется. Параметры размером 2 и 4 байта помещаются в стек в виде слова и двойного слова соответственно.

Адреса всех типов помещаются в стек в виде 32-х разрядных смещений.

Строковые параметры, переданные по значению, независимо от их размера вызывающей программой в стек не записываются. Вместо этого в стек помещается адрес копии строки (4 байта). Сама копия размещается в отведенной для этого памяти стека, предназначенной для локальных данных.

Записи, массивы и объекты, имеющие размер 1, 2 и 4 байта, передаются непосредственно через стек. Для всех остальных размеров в стек заносится указатель (4 байта) на копию данного параметра.

Множества, так же как и строки, никогда не помещаются непосредственно в стек, а передаются с помощью указателя на копию 256-ти битового представление множества. Первый бит младшего байта всегда соответствует базовому элементу множества с порядковым значением 0.

Параметры – переменные. Параметры – переменные передаются в процедуры одним и тем же способом – через указатель на их содержимое.

Например, задание списка параметров в следующем виде:

(a:smallint; b:char; s:string;var c: byte)

приведет к тому, что в стек последовательно будут помещены: 2 байта **a**, 2 байта **b** (т.к. запись в стек идет словами, второй байт останется неинициализированным), 4-х байтовый указатель на копию строки **s** и 4-х байтовый указатель на байт **c** (см. рисунок 3.2).

Рисунок 3.2 – Состояние стека после записи параметров

Возвращение результатов процедур и функций в основную программу. Процедуры Delphi Pascal возвращают результаты через параметры, передаваемые по ссылке.

Форма представления результата функции зависит от типа возвращаемых данных. Результат функций скалярных типов возвращается в регистрах процессора:

байт – в AL;

слово – в AX;

двойное слово – в EAX;

указатели – в EAX (32-х разрядное смещение).

Исключением является результат строкового типа, для размещения которого Delphi Pascal записывает в стек указатель на специально выделенную область.

Доступ к параметрам из процедур на ассемблере. При передаче управления ассемблерной процедуре вершина стека содержит адрес возврата и расположенные в старших адресах (стек растет в сторону младших адресов) передаваемые параметры. Для доступа к этим параметрам ассемблер использует регистр EBP.

По соглашениям, принятым в Pascal, вызываемая процедура должна перед возвратом управления выполнить очистку стека от переданных ей параметров. Для этого можно использовать 2 способа. Первый – заключается в указании после размера области параметров команды RET n, где n – размер области переданных параметров в байтах.

Второй способ заключается в сохранении адреса возврата в регистре или в памяти и последовательное извлечение всех параметров из стека с помощью команды POP. Применение команды POP позволяет выполнить оптимизацию программы по скорости, а также уменьшает размер процедуры, так как каждая из них занимает всего 1 байт. В этом случае возврат управления можно выполнить, используя команду безусловной передачи управления по адресу в регистре.

3.4 Компоновка модулей

Для совместной компоновки с Delphi Pascal сегмент кодов ассемблерного модуля должен носить имя **code**, а сегмент данных – **data**. Оба сегмента должны быть описаны **public**. При этом в процессе компоновки сегменты кодов и данных программы и модуля на ассемблере будут объединены, и появится возможность доступа к глобальным данным Delphi Pascal через объявление их внешними (**extern)** в сегменте данных ассемблерной части. Причем, даже, если таким образом осуществляется доступ к массиву, в **extern** достаточно указать ссылку на первый элемент, например:

extern mas:word ; mas - массив, объявленный **var mas:array[1:10] of smallint.** Доступ к последующим элементам будет осуществляться по правилам ассемблера, т.е. с использованием одной из схем адресации элементов в памяти.

В свою очередь, данные ассемблерной части программы, даже будучи размещенными в общем сегменте данных с Pascal, останутся для части программы, написанной на языке Паскаль, «невидимыми». Кроме того, ассемблерные данные, размещенные в сегменте данных **data,** по соглашениям Паскаля *нельзя инициализировать*.

Правила модульного программирования ассемблера требуют, чтобы все имена программы, использующиеся отдельно транслируемыми модулями, были описаны как внутренние **public**, а все имена, используемые ассемблером из других модулей, – как внешние **extern**.

Модуль на ассемблере необходимо транслировать, используя 32-х разрядный ассемблер фирмы Borland или фирмы Microsoft и указав необходимые опции:

tasm32 /ml <Имя исходного модуля>**.asm**

ml /c <Имя исходного модуля>**.asm**

В среде RADAsm для того, чтобы выполнить ассемблирование, необходимо в настройках проекта указать:

3,O,$B\ML.EXE /c,2.

Однако, как показывает опыт, созданный объектный модуль формирует таблицы, несовместимые с таблицами Delphi Pascal. Это объясняется тем, что среда RADAsm использует некоторые соглашения языка C++. Поэтому, модуль на ассемблере целесообраз-

но транслировать с помощью средств среды Turbo Delphi. Это можно сделать двумя способами.

1. Добавить в Turbo Delphi внешний инструмент, назначив в качестве такого инструмента программу-ассемблер ml.exe. Для этого необходимо выбрать пункт меню **Tools/Configure tools/**. После его нажатия на экране появится окно (см. рисунок 3.3), в котором нужно выбрать кнопку **Add.**

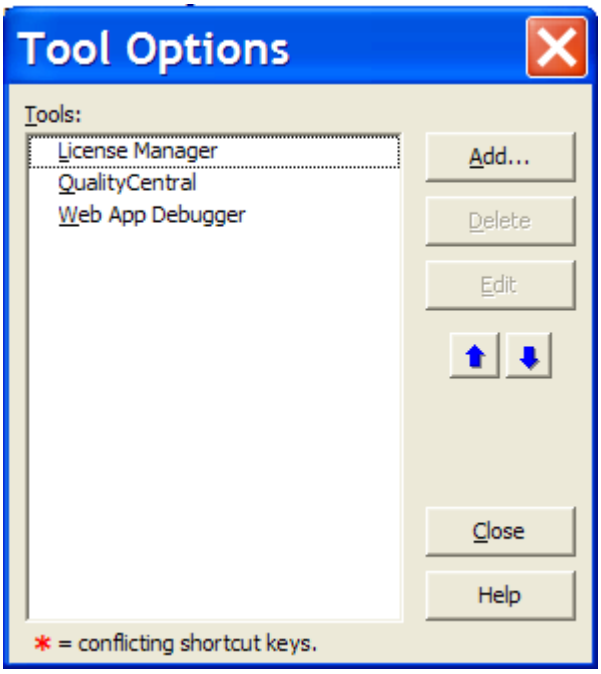


Рисунок 3.3 – Окно пункта меню **Tools/Configure tools/**

и в появившемся окне **Tool Properties** задать необходимые параметры. (см. рисунок 3.4)

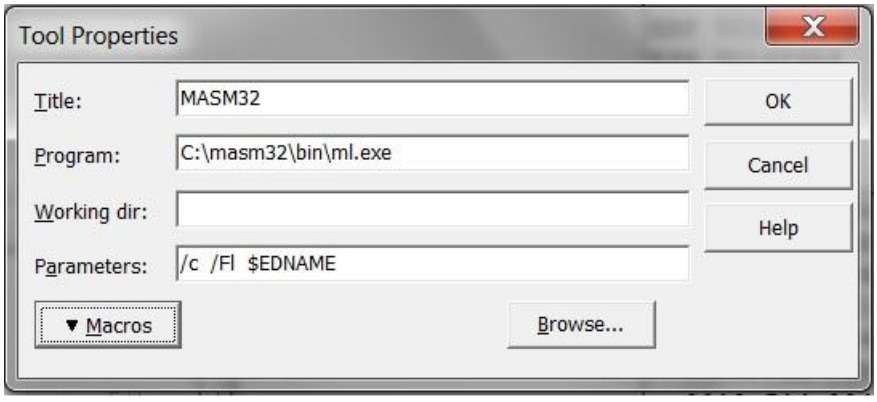


Рисунок 3.4 - Окно **Tool Properties** с заполненными полями.

