Федеральное агентство по образованию
САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМ. Н.Г. ЧЕРНЫШЕВСКОГО
Институт дополнительного профессионального образования

Кафедра естественно-математических дисциплин

Разработка приложений для платформы Win32 с использованием Flat Assembler

(информатика)

Выпускная квалификационная работа
слушателя группы 94.2
ВАХЛАЕВОЙ КЛАВДИИ ПАВЛОВНЫ

Научный руководитель
к.физ-мат.н., доцент
ФЕДОРОВА АНТОНИНА ГАВРИЛОВНА

К защите допускается
Зав. кафедрой
д.физ-мат.н., профессор
_____В.П. Рябухо
«___»_____2009 г.

САРАТОВ

2009

Содержание

ВВЕДЕНИЕ....4

ОБЗОР КОМПИЛЯТОРА FASM....5

Использование компилятора....5
Форматы выходных файлов....6

ОСОБЕННОСТИ СИНТАКСИСА FASM....7

Формат записи инструкций....7
Константы....7
Метки....8

ПРОГРАММИРОВАНИЕ ДЛЯ WINDOWS В FASM....9

СТРУКТУРА КАРКАСНОГО ОКОННОГО WINDOWS – ПРИЛОЖЕНИЯ В FASM....13

Главная функция каркасного приложения....14
Цикл обработки сообщений....16
Оконная процедура....18

РЕСУРСЫ....22

УПРАЖНЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ....24

ЗАКЛЮЧЕНИЕ....26

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ....27

Введение

В работе представлены методические указания для студентов старших курсов компьютерных специальностей, изучающих дисциплины по системному программированию и разработке приложений.

Использование данных методических указаний на практических занятиях позволяет решить задачу обучения студентов базовым принципам программирования для Windows на языке ассемблера с использованием встроенных функций интерфейса прикладного программирования Win32 и макро – инструкций ассемблера.

В качестве учебного компилятора на занятиях по системному программированию для процессоров семейства Intel используется свободно распространяемый компилятор FASM (Flat Assembler). Этот компилятор достаточно прост в установке и использовании, отличается компактностью и быстротой работы, имеет богатый макро-синтаксис. Простота входного текста на языке ассемблера, отличающая FASM от компиляторов MASM и TASM, позволяет перейти непосредственно к разработке программ. Последнюю версию FASM можно получить на официальном сайте: http://www.flatassembler.net/, который поддерживает автор компилятора Томаш Гриштар (Tomasz Grysztar). В дистрибутив для Windows входит документация [1] в формате PDF, содержащая описание, как компилятора, так и машинных инструкций процессоров Intel включая расширенный набор команд. Кроме того, FASM является одним из немногих открытых средств разработки для платформы Win64.

В первых разделах методических указаний, содержащихся в данной работе, рассматриваются основы синтаксиса компилятора FASM и структура оконных приложений Windows. Изложение материала основывается на предположении, что студенты имеют определенные знания языка ассемблера и навыки работы с MASM (TASM). Примеры программ используются для выполнения первых двух из семи упражнений, разработанных для одного учебного семестра. Методические указания для остальных заданий, не вошедшие в данную работу, содержат разделы, связанные с программированием пользовательского интерфейса – создания диалоговых окон и меню окна приложения, далее рассматриваются вопросы чтения, записи файлов и выделения, освобождения памяти.

Обзор компилятора FASM

FASM (Flat Assembler) – это многопроходной, высокоскоростной компилятор ассемблера для процессоров с архитектурой x86. Помимо базового набора инструкций процессора и сопроцессора FASM поддерживает наборы мультимедийных инструкций MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4a и 3DNow!, а также AMD x86-64. FASM обладает развитым макро-синтаксисом и гибкой системой управления форматом выходных файлов, позволяет оптимизировать генерируемый машинный код, способен транслировать сам себя и работать в разных операционных системах, таких как MS-DOS, Windows 9x/NT, Linux, BSD.

Для операционных систем Microsoft Windows разработана IDE-версия компилятора FASM, которая использует графический интерфейс и обладает встроенным текстовым редактором. Исполняемый файл IDE-версии называется **`fasmw.exe`**.

