Московский государственный технический университет
имени Н.Э. Баумана

Факультет «Информатика и вычислительная техника»
Кафедра «Компьютерные системы и сети»

Г.С. Иванова, Т.Н. Ничушкина

МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА. СВЯЗЬ РАЗНОЯЗЫКОВЫХ МОДУЛЕЙ

Электронное учебное издание

Учебное пособие по дисциплине
«Машинно-зависимые языки и основы компиляции»

Москва

УДК 004.432

Рецензенты: заведующий кафедрой ИУ5, д.т.н. профессор
Черненький Валерий Михайлович
к.т.н., доцент Коновалов Сергей Михайлович

Г.С. Иванова, Т.Н. Ничушкина

Модульное программирование на ассемблере. Связь разноязыковых модулей. Учебное пособие по дисциплине «Машинно-зависимые языки и основы компиляции». - М.: МГТУ имени Н.Э. Баумана, 2015. 58 с.

Учебное пособие содержит теоретические сведения об организации связей между модулями и способах передачи параметров при вызове подпрограмм на языке ассемблера из программ на том же языке или языках высокого уровня. Рассмотрены основные правила организации связей – конвенции о связях для операционных систем семейства Win32, а также соответствия форматов данных и правила компоновки модулей различных языков программирования. Приведены примеры, демонстрирующие особенности организации модулей при использовании различных конвенций.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по программе бакалавриата направления «Информатика и вычислительная техника» профиль «Вычислительные машины, комплексы, системы и сети».

Рекомендовано Учебно-методической комиссией НУК «Информатика и системы управления» МГТУ им. Н.Э. Баумана

Электронное учебное издание

Иванова Галина Сергеевна
Ничушкина Татьяна Николаевна

Модульное программирование на ассемблере.
Связь разноязыковых модулей.

Учебное пособие

по дисциплине Машинно-зависимые языки и основы компиляции

Оглавление

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 других языках программирования. 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 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 разноязыковых модулей.

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.

Целью настоящего пособия является последовательное изложение основ организации межмодульных связей, изучение которых позволит будущему программисту углубить свои знания в области разработки сложных программных систем.

Пособие предназначено для студентов МГТУ имени Н.Э. Баумана, обучающихся по программе бакалавриата направления «Информатика и вычислительная техника» профиль «Вычислительные машины, комплексы, системы и сети».

1 МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

1.1 Процедуры языка ассемблера

Процедура в языке ассемблера – это относительно самостоятельный фрагмент программы, к которому возможно обращение из разных ее частей. На языках высокого уровня такие фрагменты оформляют соответствующим образом и называют *подпрограммами*: функциями или процедурами.

С точки зрения структуры программы процедуры на языке ассемблера, как и подпрограммы на других процедурных языках, – это *программные модули*. Поддержка модульного принципа для языка ассемблера означает, что в нем существуют средства организации процедур (модулей).

К таким средствам относятся специальные машинные команды вызова процедуры и обратной передачи управления. Однако, в отличие от языков высокого уровня, ассемблер не требует специального оформления процедур. В любое место программы можно передать управление командой вызова процедуры, и оно вернется к вызвавшей процедуре, как только встретится команда возврата управления.

Отсутствие специального оформления может привести к трудночитаемым программам, поэтому в язык ассемблера включены директивы оформления процедур. В *простейшем* случае с использованием директив процедуры описываются следующим образом:

```
<Имя процедуры>    PROC    [<Тип вызова>][<Язык>]
            <Тело процедуры>
<Имя процедуры>    ENDP
```

Развернутый вариант описания процедуры, не являющийся обязательным, рассмотрен в разделе 1.5.2.

Квадратные скобки в формате простейшего описания процедуры согласно используемой нотации определяют, что указанные в них описатели при описании процедуры могут быть опущены. В этом случае применяются значения по умолчанию, задаваемые директивой языка ассемблера **.MODEL**, например

```
.MODEL flat,stdcall
```

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 – дальним **far**. В модели памяти **flat**, которая используется при создании 32-х битных приложений, все процедуры считаются ближними, что и подразумевается по умолчанию для данной модели.

Язык – параметр, определяющий конвенцию о связи, т. е. способ передачи параметров и управления. Конвенция, применяемая по умолчанию, как и модель памяти, определяется директивой языка ассемблера .MODEL. Так описатель **stdcall,** указанный в примере выше, назначает конвенцию «стандартный вызов» (см. далее) конвенцией, используемой по умолчанию.

1.2 Организация связи процедур языка 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).

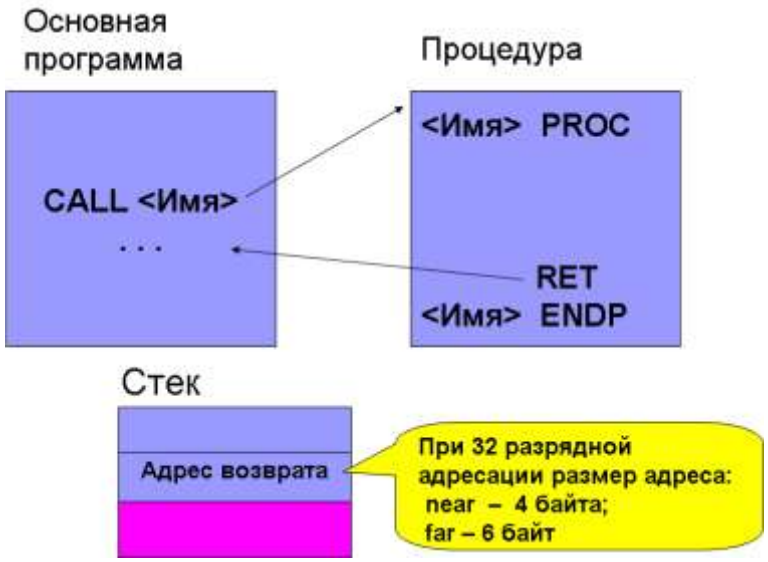


Рисунок 1.1 – Передачи управления процедуре и обратно

Машинная 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 описанию процедуры. Тип вызова неоформленной процедуры (фрагмента программы) необходимо уточнять, указывая перед именем или

адресом описатель **near ptr** – для организации ближнего и **far ptr** – для организации дальнего вызовов. Как уже упоминалось, по умолчанию в модели памяти **FLAT** все вызовы – ближние.

При 32-х разрядной адресации (модель памяти **FLAT**) в пределах сегмента кода адрес любой команды, в том числе первой команды процедуры, представляет собой 32-х разрядное смещение относительно условно нулевого адреса начала сегмента.

Для обеспечения возврата управления из вызванной процедуры команда **CALL** помещает в стек *адрес возврата* в вызывающую процедуру. Таким адресом является 32-х разрядное смещение относительно начала сегмента кода следующей за **CALL** команды.

Затем команда **CALL** загружает 32-х разрядное смещение первой команды вызываемой процедуры 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 с помощью машинной команды **RET**, которая должна быть последней выполняемой командой процедуры или непоименованного фрагмента. Формат команды **RET**:

```
RET [<Число>].
```

В зависимости от типа вызова команда извлекает из стека ближний или дальний адрес возврата и загружает его в **EIP** (при ближнем вызове) или **CS:EIP** (при дальнем вызове).

В команде **RET** может быть указан один операнд – целое положительное число. Это число после извлечения из стека адреса возврата будет добавлено к содержимому регистра указателя стека **ESP**. Это равносильно удалению из стека указанного количества байтов данных. 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 доступа к данным из процедур на языке ассемблера

Существует несколько способов организации доступа к данным из процедур.

Любой процедуре доступны данные сегмента данных непосредственно по их именам или адресам (*прямой доступ*). При *совместной трансляции*, т.е. когда программа записана в единый исходный модуль и транслируется как единое целое, такой доступ обеспечивается автоматически.

При *раздельной трансляции* текст программы разбит на несколько исходных модулей, которые потом компонуются в единое целое. Для обеспечения доступа к данным из других исходных модулей необходимо в тексты взаимодействующих модулей добавлять специальные директивы.

Недостатком прямого доступа является жесткая связь процедуры и данных. При этом процедура может работать только с теми данными, для которых она написана. Чтобы обеспечить возможность смены обрабатываемых процедурой данных, применяют косвенный доступ к данным.

При *косвенном доступе* имена или адреса данных указывают в командах не напрямую, а через промежуточное звено: регистр или область памяти. Адреса данных в регистрах или памяти можно изменить, тогда процедура будет работать с другими данными. По аналогии с языками высокого уровня способы косвенного доступа к данным называют способами передачи параметров. Рассмотрим указанные способы организации доступа процедур к данным более подробно.

1.3.1 Организация прямого обращения процедур к данным

При прямом обращении к данным вызываемая и вызывающая процедуры обращаются к одним и тем же данным по их символическим именам или адресам в сегменте данных. Способ оформления такого доступа зависит от того, находятся основная программа и процедура в одном исходном модуле (файле) или в разных.

Совместная трансляция основной программы и процедуры. При совместной трансляции вся программа представляет собой один исходный модуль, который транслируется за один вызов транслятора. В этом случае ассемблер формирует единое адресное пространство программы. Соответственно все имена данных, размещенных в сегменте данных, видимы и в программе, и в процедуре, расположенных в сегменте кода.

Пример 1.1. Программа суммирования двух чисел. Пусть необходимо найти сумму чисел A и B и поместить ее в D.

Основная программа вызывает процедуру, которая выполняет операцию сложения. При этом процедура для доступа к данным используют их символические имена (рисунок 1.2). При необходимости так же будет обращаться к этим данным и основная программа.