После этого нужно в проекте открыть файл на ассемблере через меню **File/ Open** и сделать вкладку с файлом активной. Затем, используя пункт меню **Tools/Masm32** инициировать процесс компиляции файла **<Имя>.asm**. Результат компиляции следует смотреть в файле **<Имя>.lst,** который находится в текущей директории, с помощью пункта меню

File/ Open. В случае наличия ошибок в файле листинга появится сообщение (см. рисунок 3.5)

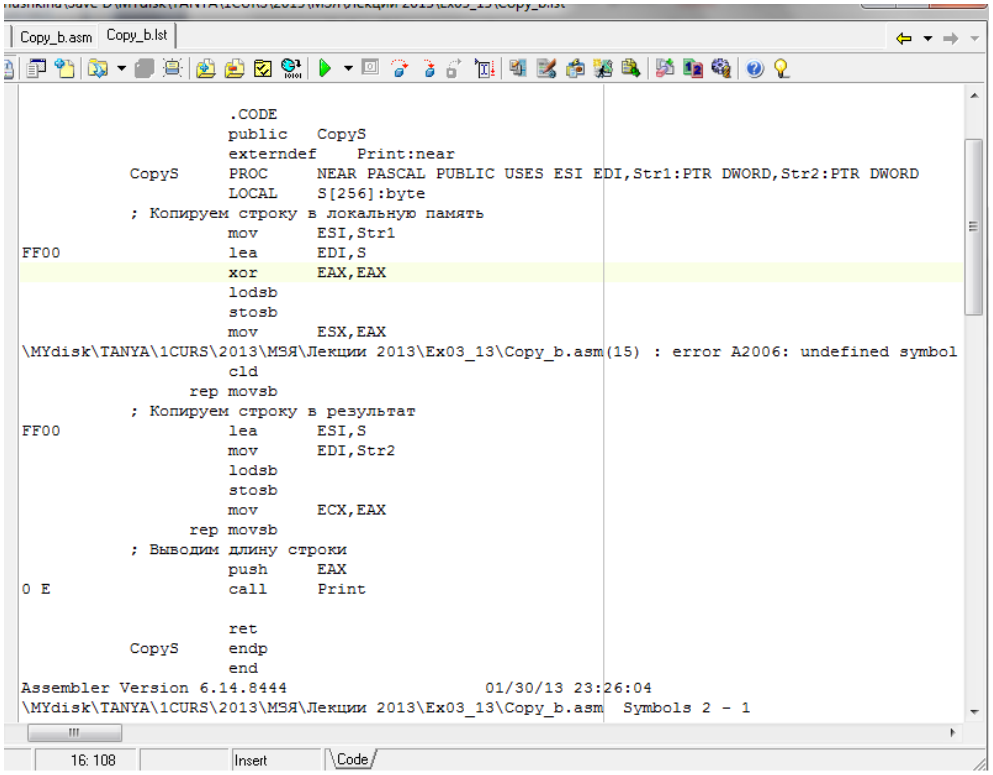


Рисунок 3.5 Вид окна с открытым файлом *.lst с ошибкой

После нахождения ошибки текст программы следует исправить, а изменения сохранить с помощью пункта меню **File/ Save**. После этого процесс компиляции повторяется. Когда компиляция проходит без ошибок листинг содержит сообщение об успешной компиляции (см. рисунок 3.6), а в текущей директории появляется файл с расширением <Имя>..Obj.

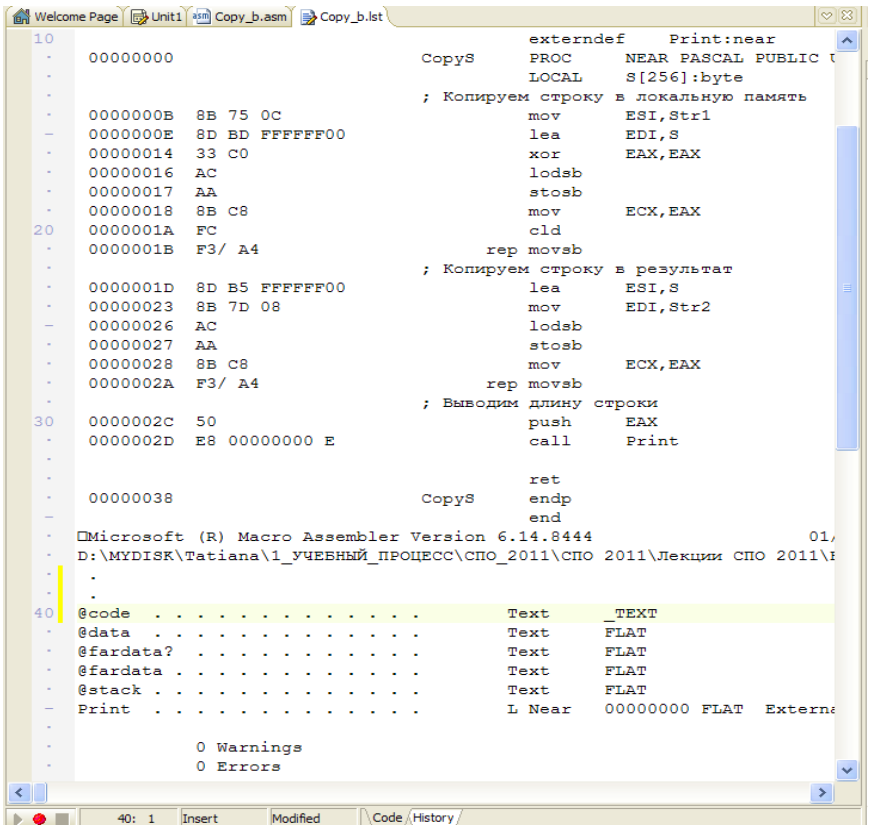


Рисунок 3.6 - Вид окна с открытым файлом *.lst без ошибок

2. Добавить в Turbo Delphi встроенный инструмент, назначив в качестве такого инструмента программу-ассемблер ml.exe. Для этого необходимо выбрать пункт меню **Tools/Build**. После его нажатия на экране появится окно (см. рисунок 3.7), в котором нужно выбрать пункт **Add .**

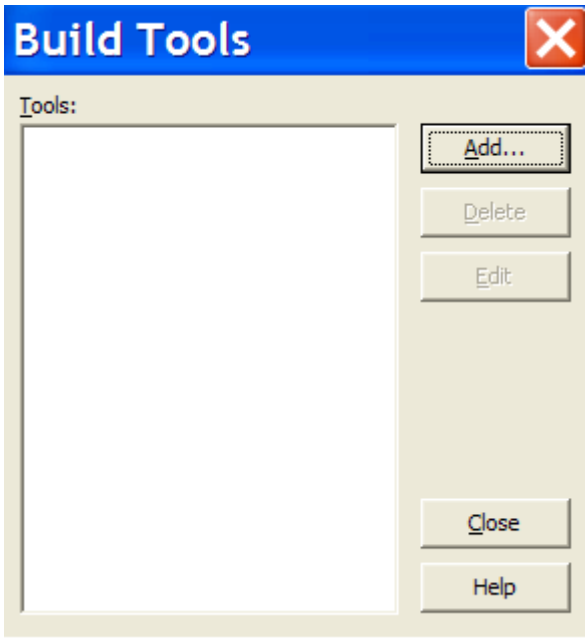


Рисунок 3.7 - Окно пункта меню **Tools/Build**.

