ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра безопасности информационных систем

Вычислительный практикум

Учебное пособие

Составитель М.А. Попов

Самара
Издательство «Универс групп»
2007

Печатается по решению Редакционно-издательского совета
Самарского государственного университета

Вычислительный практикум : учеб. пособие / сост. М.А. Попов. – Самара : Изд-во «Универс групп», 2007. – 62 с.

Учебное пособие предназначено для студентов специальностей 090102.65 Компьютерная безопасность и 090103.65 Организация и технология защиты информации. Материалы учебного пособия позволяют выполнить девять лабораторных работ по дисциплине «Вычислительный практикум». К каждой лабораторной работе приведена необходимая теоретическая информация, описание хода выполнения работы и контрольные вопросы для подготовки к защите работы.

СОДЕРЖАНИЕ

Введение 4
Лабораторная работа №1 «Использование Ассемблера в программах на языке *Delphi*» 5
Лабораторная работа №2 «Ассемблер и *WinAPI*» 13
Лабораторная работа №3 «Ассемблер и динамическое подключение библиотек» 21
Лабораторная работа №4 «Определение версии *Windows*» 29
Лабораторная работа №5 «Дизассемблирование» 34
Лабораторная работа №6 «Определение тактовой частоты центрального процессора компьютера» 43
Лабораторная работа № 7 «Анализ исходного текста программы» 46
Лабораторная работа № 8 «Программа для загрузки файлов из сети Интернет» 50
Лабораторная работа № 9 «Возобновляемая загрузка файлов из сети Интернет» 57

Введение

В данном учебном пособии представлены методические указания для выполнения девять лабораторных работ по курсу «Вычислительный практикум».

Лабораторные работы выполняются в компьютерном классе в среде разработки программ *Borland Delphi*. Это позволяет знакомить студентов с современными технологиями программирования и, в то же время, в значительной мере использовать знания языка Паскаль, полученные на первом курсе обучения. В ходе выполнения данных лабораторных работ студенты знакомятся с языком Ассемблер, его особенностями и возможностями, а также с теми преимуществами, которые дает использование ассемблерных вставок в программах на языке высокого уровня.

Тематика лабораторных работ выбрана таким образом, чтобы каждая из них была посвящена решению задачи, довольно часто возникающей перед программистом. Выполнив все работы, студент получает набор тех «кирпичиков», из которых, как из конструктора, можно составить самостоятельное приложение.

В отличие от традиционной методики, когда студентам даются небольшие задачи (по вариантам или без таковых), и предлагается самостоятельно их решить, данные методические указания рассчитаны на несколько иную методику обучения. А именно: выдаваемые студентам методические указания содержат не только формулировку задачи, краткие теоретические сведения и контрольные вопросы, но и непосредственно решение задачи. В этом случае основной контроль знаний студентов проводится в форме «защиты» лабораторной работы, т. е. устной беседы студента с преподавателем.

Такая методика исключает ситуацию, когда несколько студентов группы решают задачу не только себе, но и всем остальным.

Лабораторная работа №1
«Использование Ассемблера в программах на языке *Delphi*»

1. Цель работы

Данная работа предусматривает обучение созданию и использованию в рамках программы на языке *Delphi* фрагмента, написанного на Ассемблере. Также целью работы является наглядное знакомство с архитектурой процессоров семейства x86 фирмы *Intel*.

2. Краткие теоретические сведения

Несмотря на то, что *Delphi* является объектно-ориентированным языком программирования высокого уровня, его разработчики предусмотрели возможность использования языка Ассемблер в рамках *Delphi* практически без ограничений.

Включение ассемблерных команд в тело программы осуществляется при помощи оператора *asm*. Порядок его использования следующий:

```
asm
    {здесь помещаются ассемблерные команды}
end;
```

Пример:

```
asm
    MOV    AX, X
    IMUL   Y
end;
```

Кроме того, программист имеет возможность использовать функции или процедуры, написанные целиком на Ассемблере. Например:

```
function MultXY(X, Y: Integer):Integer;

asm
    MOV    AX,X
    IMUL   Y
end;
```

Результат, возвращаемый функцией, помещается в регистре *AL* (8 бит), *AX* (16 бит) или *EAX* (32 бита). При работе функция может свободно изменять содержимое регистров *EAX*, *ECX* и *EDX*. При необходимости использования других регистров, их содержимое должно быть сохранено (обычно в стеке), а по завершении работы функции восстановлено. Например, вам нужно использовать регистр *EBX*, для этого:

```
asm
    ……
    PUSH EBX; сохраняем EBX в стеке

    {здесь выполняются нужные вам действия}

    POP EBX; восстанавливаем содержимое регистра EBX
……..
end;
```

В интегрированной среде разработки *Delphi* предусмотрены специализированные средства отладки, дающие программисту возможность не только отслеживать и изменять содержимое всех регистров процессора, а также сегмента данных и стека, но и видеть всю свою программу в виде ассемблерных инструкций, а также осуществлять их пошаговое выполнение.

Для этого необходимо использовать так называемый «брейкпоинт» (точка останова программы). Это специальная метка устанавливаемая в нужном месте программы. Как только выполнение программы «доходит» до строки, на которой стоит брейкпоинт, выполнение временно приостанавливается.

Теперь можно, выбрав в меню пункт *View — Debug Windows — CPU*, увидеть содержимое регистров, памяти и пр. Появившееся окно разделено на пять частей (рис. 1).

Самая большая из них, слева вверху, содержит адреса и команды, находящиеся по этим адресам в сегменте кода.

Внизу слева находится окно, показывающее содержимое сегмента данных – так называемый «дамп памяти».

Крайнее окно справа вверху показывает содержимое регистра флагов.

Между этим окном и окном сегмента кода показано содержимое всех регистров процессора.

Самое нижнее окно справа дает возможность отслеживать содержимое сегмента стека.

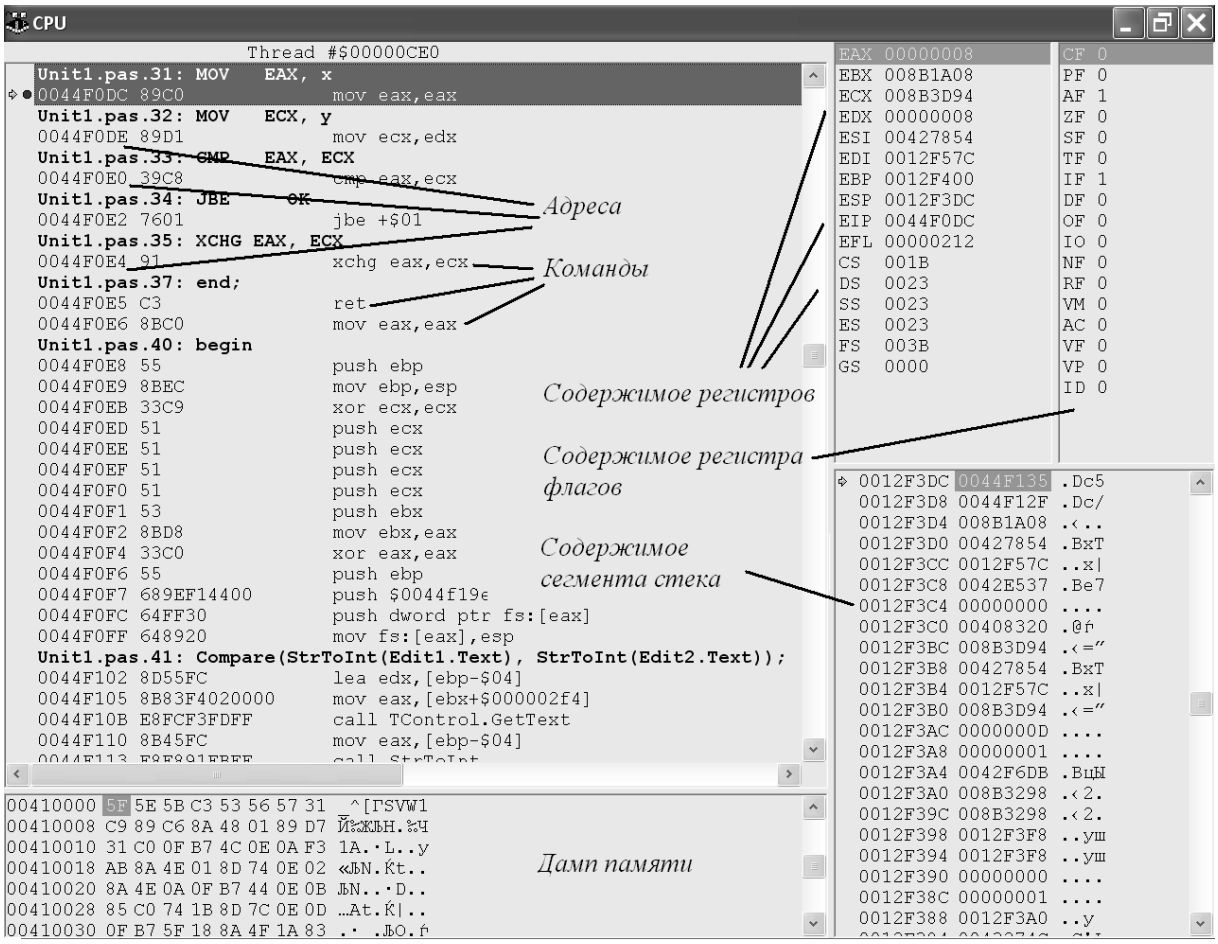


Рисунок 1. Окно отладчика

Для удобства отладки содержимое регистра, измененное в результате выполнения команды, отображается красным цветом. Содержимое любого из регистров может быть изменено, для этого на нужном регистре нужно кликнуть правой кнопкой мыши. Затем выбирается нужный вариант – инкремент, декремент, обнуление или просто ввод нового содержимого. Содержимое всех регистров представлено в шестнадцатеричном виде.

Внимание! Данное отладочное средство – чрезвычайно мощный инструмент! Вы получаете непосредственный доступ к регистрам микропроцессора! Следует соблюдать осторожность при изменении содержимого регистров! В ряде случаев можно необратимо нарушить работу своей программы, операционной системы и потерять любые несохраненные данные!

Остановимся кратко на особенностях современных процессоров применительно к данной работе.

В процессорах, начиная с 80386, используются так называемые расширенные регистры. Их разрядность составляет 32 бита. Например, регистр *EAX* – расширенный *AX*, то есть можно использовать его целиком (*MOV EAX, 0459AB1Dh*), можно – только младшие 16 бит (*MOV AX, AB1Dh*), и, наконец, можно по отделности применять две восьмибитные половины регистра *AX* (*MOV AL, 1Dh; MOV AH, ABh*). Назначение регистров не изменилось, т. е. *EAX* – это аккумулятор, *EBX* – базовый и т. д.

Кроме того, в современных процессорах не четыре, а шесть сегментных регистров: добавлены *FS* и *GS*.

Отдельно следует остановиться на регистре флагов. Он содержит 32 бита (используются не все биты). Каждый бит регистра флагов имеет определенное назначение. Некоторые из них определяют режим работы процессора, поэтому очень важны. Рассмотрим назначение основных бит регистра флагов.