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

```
          .586
          .MODEL  flat, stdcall
          OPTION CASEMAP:NONE
Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib
          .CONST    ; сегмент константных данных
MsgExit   DB    "Press Enter to Exit",0AH,0DH,0
          .DATA     ; сегмент инициализированных данных
A         DWORD  56
B         DWORD  34
          .DATA?    ; сегмент неинициализированных данных
D         DWORD  ?
inbuf     DB    100 DUP (?)
          .CODE     ; сегмент кода
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.

Раздельная трансляция процедур. При раздельной трансляции процедуры программы помещают в разные файлы, транслируют отдельно и объединяют в единую программу на этапе компоновки. Каждый файл в этом случае – отдельный модуль со своим адресным пространством. Поэтому необходимо указать компоновщику данные, по которым будет происходить «связывание» модулей. Такими данными являются:

- внутренние имена модуля, к которым будет происходить обращение из других модулей,
- 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 3.1)

<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 (имя) данных: BYTE, WORD, DWORD;

идентификатор (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:

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

В зависимости от обстоятельств может служить как **PUBLIC** или **EXTERN.**

Пример 1.2. Программа суммирования двух чисел. Рассмотрим пример 1.1, но разместим основную программу с данными и процедуру в разных модулях (рисунок 1.3).

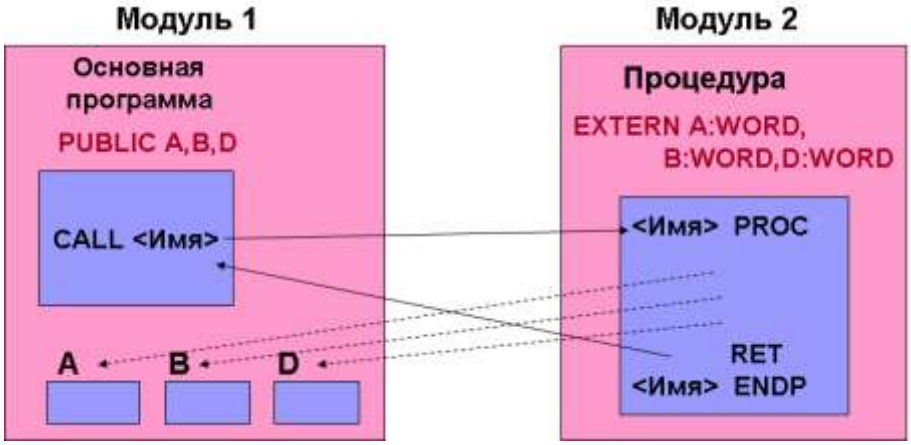


Рисунок 1.3 – Размещение основной программы и процедуры в разных исходных модулях

Основная программа:

```
.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 (?)
         PUBLIC A,B,D
         EXTERN SumDword:near   ; имя процедуры
         .CODE
Start:   call   SumDword
         Invoke StdOut,ADDR MsgExit
         Invoke StdIn,ADDR inbuf,LengthOf inbuf
         Invoke ExitProcess,0
         End    Start
```

Модуль, содержащий текст процедуры:

```
         .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 и его восстановление перед возвратом управления в основную программу позволяет гарантировать, что процедура не «испортит» содержимого этого регистра в основной программе.

Примечание – Создание многомодульных программ с использованием среды 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 **Assemble Modules**

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

Далее обсуждаются косвенные способы доступа к данным, называемые передачей параметров.

1.3.2 Передача параметров через регистры

Как упоминалось выше, косвенный доступ к данным из процедуры позволяет подменять данные, к которым обращается процедура, т.е. «передавать в процедуру» для обработки разные данные.

Если данных в процедуру передается немного (2-3 значения), то самый быстрый и простой способ – размещение параметров в регистрах. Если же данных много, или они представляют сложные структуры типа массива или записи, то использовать регистры не рационально. При необходимости перед вызовом процедуры в одни и те же регистры можно записать разные данные, т.е. процедура сможет работать с разными данными.

Пример 1.3. Программа суммирования двух чисел.

Основная программа записывает в регистры два параметра и адрес, по которому надо записать результат. Суммирование выполняет вызываемая процедура. Ниже приведен текст программы.

```
            .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        ; сложение чисел
            mov    [EDX],EAX      ; запись результата на указанное место
            ret
SumDword ENDP
            End    Start
```

1.3.3 Передача параметров через таблицу адресов

При использовании данного способа 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. Программа суммирования чисел 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 (рисунок 1.4).

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

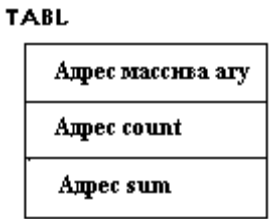


Рисунок 1.4 – Таблица адресов параметров

Текст процедуры находится в другом модуле:

```
          .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 большей частью используют языки высокого уровня.

В стек могут помещаться как копии значений параметров, так и их адреса. В первом случае мы говорим, что параметр передается «по значению», во втором – «по ссылке».

Копии значений параметров или их адреса помещают в стек командой **PUSH**, после чего управление передается вызываемой процедуре. Доступ к параметрам, хранящимся в стеке, из вызываемой процедуры осуществляют с использованием регистра **EBP**. В этот регистр помещают адрес вершины стека в момент начала работы процедуры, копируя его из регистра указателя стека **ESP**, а затем **EBP** используют как базовый регистр 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 при необходимости записывает на указанное место 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. Процедура использует 3 параметра: два исходных значения (адрес массива и количество элементов) и результат.

Все параметры передадим, как адреса: первый параметр - адрес первого элемента массива, второй параметр – адрес счетчика, содержащего количество элементов, третий параметр - адрес поля, в которое процедура поместит полученное значение (рисунок 1.5).

```
.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 адресов 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

Модуль, содержащий процедуру 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 же 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 ва-

жен, 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, свое содержимое регистров и свои 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 осуществить *структуры*.

Объявление и описание структур в языке ассемблера. 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 символических 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 объявить с помощью символических имен.

Формат описания структуры:

```
<Имя структуры>     STRUCT
                <Описание полей>
<Имя структуры>     ENDS
```

где <Имя структуры> – идентификатор или символьное имя структуры,

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

Указанная 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 используют, когда в программе многократно повторяются сложные коллекции данных с единым строением, но с различными значениями полей. В этом случае, для выделения памяти под структуру 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 однотипные 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
```

Объявить 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'>
```

Организация рекурсивных процедур с использованием структуры. При создании рекурсивных процедур структуры использует 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. Рекурсивная процедура 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, процедура во время каждой активации должна иметь доступ к текущему значению числа **N** для расчета **N*(N-1)** и сохранять результат вычисления. Кроме

того, при очередном вызове процедура должна сохранять регистр базы **EBP** для доступа к параметрам, размещенным в стене, 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
```

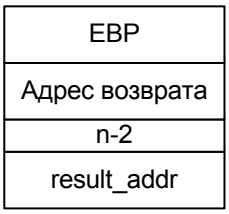


Рисунок 1.6 – Структура Frame

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).

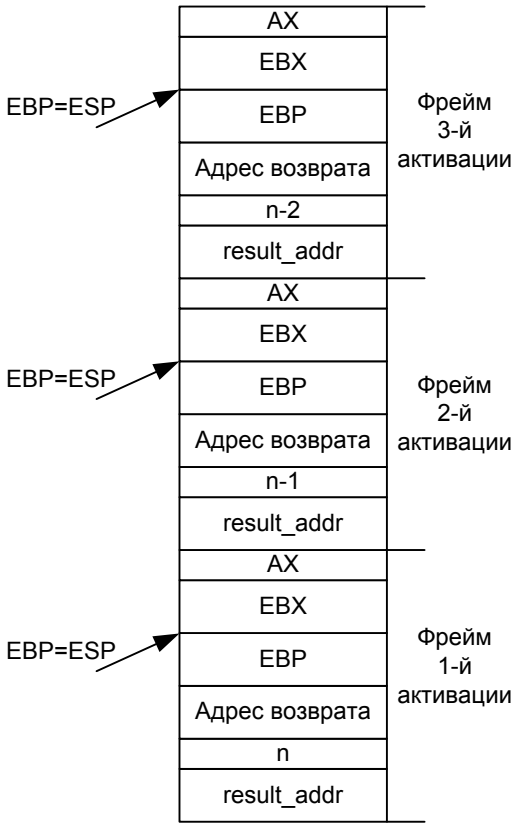


Рисунок 1.7 – Содержимое стека в процессе рекурсивного спуска

В начальный момент времени основная программа помещает в стек адрес, по которому следует поместить результат, затем копию числа n, затем при вызове процедуры туда автоматически запишется адрес возврата. Начав работу, процедура сохранит в стеке адрес базы и регистры EBX и AX. Затем в процессе следующего вызова запишет параметры для следующей активации, причем число уменьшится на 1.

После серии вызовов число станет равны 1 и полученный в процессе рекурсивного выхода результат окажется записанным в сегмент данных на указанное при первом вызове место.

1.5 Директивы описания процедур

При работе с процедурами в ассемблере необходимо соблюдать большое количество различных правил, например, правил передачи параметров, правил формирования внутренних имен, правил сохранения регистров и т.п. Использование директив описания позволяет существенно упростить эти операции, поскольку необходимые команды будут добавлены в текст программы автоматически.

1.5.1 Директива объявления прототипа процедуры

Директива предварительно объявляет процедуру, указывая ее имя, список параметров и основные описатели, т.е. создает «*прототип*» процедуры. Наличие прототипов процедур в тексте программы не является обязательным, но добавление в начало программы прототипов всех используемых процедур позволяет описывать уже объявленные процедуры в любом порядке, а не обязательно до первого вызова, как требует транслятор языка.

Формат директивы:

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

где <Тип вызова>:

far – дальний – межсегментный – используется, когда программа и процедура находятся в различных сегментах,

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

<Конвенция о связи> – имя конвенции о связях (см. раздел 2.1), которая определяет способ передачи параметров, формирование внутренних имен и т.п., по умолчанию используется конвенция, указанная в директиве **.MODEL**. Возможны следующие варианты конвенций:

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.

<Параметр> – имя параметра процедуры.

PTR – спецификатор указателя.

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

Прототип должен совпадать с описанием той же процедуры директивой PROC.

Примеры:

```
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
```

1.5.2 Директива заголовка процедуры

Директива заголовка процедуры позволяет объявить основные характеристики процедуры. Полный формат директивы:

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

Способы задания типа вызова, конвенции о связи, доступности и параметров описаны в предыдущем разделе.

<Список используемых регистров> – должен содержать перечень регистров, используемых в тексте процедуры, применяется для автоматического сохранения и восстановления этих регистров.

Примеры:

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

или с учетом умолчаний:

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

1.5.3 Директива описания локальных переменных

Локальные переменные – переменные, объявленные в процедуре (в C++ – автоматические переменные **auto**). Память для размещения локальных переменных отводится в стеке при вызове процедуры и должна освобождаться при ее завершении.

Директива описания локальных переменных используется для выделения памяти для указанных в ней локальных переменных процедуры. При завершении процедуры эта память освобождается автоматически. Директива помещается в процедуру сразу после **PROC**.

Формат директивы:

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

Пример:

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

В данном случае по директиве LOCAL выделяется 20 байт памяти, к которым можно обращаться, используя символическое имя ARRAY в качестве адреса первого байта. По завершении процедуры память будет автоматически освобождена.

1.5.4 Директива вызова процедуры

Директива вызова процедуры существенно упрощает вызов процедуры, поскольку автоматически размещает в стеке перечисленные в процедуре аргументы.

Формат директивы:

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

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

В качестве аргументов директивы INVOKE можно указать:

- целое значение, например: **27h**, **-128**;
- выражение целого типа, в том числе использующее операторы получения атрибутов полей данных (см. далее), например:

```
(10*20), TYPE mas, SYZEOF mas+2, OFFSET AR;
```

- имя регистра, например: **EAX, BH**;
- символический адрес переменной, например: **Ada1**, **var2_2**;
- адресное выражение, например: **4[EDI+EBX], Ada+24, ADDR AR**.

В адресные выражения и выражения целого типа могут включаться операторы получения атрибутов полей данных:

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 для согласования с формальным параметром.

Пример 1.7. Определение максимального из двух целых чисел.

В тексте программы выделены директивы и макрокоманды описания процедур, облегчающие написание программы.

```
        .586
        .MODEL  flat, stdcall
        OPTION CASEMAP:NONE
Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib
```

```
; прототип

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  ; вызов процедуры

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

; Тело процедуры

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
```

Вопросы для самоконтроля

1. Определите процедуру ассемблера. Чем процедура ассемблера отличается от подпрограмм языков Delphi Pascal и C++?

2. Как организуется передача управления в процедуру и обратно?

3. Какие способы доступа к данным вы знаете? Охарактеризуйте их. Почему часть способов доступа называют «передачей параметров»?

4. Что такое структура в ассемблере и для чего она используется?

5. Как в ассемблере реализуют рекурсивные процедуры?

6. Что такое прототип? Для чего он используется?

7. Какие описатели включены в описание процедуры?

8. Зачем используются локальные данные в процедурах? Как их описать?

9. Какие команды будут автоматически вставлены в программу примера 1.7?

2 СВЯЗЬ РАЗНОЯЗЫКОВЫХ МОДУЛЕЙ В WINDOWS

При разработке сложных программных систем на универсальных языках высокого уровня иногда возникает необходимость использования подпрограмм, написанных на других языках. Чтобы понять, как организуются программы, включающие модули на разных языках программирования, рассмотрим вызовы ассемблерных процедур из программ, создаваемых в средах программирования Turbo Delphi и Visual Studio 2008 (язык C++).

Основные проблемы, решаемые при создании программы из разноязыковых модулей:

- организация передачи и возврата управления;
- передача данных в подпрограмму на другом языке программирования:
  - через глобальные переменные – прямой доступ к данным вызывающей программы,
  - через размещение параметров или их адресов в стеке – косвенный доступ к данным вызывающей программы,
- обеспечение возврата результата функции;
- обеспечение корректного использования регистров процессора;
- осуществление совместной компоновки модулей.

2.1 Организации связи разноязыковых модулей. Конвенции о связи модулей

Корректное обращения к процедурам, написанным на языке ассемблера, из приложений, написанных на разных языках программирования, и обращение к подпрограммам на этих языках из процедур ассемблера предполагает соблюдение определенных правил. Эти правила определяют способ передачи параметров, закономерности формирования внутренних имен подпрограмм и глобальных данных и применяемую модель памяти.

Существует два варианта записи параметров в стек при передаче данных между основной программой и подпрограммой. Первый вариант предполагает, что параметры заносятся в стек в порядке их записи при описании подпрограммы. Этот вариант используется в программах на Паскале (рисунок 2.1, *а*). Другой, использующий обратный порядок записи параметров в стек, реализован в языке C (C++) (рисунок 2.1, *б*). Второе отличие

передачи параметров заключается в том, что в Паскале стек освобождает от параметров вызываемая подпрограмма, а в С (С++) это делает вызывающая программа.

а *б*

Рисунок 2.1 – Структура стека при передаче параметров:
а – по варианту, принятому в Паскале, *б* – по варианту, принятому в языке С (С++)

Правила, декларирующие способы передачи параметров при организации связи модулей, получили название «конвенции». Названия основных конвенций связано с именами двух основных универсальных языков программирования, для которых эти правила разрабатывались: Паскаль и С (С++). Остальные получили свои имена в соответствии с основными свойствами: стандартная Windows, защищенная и регистровая (таблица 2.1).

Таблица 2.1 – Конвенции по передаче параметров

	Конвенция	**Delphi**	**C++ Builder и Visual C++**	**Порядок параметров в стеке**	**Процедура, выполняющая очистку стека**	**Использование регистров**
1	Паскаль	pascal	_ _pascal	Слева направо	Вызываемая процедура	Нет
2	С	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

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

Стандартная и **Защищенная 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.

Согласно таблице 2.1 только при использовании конвенции C параметры из стека удаляет вызывающая программа. Это связано с тем, что только в языке C существует возможность описания подпрограмм с переменным количеством параметров. При реализации подобных процедур удаление параметров из стека целесообразно выполнять в вызывающей программе, которая «знает», сколько и каких параметров она передавала.

2.2 Правила формирования внутренних имен подпрограмм и глобальных данных

Компоновка программы из модулей на различных языках программирования выполняется с использованием таблиц внешних ссылок, содержащих сведения о подпрограммах, их вызовах и данных, описанных в одном месте и востребованных в другом. При этом в модулях на различных языках подпрограммы и данные должны называться одинаково, иначе компоновщик не сможет «разрешить внешние ссылки», сопоставляя вызов и вызываемую подпрограмму или обращение к внешним данным и их реальное местоположение. В таблице 2.2 перечислены особенности формирования внутренних имен компиляторами языков Паскаль и C (C++).

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

Так компилятор Delphi Pascal изменяет внутренние имена подпрограмм и глобальных переменных, заменяя строчные буквы на прописные. Это позволяет не учитывать регистр при написании программ на этом языке.

Компиляторы языка C (C++) изменяют имена всех глобальных («extern») переменных программы, добавляя перед ними символ подчеркивания «_». При этом строчные и прописные буквы в языке C различаются. Компиляторы языка C++ дописывают к именам функций специальные комбинации символов, отражающие используемый способ передачи параметров и их тип.

2.3 Сохранение регистров и модель памяти

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

Согласование типа вызова не требуется, поскольку во всех случаях используется модель памяти **`FLAT`**, для которой все вызовы ближние **`near`** и смещение имеет размер 32 бита.

Вопросы для самоконтроля

1. Что такое «конвенции о связи»? Зачем они нужны?
2. Назовите конвенции о связях. Что именно они фиксируют?
3. Каковы правила формирования внутренних имен в средах Turbo Delphi, Turbo C++ Builder?
4. Какие регистры следует обязательно сохранять?

3 РАЗРАБОТКА ПРИЛОЖЕНИЙ, ВКЛЮЧАЮЩИХ МОДУЛИ НА ЯЗЫКЕ АССЕМБЛЕРА, В СРЕДЕ TURBO DELPHI

3.1 Соглашения о передаче управления между подпрограммами

Основные соглашения, используемые Delphi Pascal при вызове подпрограмм, следующие:

1. Приложения, написанные на Delphi Pascal, используют модель памяти **FLAT**, а потому подстыковываемые к ним модули на языке ассемблера должны быть разработаны с использованием той же модели. Таким образом, все адреса в процедуре на ассемблере должны быть ближними и состоять только из смещения в сегменте (4 байта).

2. По правилам Delphi Pascal параметры передаются в вызываемую подпрограмму через стек, и там же размещаются локальные переменные подпрограмм. Вызов подпрограммы реализуется в следующей последовательности:

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

3. Подпрограммы Delphi Pascal имеют стандартно оформленные вход – *пролог* и выход – *эпилог*.

Пролог:

```
<Имя>  proc  near
       push  EBP       ; сохранить старое значение регистра 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>
```

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

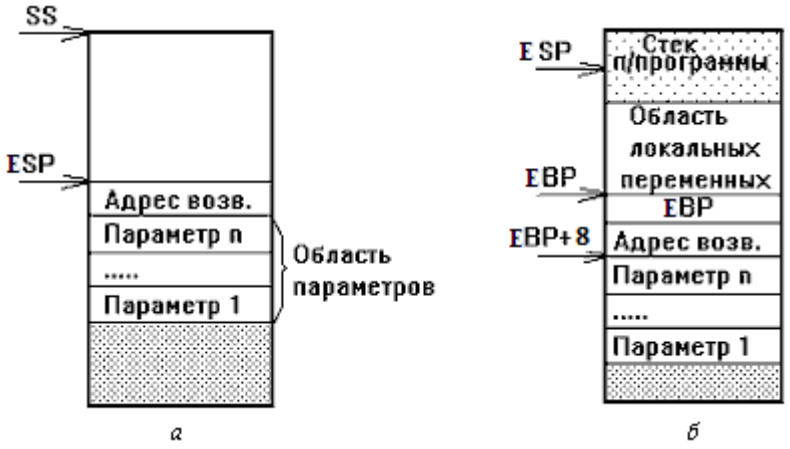


Рисунок 3.1 – Содержимое стека:
а – в момент передачи управления подпрограмме; *б* – во время работы подпрограммы

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

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

3.2 Соответствие форматов данных

Язык Delphi Pascal использует следующие внутренние представления данных (форматы).

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 – **ansichar (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[<длина>]** – символьный массив указанной при объявлении строки 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 соответствующие типы переменных (таблица 3.1).

Таблица 3.1 – Соответствие типов данных языков ассемблера и Delphi Pascal

№	Тип данных ассемблера	Размер памяти	Соответствующий тип данных языка 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 в языке ассемблера моделируют, указав тип и адрес первого элемента и используя их затем для адресации всей структуры.

3.3 Передача параметров по значению и ссылке. Возврат результатов функций

В языке Delphi Pascal параметры могут передаваться двумя способами: по значению и по ссылке. Параметры, передаваемые в подпрограмму по ссылке, описывают как **var** или **const**. Независимо от способа передачи параметра копии параметров или их адреса заносятся в стек.

При передаче по значению подпрограмме в стеке передаются копии значений параметров, и, соответственно, она не имеет возможности менять значения передаваемых параметров в вызывающей программе. При передаче по ссылке подпрограмма также в стеке

получает адреса передаваемых значений. Если параметр описан служебным словом **`var`**, то подпрограмма может не только читать значения, но и менять их.

Параметры-значения. Копии параметров-значений *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 очищается. Параметры размером 2 и 4 байта помещаются в стек в виде слова и двойного слова соответственно.

Параметры *строкового типа*, переданные по значению, независимо от их размера вызывающей программой в область параметров в стеке 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 значения множества. Первый бит младшего байта всегда соответствует базовому элементу множества с порядковым значением 0.

Параметры-переменные. Параметры-переменные или параметры, передаваемые по ссылке, передаются в процедуры одним и тем же способом – через 32-х разрядный указатель на их содержимое.

Так 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`** (рисунок 3.2).

Рисунок 3.2 – Состояние стека после записи параметров

Возвращение результатов процедур и функций в основную программу. Подпрограммы на языке Delphi Pascal возвращают результаты через параметры, передаваемые по ссылке (описанные с использованием **var**).

Конкретное место нахождения результата функции зависит от типа и размера возвращаемых данных. Результаты функций скалярных типов возвращаются 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 в стеке область.

Доступ к параметрам из процедур на языке ассемблера. 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**.

Очистка стека после выполнения подпрограммы. По соглашениям, принятым в Delphi 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 многоязыковой программы

Для совместной компоновки с программой на языке Delphi Pascal сегмент кодов ассемблерного модуля должен носить имя **code**, а сегмент данных – **data**. Оба сегмента должны быть объявлены общедоступными, т.е. **public**. При таком варианте описания в

процессе компоновки сегменты кодов и данных программы и модуля на ассемблере будут объединены, и появится возможность доступа к глобальным переменным основной программы на языке 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 как
   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.

По правилам языка Delphi Pascal подпрограммы не могут объявлять глобальные переменные. Соответственно данные ассемблерной части программы, даже будучи размещенными в общем сегменте данных с программой, написанной на языке Delphi Pascal, останутся для этой части программы «невидимыми» (локальными для подпрограммы на ассемблере). Кроме того, эти ассемблерные данные *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**.

Трансляция ассемблерных модулей. Модуль на ассемблере необходимо транслировать, используя 32-х разрядный ассемблер фирмы Borland (`tasm32`) или фирмы Microsoft (`ml`) и указав необходимые опции:

```
tasm32 /ml  <Имя исходного модуля>.asm
ml /c   <Имя исходного модуля>.asm
```

В среде RADAsm для того, чтобы выполнить ассемблирование с указанными опциями, необходимо в настройках проекта задать опции:

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

Если использовать текущие опции среды, заданные как:

```
3,O,$B\ML.EXE /c /coff /Cp /nologo /I"$I",2
```

то из-за присутствия опции **/coff** получим объектный код формата **MS Common Object File Format**, который компоноваться с модулями Delphi Pascal не будет.

Полученный объектный модуль <Имя исходного модуля>.**obj** следует переместить в папку проекта, чтобы при компоновке приложения он был найден и включен в программу.

Существует и более удобный вариант организации работы с ассемблерным модулем: среда Turbo Delphi содержит средства, которые позволяют транслировать модуль на языке ассемблера, не выходя из нее. Для этого в Turbo Delphi необходимо добавить внешний инструмент, назначив в качестве такого инструмента программу-ассемблер **ml.exe**.

Добавление внешнего инструмента реализуется следующим образом:

- выбрать пункт меню **Tools/Configure tools**. После щелчка мышкой по указанной строке меню на экране появится окно (рисунок 3.3), в котором нужно щелкнуть кнопку **Add;**
- в появившемся окне **Tool Properties** необходимо задать параметры добавляемой программы (рисунок 3.4).

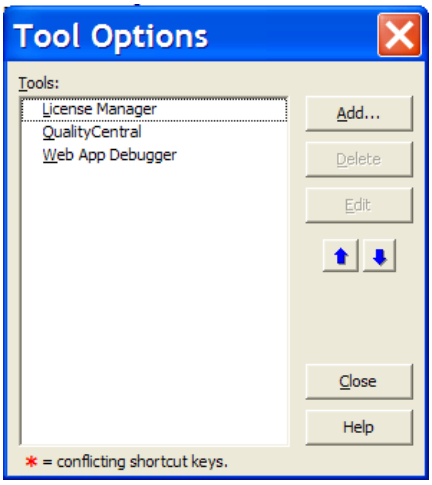


Рисунок 3.3 – Окно пункта меню **Tools/Configure tools**

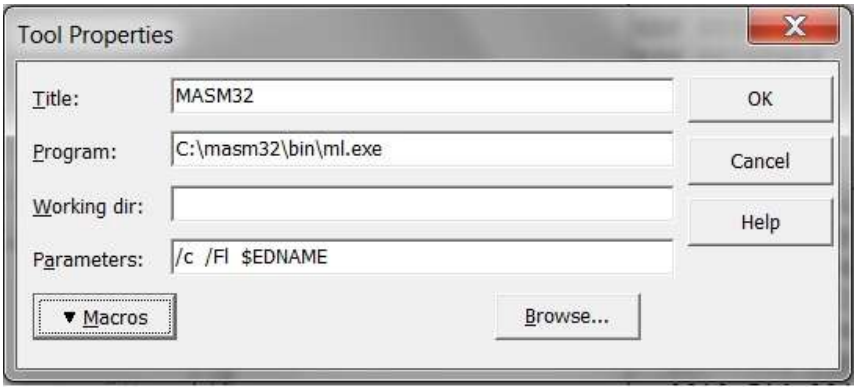


Рисунок 3.4 – Окно **Tool Properties** с заполненными полями

Теперь для трансляции модуля на языке ассемблера достаточно открыть файл с модулем посредством меню **File/Open**. Затем, используя пункт меню **Tools/Masm32,** выполнить процесс трансляции файла **<Имя>.asm**.

ВНИМАНИЕ! При выполнении трансляции вкладка с файлом должна быть активной («верхней» в многооконном редакторе среды Delphi).

При ассемблировании программы в среде Turbo Delphi ошибки трансляции удобнее всего искать в листинге программы. Для этого при настройке внешнего инструмента в строке параметров указан параметр /Fl, наличие которого указывает ассемблеру на необходимость создания файла с листингом **<Имя>.lst.** Этот файл появится после ассемблирования модуля в текущей директории проекта, и его можно открыть в среде Delphi с помощью пункта меню **File/Open** или блокнотом из Windows.

В случае наличия ошибок в файле листинга появится сообщение (рисунок 3.5).

После устранения ошибки текст программы следует сохранить с помощью пункта меню **File/Save**. Затем процесс трансляции следует повторить. Если трансляция прошла без ошибок, листинг содержит сообщение об успешной трансляции (рисунок 3.6), а в текущей директории появляется файл с расширением **<Имя>.Obj.**

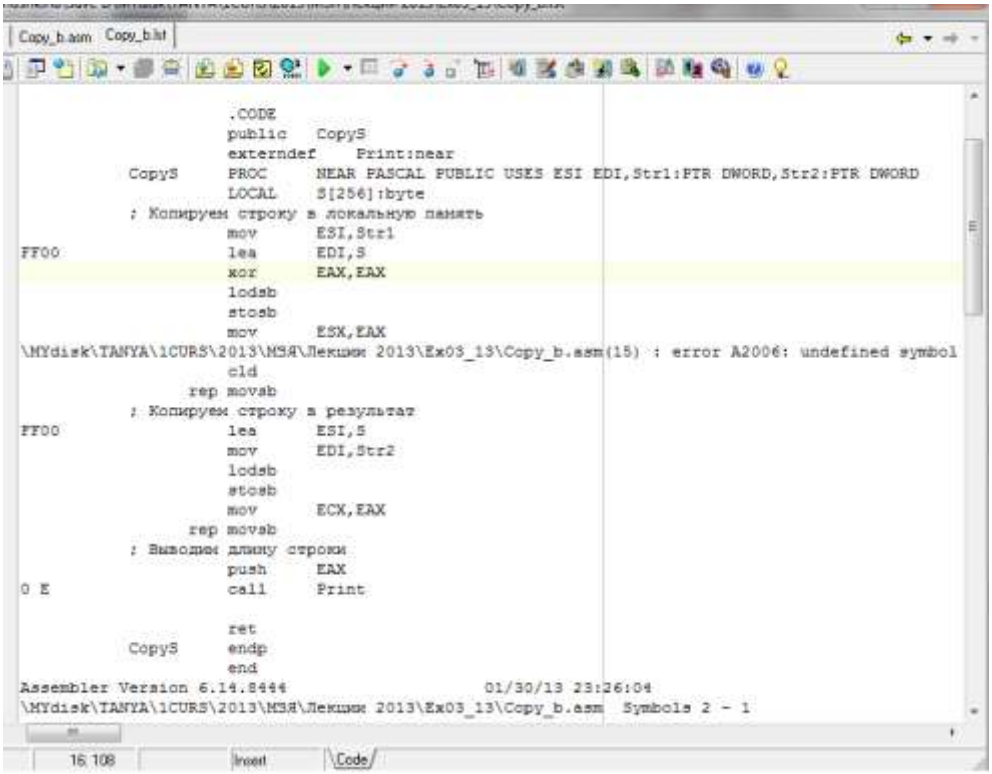


Рисунок 3.5 - Вид окна среды Turbo Delphi с открытым файлом ***.lst** с ошибкой

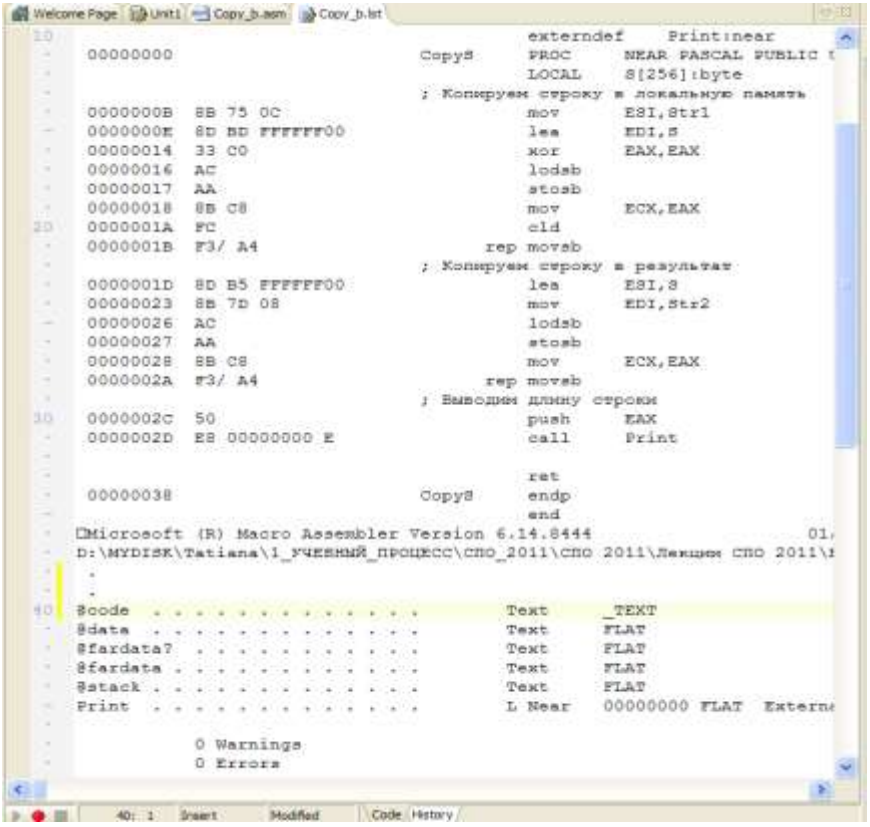


Рисунок 3.6 - Вид окна с открытым файлом **`*.lst`** без ошибок

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

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

Саму процедуру необходимо в программе на языке Delphi Pascal описать как внешнюю, указав используемую конвенцию, например:

```
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 подпрограммы на ассемблере в теле программы на языке Delphi Pascal независимо от конвенции осуществляется одинаково по имени, например:

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

При вызове будет автоматически организована передача данных во внешнюю процедуру в соответствии с указанной конвенцией.

3.5 Примеры

Пример 3.1. Процедура сложения целых чисел формата `integer`.

Рассмотрим реализации, использующие различные конвенции.

а) **Конвенция `pascal`**. Структура стека показана на рисунке 3.7.

```
      . 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
```

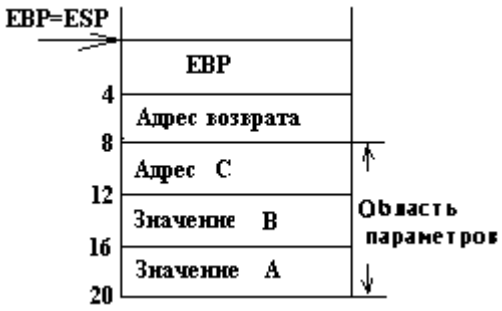


Рисунок 3.7 – Структура стека для конвенции **`pascal`**

б) **Конвенция cdecl**. Структура стека показана на рисунке 3.8.

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

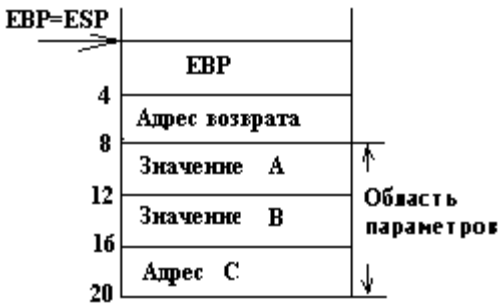


Рисунок 3.8– Структура стека для конвенции cdecl

```
      mov  EDX,[EBP+16]
      mov  [EDX],EAX
      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 3.9.

```
     .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 3.9 – Структура стека для конвенции stdcall

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

Особенности передачи строки, как параметра – значения.

Пример 3.2. Функции, принимающие строку-параметр и возвращающие строку-результат.

Для того, чтобы правильно организовать функции, получающую строку и возвращающую строку, рассмотрим функцию копирования строки-параметра в строку-результат:

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

В этом случае Delphi Pascal 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 3.10.

Рисунок 3.10 – Структура стека при работе функции Dell1

Дисассемблированный текст программы выглядит следующим образом:

```
        .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
@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
```

Из текста программы следует, что Delphi Pascal вначале копирует строку, помещая в стек копию параметра, так как строка передается по значению, а уже затем копирует копию строки по адресу результата.

3.6 Отладка программ на Delphi Pascal с модулями на ассемблере

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 точки останова **BreakPoints**, просмотреть значения переменных **Watches** и содержимое стека **Call 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** (рисунок 3.11).

После этого появится окно CPU режима отладки (рисунок 3.12). В этом окне представлена вся отладочная информация: текст программы с точки останова, содержимое стека, регистров и сегмента данных.