и в появившемся окне **Edit Tool** задать необходимые параметры. (см. рисунок 3.8)

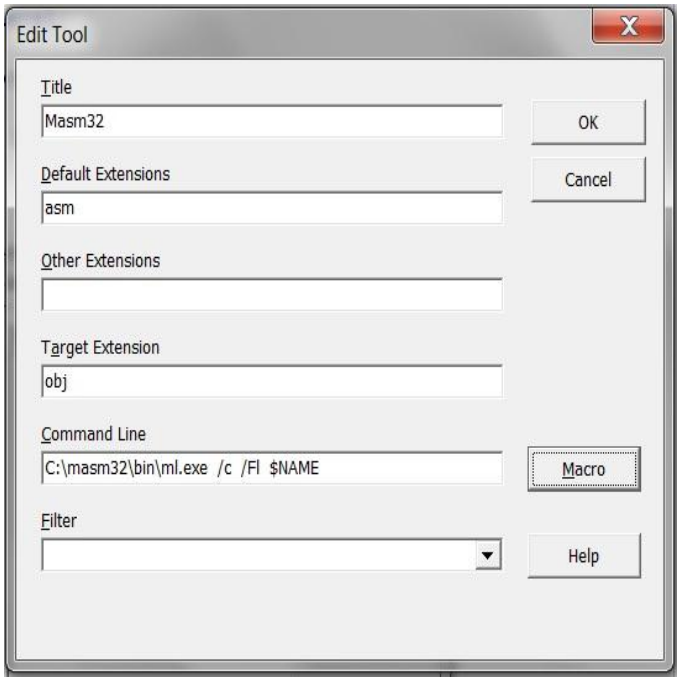


Рисунок 3.8 - Окно **Edit Tool** с заполненными полями.

Затем, для выполнения автоматической компиляции, файл с расширением **<Имя>.asm** необходимо подключить к проекту с помощью пункта меню **Project/Add Existing Item**…(см. рисунок 3.9).

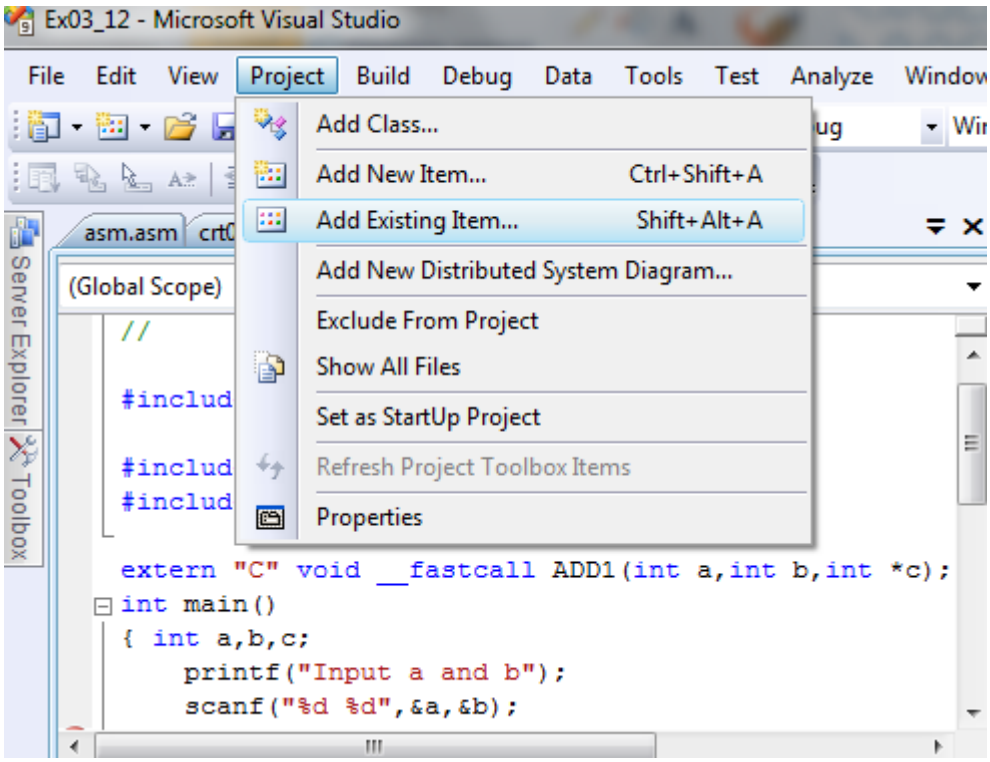


Рисунок 3.9 – Окно с выбранным пунктом меню **Project/Add Existing Item**…

В появившемся окне диалога выбора файла выбрать нужный ассемблерный файл и нажать кнопку Открыть (см. рисунок 3.10). После этого файл появится в дереве проекта. Теперь проект можно запускать на компиляцию. При этом, для ассемблерного файла будет автоматически вызван компилятор с ассемблера. Результат компиляции, как и в предыдущем случае, следует смотреть в файле **<Имя>..lst,**

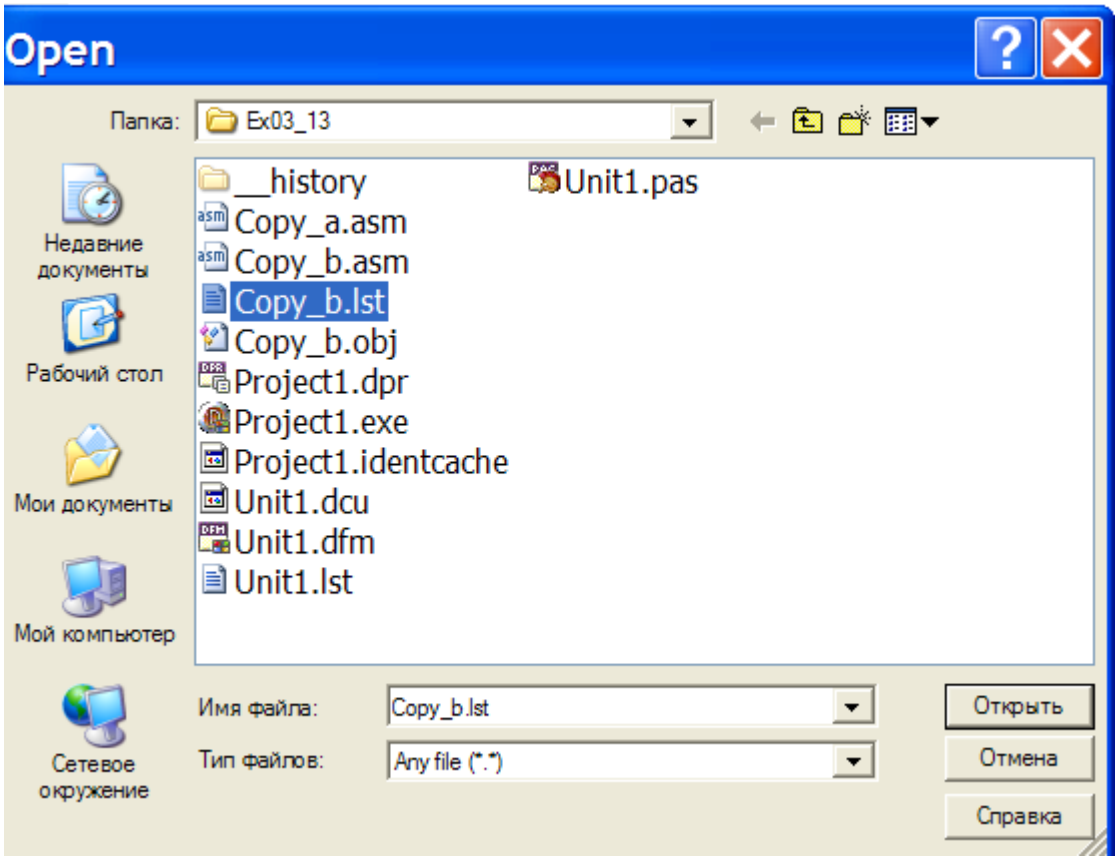


Рисунок 3.10 – Окно диалога выбора файла.

Полученный в результате успешной компиляции объектный модуль, в котором находится ассемблерная процедура, необходимо подключить в секции реализации основного модуля, используя директиву компилятора **{$L <имя файла>}**:

```
Implementation
{$L Add.obj}
...
```

Саму процедуру необходимо описать как внешнюю, указав используемую конвенцию:

```
procedure ADD1(A,B:integer;Var C:integer);  pascal; external;
procedure ADD1(A,B:integer;Var C:integer);  cdecl; external;
procedure ADD1(A,B:integer;Var C:integer); register; external;
procedure ADD1(A,B:integer;Var C:integer); stdcall; external;
```

```
procedure ADD1(A,B:integer;Var C:integer); safecall; external;
```

Вызов описанной функции в теле программы на языке DELPHI PASCAL независимо от конвенции осуществляется одинаково по имени:

```
ADD1(A,B,C);
```

При этом будет организована передача данных в процедуру в соответствии с указанной конвенцией.