OF (бит 11) – флаг переполнения. Равен 1, если результат сложения или вычитания выходит за допустимые пределы. В сочетании с флагом *CF* показывает длину результата умножения. Если старшая половина произведения не равна нулю, то *CF* и *OF* равны 1.

DF (бит 10) – флаг направления. Показывает направление обработки строки (цепочки) «слева направо» или «справа налево».

IF (бит 9) – флаг прерывания. При сбросе его в 0 микропроцессор не реагирует на прерывания от внешних устройств.

TF (бит 8) – флаг трассировки. При установке в 0 заставляет процессор выполнять программы в пошаговом режиме (используется при отладке программ).

SF (бит 7) – флаг знака. Имеет значение только при обработке чисел со знаком. Равен 1, если результат операции меньше нуля.

ZF (бит 6) – флаг нуля. Равен 1, если в результате операции получился нуль.

AF (бит 4) – дополнительный флаг переноса.

PF (бит 2) – флаг четности. Равен 1, если в результате операции получено число с четным числом единичных бит.

CF (бит 0) – флаг переноса. Равен 1, если произошел перенос единицы при сложении или заем единицы при вычитании.

Остальные биты регистра флагов либо зарезервированы для будущего использования, либо относятся к защищенному режиму работы микропроцессора и в данной работе не рассматриваются.

3. Порядок выполнения работы

Внимание! *Не удаляйте символы в квадратных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет.*

В начале данной работы создается программа, которая затем будет запущена в режиме отладки.

Программа выполняет несложную задачу (для примера): нужно найти большее из двух чисел. Однако значительная часть программы реализуется на Ассемблере.

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File—Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File—New—Application.* Это будет основная программа.

В разделе *Implementation* необходимо описать процедуру, которая и будет производить операцию сравнения.

```
procedure Compare (x, y: LongInt);

label OK;

asm
      MOV   EAX, x
      MOV   ECX, y
      CMP   EAX, ECX
      JBE     OK
      XCHG EAX, ECX
OK:
end;
```

В разделе *Var* нужно описать 2 переменные: *x* и *y*:

```
x, y: LongInt;
```

Теперь нужно поместить в окне основной программы два поля для ввода текста. Они будут использованы для ввода пользователем сравниваемых чисел. Для этого в панели компонентов в разделе *Standard* дважды кликнуть мышью на объекте *Edit*. В разделе *Properties* (окно *Object Inspector*) следует найти свойство *Text* и сделать его пустым. Затем следует аналогичным образом создать второе поле ввода. Рекомендуется размещать поля или одно под другим, или горизонтально на одном уровне для более логичного восприятия результата.

Затем нужно поместить в окне будущей программы кнопку. Для этого в панели компонентов в разделе *Standard* дважды кликнуть мышью на объекте *Button*. Местоположение и размеры кнопки можно изменить по вашему желанию. В окне *Object Inspector* выбрать объект *Button1*, затем в разделе *Properties* найти свойство *Caption* и присвоить ему значение «Сравнить!», это будет текстом надписи на кнопке. В разделе *Events* найти событие *OnClick* и поставить ему в соответствие процедуру *Button1Click*.

В окне редактирования текста программы появится заголовок процедуры. Введите следующие строки:

```
Compare(StrToInt(Edit1.Text), StrToInt(Edit2.Text));

asm
      MOV  x, EAX
      MOV  y, ECX
end;

Edit1.Text := IntToStr(x);
Edit2.Text := IntToStr(y);
```

Запустите программу на выполнение, нажав клавишу *F9*. Задавая различные числа (только целые), проверьте правильность работы программы.

Завершите работу программы.

Открыв исходный текст, найдите процедуру сравнения и установите брейкпоинт на строке:

```
MOV   EAX, x.
```

Для этого кликните мышью на сером поле слева рядом с нужной строкой. На нем появится красная точка, а строка будет выделена красным.

Запустите программу. Введите числа и нажмите кнопку “Сравнить!”. Кнопка *Delphi* на панели задач должна мигать синим цветом. Нажмите на нее. В окне для редактирования текста программы будет показан фрагмент текста со строкой, содержащей брейкпоинт. На красной точке слева будет стоять галочка. Это означает, что выполнение остановлено на этом месте.

Теперь можно, выбрав в меню пункт *View—Debug Windows—CPU*, увидеть содержимое регистров, памяти и пр. В окне сегмента кода вы увидите те ассемблерные команды, которые содержались в процедуре *Compare*. Нажав дважды клавишу *F8*, выполните следующие две команды. Теперь переключитесь в вашу программу (через панель задач). Запишите введенные вами числа и переведите их в шестнадцатиричную систему счисления. Теперь вернитесь в *Delphi* и посмотрите на значения в регистрах *EAX* и *ECX*. Они должны совпадать с введенными вами числами.

Нажимая клавишу *F8*, наблюдайте как выполняется программа: изменяются значения регистров, значения различных бит в регистре флагов и т. д.

Закройте окно *CPU*.

Завершите работу вашей программы.

4. Вопросы для подготовки к защите работы.

1. Что такое Ассемблер? Каковы его преимущества и недостатки?
2. Упрощенная структурная схема микропроцессора 8086.
3. Назначение всех программно-доступных регистров процессора.
4. Регистр флагов, его назначение. Назначение всех его бит.
5. Перевод чисел из десятичной в двоичную систему счисления и обратно.
6. Перевод чисел из десятичной в шестнадцатеричную систему счисления и обратно.
7. Особенности современных микропроцессоров (применительно к данной работе).
8. Что такое сегментная адресация?
9. Каким образом формируется адрес команды, выполняемой процессором (назвать регистры и привести фрагмент структурной схемы)?
10. Каким образом оформляется фрагмент на Ассемблере в *Delphi*?
11. Что такое «брейкпоинт»? Каким образом он устанавливается (уметь показать на компьютере)?

12. Окно *CPU*. Назначение всех его частей. Как его вызвать (уметь показать на компьютере)?

13. Приведите пример ассемблерной команды.

14. Присвойте регистру *EBX* значение *12345678h*, используя стек.

Лабораторная работа №2
«Ассемблер и *WinAPI*»

1. Цель работы

Целью данной работы является изучение способов взаимодействия программы на Ассемблере и функций *WinAPI*. Также в работе рассматриваются способы передачи параметров в вызываемую функцию.

2. Теоретическая часть

В данной работе будут использоваться несколько функций *WinAPI* (*Windows Application Program Interface* – набор стандартных процедур и функций, предоставляемых операционной системой *Windows* для работы прикладных программ).

Как правило, в любую функцию или процедуру передаются параметры – те числовые величины, строки, указатели и пр., которые используются в теле функции для вычисления результата или выполнения каких-либо действий.

Рассмотрим два способа передачи параметров – через регистры и через стек.

Передача параметров через регистры предусматривает, что перед вызовом функции ее параметры записываются в регистры микропроцессора. Программисту на *Delphi* вряд ли приходилось заботиться о способе передачи параметров, так как все эти операции происходят без его ведома. Рассмотрим пример.

Пусть описана функция:

function Func0 (par1: LongInt; par2: LongInt; par3: LongInt): LongInt;

В нужном месте программы производится вызов этой функции, например:

i := Func0 (809, 46, 478);

На самом деле при компиляции генерируется машинный код, соответствующий следующему фрагменту на Ассемблере:

MOV EAX, 809

MOV EBX, 46
MOV ECX, 478
CALL Func0
MOV i, EAX

В теле функции значения, переданные в регистрах, будут использованы в соответствии с алгоритмом работы функции. Как видно из рассмотренного примера, присвоение идет «слева направо», т.е. первый параметр будет записан в *EAX*, второй в *EBX* и т.д. Результат работы функции будет возвращен в регистре *EAX*.

Преимуществом регистрового способа передачи параметров является некоторое ускорение работы программы, сокращение ее размера и экономия памяти. Однако применение его ограничено в силу очевидных соображений: параметров может быть много, а регистров общего назначения всего четыре. Кроме того, разрядность регистров в современных процессорах составляет 32 бита, что не удобно, например, для передачи дальнего указателя (сегмент 16 разрядов и смещение 32 разряда).

Поэтому обычно, в т.ч. и при вызове функций *WinAPI*, используется передача параметров через стек. Ее суть состоит в том, что перед вызовом функции параметры записываются в стек, а затем извлекаются оттуда уже в теле функции. Предположим, что в рассмотренном выше примере используется не регистровая, а стековая передача параметров. Тогда эквивалентный ассемблерный код будет другим:

PUSH 809
PUSH 46
PUSH 478
CALL Func0
MOV i, EAX

Передача параметров через стек позволяет передавать практически любое число параметров и, самое главное, установить единое соглашение о передаче параметров. Например, в *DLL* могут использоваться функции, написанные на любом языке высокого уровня, и необходимо обеспечить работу всех этих *DLL* совместно с программами, которые также могли писаться на любом языке программирования. Стековая передача параметров позволяет ввести такую унификацию. Стандартный вызов (*stdcall*) преду-

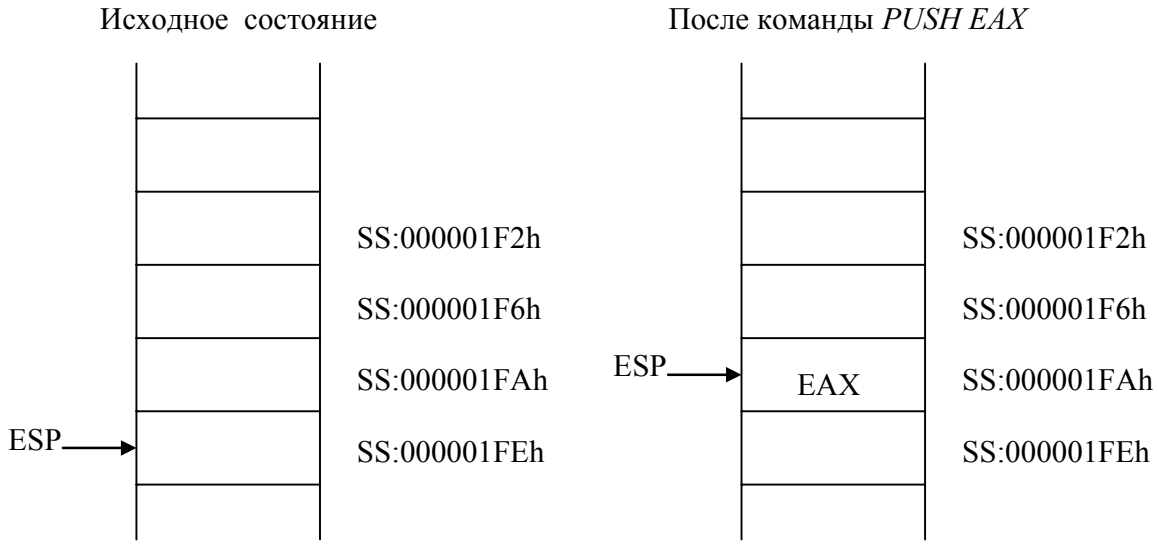
сматривает запись параметров в стек «справа налево» (первый параметр записывается в стек последним, последний – первым) и, как правило, возврат результата в регистре *EAX*.