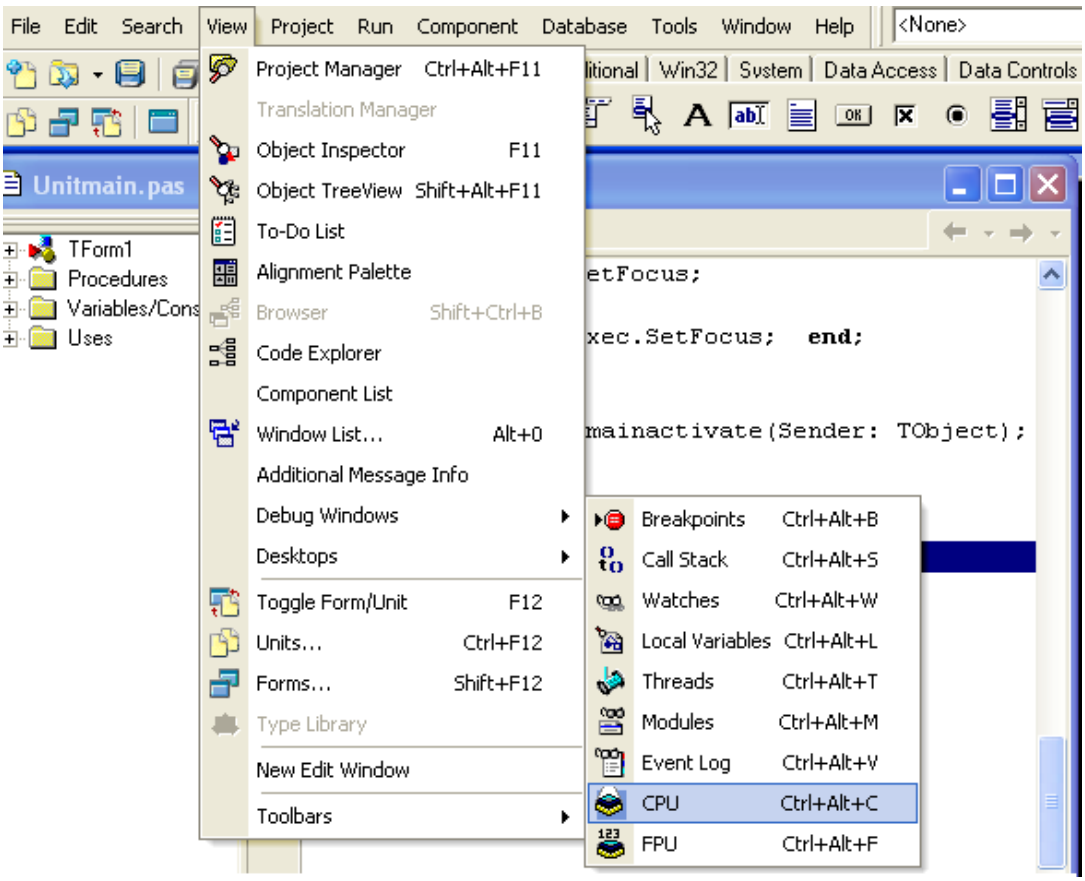


Рисунок 3.11 – Вид окна настройки режима отладки

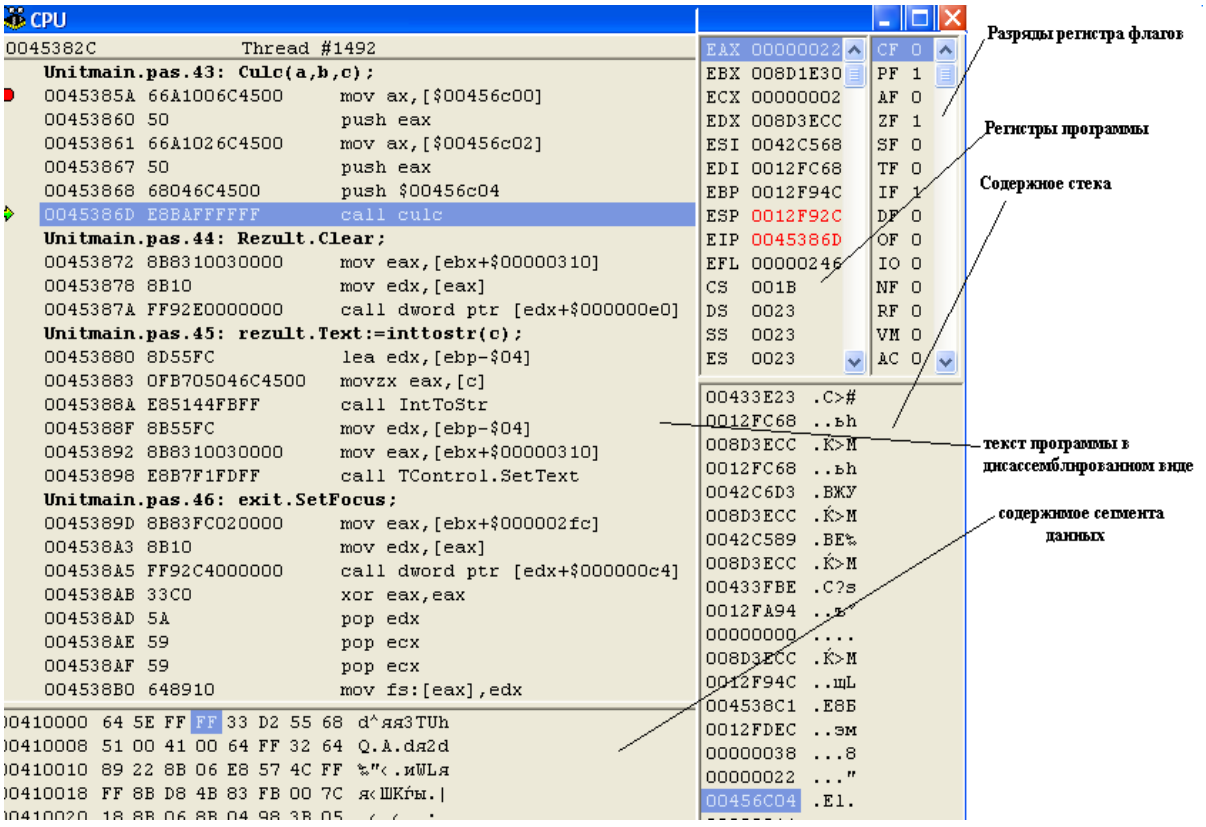
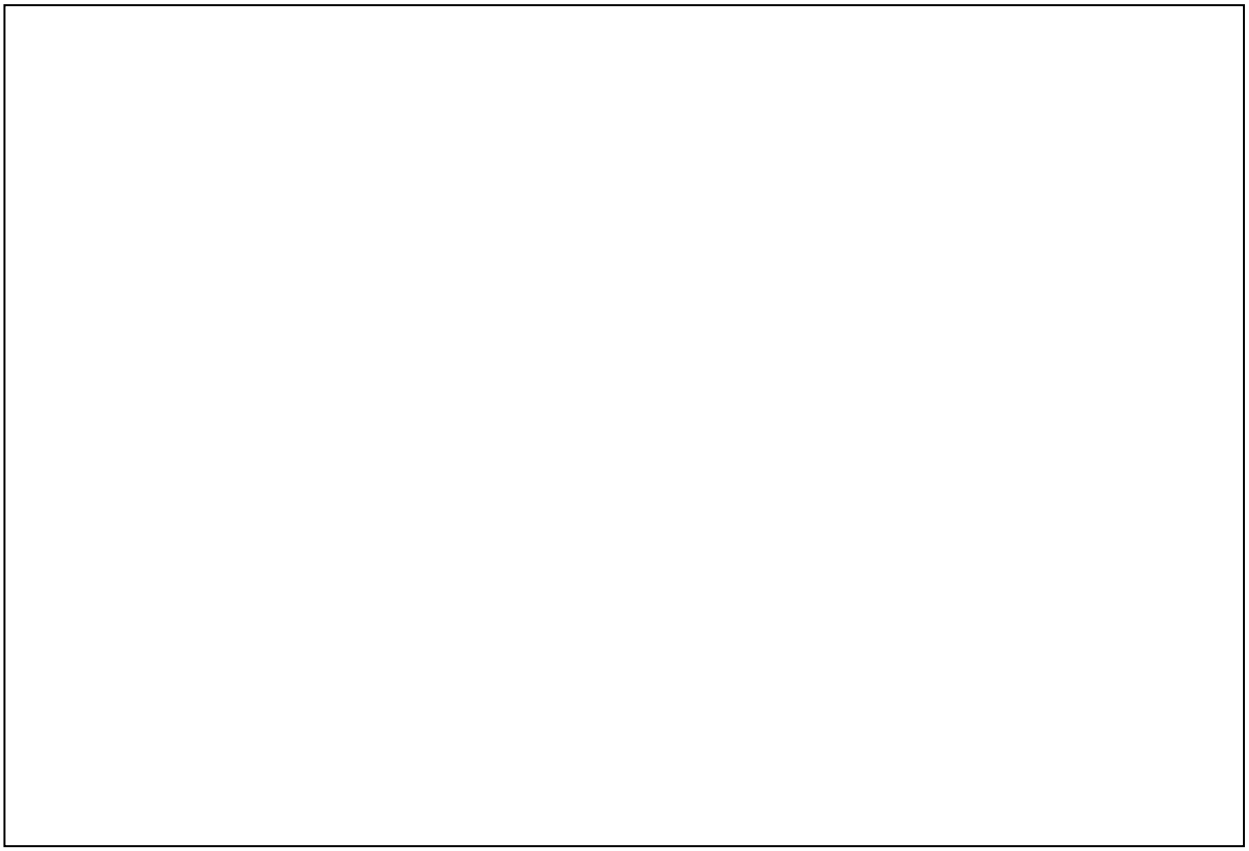


Рисунок 3.12 – Вид окна отладки Turbo Delphi в режиме CPU

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.



Вопросы для самоконтроля

1.Какие соглашения о передаче управления приняты в языке Delphi Pascal?

2. Какие способы передачи параметров использует Delphi Pascal?

3. Зачем необходимо знать внутренний формат представления данных в Delphi Pascal?

4. Как в среде Delphi выполнить ассемблирование модуля? Где искать сообщения об ошибках, найденных в процессе ассемблирования?

5. Как выполнить совместную компоновку модулей на языках ассемблера и Delphi Pascal?

6. Зачем необходимо рисовать стек в момент передачи управления из программы на языке Delphi Pascal в процедуру на ассемблере?

4 РАЗРАБОТКА ПРИЛОЖЕНИЙ, ВКЛЮЧАЮЩИХ МОДУЛИ НА ЯЗЫКЕ АССЕМБЛЕРА, В СРЕДЕ VISUAL STUDIO 2008

4.1 Передача параметров и возвращение результатов функции в C++

В отличие от языка Паскаль, в котором параметры могут передаваться в подпрограмму по значению и по ссылке, по правилам языка С (С++) параметры передаются в подпрограмму только *по значению*: если передается значение, то в стек помещается значение, если передается указатель или ссылка, 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 2.1, *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.

Значения, возвращаемые функцией, 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 С и С++: компилятор языка изменяет внутренние имена. Так перед всеми глобальными именами в языке С указывается символ подчеркивания, а в имена функций языка С++ 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 языках С или С++ 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 языка С в программах написанных на языке 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 С. Указанная в примере функция **ADD** будет иметь внутреннее имя **_ADD.**

4.2 Внутренний формат данных C++

Язык программирования C++ «под Windows» использует следующие типы целочисленных 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.

4.3 Объявление глобальных переменные в модуле и внешние имена

В отличие от Паскаля языки С и C++ позволяют 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 С должно начинаться со знака подчеркивания. Прочие модули, использующие данное имя, должны включать его описание как EXTERN (на ассемблере, на языках С или C++). Пример такого расширения списка переменных приведен в разделе 4.2.2 (случай *б*).

4.4 Компоновка модулей

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

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

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

Ассемблирование удобнее выполнять 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 4.1.

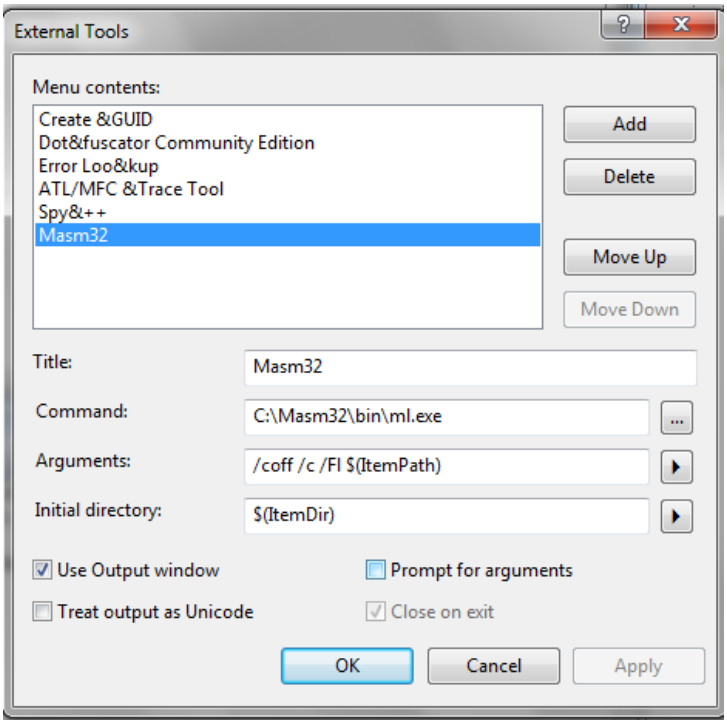


Рисунок 4.1 – Окно добавления инструмента с настройками

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 в окне редактора, в котором высвечивается модуль на ассемблере, ошибочные строки будут помечены зеленым (рисунок 4.2) .