3.5 Примеры

а) **Конвенция pascal**. Структура стека показана на рисунке 3.11.

```
      . 386
      . model flat
      . code
      public ADD1
ADD1 proc
      push EBP
      push EBP, ESP
      mov  EAX, [EBP+16]
      add  EAX,  [EBP+12]
      mov EDX,  [EBP+8]
      mov [EDX], EAX
      pop EBP
      ret 12  ; стек освобождает процедура
ADD1 endp
      end
```

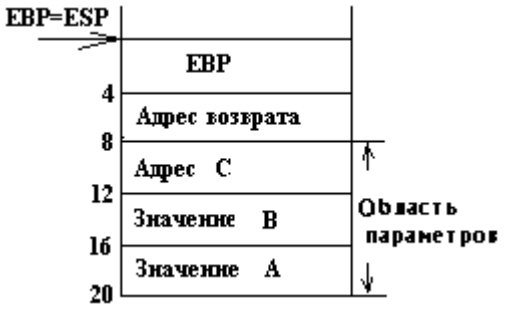


Рисунок 3.11 – Структура стека для конвенции pascal

б**) Конвенция cdecl**. Структура стека показана на рисунке 3.12.

```
      . 386
      . model flat
      . code
      public ADD1
ADD1 proc
      push EBP
      push EBP, ESP
      mov  EAX, [EBP+8]
      add  EAX,  [EBP+12]
      mov EDX,  [EBP+16]
      mov [EDX], EAX
```

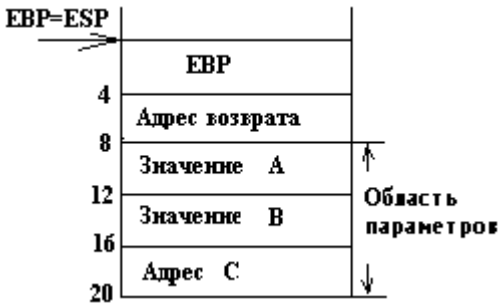


Рисунок 3.12– Структура стека для конвенции cdecl

```
      pop EBP
      ret ;стек освобождает
          ;вызывающая программа
ADD1 endp
      end
```

в) **Конвенция register**

```
      .386
      .model flat
      .code
      public ADD1
ADD1 proc
      add  EDX,EAX
      mov [ECX],EDX
      ret   ; стек освобождает вызывающая программа
ADD1 endp
      end
```

Размещение параметров:

первый параметр A в регистре EAX

второй параметр B в регистре EDX

третий параметр адрес C в регистре ECX

г) **Конвенция stdcall**. Структура стека показана на рисунке 3.13.

```
     .386
     .model flat
     .code
     public ADD1
ADD1 proc
      push EBP
      push EBP, ESP
      mov  EAX, [EBP+8]
      add  EAX,  [EBP+12]
      mov EDX,  [EBP+16]
      mov [EDX], EAX
       pop EBP
       ret 12        ; стек освобождает процедура
ADD1  endp
       end
```

Рисунок 3.13 – Структура стека для конвенции stdcall

д) **Конвенция safecall**= stdcall + формирование исключения при ошибке

Особенности передачи строки, как параметра – значения.

Пусть описана функция, которая получает строку по значению и возвращает эту же строку в качестве результата работы функции, например:

```
Function DeLL1(S:Shortstring):Shortstring; pascal;
  Begin Result:=s; end;
```

В этом случае DELPHI передает в функцию адрес исходной строки, а в стеке создает локальную копию строки, с которой и работает процедура. Структура стека в момент работы подпрограммы показана на рисунке 3.14.

```
        .386
        .model flat
        .code
        public   Dell1
Dell1 proc
        push  EBP
        mov   EBP,ESP
        add   ESP,0FFFFFF00h ;выделение памяти под копию строки
        push  ESI
        push  EDI
        mov   ESI,[EBP+0ch] ; адрес исходной строки
        lea   EDI,[EBP-000000100h]; адрес копии исходной строки
        xor   ECX,ECX
        mov   CL,[ESI]  ; длина исходной строки
        inc   ECX       ; адрес начала исходной строки
        rep   movsb     ; копирование исходной строки
        mov   EAX,[EBP+8]    ;адрес результата
        lea   EDX,[EBP-00000100h]  ;адрес копии строки
        call  @PStrCpy
        pop   EDI
        pop   ESI
        mov   ESP,EBP
        pop   EBP
        ret   8
Dell1 endp
```

Рисунок 3.14 – Структура стека при работе функции Dell1

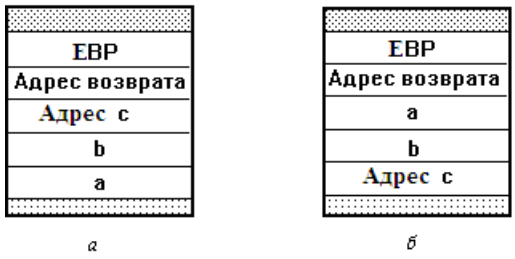
```
@PStrCpy  proc
          xor  ECX,ECX
          push ESI
          push EDI
          mov  CL,[EDX]
          mov  EDI,EAX
          inc  ECX
          mov  ESI,EDX
          mov  EAX,ECX
          shr  ECX,02h
          and  EAX,03h
          rep  movsd
          mov  ECX,EAX
          rep  movsb
          pop  EDI
          pop  ESI
          ret
@PStrCpy  endp
          end
```

4 ВЗАИМОДЕЙСТВИЕ C++ И АССЕМБЛЕРА

4.1 Основные принципы

4.1.1 Передача параметров и возвращение результатов функции

В настоящее время реализованы два варианта передачи параметров между основной программой и подпрограммой. Один, описанный выше, используется в программах на Паскале (см. рисунок 4.1, *а*). Другой, использующий обратный порядок записи параметров в стек, реализован в Си (см. рисунок 4.1, *б*). Кроме этого, в отличие от Паскаля вызываемая программа на Си не освобождает область параметров. В Си это делает вызывающая программа.



а *б*

Рисунок 4.1 – Структура стека при передаче параметров:
а – по варианту, принятому в Паскале, *б* – по варианту, принятому в Си

Все параметры в C++ передаются в подпрограмму *по значению*. Если передается значение, то в стек помещается значение, если передается адрес, то в стек помещается адрес. Так при вызове функции с прототипом:

```
void pro(int a, int b, int * c);
```

в стек сначала будет занесено смещение параметра **c** длиной 4 байта, затем **b** и **a** по четыре байта каждый, а затем, как обычно, ближний адрес возврата (см. рисунок 2.7, *б*).

Если используется функция с переменным числом параметров, то это отразится только на размере области параметров, так как каждый параметр будет помещен в стек, а удаление параметров будет выполнять вызывающая программа.

Возвращаемые значения должны быть записаны в регистры:

char, short, enum, – в регистр AX;

int, указатель **near** – в регистр EAX;

float, double – в регистры TOS и ST(0) сопроцессора;

struct – записывается в память, а в регистр записывается указатель на нее. В качестве исключения структуры длиной в 1 и 2 байта возвращаются в AX, а 4 байта – в EAX.

Существует еще одна особенность внутреннего представления функций на языках Си и C++: компилятор языка изменяет внутренние имена. Так перед всеми глобальными

именами в Си указывается символ подчеркивания, а в имена функций C++ добавляется информация о ее аргументах. Причем правила передачи информации об аргументах различны для сред Visual Studio и Builder.

Обработка имени выполняется автоматически и скрыта от программиста. Однако если какой-то модуль написан на ассемблере, то при подключении к программе на Си или C++ программист должен самостоятельно выполнить обработку указанных имен в этом модуле.

Обработку имен ассемблерных функций можно и не выполнять, например, чтобы избежать несовместимости с последующими версиями компиляторов, в которых возможны изменения алгоритма этой обработки. С этой целью C++ предоставляет возможность использования стандартных имен функций Си в программах написанных на C++. Такие функции должны объявляться с квалификатором **"C"** например:

```
extern "C" { int ADD (int *a,int b; ...}
```

Все функции, объявление которых заключено в фигурные скобки, будут иметь имена, соответствующие соглашениям, принятым в языке Си. Указанная в примере функция ADD будет иметь внутреннее имя **_ADD.**

4.1.2 Внутренний формат данных C++

Язык C++ «под Windows» использует следующие типы данных.