Для успешной компиляции исходного кода программы, сохраненного в файле **`*.asm`**, требуется, чтобы переменная окружения **`include`** хранила полный путь к папке **`INCLUDE`**, входящей в дистрибутивный пакет FASM. Для IDE-версии можно установить значение этой переменной редактированием файла конфигурации **`fasmw.ini`**, который создается компилятором **`fasmw.exe`** при первом запуске.

FASM является одновременно и компилятором и компоновщиком, получив на входе текст программы на языке ассемблера, FASM выдает готовую к выполнению машинную программу. С одной стороны, это упрощает процесс получения исполняемого файла, с другой – делает невозможным использование традиционных **`OBJ`**- и **`LIB`**-модулей, но хотя FASM и не поддерживает объектных модулей, он имеет средства, позволяющие создавать модульные программы с использованием директив **`include`**.

Использование компилятора

Для того чтобы начать работу с FASM, необходимо запустить файл **`fasmw.exe`**. В появившемся окне редактора разместить текст программы и сохранить его в файле **`*.asm`**, или открыть уже существующий ассемблерный файл. Выполнить компиляцию, выбрав команду **`Compile`** (**`Ctrl+F9`**) из меню **`Run`**, и если компиляция пройдет успешно, компилятор отобразит окно результатов компиляции, иначе выведет сводку ошибок. Окно результатов компиляции содержит информацию о количестве проходов, длительности

компиляции и количестве байт, записанных в выходной файл. Сводка ошибок содержит сообщение об ошибке, показывает препроцессированную форму вызвавшей ее инструкции и расположение всех строк исходного кода, связанных с ошибкой.

Команда **Run** (**F9**) также выполняет компиляцию, и после успешного завершения запускает скомпилированную программу в том случае, если она относится к одному из исполняемых в среде Windows форматов.

Форматы выходных файлов

Для написания в FASM работающего приложения необходимо указать компилятору, какого формата должен быть создаваемый файл. Это делается директивой **format** после, которой следует указание форматов:

- **PE** (Portable Executable) — формат исполняемых файлов Windows. Далее должно следовать уточнение **GUI** (графический интерфейс), **console** (консольное приложение) или **native**. Если выбран графический интерфейс, то следует также указать версию графического интерфейса **4.0**, а так же зарезервированное слово **DLL**, если собирается динамическая библиотека.
- **MZ** — формат исполняемых файлов MS DOS.
- **COFF** или **MS COFF** (Common Object File Format) — объектный файл, который в дальнейшем будет линковаться к другому проекту или к ресурсам.
- **ELF** (Executable and Linkable Format) — формат для создания исполняемых файлов UNIX подобных систем (Linux).

При отсутствии директивы **format**, указав смещение **100h** (**org 100h**), можно получить исполняемый файл формата **com**, или, по умолчанию, компилятор создаст плоский двоичный файл.

Далее рассмотрим процесс создания ***.exe** файлов в формате PE для 32-разрядных систем Microsoft Windows. PE-файл начинается с заголовков, за которыми располагаются несколько секций. В секциях размещаются код и данные исполняемого файла, а также служебная информация, необходимая загрузчику. В названии формата PE присутствует слово "portable" ("переносимый") в силу того, что он слабо зависит от типа процессора и поэтому подходит для разных платформ. Последняя редакция формата PE может использоваться на 64-разрядных аппаратных платформах, однако, 64-разрядный PE формат незначительно отличается от 32-разрядного. Основное отличие касается разрядности полей структур PE-файла.

Особенности синтаксиса FASM

Формат записи инструкций

FASM использует синтаксис Intel для ассемблерных инструкций. Единственное существенное отличие от формата, принятого в других ассемблерах (MASM, TASM) – значение ячейки памяти всегда записывается как **`[var_name]`**, а просто **`var_name`** означает адрес ячейки, что позволяет обходиться без ключевого слова **`offset`**. Например, **`mov eax, var_1`** загрузит в регистр EAX адрес переменной **`var_1`**, тогда как для загрузки значения переменной нужно воспользоваться командой **`mov eax, [var_1]`**, аналогично, для того, чтобы поместить значение из регистра EAX в переменную **`var_1`**, необходимо указать имя переменной в квадратных скобках: **`mov [var_1],eax`**. При переопределении размера операнда в FASM вместо **`byte ptr`** пишется просто **`byte`**, вместо **`word ptr`** – **`word`** и т. д. Не позволяется использовать несколько квадратных скобок в одном операнде, таким образом, вместо **`[bx][si]`** необходимо писать **`[bx+si]`**. Эти изменения синтаксиса привели к более унифицированному и легкому для прочтения коду.