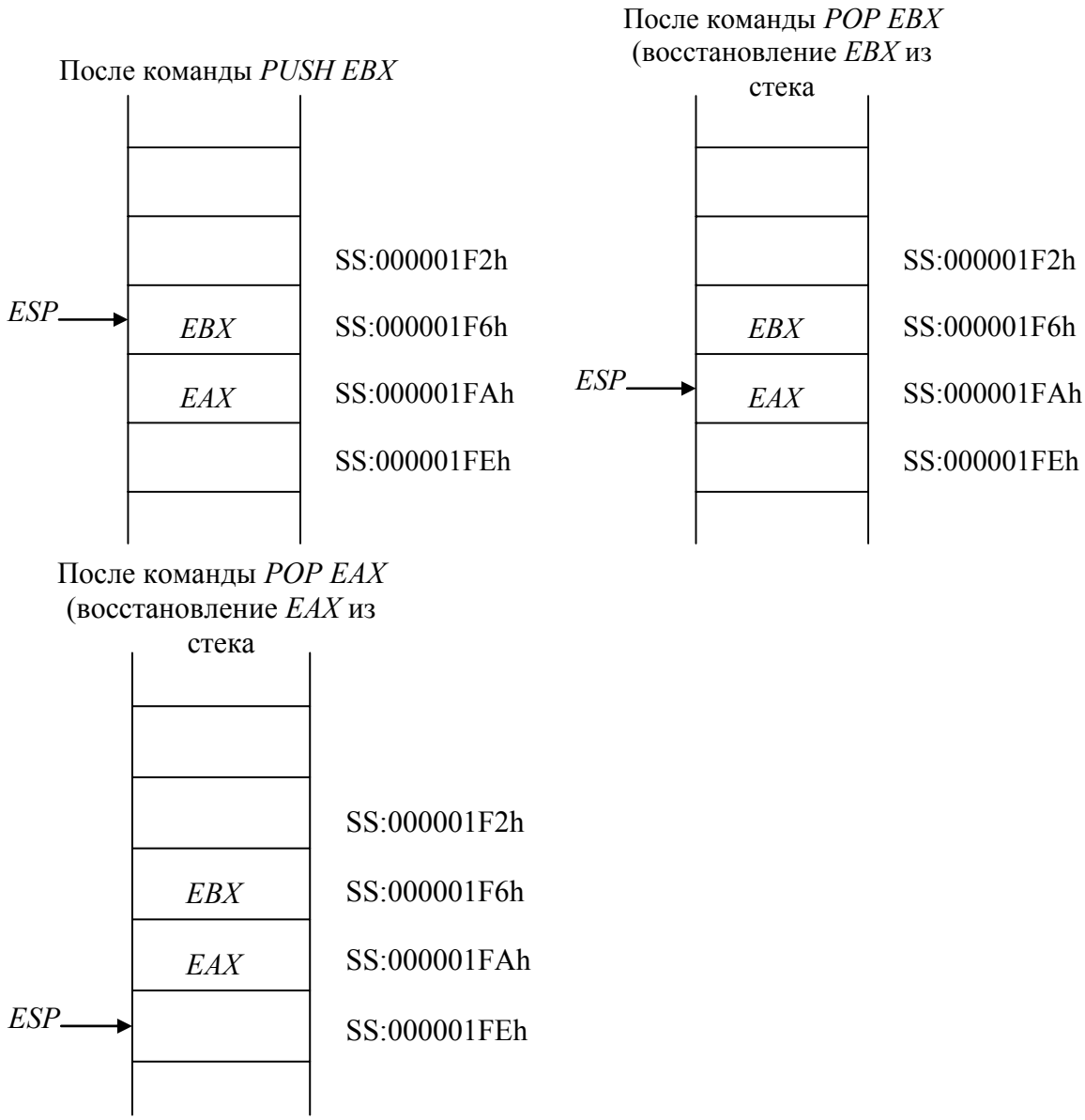
Остановимся более подробно на работе стека. Фактически стек представляет собой область памяти для временного хранения данных. Однако его работа характеризуется рядом особенностей, которые нужно пояснить на примере.

Предположим, что нужно сохранить в стеке значения регистров *EAX*, *EBX*, а затем восстановить их.

SS:ESP указывает на вершину стека. Первые два байта (два байта принято называть «словом») информации запоминается в ячейке с наибольшим адресом, следующие – в ячейке с адресом на 2 меньше и т.д. Регистр *ESP* всегда указывает на слово, помещенное в стек последним. Следовательно, команда *PUSH* вычитает 2 из значения указателя стека (*ESP*), а затем посылает слово в стек. Действуя обратным образом, команда *POP* присваивает своему операнду слово из стека и увеличивает значение регистра *ESP* на 2.

Выше была рассмотрена ситуация, когда помещается в стек и извлекается слово (т.е. 2 байта или 16 бит). Регистры *EAX*, *EBX* и пр. содержат 32 бита, т.е. 4 байта или 2 слова. Этот случай представлен на рисунке.





В данной работе используются функции *WinAPI*: *MessageBox* и *ExitProcess*.

Функция *MessageBox* предназначена для вывода на экран окна с сообщением и имеет следующие параметры:

MessageBox (HWnd: Integer; Text, Title: PChar; Flags: Integer): Integer;

HWnd – идентификатор родительского окна; если такого окна нет, равен нулю.
Text – указатель на строку, являющуюся заголовком окна.
Title – указатель на текст, который должен быть размещен в окне.
Flags – число, определяющее вид окна сообщения и количество кнопок.

Возвращаемое функцией число (регистр *EAX*) дает информацию о том, какая из кнопок была нажата.

Процедура *ExitProcess* служит для завершения работы программы. Она имеет единственный параметр – код завершения программы (тип *Integer*).

3. Выполнение работы

Для правильной работы большинства программ (в т. ч. *Delphi*) под *Windows NT* необходимо предварительно настроить переменные среды. На рабочем столе найдите значок «Мой компьютер» и кликните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным *Temp* и *Tmp* значение *%UserProfile%\Temp*.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет.*

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File—Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File—New—Other*. Затем в появившемся окне выбрать *DLL Wizard*. Создается заготовка *DLL*.

Введите текст программы:

```
library asmdll;

uses
    SysUtils,
    Classes;

function MessageBox(HWnd: Integer; Text, Title: PChar; Flags: Integer):
Integer; stdcall; external 'user32.dll' name 'MessageBoxA';

procedure ShowMessage(); stdcall;

label
start, text, title;
```

```
asm
        JMP START
TEXT:
        DB 'Библиотека ASMDLL.DLL успешно загружена',0
TITLE:
        DB  'Процедура ShowMessage',0
START:
        PUSH 0
        PUSH OFFSET(TITLE)
        PUSH OFFSET(TEXT)
        PUSH 0
        CALL MESSAGEBOX
end;

exports Showmessage;

begin
end.
```

Как видно из вышеприведенного исходного текста *DLL*, будет использоваться статический метод загрузки как в самой *DLL*, так и в основной программе. Это сделано для упрощения.

Интересным моментом является то, что создаваемая вами *DLL asmdll.dll* будет использовать, в свою очередь, еще одну *DLL user32.dll* (это одна из составных частей ядра *Windows*), в которой находится функция *MessageBox*. Она должна быть объявлена как внешняя уже известным вам способом. Нужно лишь добавить зарезервированное слово *name* (см. исходный текст), которое указывает то имя функции, под которым она находится в *DLL*. В данной работе это имя и имя, под которым функция объявлена в программе, не совпадают.

Компилировать *DLL*, нажав *Ctrl–F9*. Выполнить команду *File–Close*.

Выполнить команду *File–New–Application.* Это будет основная программа.

В разделе *Implementation* поместите строки:

```
function MessageBox(HWnd: Integer; Text, Title: PChar; Flags: Integer):
 Integer; stdcall; external 'user32.dll' name 'MessageBoxA';
```

```
function ExitProcess(Code: Integer): Integer; stdcall; external 'kernel32.dll' name 'ExitProcess';
procedure ShowMessage; stdcall; external 'asmdll.dll' name 'ShowMessage';
```

Таким образом, вы объявили внешние функции, используемые вашей программой. Метод загрузки *DLL*, как уже говорилось, статический.

Вся основная программа будет написана на Ассемблере, поэтому целесообразно для упрощения вызова функций *WinAPI* разместить строковые константы в сегменте кода. В таком случае для реализации доступа к ним нужно знать их адрес, что обеспечивается при помощи меток, которые необходимо объявить:

```
label
text, title, start;
```

Введите текст основной программы:

```
begin

asm
    JMP START
TEXT:
    DB 'Будем загружать DLL статическим методом',0
TITLE:
    DB 'Программа для Windows на Ассемблере',0
START:
    PUSH 0
    PUSH OFFSET(TITLE)
    PUSH OFFSET(TEXT)
    PUSH 0
    CALL MessageBox
    CALL ShowMessage
    PUSH 0
    CALL ExitProcess
end;

end.
```

Запустите программу на выполнение, нажав клавишу *F9*. Если программа работает неправильно, проверьте наличие *DLL* по указанным вами маршрутам.

4. Контрольные вопросы для подготовки к защите работы

1. Что такое *WinAPI*?
2. Что такое стек?
3. Назовите все команды работы со стеком.
4. Как работает стек? Привести рисунок.
5. Что такое регистровая передача параметров? Приведите пример.
6. Что такое стековая передача параметров? Приведите пример.
7. Зачем нужна унификация способов передачи параметров? Что подразумевает стандартный вид передачи параметров и вызова функций *StdCall*?
8. Зачем при описании внешних функций используется зарезервированное слово *name*?
9. Функция *MessageBox*. Ее назначение, параметры.
10. Процедура *ExitProcess*. Ее назначение, параметры.
11. В каком из регистров обычно возвращается результат функции?
12. Функция *MessageBox* после завершения своей работы возвращает результат. Используя отладочные возможности среды *Delphi*, выяснить, чему он равен.
13. Используя отладочные возможности среды *Delphi*, продемонстрировать наглядно механизм работы стека при выполнении основной программы, созданной в данной работе.

Лабораторная работа №3
«Ассемблер и динамическое подключение библиотек»

1. Цель работы

Целью данной работы является изучение особенностей применения динамического метода загрузки *DLL* в программах на Ассемблере. Также в работе рассматривается ряд новых команд, например, сравнение и условный переход.

2. Краткие теоретические сведения

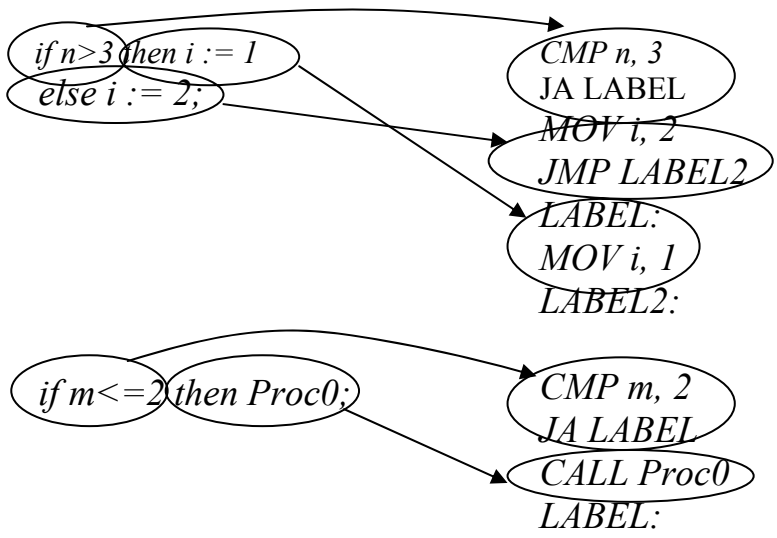
В данной работе будут использоваться несколько функций *WinAPI*, с которыми вы уже знакомы: *MessageBox, ExitProcess, LoadLibrary, GetProcAddress* и *FreeLibrary*.