Целое -

short int: -32768..32767 – слово со знаком;

unsigned short int: 0..65535 – слово без знака;

int, long int: – двойное слово со знаком;

unsigned long int: – двойное слово без знака;

char: – -128..127 байт **со знаком** (передается слово);

unsigned char: 0..255 – байт без знака (передается слово).

***char**: – строки – указатель на массив символов с нулем на конце.

Указатель, массив – **near –** 32-х разрядное смещение.

4.1.3 Определение глобальных и внешних имен

В отличие от Паскаля Си позволяет ассемблеру объявлять новые глобальные переменные, доступные для всех модулей. Это достигается за счет размещения этих переменных в сегменте данных, отведенном для глобальных переменных, и описания его внутренней директивой PUBLIC**.** Имя такой переменной по правилам Си должно начинаться со знака подчеркивания. Прочие модули, использующие данное имя, должны включать

его описание как EXTERN (на ассемблере или на Си). Пример такого расширения списка переменных приведен в разделе 4.2.2 (случай б).

4.2 Особенности взаимодействия Visual C++ и ассемблера

4.2.1 Компоновка модулей

Подключаемый модуль, написанный на ассемблере, необходимо предварительно отранслировать, используя один из 32-х разрядных ассемблеров.

MASM32: **`ml/c/coff <Имя файла>.asm`**

TASM32: **`tasm32 /ml <Имя файла>.asm`**

Ассемблирование можно выполнить и в среде Visual Studio. Для этого необходимо добавить внешний инструмент с помощью **Tools/External Tools…/Add.** В открывшемся после нажатия **Add** окне заполнить поля в соответствии с рисунком 4.2.

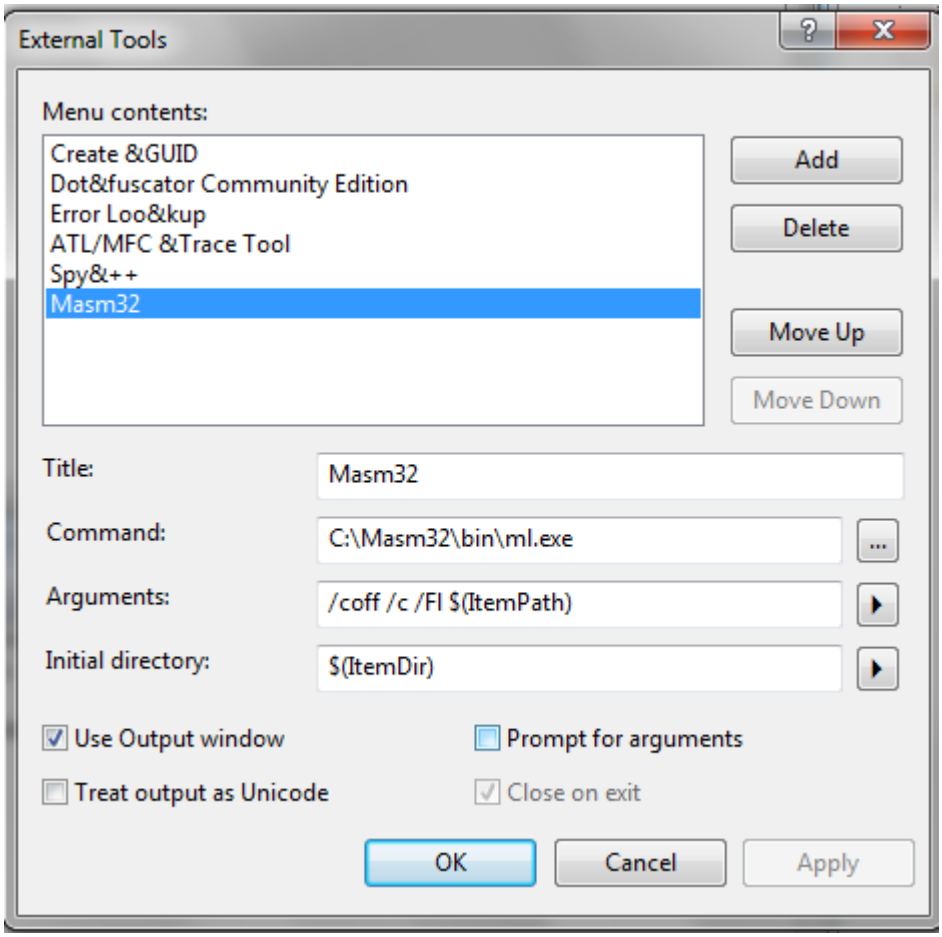


Рисунок 4.2 – Окно добавления инструмента с настройками

После этого нужно в проекте открыть файл на ассемблере через меню **File/ Open/File** и сделать вкладку с файлом активной. Затем, используя пункт меню **Tools/Masm32** инициировать процесс компиляции файла **<Имя файла>.asm**. Результаты компиляции будут отражены в окне **Output.** В случае наличия ошибок в окне появится сообщение, а модуле на ассемблере ошибочные строки будут помечены зеленым. (см. рисунок 4.3) .

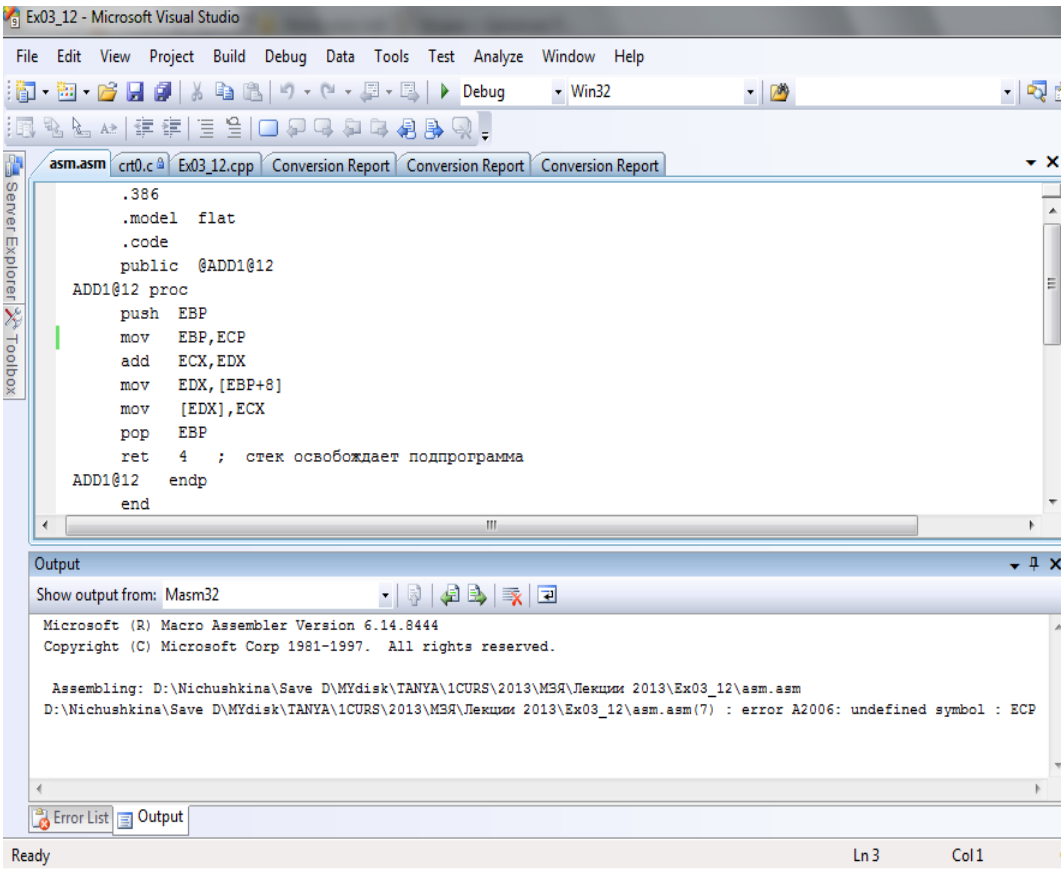


Рисунок 4.3 Вид окна **Output** с ошибкой.