Знак "**`\`**" используется в синтаксисе FASM для объединения нескольких строк в одну, его расположение определяет место разрыва строки. После знака "**`\`**" строка не должна содержать ничего, кроме комментариев, начинающихся со знака "**`;`**".

Константы

Все символические имена, определяемые в исходном коде, чувствительны к регистру написания символов. Например, идентификаторы **`max_size`** и **`MAX_SIZE`** воспринимаются как два разных имени.

Константа может переопределяться произвольное количество раз, в этом случае она доступна только после объявления, и всегда имеет значение, данное ей в последнем переопределении перед местом использования, если константа объявлена только один раз, она доступна из любой части кода. Объявление константы состоит из имени константы, символа "**`=`**" и числового выражения, которое после вычисления становится значением константы. Например, можно объявить константу **`count`** с помощью директивы **`count = 17`** и затем использовать ее в инструкции ассемблера **`mov cx,count`** – которая превратится в **`mov cx,17`** во время процесса компиляции.

Метки

Каждая метка может быть определена только один раз и использована в любом месте программы, даже раньше своего объявления. Простейший способ объявления метки – поставить двоеточие сразу после ее имени, например, "**metka1:**". За таким объявлением метки может следовать очередная команда в той же строке, хотя для удобства чтения текста программы, лучше объявлять метки в отдельных строках кода. Для того чтобы обеспечить уникальность меток в подпрограммах, подключаемых с помощью директив **include**, FASM предлагает удобный способ описания локальных меток, а именно, любая метка, которая начинается с точки, является локальной по отношению к предшествующей глобальной метке, например, "**.local1:**". Имя локальной метки автоматически присоединяется к имени последней глобальной метки (без точки в начале имени). Это позволяет сосредоточить внимание только на уникальности меток в контексте подпрограммы. В то же время на локальную метку можно ссылаться из любого места программы по ее полному имени: **jmp metka1.local1**.

Метка, имя которой начинается с двух точек, является исключением из правил – такая метка имеет свойства глобальной метки, но следующие за ней локальные метки к ней не привязываются.

Для организации **LOOP**-циклов удобно использовать так называемые анонимные метки "**@@**". В программе можно ссылаться на предшествующую или последующую анонимную метку с помощью идентификаторов "**@b**" (или "**@r**") и "**@f**" соответственно.

Программирование для Windows в FASM

Основной тип приложений в Windows – оконные, поэтому начнем знакомство с процессом разработки программ для этой операционной системы с создания простых окон.

Окно – это прямоугольная часть экрана, в которой приложение производит отображение выводимой информации и из которой получает вводимую информацию. Пользователь может использовать клавиатуру, мышь или другое устройство ввода для взаимодействия с окном. В результате загрузки операционной системы Windows, создается окно, занимающее весь экран (desktop window). Каждое Windows-приложение создает хотя бы одно окно, называемое **главным окном приложения**. На рис. 1 показаны некоторые компоненты, из которых может состоять главное окно приложения.

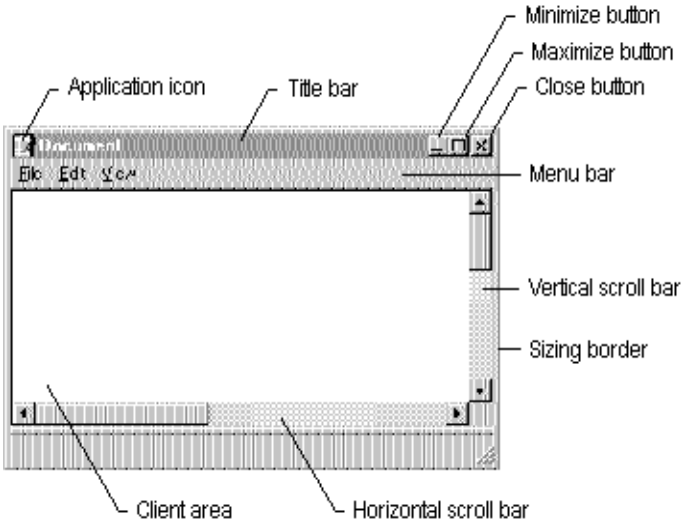


Рис. 1. Компоненты главного окна приложения.