Как вы знаете из предыдущей работы, функции *WinAPI* используют передачу параметров через стек и возвращают результат своей работы в регистре *EAX*.

В данной работе вы будете пользоваться несколькими новыми командами языка Ассемблер, а именно командами сравнения и условного перехода. Очевидно, что любая реальная программа обязательно использует операции сравнения, хотя об этом начинающий программист может и не догадываться.

Рассмотрим пример, показывающий соответствие операторов языка *Delphi* и ассемблерных команд. Следует подчеркнуть, что в исполняемой программе в любом случае будут стоять машинные коды (которые однозначно соответствуют ассемблерным командам), однако если вы изначально писали свою программу на Ассемблере, то компилироваться и выполняться она будет быстрее.

В первом примере команды сравнения (*CMP n, 3*) и условного перехода (*JA LABEL*) эквивалентны *if n>3*. Команда условного перехода заставляет процессор продолжить выполнение программы с того места, где стоит метка *LABEL* только в том случае, если *n* больше 3. Это видно из расшифровки команды *JA*: ***J**ump, if **A**bove* – «перейти, если больше». Аналогично можно понять смысл и других команд: *JNA – **J**ump, if **N**ot **A**bove* («перейти, если не больше»), *JE – **J**ump, if **E**qual* («перейти, если равно»).



Соответственно, если условие не выполняется (*n<=3*), то перехода не будет, и выполнение продолжится с команды следующей после *JA LABEL*. Таким образом реализуется *else*.

Во втором примере логика работы несколько другая: несмотря на то, что варианте на языке *Delphi* используется сравнение «меньше или равно», в ассемблерном эквиваленте по-прежнему используется команда *JA LABEL*. Дело в том, что здесь нет *else*, и в случае, когда условие не выполняется, нужно просто продолжить работу программы (не будет вызова *Proc0*). Если бы мы стали использовать более подходящую на первый взгляд инструкцию *JNA*, то затем пришлось бы применять команду безусловного перехода *JMP*.

В предыдущих работах вы использовали только прямой вызов функций и процедур:

CALL <метка>

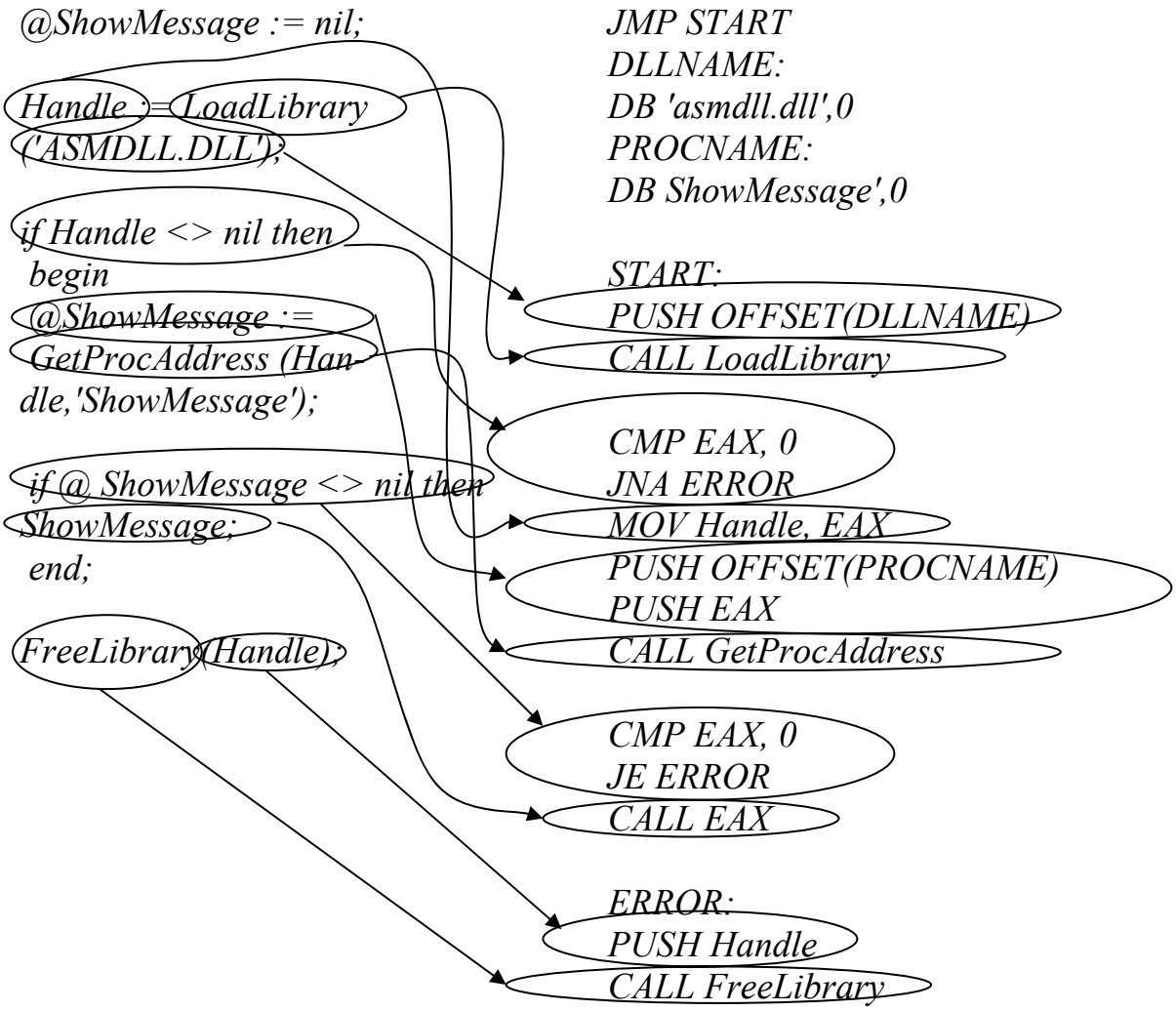
Очевидно, что для написания любой сколько-нибудь сложной программы этого оказывается недостаточно. В данной программе вы будете пользоваться косвенным вызовом процедуры. Его отличие заключается в том, что адрес вызываемой процедуры может быть не задан жестко еще на этапе компиляции (так происходит при использовании прямого вызова – метка при компиляции переводится в адрес, который указывается непосредственно в теле машинного эквивалента команды *CALL* и **не может быть изменен** на этапе выполнения программы). Вместо адреса задается регистр общего назначения, где хранится этот адрес, например:

CALL EAX

Присваивая регистру *EAX* значение, соответствующее тому адресу в памяти, где находится процедура, а затем используя косвенный вызов, можно вызывать процедуры, конкретный адрес которых на момент написания программы вам не известен.

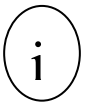
Следует особо отметить, что косвенный метод вызова требует от программиста дополнительной осторожности: если в регистре, который используется командой *CALL*, указан неверный адрес вместо адреса нужной вам процедуры, то результатом будет выполнение неизвестных инструкций, которые по этому неверному адресу находятся. Работа вашей программы (и даже операционной системы) в этом случае будет необратимо нарушена.

Произведем сравнение двух эквивалентных вариантов одной и той же программы, использующей динамическую загрузку *DLL*.



Можно видеть, что практически все инструкции программы на *Delphi* имеют эквивалент в программе на Ассемблере. Однако если вы, воспользовавшись отладчиком, посмотрите на то, каким образом компилятор «перевел» первый вариант на машинные коды (в отладчике вам видны эквивалентные им ассемблерные команды), то ничего подобного второму варианту вы не увидите! Команд будет намного больше, и возможно, что их последовательность будет нарушена. Очевидно, что такая программа занимает больше места в памяти и медленнее выполняется. Это неизбежная плата за те преимущества, которые дает вам использование языка высокого уровня, например, *Delphi*.

3. Выполнение работы



Для правильной работы большинства программ (в т. ч. Delphi) под Windows NT необходимо предварительно настроить переменные среды. На рабочем столе найдите значок «Мой компьютер» и кликните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным Temp и Tmp значение %UserProfile%\Temp.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет.*

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File—Close All* для закрытия тех окон проектов, которые были загружены автоматически.

В данной работе используется та же библиотека *ASMDLL.DLL*, что и в предыдущей работе. Поэтому если вы сохранили ее, то заново создавать эту библиотеку не нужно, а следует сразу же перейти к написанию основной программы (см. ниже).

Выполнить команду *File—New—Other*. Затем в появившемся окне выбрать *DLL Wizard*. Создается заготовка *DLL*.

Введите текст:

```
library asmdll;
```

```
uses
  SysUtils,
  Classes;

function MessageBox(HWnd: Integer; Text, Title: PChar; Flags: Integer): Integer; stdcall; external 'user32.dll' name 'MessageBoxA';

procedure ShowMessage; stdcall;

label
start, text, title;

asm
        JMP START
TEXT:
        DB 'Библиотека ASMDLL.DLL успешно загружена',0
TITLE:
        DB 'Процедура ShowMessage',0
START:
        PUSH 0
        PUSH OFFSET(TITLE)
        PUSH OFFSET(TEXT)
        PUSH 0
        CALL MessageBox
end;

exports ShowMessage;

begin
end.
```

Компилировать *DLL*, нажав *Ctrl–F9*. Выполнить команду *File–Close*.

Выполнить команду *File–New–Application.* Это будет основная программа.