Если компиляция прошла успешно, то в окне **Output** появится соответствующее сообщение (см. рисунок 4.4), а в папке проекта появится файл **<Имя файла>.obj**.

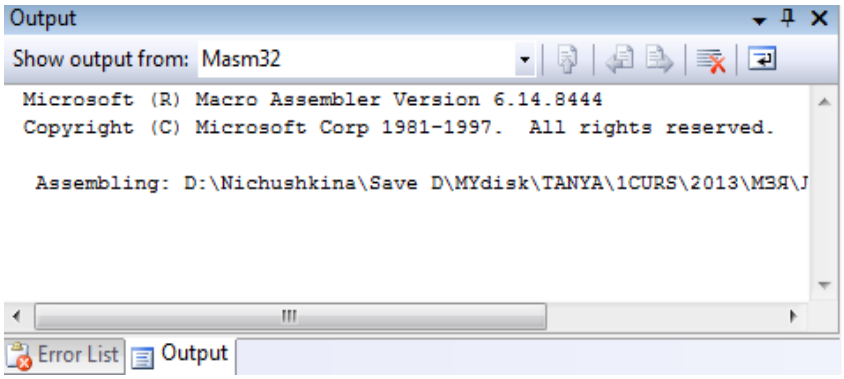


Рисунок 4.4 Вид окна **Output** при успешной компиляции.

Полученный в результате успешной компиляции объектный модуль **<Имя файла>.obj** необходимо подключить к приложению, используя пункт меню **Project/Add Existing Item...** После выполнения подключения файл появится на вкладке **Solution Explorer** (см. рисунок 4.5) и можно приступать к сборке проекта.

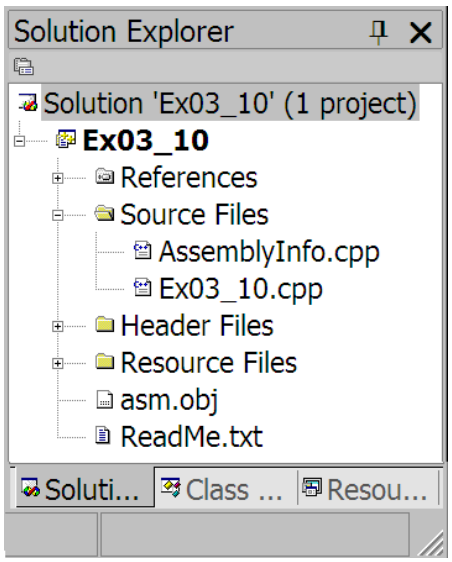


Рисунок 4.5 – Вкладка Solution Explorer с добавленным файлом **asm.obj**

Для корректной компоновки проекта, в тексте программы внешняя процедура на ассемблере должна быть описана как

```
extern void __<Конвенция>  <Имя>(<Список форм. параметров>);
```

4.2.2 Примеры

а) **Конвенция cdecl.** Структуру стека см. на рисунке 3.12.

Текст модуля на C++:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
extern "C" void __cdecl ADD1(int a,int b,int *c);
int main()
{ int a,b,c;
    printf("Input a and b:\n");
    scanf("%d %d",&a,&b);
    ADD1(a,b,&c);
    printf("c=%d.",c);
    getch();
    return 0;
};
```

Текст модуля на ассемблере:

```
.586
.model  flat
.code
```

```
          public  _ADD1
_ADD1     proc
          push    EBP
          mov    EBP,ESP
          mov    EAX,[EBP+8]
          add    EAX,[EBP+12]
          mov   EDX,[EBP+16]
          mov    [EDX],EAX
          pop    EBP
          ret
_ADD1     endp
          end
```

б) **Создание внешних переменных в модуле на ассемблере**

Текст программы на C++:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
extern "C" void __cdecl ADD1(int a,int b);
extern int d;
int main()
{ int a,b;
    printf("Input a and b:\n");
    scanf("%d %d",&a,&b);
    ADD1(a,b);
    printf("d=%d.",d);
    getch();
    return 0;
};
```

Текст программы на ассемблере:

```
          .586
          .model  flat
          .data
          public ?d@@3HA
?d@@3HA   DD     ?
          .code
          public ADD1
```

```
_ADD1     proc
          push   EBP
          mov    EBP,ESP
          mov    EAX,[EBP+8]
          add    EAX,[EBP+12]
          mov    ?d@@3HA,EAX
          pop    EBP
          ret
_ADD1     endp
          end
```

в) **Конвенция stdcall**/ Структура стека показана на рисунке 3.13.

Программа на C++:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
extern void __stdcall ADD1(int a,int b,int *c);
int main()
{ int a,b,c;
    printf("Input a and b:\n");
    scanf("%d %d",&a,&b);
    ADD1(a,b,&c);
    printf("c=%d.",c);
    getch();
    return 0;
};
```

Программа на ассемблере:

```
      .386
      .model  flat
      .code
      public ?ADD1@@YGXHHPAH@Z
?ADD1@@YGXHHPAH@Z proc
      push EBP
      mov  EBP,ESP
      mov  ECX,[EBP+8]
      add  ECX,[EBP+12]
      mov  EAX,[EBP+16]
```

```
      mov   [EAX],ECX
      pop   EBP
      ret   12
?ADD1@@YGXHHPAH@Z endp
      end
```

г) **Конвенция fastcall**

Поскольку Visual C++ в регистрах передает только два параметра (1-й – в ECX, 2-й – в EBX), третий параметр будет передаваться адресом через стек.

Текст модуля на C++:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>

extern "C" void __fastcall ADD1(int a,int b,int *c);

int main()
{    int a,b,c;
     printf("Input a and b");
     scanf("%d %d",&a,&b);
     ADD1(a,b,&c);
     printf("c=%d.",c);
     getch();
     return 0;
}
```

Текст модуля на ассемблере:

```
      .386
      .model  flat
      .code
      public  @ADD1@12
@ADD1@12 proc
      push  EBP
      mov   EBP,ESP
      add   ECX,EDX
      mov   EDX,[EBP+8]
      mov   [EDX],ECX
      pop   EBP
      ret 4   ; стек освобождает подпрограмма
```

```
@ADD1@12    endp
        end
```

4.3 Особенности взаимодействия модулей на C++ Builder и ассемблере

4.3.1 Правила формирования внутренних имен

К именам функций при использовании компиляторов фирмы Borland C++ в начало добавляется символ **@**, а в конец дописываются символы **$q** и символы, кодирующие типы параметров функции в виде:

@ <Имя функции> $q<Коды типов параметров>

Коды типов параметров:

void - v char - zc int - i float - f double - d

short - s long - l *, [ ] - p ... - e

Например, **fa(int *s[], char c, short t) => @fa$qppizcs.**

4.3.2 Компоновка модулей

Подключаемый модуль на ассемблере необходимо предварительно оттранслировать. Затем объектный модуль, в котором находится ассемблерная процедура, необходимо подключить к приложению в файл проекта следующим образом:

Файл Project1.cpp должен включать директиву:

```
USEOBJ("add.obj");
```

Файл Unit1.cpp должен включать директиву:

```
Extern void __<Конвенция> ADD1(int a,int b,int &c);
```

Модуль на Ассемблере необходимо транслировать с опцией /mx:

tasm /mx Add.asm

Вызов процедуры выполняется по имени:

ADD1(a,b,c);

4.3.3 Примеры

а) **Конвенция pascal.** Структуру стека см. на рисунке 3.11.

```
        . 386
        . model flat
        . code
       public @ADD1$qiipi
@ADD1$qiipi proc
        push EBP
        push EBP, ESP
```

```
        mov  EAX, [EBP+16]
        add  EAX,  [EBP+12]
        mov EDX,  [EBP+8]
        mov [EDX], EAX
        pop EBP
        ret 12  ; стек освобождает процедура
@ADD1$qiipi   endp
        end
```