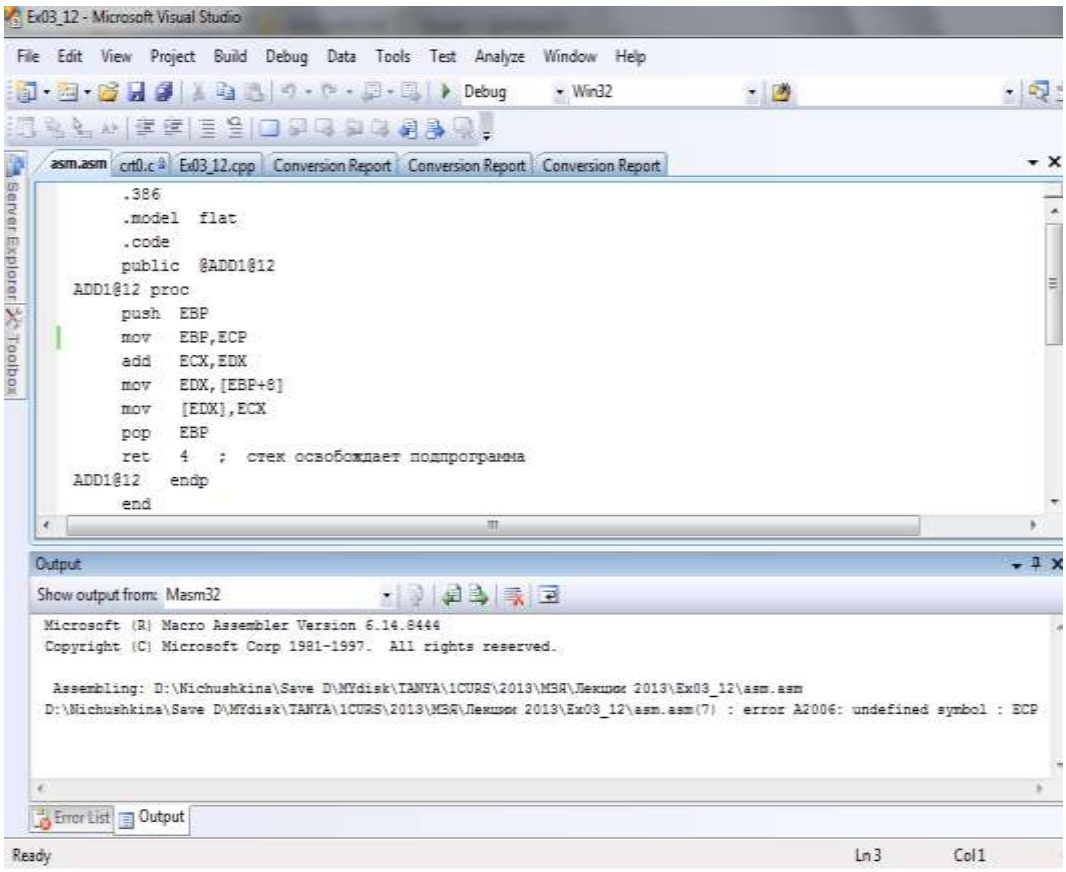


Рисунок 4.2 - Вид окна **Output** с ошибкой

Если компиляция прошла успешно, то в окне **Output** появится соответствующее сообщение (рисунок 4.3), а в папке проекта появится файл **<Имя файла>.obj**.

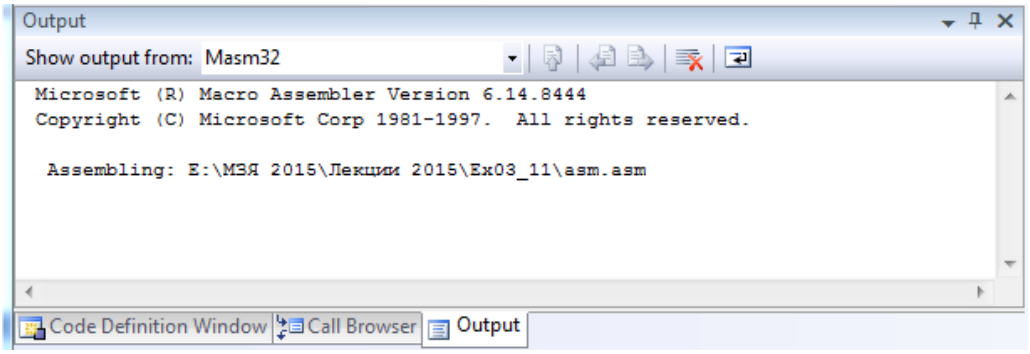


Рисунок 4.3 - Вид окна **Output** при успешной компиляции

Полученный в результате успешной компиляции объектный модуль **<Имя файла>.obj** необходимо подключить к приложению, используя пункт меню **Project/Add Existing Item** (рисунок 4.4).

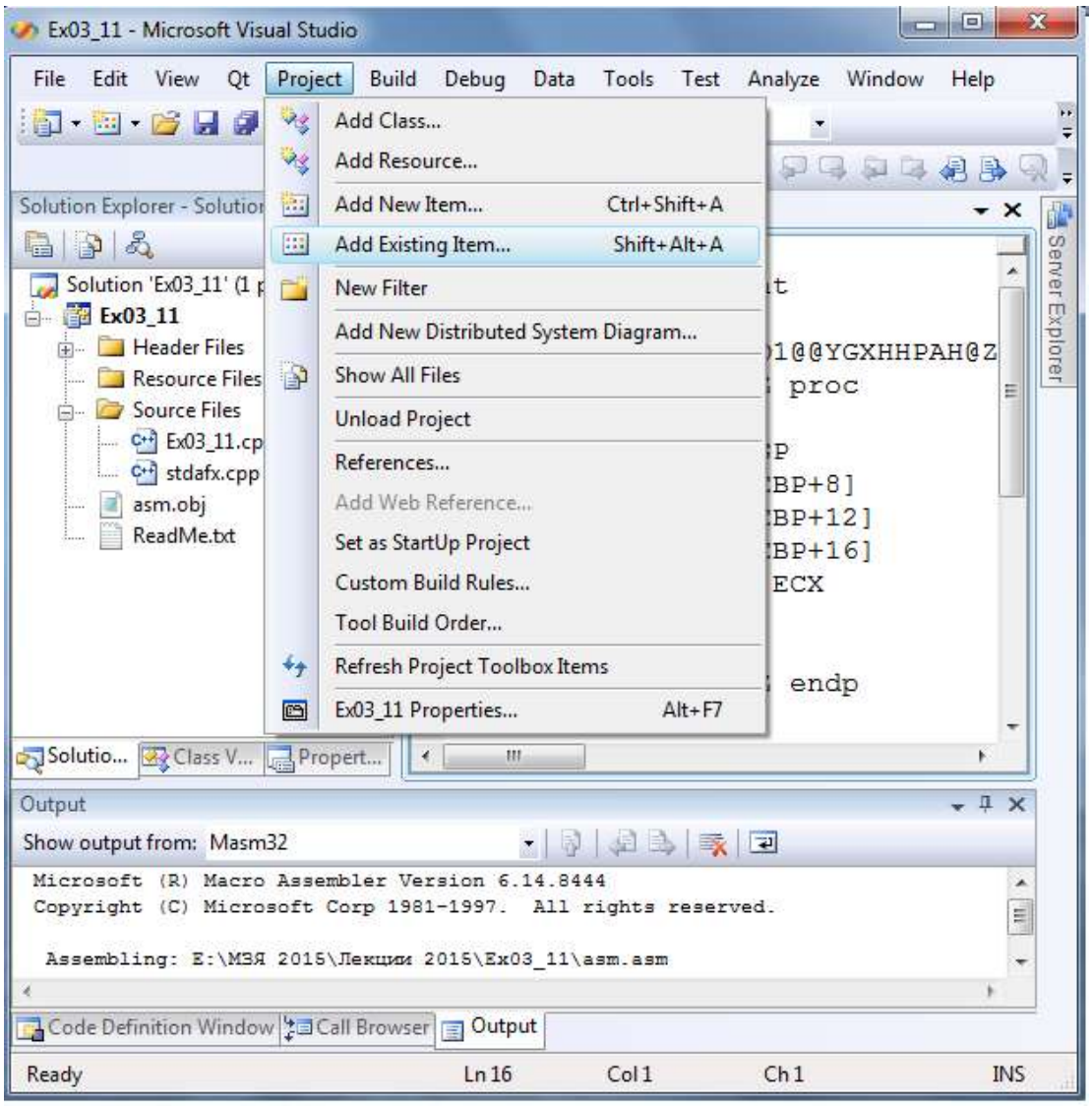


Рисунок 4.4 – Добавление существующего объектного модуля ассемблера

В появившемся окне выбираем объектный модуль **<Имя файла>.obj**, после чего он появится на вкладке **Solution Explorer** (рисунок 4.5). Теперь приступать к компоновке проекта.

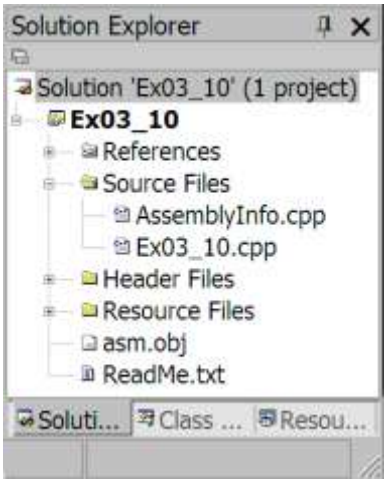


Рисунок 4.5 – Вкладка Solution Explorer с добавленным файлом **asm.obj**

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

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

4.5 Примеры

Пример 4.1. Процедура сложения целых чисел формата integer.

Рассмотрим реализации, использующие различные конвенции.

а) **Конвенция cdecl.** Структуру стека см. на рисунке 3.8.

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 языке 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      // измененное имя переменной d
?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 3.9.