В разделе *var* укажите:

```
Handle: LongInt;
```

В разделе *Implementation* поместите строки:

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer; stdcall; external 'user32.dll' name 'MessageBoxA';
function ExitProcess(Code: Integer): Integer; stdcall; external 'kernel32.dll' name 'ExitProcess';
function LoadLibrary(LibName: PChar): Integer; stdcall; external 'kernel32.dll' name 'LoadLibraryA';
function GetProcAddress(Handle: pointer; ProcName: PChar): Integer; stdcall; external 'kernel32.dll' name 'GetProcAddress';
procedure FreeLibrary(LibName: PChar); stdcall; external 'kernel32.dll' name 'FreeLibrary';
```

Укажите метки, которые используются в программе:

```
label
START, TEXT, TITLE, DLLNAME, PROCNAME, ERROR;
```

Введите текст основной программы:

```
begin
asm
JMP START
TEXT:
DB 'Будем загружать DLL динамическим методом',0
TITLE:
DB 'Программа для Windows на Ассемблере',0
DLLNAME:
DB 'asmdll.dll',0
PROCNAME:
DB 'ShowMessage',0
START:
PUSH 0
PUSH OFFSET(TITLE)
PUSH OFFSET(TEXT)
PUSH 0
CALL MessageBox
PUSH OFFSET(DLLNAME)
CALL LoadLibrary
```

```
CMP EAX, 0
JNA ERROR
MOV Handle, EAX
PUSH OFFSET(PROCNAME)
PUSH EAX
CALL GetProcAddress
CMP EAX, 0
JE ERROR
CALL EAX
ERROR:
PUSH Handle
CALL FreeLibrary
PUSH 0
CALL ExitProcess
end;
end.
```

Запустите программу на выполнение, нажав клавишу *F9*. Если программа работает неправильно, проверьте наличие *DLL* по указанным вами маршрутам.

4. Контрольные вопросы для подготовки к защите работы

1. Что такое *WinAPI*?
2. Что включает в себя соглашение о вызове *stdcall*?
3. Расскажите о командах сравнения и условного перехода. Приведите примеры.
4. Привести ассемблерный эквивалент выражения: *if n=0 then m:=0 else t:=34*.
5. Поясните смысл использования команды *JMP LABEL2* в первом примере. Что будет, если ее удалить?
6. Во втором примере использована команда *JA LABEL*. Переделайте ассемблерный вариант так, чтобы алгоритм работы остался тем же, но использовалась команда *JNA LABEL*.
7. Косвенный вызов процедур и функций. Его особенности и отличия от прямого вызова.

8. Зачем при описании внешних функций используется зарезервированное слово *name*?

9. Есть ли разница между командами *CALL EAX* и *CALL [EAX]*? Изменится ли работа программы при замене первой на вторую? Почему? Уметь продемонстрировать при помощи отладчика.

10. В данной работе вы используется несколько внешних процедур и функций. Какие из них загружаются статически, а какие – динамически?

11. Можно ли обойтись лишь динамическим методом загрузки *DLL*, не прибегая к статическому нигде в своей программе?

12. В третьем примере в варианте на *Delphi* есть строки, не имеющие прямого соответствия в ассемблерном варианте, и наоборот. Почему?

13. Почему в ассемблерном варианте примера 3 нигде не встречается эквивалент операции *@ShowMessage := nil*?

14. Существует ли реально «указатель, никуда не указывающий», который в *Delphi* обозначается *nil*? Для ответа на данный вопрос вспомните, что входит в соглашение *stdcall* и затем воспользуйтесь отладочными возможностями среды *Delphi* при запущенной программе. Потом укажите заведомо неправильный путь к библиотеке *asmdll.dll* и вновь воспользуйтесь отладчиком. Сделайте необходимые выводы.

Лабораторная работа №4
«Определение версии *Windows*»

1. Цель работы

Целью данной работы является изучение одного из способов определения версии операционной системы *Windows*. Также в работе рассматривается способ организации многовариантного выбора при помощи команд условного перехода.

2. Теоретическая часть

В данной работе будут использоваться несколько функций *WinAPI*, с которыми вы уже знакомы: *MessageBox, ExitProcess*, а также новая для вас функция *GetVersion*.

В ряде случаев важно знать, под управлением какой операционной системы выполняется ваша программа, причем сделать это заранее не представляется возможным. Например, программа должна работать внезависимости от версии *Windows*. Ясно, что в более поздних реализациях появляются новые возможности, устраняются ошибки и т. д. Может оказаться, что, например, программа, написанная для *Windows 95*, не будет выполняться под *Windows NT*.

Для решения подобных проблем в *WinAPI* и предусмотрена функция *GetVersion*. Она не имеет параметров. В соответствии с соглашением о вызове *StdCall*, результат возвращается в регистре *EAX*.

Информация о версии содержиться в младшем слове *EAX*, которое, как известно, представляет собой регистр *AX*. Согласно традиции, версии программного обеспечения, в том числе Windows нумеруются двумя числами, разделенными точкой (например, *Windows NT 4.0*). Старшая часть номера версии (в данном примере – число «4») располагается в регистре *AL*, а младшая часть – в *AH*.

Например, для *Windows NT 4.0 AX = $0004* (в *Delphi* «символ доллара», стоящий перед числом, обозначает, что оно записано в шестнадцатиричной системе счисления, в Ассемблере для аналогичной цели используется буква *h* после числа).

Помимо собственно определения версии, часто возникает задача сообщить эту информацию пользователю, так как она не всегда очевидна,

особенно при его низкой квалификации, существенных изменениях интерфейса и пр.

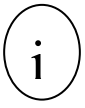
Из вышесказанного видно, что информация о версии возвращается в весьма специфичном виде, следовательно, невозможен ее непосредственный вывод на экран. Если на языке высокого уровня эта проблема легко решаема, то в случае применения Ассемблера следует рассмотреть ее особо.

Одним из вариантов, наиболее удачно реализуемом на языке низкого уровня, может быть организация последовательных сравнений и следующих за ними команд условного перехода. Принцип такой: сравниваем *AX* с тем значением, которое он имел бы для определенной версии, если условие выполняется, переходим к выводу сообщения, если не выполняется, то сравниваем с другим значением и т. д. Например:

```
CMP AX, $0105
JE _END
CMP AX, $0005
JE _END
...
...
...
_END:
CALL MessageBox
```

Конечно, необходимо выводить текстовые сообщения, поэтому в данном примере нужно выполнять дополнительные действия. Об этом будет сказано ниже.

3. Выполнение работы



Для правильной работы большинства программ (в т. ч. Delphi) под Windows NT необходимо предварительно настроить переменные среды. На рабочем столе найдите значок «Мой компьютер» и кликните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным Temp и Tmp значение %UserProfile%\Temp.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не коммента-*

рии, а директивы компилятора, без них ваша программа компилироваться не будет.

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File–Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File–New–Application*.

Укажите метки, которые используются в программе:

```
label
TITLE, _UNKNOWN, _XP, _2000, _9xNT40, _NT35, _NT351, START, _END;
```

Введите текст основной программы:

```
begin
asm
JMP START
TITLE:
DB 'Determining Windows Version...', 0
_UNKNOWN:
DB 'Unknown version of Windows!', 0
_XP:
DB 'Windows XP', 0
_2000:
DB 'Windows 2000', 0
_9xNT40:
DB 'Windows 95, 98 or NT 4.0', 0
_NT35:
DB 'Windows NT 3.5', 0
_NT351:
DB 'Windows NT 3.51', 0
START:
CALL GETVERSION
MOV EBX, OFFSET _XP
CMP AX, $0105
JE _END
MOV EBX, OFFSET _2000
```

```
CMP AX, $0005
JE _END
MOV EBX, OFFSET _9xNT40
CMP AX, $0004
JE _END
MOV EBX, OFFSET _NT35
CMP AX, $0003
JE _END
MOV EBX, OFFSET _NT351
CMP AX, $5103
JE _END
MOV EBX, OFFSET _UNKNOWN
_END:
PUSH 0
PUSH OFFSET TITLE
PUSH EBX
PUSH 0
CALL MessageBox
PUSH 0
CALL ExitProcess
end;
end.
```

Запустите программу на выполнение, нажав клавишу *F9*.

4. Контрольные вопросы для подготовки к защите работы

1. Что такое *WinAPI*?
2. Что включает в себя соглашение о вызове *stdcall*?
3. Расскажите о командах сравнения и условного перехода. Приведите примеры.
4. Зачем бывает нужно определять версию операционной системы на этапе выполнения программы?
5. Предложите свой вариант решения проблемы вывода текстового сообщения о номере версии.
6. Поясните, в каком виде функция *GetVersion* возвращает результат.

7. Почему все строки сообщений в данной программе заканчиваются нулем?

8. Поясните последовательность выполнения данной программы.

9. Для какой цели в программе используется регистр *EBX*?

Лабораторная работа №5
«Дизассемблирование»

1. Цель работы

Знакомство с понятием «дизассемблирование», создание простейшей программы-дизассемблера.

2. Теоретическая часть

Вы знаете, что Ассемблер – язык низкого уровня, и входящие в его состав мнемонические команды (например, *PUSH AX*) имеют однозначное соответствие среди команд машинного языка, понятного процессору.

Таким образом, Ассемблер предоставляет программисту возможность иметь дело с простыми для запоминания командами-словами, а не с непосредственно машинными командами, т. е. числами. Фактически, Ассемблер позволяет использовать все возможности программирования на машинных кодах, но с намного большим удобством.

После составления программы в виде текстового файла, содержащего мнемонические команды, вы запускаете компилятор, который переводит эти команды в машинные команды, понятные процессору.

Когда вы хотите запустить полученный исполняемый файл, вы сообщаете об этом операционной системе тем или иным способом (например, щелкая мышкой по имени файла). Программа загружается в свободное место оперативной памяти и операционная система передает ей управление.

Файлы, которые можно запустить (т. е. они содержат исполняемые процессором команды и являются законченными программами) имеют определенные расширения (часто называемое «типом файла»), а именно *.com* и *.exe*. Файлы первого типа сейчас можно увидеть очень редко (это исполняемый файл *DOS*, его размер не может превышать размер одного сегмента процессора 8086, т. е 65536 байт, что в настоящее время совершенно недостаточно, однако для обеспечения совместимости такие файлы поддерживаются всеми версиями *Windows*, кроме 64-разрядных). Такой файл имеет простейшую структуру – состоит из исполняемых команд и констант, нужных программе. Его первый байт уже будет выполнимой командой.

Все современные программы имеют запускаемый файл типа *.exe*. Такие файлы, помимо исполняемого кода, содержат специальную информа-

цию, предназначенную для операционной системы. Ее рассмотрение выходит за рамки данной работы.

Поскольку между командами Ассемблера и машинными командами существует однозначное соответствие, появляется возможность «перевести» вторые в первые, осуществив тем самым «дизассемблирование» программы (операция, обратная «ассемблированию», т. е. переводу мнемонических команд в машинные коды). Очевидно, что такая возможность чрезвычайно полезна – можно разобраться, каким образом построена программа, каков ее алгоритм. Можно сделать необходимые изменения и заново компилировать программу.

Следует особо отметить, что в настоящее время подавляющее большинство программ пишется на языках высокого уровня и имеет большой размер, поэтому ясное понимание дизассемблированного варианта программы, а зачастую и правильное дизассемблирование может быть весьма затруднено.

Очевидно, что выполнить дизассемблирование хоть сколько-нибудь большой программы вручную нереально. Это и не нужно, поскольку существуют специальные программы-дизассемблеры. Они имеют удобный интерфейс и ряд дополнительных возможностей.

Однако основной задачей программы-дизассемблера по-прежнему остается перевод машинных команд, содержащихся в исполняемом файле, в ассемблерные команды, и отображение последних на экране.

В этой работе вы создадите исполняемый файл типа *.com*, вводя в текстовом редакторе символы, *ASCII*-коды (*American Standard Code for Information Interchange – американский стандартный код для информационного обмена*) которых совпадают с машинными кодами. Таким образом, выполняя эту процедуру вы как бы пропускаете стадию компиляции программы, переводя ассемблерные команды в машинные коды самостоятельно.

После этого вы создадите простейшую программу-дизассемблер. В одном окне будет отображено содержимое исполняемого файла, а в другом – ассемблерные команды, полученные в результате дизассемблирования данного файла.

Для правильного понимания принципа работы дизассемблера необходимо рассмотреть более подробно вопрос о том, что же представляет собой машинная команда.

Машинная команда – информационная последовательность, однозначно указывающая процессору ту операцию, которую нужно выполнить.

Машинная команда может состоять из одного, двух и более байт. В наиболее общем случае формат машинной команды следующий:

Код операции	Операнд 1	Операнд 2

Поскольку существуют команды с различным количеством операндов (от 0 до 2), и сами операнды могут иметь различную длину (от 8 до 32 бит), возникает то разнообразие длин команд о котором сказано выше.

Тем не менее, код операции, очевидно, присутствует в любой команде. В большинстве случаев он занимает один байт – первый байт машинной команды.

На этом факте и основана работа рассматриваемой в данной работе программы. Поскольку код операции занимает 1 байт, то всего различных команд может быть 256. Поэтому для отображения на экране достаточно создать массив из 256 строк, каждая из которых содержит мнемоническую ассемблерную команду:

```
var
OpCode: array [0..255] of string;
```

Затем необходимо заполнить данный массив, поставив в соответствие каждому элементу массива строку, содержащую ассемблерную команду. При этом порядковый номер элемента массива будет равен машинному коду, соответствующему этой ассемблерной команде. Например:

```
begin
.....
OpCode[$40]:='INC AX';
OpCode[$50]:='PUSH AX';
.....
end.
```

Машинные коды, соответствующие той или иной ассемблерной команде, указываются в специальных справочниках. В данной учебной программе заполняется лишь малая часть массива (это означает, что создавае-

мый нами простейший дизассемблер не будет «знать» машинные коды всех команд).

Например, команде *PUSH AX* соответствует машинный код *$50* (шестнадцатеричная система счисления) и т. д.

Фактически, после заполнения массива, программа-дизассемблер должна лишь читать содержимое исполняемого файла байт за байтом. Считав из файла один байт, нужно использовать его в качестве номера элемента массива и вывести на экран соответствующую строку:

Memo2.Lines.Add(OpCode[Ord(Memo1.Text[i])]);

В данной программе для упрощения сначала в поле *Memo1* выводится все содержимое файла, а затем уже при помощи свойства *Text* производится побайтовое чтение. Очевидно, в реальной программе-дизассемблере чтение производится непосредственно из файла (либо из области памяти, куда предварительно загружен файл).

Важно отметить, что открыв любой исполняемый файл при помощи текстового редактора, мы увидим набор самых разнообразных символов (ряд символов не отображается, обычно редактор выводит вместо них пробел). Это объясняется тем, что текстовый редактор пытается интерпретировать каждый байт исполняемого файла как символ. Существует так называемая *ASCII*-таблица, в которой содержатся 256 символов и их цифровые коды. Например, команде *INC AX* соответствует машинный код *$40*. Однако этот же код в таблице символов соответствует «собачке» @. Поэтому, если где-то в исполняемом файле содержится машинный код команды *INC AX*, то текстовый редактор, в котором мы будем просматривать этот файл, отобразит в этом месте символ @. С этой интересной особенностью вы познакомитесь практически при выполнении работы.

Рассмотрим подробно работу фрагмента программы:

Memo2.Lines.Add(OpCode[Ord(Memo1.Text[i])]);

Как уже говорилось, можно считывать любой символ из поля *Memo* при помощи выражения *Memo1.Text[i]*. То есть, чтобы поочередно считать и дизассемблировать каждый байт из исполняемого файла, нужно организовать цикл, изменяя значение *i* от *1* до того значения, которое показывает последний байт в файле. Это значение равно размеру файла в байтах.

Функция *Ord* позволяет определить тот *ASCII*-код, который соответствует считанному символу. Таким образом, возвращаемый функцией *Ord* результат и будет равен машинному коду. Теперь уже можно выводить на экран (при помощи метода *Memo2.Lines.Add*) соответствующую ассемблерную команду.

Сначала мы определяем размер файла:

```
AssignFile(f, Edit1.Text);
Reset(f);
Size:=FileSize(f);
CloseFile (f);
```

Затем организуем цикл:

```
i:=1;
Memo1.Lines.LoadFromFile(Edit1.Text);
while i<=Size do
begin
Memo2.Lines.Add(OpCode[Ord(Memo1.Text[i])]);
Inc(i);
end;
```

Важно отметить, что данный вариант программы имеет существенный недостаток: правильно дизассемблируются команды размером только в один байт. Это связано с тем, что каждый байт исполняемого файла в данной программе рассматривается как машинный код операции. Т. е. если машинная команда состоит из двух байт (кода операции и операнда, например), то каждый из них будет рассматриваться как код операции, что неверно.

Однако этот недостаток весьма несложно устранить. В целом построение практически ценной программы-дизассемблера представляет собой весьма сложную задачу и в данном курсе не рассматривается.

3. Порядок выполнения работы

Для правильной работы большинства программ (в т. ч. Delphi) под Windows NT необходимо предварительно настроить переменные среды. На рабочем столе найдите значок «Мой компьютер» и клик-

ните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным Temp и Tmp значение %UserProfile%\Temp.

Сначала вам необходимо создать исполняемый файл *test.com*, который будет дизассемблирован создаваемой вами программой. Запустите текстовый редактор *Блокнот*.

В таблице приведены ассемблерные команды, эквивалентные им машинные коды (в двух системах счисления) и символы, *ASCII*-коды которых совпадают с машинными кодами команд. Именно эти символы вы и будете вводить в текстовом редакторе.

Ассемблерная команда	**Машинный код**		**Символ, отображаемый в текстовом редакторе**
	шестнадцатеричный	**десятичный**	
INC AX	*40*	*64*	*@*
INC BX	*43*	*67*	*C*
INC CX	*41*	*65*	*A*
INC DX	*42*	*66*	*B*
DEC AX	*48*	*72*	*H*
DEC BX	*4B*	*75*	*K*
DEC CX	*49*	*73*	*I*
DEC DX	*4A*	*74*	*J*
POP AX	*58*	*88*	*X*
POP BX	*5B*	*91*	*[*
POP CX	*59*	*89*	*Y*
POP DX	*5A*	*90*	*Z*
PUSH AX	*50*	*80*	*P*
PUSH BX	*53*	*83*	*S*
PUSH CX	*51*	*81*	*Q*
PUSH DX	*52*	*82*	*R*

Создаваемый в данной работе простейший дизассемблер может распознать только те машинные коды, которые перечислены в данной таблице.

Введите в окне текстового редактора *Блокнот* произвольную последовательность, состоящую из символов, указанных в правом столбце таб-

лицы. Фактически это будет последовательность машинных команд. Для ввода символов можно, удерживая клавишу *Alt*, набрать его десятичный код цифровыми клавишами (они находятся на клавиатуре справа). Символ появится в окне редактора.

Сохраните файл под именем *test.com* в том же каталоге, где будет находиться ваша программа.

Внимание! Данный файл является выполняемым, однако запускать его не следует! Введенная вами бессмысленная последовательность команд может при ее выполнении процессором нарушить стабильную работу операционной системы.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет.*

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File–Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File–New–Application*.

В разделе *Var* объявите необходимые для работы программы переменные:

```
OpCode: array [0..255] of string;
i: Integer;
f: file of byte;
Size: LongInt;
```

Поместите в окне вашей программы два поля вывода текста (компонент *Memo*, его можно найти панели в компонентов в разделе *Standard*).

Теперь поместите поле для ввода имени дизассемблируемого файла – компонент *Edit*. В окне *Object Inspector* выбрать объект *Edit1*, затем в разделе *Properties* найти свойство *Text* и присвоить ему значение «*test.com*». Это означает, что по умолчанию ваша программа будет открывать именно этот файл.

Затем нужно поместить в окне будущей программы кнопку. Для этого в панели компонентов в разделе *Standard* дважды кликнуть мышью на объ-

екте *Button*. Местоположение и размеры кнопки можно изменить по вашему желанию. В окне *Object Inspector* выбрать объект *Button1*, затем в разделе *Properties* найти свойство *Caption* и присвоить ему значение «*Open*», это будет текстом надписи на кнопке. В разделе *Events* найти событие *OnClick* и дважды щелкнуть мышью по нему. В окне исходного текста программы появится заголовок процедуры.

Затем введите исполняемую часть процедуры:

```
Memo1.Clear;
Memo2.Clear;
AssignFile(f, Edit1.Text);
Reset(f);
size:=FileSize(f);
CloseFile (f);
Memo1.Lines.LoadFromFile(Edit1.Text);
i:=1;
while i<=Size do
begin
Memo2.Lines.Add(OpCode[Ord(Memo1.Text[i])]);
inc(i);
end;
```

После описания процедуры введите:

```
begin
OpCode[$40]:='INC AX';
OpCode[$43]:='INC BX';
OpCode[$41]:='INC CX';
OpCode[$42]:='INC DX';
OpCode[$48]:='DEC AX';
OpCode[$4B]:='DEC BX';
OpCode[$49]:='DEC CX';
OpCode[$4A]:='DEC DX';
OpCode[$58]:='POP AX';
OpCode[$59]:='POP CX';
OpCode[$5A]:='POP DX';
OpCode[$5B]:='POP BX';
```

OpCode[$50]:='PUSH AX';
OpCode[$51]:='PUSH CX';
OpCode[$52]:='PUSH DX';
OpCode[$53]:='PUSH BX';

Запустите программу. Нажмите кнопку *Open* для дизассемблирования выбранного файла. В левом окне вы видите содержимое исполняемого файла, в правом – дизассемблированный вариант. Проверьте правильность дизассемблирования.

4. Контрольные вопросы для подготовки к защите работы

1. Что такое «машинная команда»?
2. Какие типы исполняемых файлов вам известны? Расскажите о них.
3. Почему язык машинных команд не используется на практике для написания программ?
4. Что такое «дизассемблирование»? Для чего оно нужно?
5. Поясните, почему, открыв исполняемый файл в текстовом редакторе, вы видите бессмысленный на первый взгляд набор символов, в том числе нечитаемых?
6. Приведите наиболее общий вид машинной команды. Какой размер в байтах она может иметь?
7. Почему созданный вами дизассемблер может правильно интерпретировать лишь однобайтовые машинные команды?
8. Зачем нужен массив *OpCode: array [0..255] of string*?
9. Поясните алгоритм работы дизассемблера, созданного в данной работе.
10. Каковы недостатки и ограничения дизассемблера, созданного в данной работе?
11. Вставьте в файл test.com символ, эквивалентный машинному коду, заведомо неизвестному созданному вами дизассемблеру. Например, это может быть символ *«пробел»*. Попытайтесь дизассемблировать этот файл. Объясните полученный результат.

Лабораторная работа №6
«Определение тактовой частоты центрального процессора компьютера»

1. Цель работы

Целью данной работы является изучение одного из способов определения тактовой частоты процессора.

2. Теоретическая часть

Тактовая частота является одной из основных характеристик центрального процессора и компьютера в целом. В связи с этим перед выполнением некоторых программ (например, требующих высокой производительности) необходимо определить тактовую частоту.

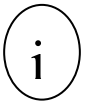
В ранних процессорах для «вычисления» тактовой частоты приходилось прибегать к определенным ухищрениям. Однако в настоящее время разработчики процессоров предусмотрели специальные команды для упрощения этой процедуры.

Команда *RDTSC* позволяет считать значение, хранящееся в 64-разрядном счетчике тактов процессора. Младшие 32 бита этого счетчика помещаются в регистр *EAX*, а старшие 32 бита – в *EDX*.

Указанный счетчик имеет ряд интересных особенностей. При сбросе процессора происходит сброс счетчика, т. е. если компьютер длительно время работает непрерывно, то счетчик переполниться и опять начнется отсчет тактов с нуля. Несложно оценить время, за которое счетчик будет переполнен; например, при тактовой частоте 2000 МГц оно составляет около 300 лет! Таким образом, проверку переполнения счетчика производить не имеет смысла.

Более того, на существующих сегодня процессорах нет смысла и в использовании старших 32 бит счетчика тактов. В данной работе, например, запоминается значение младших 32 бит, затем «засекается» временной интервал 0.5 с, после чего вновь считывается значение младших 32 бит счетчика тактов и из него вычитается ранее сохраненное значение. Затем результат делится на 500000, чтобы получить значение частоты в мегагерцах. Этот результат и выводится на экран.

3. Выполнение работы



Для правильной работы большинства программ (в т. ч. Delphi) под Windows NT необходимо предварительно настроить переменные среды. На рабочем столе найдите значок «Мой компьютер» и кликните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным Temp и Tmp значение %UserProfile%\Temp.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет.*

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File–Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File–New–Application*.

В разделе *Var* укажите:

```
dwTimerLow: dword;
```

В разделе *Implementation* введите:

```
function CPUSpeed: dword;
asm
RDTSC
MOV dwTimerLow, EAX
PUSH 500
CALL Sleep
RDTSC
SUB EAX, dwTimerLow
MOV EBX, 500000
XOR EDX,EDX
DIV EBX
end;
```

Введите текст основной программы:

begin

MessageBox (0, PChar ('Тактовая частота равна ' + IntToStr (CPUSpeed) + ' МГц'),

'Определение тактовой частоты процессора', MB_OK);

ExitProcess (0);

end.

Запустите программу на выполнение, нажав клавишу *F9*.

4. Вопросы для подготовки к защите работы

1. Что характеризует тактовая частота процессора?
2. В каких случаях ее бывает нужно определять?
3. Каково назначение команды *RDTSC*?
4. Является ли переполнение счетчика тактов частым событием?
5. Каков алгоритм работы созданной вами программы?
6. Что нужно изменить в программе, чтобы значение тактовой частоты выводилось в гигагерцах?

Лабораторная работа № 7
«Анализ исходного текста программы»

1. Цель работы

Целью данной работы является получение практических навыков анализа исходного текста программы с целью выяснения ее назначения и алгоритма работы.

2. Теоретическая часть

Данная работа является упрощенной моделью ситуации, когда у вас в распоряжении оказывается приложение (например, в виде исполняемого файла) неизвестного назначения (возможно, опасное). Ваша задача – выяснить назначение этой программы и определить алгоритм ее работы.

В подобной ситуации вы должны применить такие средства, как дизассемблер, отладчик и пр. Желательно использовать для запуска отдельный компьютер на тот случай, если опасная программа все же «выйдет из-под контроля».

Конечно, в данной работе ситуация значительно упрощена. Так, исследуемая программа совершенно не опасна. Кроме того, вам будет предоставлен не исполняемый файл, а исходный текст на языке высокого уровня.

Ваша задача состоит в следующем:

- определить назначение программы
- выяснить алгоритм работы программы, вплоть до назначения всех используемых переменных и функций

Рекомендуется прежде всего ввести исходный текст и запустить программу. Затем, анализируя выводимые программой сообщения, сделать предположение о ее назначении.

После этого следует переходить к детальному анализу исходного текста. При необходимости следует применять встроенные отладочные возможности *Delphi*. Для поиска информации об используемых в программе процедурах, функциях и другой информации рекомендуется пользоваться системой *Справки*, предусмотренной средой разработки *Delphi*. Также разрешается пользоваться Интернетом.

3. Исходный текст


```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, WinSock, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  WSData: WSAData;
  HostEnt: PHostEnt;
  Sock: TSocket;
  SockAddrIn: TSockAddrIn;
  Addr: PChar;
```

```
 implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
Memo1.Lines.Add ('Trying to connect to ' + Edit1.Text + '...');
if WSAStartup($0101, WSData)= 0 then
 begin
 Memo1.Lines.Add ('Winsocket library init... OK');
 HostEnt := GetHostByName(PChar(Edit1.Text));
  if HostEnt <> nil then
  begin
  Memo1.Lines.Add ('Getting Host Information... OK');
  Addr := HostEnt^.H_addr^;
    if Addr<>nil then
    Memo1.Lines.Add('IP: '+ Format('%d.%d.%d.%d', [Byte (Addr [0]),
    Byte (Addr [1]), Byte (Addr [2]), Byte (Addr [3])]));
   Sock := Socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if Sock <> INVALID_SOCKET then
   begin
   Memo1.Lines.Add('Creating Socket... OK');
   SockAddrIn.Sin_family := AF_INET;
   SockAddrIn.Sin_port := Htons(80);
   SockAddrIn.Sin_Addr.S_Addr := Byte(Addr[3]) shl 24 + Byte(Addr[2]) shl 16
+ Byte(Addr[1]) shl 8 + Byte(Addr[0]);
   if Connect(Sock, SockAddrIn, SizeOf(SockAddrIn)) = 0 then
   Memo1.Lines.Add('SUCCESSFULLY CONNECTED')
   else
   begin
   Memo1.Lines.Add('FAILED TO CONNECT');
   Memo1.Lines.Add(IntToStr(WSAGetLastError));
   end;
   end
   else Memo1.Lines.Add ('Creating Socket... Failed');
  end
```

```
  else Memo1.Lines.Add ('Getting Host Information... Failed')
 end
 else Memo1.Lines.Add ('Winsocket library init... Failed');
CloseSocket(Sock);
WSACleanup;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
Memo1.Lines.Clear;
end;

end.
```

4. Для получения зачета по работе необходимо:

- уметь сформулировать назначение программы,
- знать алгоритм ее работы,
- уметь ответить на произвольный вопрос по исходному тексту программы.

Лабораторная работа № 8
«Программа для загрузки файлов из сети Интернет»

1. Цель работы

Изучение способа загрузки файлов из сети Интернет на локальный компьютер.

2. Теоретическая часть

Загрузка (или «скачивание») файлов из Интернета – очень часто встречающаяся в настоящее время операция. Наиболее очевидный пример – сохранение музыки, архивов и другой информации у себя на компьютере. Когда вы обновляете свое программное обеспечение (например, операционную систему или антивирусную программу) через Интернет, также происходит загрузка файлов на ваш компьютер. И даже когда вы просто просматриваете страничку какого-нибудь сайта, ваш Интернет-браузер обязательно «скачивает» эту страницу на ваш компьютер, и лишь потом интерпретирует код этой страницы, показывая результат на экране.

В данной работе мы рассмотрим один из самых простых способов загрузки файлов. Он имеет существенный недостаток: при разрыве связи вам придется начать скачивание заново. Однако этот способ прост и нагляден. Кстати, *Microsoft Internet Explorer* до сих пор использует похожий метод.

Реализовать скачивание можно разными способами. Однако большинство этих способов подразумевает использование определенных «промежуточных звеньев» между вашей программой и *WinAPI*, в котором уже предусмотрены все возможности для организации скачивания. Поэтому целесообразно использовать непосредственно функции *WinAPI*. При этом ваша программа будет компактнее и быстрее.

Итак, что же потребуется для организации загрузки файла из сети?

Первой функцией, которая должна вызываться перед всеми прочими (в рассматриваемом способе), должна быть *InternetOpen*. В случае удачного завершения она вернет дескриптор, который вы будете использовать во всех остальных вызовах функций. В случае неудачного завершения данная функция вернет нуль. Номер происшедшей ошибки можно узнать при помощи функции *GetLastError*. Текстовую «расшифровку» ошибки дает функция *SysErrorMessage*.

function InternetOpen (AppName: PChar; AccessType: DWord;
Proxy, ProxyBypass: PChar; Flags: DWord): hInternet;

AppName – имя программы, запрашивающей доступ к Интернету;
AccessType – тип доступа; вы будете использовать для задания типа доступа константу *INTERNET_OPEN_TYPE_PRECONFIG*, которая указывает, что все настройки нужно взять из реестра *Windows*;
Proxy, ProxyBypass – списки имен или адресов, при обращении к которым нужно или, соответственно, не нужно, использовать прокси-сервер; эти параметры имеют смысл только в случае указания предыдущим параметром на необходимость применения прокси-сервера;
Flags – вспомогательный параметр, устанавливается в нуль.

После того, как в результате вызова функции *InternetOpen* работы получен дескриптор, можно получить доступ к нужному вам файлу. Делается это при помощи функции *InternetOpenURL*. *URL* - сокращение от *Universal Resource Locator*, что означает «унифицированный определитель адресов ресурсов». Его формат: *протокол://доменное имя (или IP-адрес)/путь*. Например: *http://yandex.ru/index.html*. Сокращение *HTTP* означает *HyperText Transfer Protocol* (протокол передачи гипертекста), *FTP – File Transfer Protocol* (протокол передачи файлов). Вы должны обязательно указывать URL полностью для правильной работы рассматриваемой функции. То, что вы можете не делать этого, например, в браузере, не должно вызывать удивления: он дополняет введенное вами имя до полного варианта автоматически.

В результате удачного завершения функция вернет дескриптор нужного вам файла, который вы будете использовать для скачивания. В случае ошибки будет возвращен нуль. Как и ранее, можно получить описание ошибки.

function InternetOpenUrl (hInet: hInternet; hUrl: PChar; Headers: PChar; HeadersLength: DWord; Flags: DWord; Context: DWord): hInternet;

hInet – указатель, полученный после вызова функции *InternetOpen*;
hUrl – URL, к которому нужно получить доступ; обязательно должен начинаться с указания протокола, по которому будет происходить соединение.
Headers, HeadersLength, Flags, Context – дополнительные параметры; в данной работе не используются и не рассматриваются; устанавливаются в нуль.

После получения дескриптора файла можно начинать его чтение при помощи функции *InternetReadFile*.

function InternetReadFile (hURL: hInternet; Buffer: Pointer; NumberOfBytes-ToRead: DWord; var NumberOfBytesRead: DWord): Boolean;

hURL – дескриптор файла, полученный после вызова функции *InternetOpenUrl*;
Buffer – указатель на буфер, куда будут записываться данные;
NumberOfBytesToRead – количество байт, которое нужно считать.
NumberOfBytesRead – содержит количество фактически считанных байтов.

Следует вызывать эту функцию в цикле, считывая за один раз какое-то количество данных, например 1 кб. В этом же цикле можно производить запись на диск. Как только функция *InternetReadFile* вернет в переменной *NumberOfBytesRead* нуль, файл скачан полностью, и можно завершать цикл.

Например:

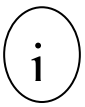
```
repeat
 asm
 PUSH PBufferLen
 PUSH BufferSize
 PUSH PBuffer
 PUSH hURL
 CALL InternetReadFile
 end;
   BlockWrite(f, Buffer, BufferLen)
until BufferLen = 0;
```

После завершения использования дескрипторов нужно их «освободить» при помощи функции *InternetCloseHandle*.

function InternetCloseHandle (hInet: hInternet): Boolean;

В качестве параметра указывается дескриптор, который нужно «освободить».

3. Порядок выполнения работы



Для правильной работы большинства программ (в т. ч. Delphi) под Windows NT необходимо предварительно настроить перемен-

ные среды. На рабочем столе найдите значок «Мой компьютер» и кликните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным Temp и Tmp значение %UserProfile%\Temp.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет.*

Для правильной работы данной программы следует отключить обработку исключительных ситуаций средой *Delphi*. Для этого выберите *Tools–Debugger Options–Language Exceptions* и снимите флажок *Stop on Delphi Exceptions*.

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File–Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File–New–Application*.

Поместите в окне программы компоненты *Edit* (два поля: для ввода *URL* и имени файла, под которым он будет сохранен), *Button* (управляющая кнопка), две надписи *Label* (пояснения над полями ввода) и *StatusBar* для вывода информации.

Свойству *Text* компонента *Edit1* присвойте значение *http://yandex.ru/index.html*. Свойству *Text* компонента *Edit2* присвойте значение *test.html*. Это позволит вам не вводить эти строки каждый раз при проверке работоспособности программы.

Свойству *SimplePanel* компонента *StatusBar1* присвойте значение *True*.

В раздел *Uses*, в котором объявляются используемые модули, добавьте *WinInet.*

В разделе *Implementation* введите:

```
function GetInetFile (fileURL: PChar; FileName: String): Boolean;
const BufferSize = 1024;
var
  sAppName: PChar;
  hSession, hURL: hInternet;
  Buffer: array[1..BufferSize] of Byte;
```

```
  PBuffer, PBufferLen: ^Byte;
  BufferLen: DWORD;
  f: File;

begin
sAppName := PChar(ExtractFileName(Application.ExeName));
PBuffer := @Buffer;
PBufferLen := @BufferLen;
Result := False;
asm
 PUSH OFFSET sAppName
 PUSH INTERNET_OPEN_TYPE_PRECONFIG
 PUSH 0
 PUSH 0
 PUSH 0
 CALL InternetOpen
 MOV hSession, EAX
end;
 try
 asm
  PUSH DWORD (0)
  PUSH DWORD (0)
  PUSH DWORD (0)
  PUSH DWORD (0)
  PUSH FileURL
  PUSH hSession
  CALL InternetOpenURL
  MOV hURL, EAX
  end;
  try
   AssignFile(f, FileName);
   Rewrite(f,1);
   repeat
    asm
    PUSH PBufferLen
    PUSH BufferSize
```

```
   PUSH PBuffer
   PUSH hURL
   CALL InternetReadFile
   end;
      BlockWrite(f, Buffer, BufferLen)
  until BufferLen = 0;
     Result:=True;
 finally
 asm
 PUSH hURL
 CALL InternetCloseHandle
 end;
 CloseFile(f);
 end;
finally
 asm
 PUSH hSession
 CALL InternetCloseHandle
 end;
 end;
 end;
```

В теле процедуры обработки нажатия на кнопку укажите:

```
if GetInetFile(PChar(Edit1.Text),Edit2.Text) then
 StatusBar1.SimpleText := 'File is Successfully Downloaded'
else
 StatusBar1.SimpleText := 'Error downloading file';
```

Запустите программу на выполнение, нажав клавишу *F9*.

Проверьте правильность работы программы. Учтите, что обработка ошибок в этой программе минимальна, поэтому правильно задавайте *URL* и имя локального файла. Правильный *URL* можно узнать при помощи любого браузера. Сохраненный файл будет находиться в том каталоге, куда вы сохранили проект.

4. Вопросы для подготовки к защите работы.

1. Когда требуется загружать файлы из Интернета? Приведите примеры.
2. Что такое *URL*?
3. Назначение и параметры функции *InternetOpen*.
4. Назначение и параметры функции *InternetOpenURL*.
5. Назначение и параметры функции *InternetReadFile*.
6. Назначение и параметры функции *InternetCloseHandle*.
7. Каков алгоритм работы программы?
8. Зачем в данной программе применен блок *try...finally*? Почему он используется дважды?

Лабораторная работа № 9
«Возобновляемая загрузка файлов из сети Интернет»

1. Цель работы

Изучение способа загрузки файлов из сети Интернет на локальный компьютер с возобновлением после разрыва соединения.

2. Теоретическая часть

В предыдущей работе вы создали программу для загрузки файлов из Интернета. Как уже было сказано, ее существенным недостатком является то, что при разрыве соединения скачивание придется начать заново.

Если речь идет о том, чтобы скачать снова файл небольшого размера, то проблем обычно не возникает. Однако уже при объеме файла в несколько мегабайт непредвиденный разрыв соединения (особенно, когда уже загружено 90 % файла) вызывает как моральные, так и материальные потери.

Отметим, что отсутствие возможности продолжить загрузку с момента разъединения может создавать большие неудобства не только в случае сравнительно медленного и неустойчивого модемного соединения. Даже при использовании выделенного канала вы можете столкнуться с подобной проблемой. Например, вы скачиваете большой объем данных. Скорость обмена достаточна, а вероятность случайного разрыва связи близка к нулю. Но всегда есть вероятность сбоя в операционной системе, «зависания» компьютера и пр. Даже если вам просто нужно перезагрузить его по каким-то причинам, вы не сможете сделать этого, не прервав скачивание. Если вам просто нужно выключить компьютер, и продолжить загрузку в следующий раз, это также невозможно без потери уже загруженной информации.

Итак, отсутствие возможности «докачать» лишь оставшуюся часть загружаемого файла создает существенные ограничения в использовании тех возможностей, которые дает нам сеть Интернет. Каким же образом можно устранить эту проблему?

Очевидно, что для этого нужно решить несколько задач:

1) сохранять уже загруженную часть файла,
2) запоминать объем скачанной информации,
3) при возобновлении работы начинать загрузку файла не с начала, а с места предшествующей остановки.

В действительности, для решения второй задачи достаточно решить первую, таккак размер сохраненной части файла и будет показывать объем скачанной информации!

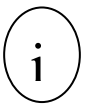
Помимо функций и процедур, известных вам по предыдущей работе, для решения третьей задачи (первая решается чисто алгоритмически) вам будет нужна еще одна функция *WinAPI*.

Она позволяет установить «указатель» в любое место нужного файла и продолжить скачивание с этого места, а не с начала.

function InternetSetFilePointer (hFile: hInternet;
lDistanceToMove: LongInt; pReserved: Pointer;
dwMoveMethod, dwContext: DWord): DWord;

hFile – дескриптор файла, созданный функцией *InternetOpenUrl*;
lDistanceToMove – показывает, на сколько байт нужно смещать указатель;
pReserved, *dwMoveMethod*, *dwContext* – неиспользуемые параметры.

3. Порядок выполнения работы



Для правильной работы большинства программ (в т. ч. Delphi) под Windows NT необходимо предварительно настроить переменные среды. На рабочем столе найдите значок «Мой компьютер» и кликните по нему правой кнопкой мыши. В меню выберите пункт «Свойства», затем закладку «Переменные среды». Необходимо присвоить переменным Temp и Tmp значение %UserProfile%\Temp.

Внимание! *Не удаляйте символы в фигурных скобках, например, {$R *.res}, находящиеся в шаблоне программы! Это не комментарии, а директивы компилятора, без них ваша программа компилироваться не будет*.

Запустить интегрированную среду разработки программ *Delphi*. Выполнить команду *File–Close All* для закрытия тех окон проектов, которые были загружены автоматически.

Выполнить команду *File–New–Application*.

Поместите в окне программы компоненты *Edit* (для ввода *URL*), *Button* (две кнопки: первая для начала загрузки, вторая – для приостановки).

Свойству *Text* компонента *Edit1* присвойте значение *http://yandex.ru/index.html*.

В раздел *Uses*, в котором объявляются используемые модули, добавьте *WinInet.*

В разделе *Var* введите:

```
Stop: Boolean;
```

В теле процедуры обработки нажатия на кнопку *Button1* введите:

```
var
hInet, hURL:hInternet;
ReadLen, RestartPos: DWord;
fBuf: array[1..1024] of Byte;
f: file;
```

Затем (между автоматически созданными *begin* и *end*) введите:

```
RestartPos:=0;
Button1.Enabled := False;
Button2.Enabled := True;
if FileExists ('Temporary file.tmp') then
 begin
 AssignFile (f, 'Temporary file.tmp');
 Reset (f, 1);
 RestartPos:=FileSize (f);
 Seek (f, FileSize (f));
 end
else
 begin
 AssignFile (f, 'Temporary file.tmp');
 ReWrite (f, 1);
 end;

hInet := InternetOpen (PChar (ExtractFileName (Application.ExeName)),
INTERNET_OPEN_TYPE_PRECONFIG, nil, nil, 0);
hURL := InternetOpenURL (hInet, PChar (Edit1.Text), nil, 0, 0, 0);
if RestartPos > 0 then InternetSetFilePointer (hURL, RestartPos, nil, 0, 0);
```

```
while (ReadLen <> 0) and (Stop = False) do
 begin
 InternetReadFile (hURL, @fBuf, SizeOf (fBuf), ReadLen);
 BlockWrite (f, fBuf, ReadLen);
 Application.ProcessMessages;
 end;
Stop := False;
Button1.Enabled := True;
Button2.Enabled := False;
InternetCloseHandle (hURL);
InternetCloseHandle (hInet);
CloseFile (f);
```

В теле процедуры обработки нажатия на кнопку *Button2* введите:

```
Stop := True;
```

Теперь нужно создать обработчик, вызываемый сразу после создания основного окна программы. Для этого дважды щелкните мышью на самой форме *Form1*.

В теле обработчика введите:

```
Stop := False;
Button2.Enabled := False;
```

Запустите программу на выполнение, нажав клавишу *F9*.

Проверьте правильность работы программы. Учтите, что обработка ошибок в этой программе минимальна, поэтому правильно задавайте *URL*. Правильный *URL* можно узнать при помощи любого браузера. Сохраненный файл (с именем *Temporary file.tmp*) будет находиться в том каталоге, куда вы сохранили проект.

Если вы полностью скачали нужный файл, то для скачивания с нового URL необходимо переименовать или удалить файл *Temporary file.tmp*! В противном случае программа будет считать его частью того файла, который вы собираетесь загрузить (проверки размера, первоначального имени файла и пр. не производится).

4. Вопросы для подготовки к защите работы.

1. Три основных проблемы, возникающих при организации «докачки». Как они решаются?
2. Назначение и параметры функции *InternetSetFilePointer*.
3. Каков алгоритм работы данной программы?
4. Назначение переменной *Stop*.

5. Домашнее задание.

Доработать интерфейс созданной вами программы. Например, отображать размер загружаемого файла, состояние скачивания (сколько еще нужно скачать), примерную скорость и требуемое время, ввести возможность задания пользователем имени, под которым файл будет сохранен и т. д.

Организовать обработку ошибок (неверный *URL*, нет соединения с Интернетом и пр.)

Результат предоставить преподавателю на следующем занятии.

Печатается в авторской редакции
Компьютерная верстка, макет В.И. Никонов

Подписано в печать 06.03.07
Гарнитура Times New Roman. Формат 60x84/16. Бумага офсетная. Печать оперативная.
Усл.-печ. л. 4,00. Уч.-изд. л. 2,11. Тираж 100 экз. Заказ № 628
Издательство «Универс групп», 443011, Самара, ул. Академика Павлова, 1

Отпечатано ООО «Универс групп»