б) **Конвенция cdecl.** Структуру стека см. на рисунке 3.12.

```
        . 386
        . model flat
        . code
        public @ADD1$qiipi
@ADD1$qiipi proc
        push EBP
        push EBP,ESP
        mov  EAX,[EBP+8]
        add  EAX,[EBP+12]
        mov  EDX,[EBP+16 ]
        mov  [EDX],EAX
        pop  EBP
        ret   ; стек освобождает вызывающая программа
@ADD1$qiipi   endp
        end
```

Как сказано выше, обработку имен ассемблерных функций можно не выполнять, если использовать описание следующего формата

```
extern  "C" void __cdecl  ADD1(int a,intb,int &c);
```

тогда компилятор сгенерирует имя процедуры:

```
_ADD1 proc
```

в) **Конвенция fastcall**

При использовании этой конвенции часть параметров хранится в регистрах, однако, сначала данные из регистров сохраняются в область локальных данных (см. рисунок 4.6), а затем из нее грузятся в регистры и используются.

```
        . 386
        . model flat
```

```
      . code
      public @ADD1$qiipi
@ADD1$qiipi proc
      push EBP
      mov EBP, ESP
      mov   [EBP-12], ECX
      mov  [EBP-8], EDX
      mov  [EBP-4], EAX
      mov  EAX,  [EBP-4]
      add EAX,  [EBP-8 ]
      mov EDX, [EBP-12]
      mov [EDX], EAX
      mov ESP,EBP ; очистка области локальных данных
      pop EBP
      ret
@ADD1$qiipi   endp
            end
```

Рисунок 4.6 – Структура стека конвенции fastcall

5 Отладка разноязыковых модулей в Delphi и Visual C++

Процесс отладки программ, содержащих модули на различных языках, в средах Windows можно инициировать непосредственно в визуальной среде.

В среде Delphi для запуска процесса отладки необходимо использовать один из пунктов основного меню RUN и перейти в режим пошагового выполнения приложения без захода (**Step Over** – клавиша F8) или с заходом (**Trace Into** – клавиша F7) в подпрограммы. В процессе пошагового выполнения становится доступным пункт основного меню View. Подпункт этого пункта меню **View\Debug Windows** позволяет определить режим индикации отладки. Этот подпункт дает возможность определить точку останова (**BreakPoints**), просмотреть значения переменных (**Watches**) и содержимое стека (**Cal Stack**) и т.д. Однако, пошаговое выполнение приложения в Windows довольно длительный процесс. Поэтому целесообразно в том месте программы, в котором может быть ошибка, поставить точку останова или установить курсор. После этого в пункте меню **RUN** следует выбрать подпункт **Run** – при установленной точке останова или **Run to Cursor** (Выполнить до курсора). После остановки в указанной точке, необходимо с помощью подпункта **View\Debug Windows\CPU** выбрать режим отладки CPU (см. рисунок 5.1).

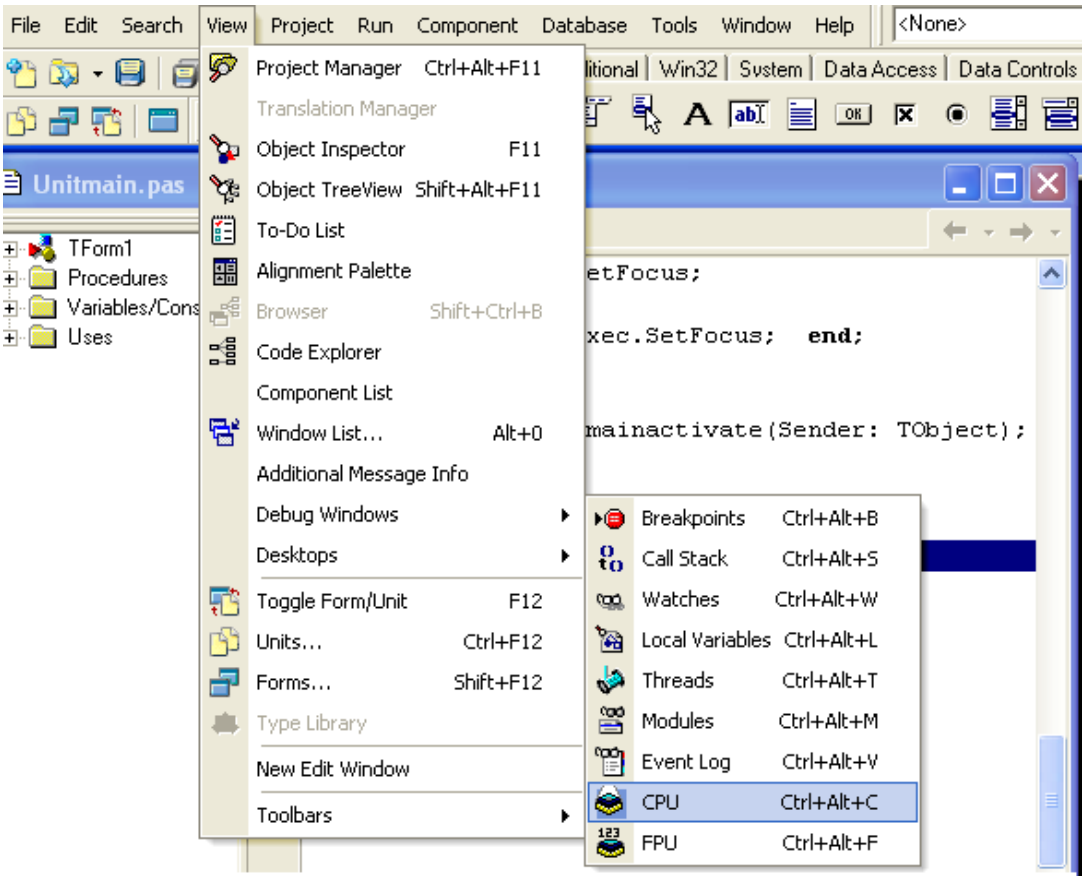


Рисунок 5.1 – Вид окна настройки режима отладки

После этого появится окно CPU режима отладки (см. рисунок 5.2). В этом окне представлена вся отладочная информация: текст программы с точки останова, содержимое стека, регистров и сегмента данных.