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
```

4.6 Отладка программ с модулями на ассемблере

Для отладки программ на C++ с модулями на языке ассемблера 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** (рисунок 4.6).

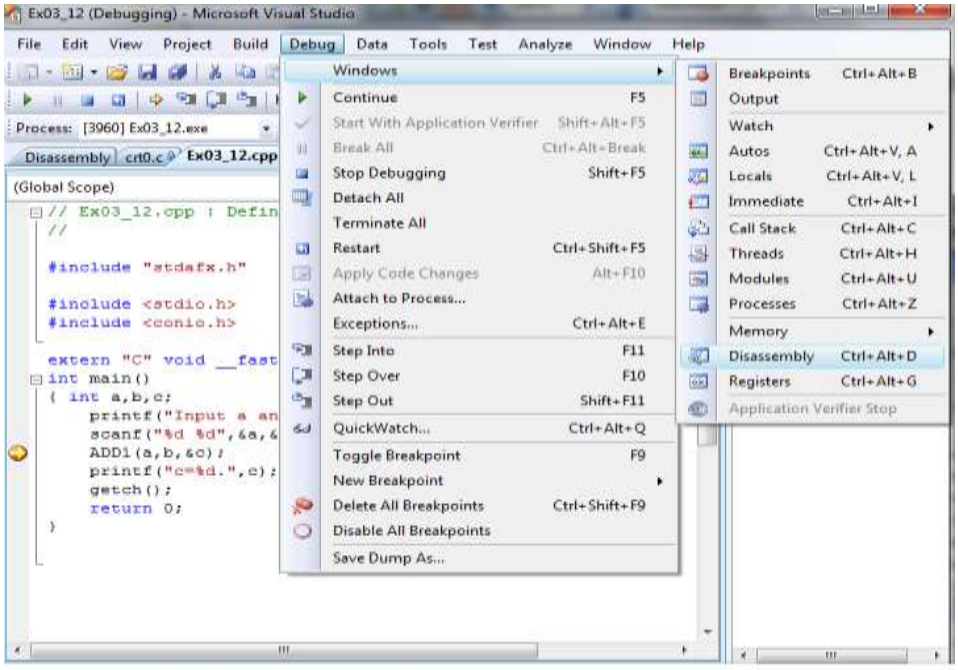


Рисунок 4.6 – Вид окна настройки режима отладки в среде Visual Studio 2008

Однако, в отличие от среды Delphi, в которой окно дисассемблера содержит всю необходимую информацию (содержимое регистров, содержимое памяти и т.д.) сразу, в момент открытия, появившееся окно содержит только текст модуля.

Для просмотра содержимого регистров или памяти необходимо использовать пункты меню **Debug/Windows/Registers** и **Debug/Windows/Memory.** После выбора этих пунктов, на экране в соответствующих окнах появится нужная информация (рисунок 4.7).

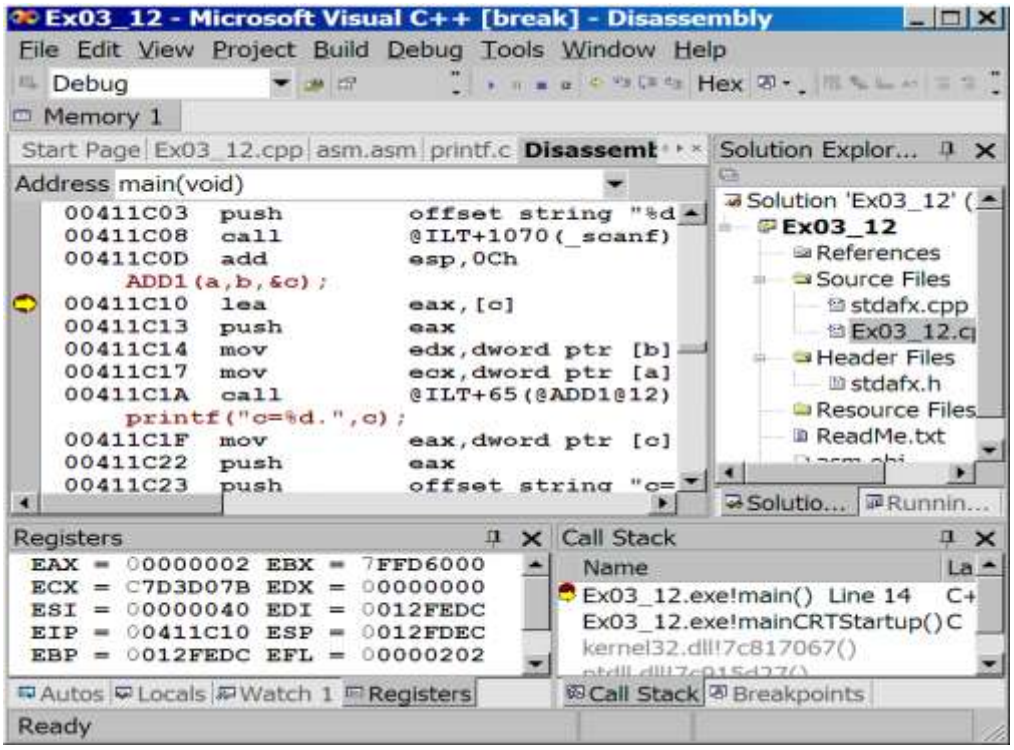


Рисунок 4.7 – Окно Дисассемблера Visual C++

Отладка программы ведется в покомандном режиме, как и в среде Turbo Delphi.

Вопросы для самоконтроля

1. Перечислите основные особенности передачи управления и параметров из программ, написанных на языке C++, в процедуры ассемблера.

2. Можно ли в подпрограммах на языке C++ объявлять глобальные переменные?

3. Как компоновать программу на языке C++, содержащую модуль на языке ассемблера в средеVisual Studio 2008?

4. Как определить внутреннее имя подпрограммы, написанной на языке C++?

5. Как отлаживаются программы, содержащие модули на языке ассемблера?

5 РАЗРАБОТКА ПРИЛОЖЕНИЙ, ВКЛЮЧАЮЩИХ МОДУЛИ НА ЯЗЫКЕ АССЕМБЛЕРА, В СРЕДЕ TURBO C++ BUILDER

5.1 Правила формирования внутренних имен

К именам функций при использовании компилятора C++ Builder 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.**

5.2 Особенности компоновки модулей

Подключаемый модуль на ассемблере необходимо предварительно транслировать. Для этого следует использовать 32-х разрядный компилятор **tasm**, который есть в библиотеках визуальной среды C++ Builder. Модуль на ассемблере необходимо транслировать с опцией **/mx**:

```
tasm /mx Add.asm
```

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);
```

Вызов процедуры выполняется по имени:

```
ADD1(a,b,c);
```

5.3 Примеры

Пример 4.3. Процедура сложения целых чисел формата `integer`.

Рассмотрим реализации, использующие различные конвенции.

а) **Конвенция `pascal`.** Структуру стека см. на рисунке 3.7.

```
      . 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  ; освобождение стека процедурой
@ADD1$qiipi   endp
      end
```

б) **Конвенция cdecl.** Структуру стека см. на рисунке 3.8.

```
      . 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 (рисунок 4.8), 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
```

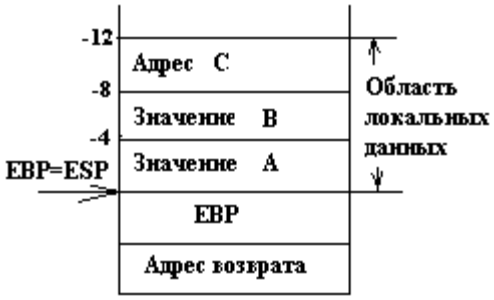


Рисунок 5.8 – Структура стека конвенции fastcall

Вопросы для самоконтроля

1. Совпадают ли правила передачи управления из программ, написанных на языке C++, в процедуры ассемблера и обратно в средах Visual Studio 2008 и C++ Builder и почему?

2. Как определить внутреннее имя подпрограммы, написанной на языке C++?

3. Как компоновать программу на языке C++, содержащую модуль на языке ассемблера, в Turbo C++ Builder?

Заключение

В учебном пособии приведены базовые сведения по организации межмодульных связей на языке ассемблера, а также особенности реализации модульного принципа в различных языках программирования. Рассмотрены системные соглашения о передаче управления и параметров в подпрограммы в конкретной операционной системе.

Изучение представленного материала позволит будущему программисту расширить и углубить свои знания в области разработки сложных программных систем.

Кроме того, рассмотренные в учебном пособии материалы помогут студентам выполнять лабораторные и контрольные работы, домашние задания и самостоятельную работу по дисциплине «Машинно-зависимые языки и основы компиляции».

Авторы надеются, что изучение данного пособия будет полезно всем читателям в их практической деятельности по проектированию сложного программного обеспечения.

Литература

1. Иванова Г.С. Программирование: Учебник для вузов. М.: Кнорус, 2013. 425 с.
2. Иванова Г.С., Ничушкина Т.Н., Самарев Р.С. Средства процедурного программирования Microsoft Visual C++ 2008: Учебное пособие. М.: Издательство МГТУ им. Н.Э. Баумана, 2012. 138 с.
3. Иванова Г.С., Ничушкина Т.Н. Объектно-ориентированное программирование: Учебник для вузов. М.: Издательство МГТУ им. Н.Э. Баумана, 2014. 455 с.
4. Иванова Г.С., Ничушкина Т.Н. Основы программирования на ассемблере IA-32 [Электронный ресурс]: электронное учебное издание: учебное пособие по дисциплине «Системное программное обеспечение». М.: МГТУ им. Н.Э. Баумана, 2010. Шифр Информрегистра: 0321001246/2010. – Режим доступа: http://e-learning.bmstu.ru/moodle/file.php/1/common_files/library/SPO/Basics/index.htm (дата обращения 16.03.2015).
5. Ирвин К. Язык ассемблера для процессоров Intel. М.: ИД «Вильямс», 2005.
6. Юров В.И. Assembler: Учебник для вузов. СПб.: Питер, 2008.
7. Пирогов В.Ю. Ассемблер. Учебный курс. СПб.: БХВ-Петербург, 2003.