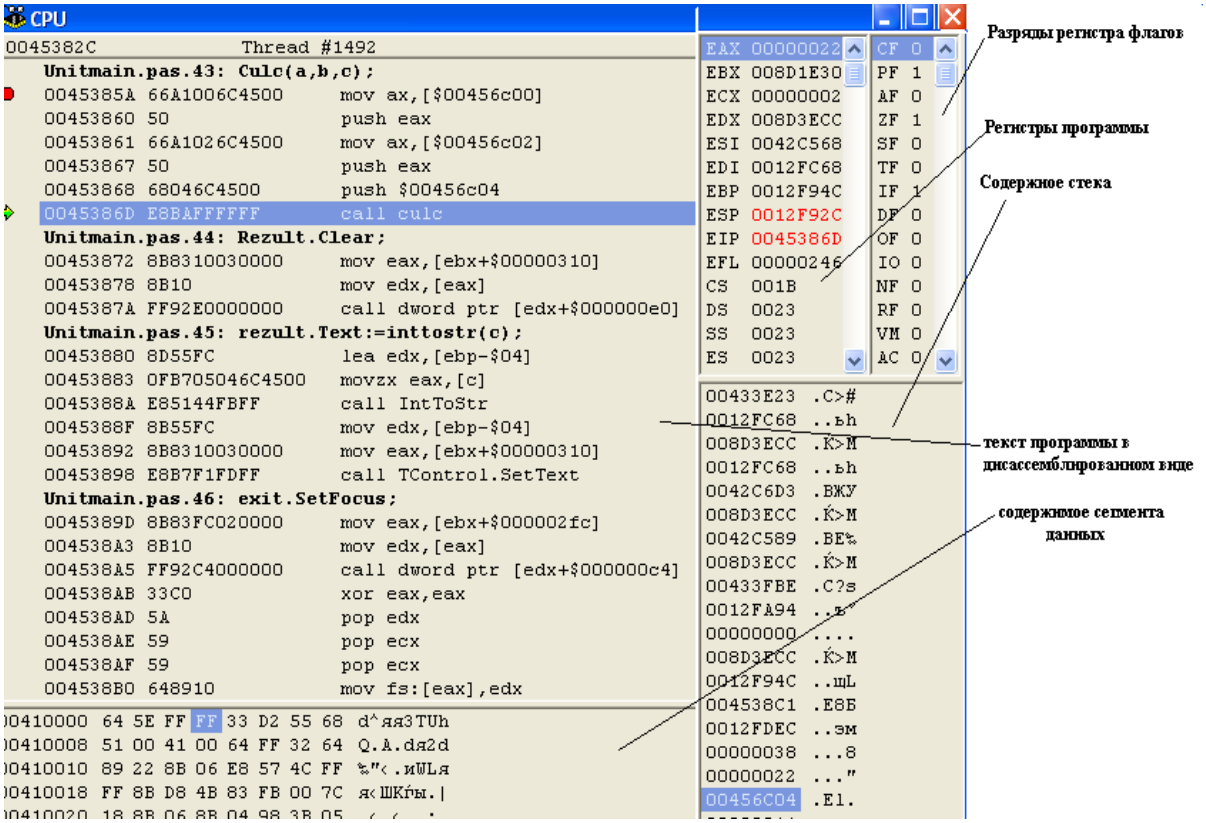


Рисунок 5.2 – Вид окна отладки Turbo Delphi в режиме CPU

Выполняя дальше программу в пошаговом режиме (клавиша F8 или клавиша F7), можно просмотреть все необходимые данные и определить источник ошибки. После исправления обнаруженной ошибки вновь выполняют программу. При выявлении новой ошибки процесс прогона программы в отладочном режиме следует повторить. Приведенная последовательность действий выполняется до получения правильного результата.

В среде Visual Studio также необходимо запустить программу на выполнение в пошаговом режиме. Для этого используется один из пунктов меню Debug **Debug/Step Into (F11)** или **Debug/Step Over (F10)**. Затем необходимо открыть окно дисассемблера, используя пункт меню **Debug/Windows/Disassemble** (см. рисунок 5.3)

Однако, в отличие от среды Delphi, в которой окно содержит всю необходимую информацию (содержимое регистров, содержимое памяти и т.д.), появившееся окно содержит только текст модуля. Для просмотра содержимого регистров или памяти необходимо использовать пункты меню **Debug/Windows/Registers** и **Debug/Windows/Memory.** После нажатия этих пунктов, на экране в соответствующих окнах появится нужная информация (см. рисунок 5.4).

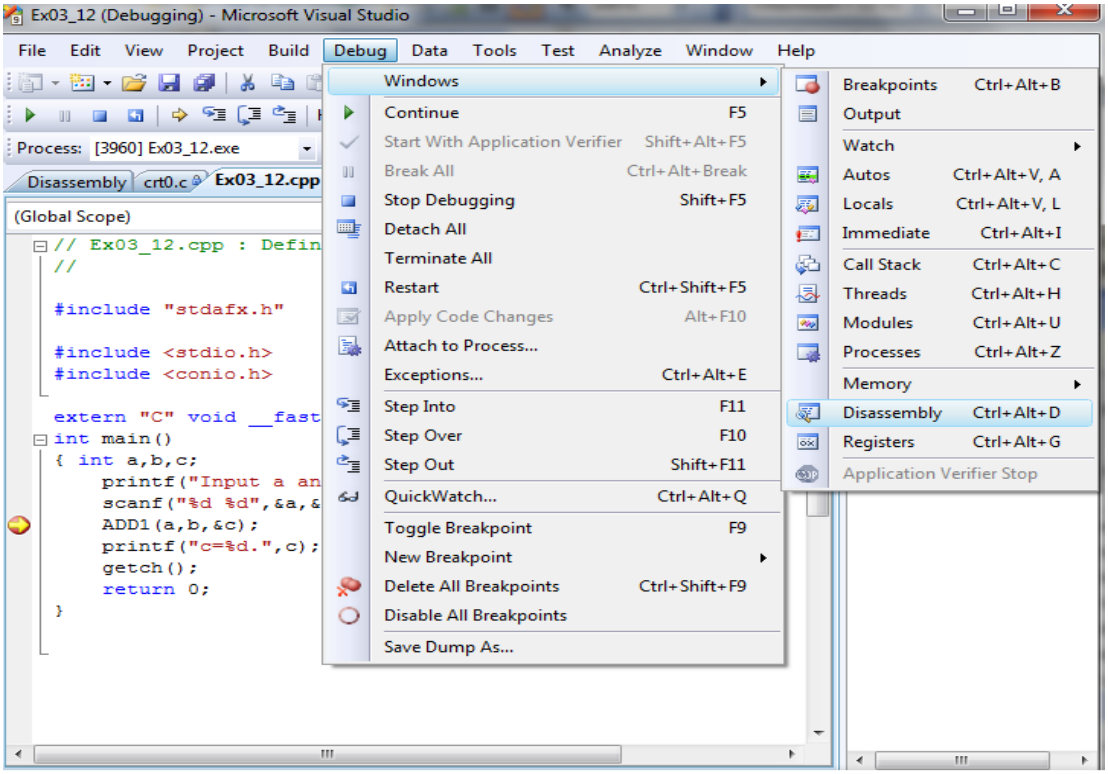


Рисунок 5.3 – Вид окна настройки режима отладки в среде **Visual Studio**

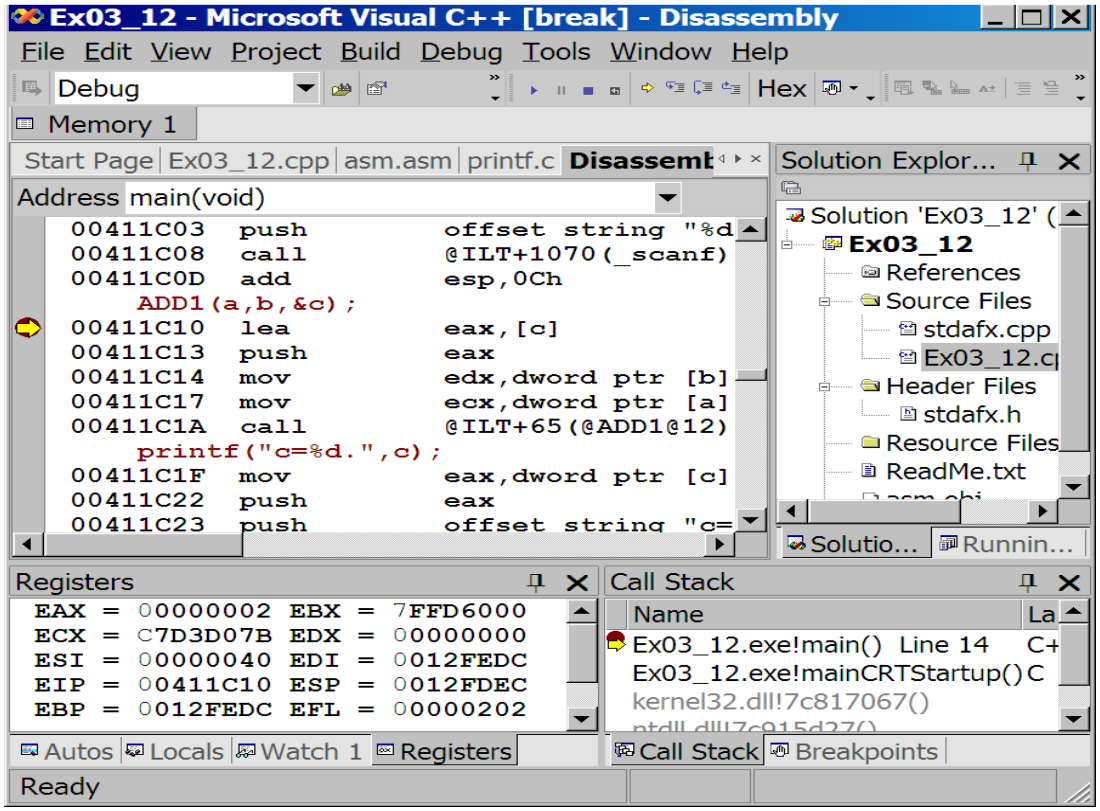


Рисунок 5.3 – Окно Дисассемблера Visual C++