Многие приложения создают дополнительные окна:

- элементы управления (controls);
- окна диалога (dialog boxes);
- окна сообщений (message boxes).

Для того чтобы вывести на экран любое окно, программа должна сначала описать его внешний вид и все свойства, то есть то, что называется **классом окна**. Для начала выведем окно с предопределенным классом – окно типа MessageBox. MessageBox – это маленькое окно с указанным текстовым сообщением и одной или несколькими кнопками.

Пример 1. Создание окна типа MessageBox

Создайте Windows-приложение, отображающее окно типа MessageBox с текстом: "FASM – инструмент разработчика".

```
format PE GUI 4.0                    ; Создать PE-файл графическим приложением Windows.
```

```
entry start                                    ; Определить точку входа в PE-файл.
include 'win32a.inc'                           ; Подключить стандартный набор макро-инструкций.

section '.data' data readable writeable        ; Секция данных.
   Caption db 'Первая программа',0
   Text db 'FASM – инструмент разработчика',0

section '.code' code readable executable       ; Секция кода.
start:
   invoke MessageBox,0,Text,Caption,MB_OK
   invoke ExitProcess,0

section '.idata' import data readable writeable ; Секция импорта.
   library KERNEL32, 'KERNEL32.DLL',\
           USER32, 'USER32.DLL'
   import KERNEL32,\
           ExitProcess, 'ExitProcess'
   import USER32,\
           MessageBox, 'MessageBoxA'
```

Директива **include** в начале программы подключает текст из файла **win32a.inc**, находящегося в папке **INCLUDE** пакета FASM. Подключаемый файл содержит еще несколько подключаемых файлов, таким образом, к ассемблерному тексту добавляются целые блоки описания структур, констант и макросов, которые могут использоваться в коде программы.

Далее создаются три секции: данных, кода и секция импорта. Каждая секция начинается директивой **section**, после которой в кавычках следует название секции, назначение и атрибуты. Первой рассмотрим секцию импорта, так как в коде программы вызываются функции интерфейса прикладного программирования (Application Programming Interface – API) Win32.

Win32 API – это набор готовых к использованию функций, расположенных внутри Windows и функционально объединенных в динамически связываемых библиотеках (DLL), таких как **KERNEL32.DLL**, **USER32.DLL**, **GDI32.DLL** и т.д. Библиотека **KERNEL32.DLL** содержит API-функции, работающие с памятью и управляющие процессами, **USER32.DLL** управляет интерфейсом программы, **GDI32.DLL** содержит графические функции. Для того чтобы использовать в программе функции из динамических библиотек, в PE-файле существует специальная секция импортируемых функций.

Секция импорта создается при помощи двух макросов – **library** и **import**. В самом начале секции должна быть макро-инструкция **library**, которая определяет, из каких библиотек будут импортированы функции и формирует для этих библиотек заголовки таблиц импорта. После **library** парами следуют параметры, разделенные запятыми: первый – имя таблицы

импорта, второй – символьная строка, определяющая название библиотеки. Далее следуют инструкции **import**, они создают тело таблицы импорта для каждой из импортируемых библиотек. После макро-иструкции **import** через запятую следуют ее параметры, первый параметр – имя таблицы импорта, объявленное ранее в макрокоманде **library**, затем пары параметров, которые содержат псевдоним и название импортируемой функции, заключенное в кавычки. Название импортируемой функции должно быть точным, так как оно будет передано операционной системе в момент запуска программы для получения адреса функции, а псевдоним может отличаться от реального имени. Таким образом, псевдоним является именем указателя, содержащего адрес импортируемой функции, и используется в дальнейшем для ее вызова.

В программе макро-инструкции **import** определяют указатель **ExitProcess** на функцию с тем же именем из библиотеки **KERNEL32.DLL** и указатель **MessageBox** на функцию **MessageBoxA** из библиотеки **USER32.DLL**.

Существует два типа API-функций, работающих с текстовыми строками: ASCII и Unicode. Имена ASCII функций оканчиваются символом "**A**", например **MessageBoxA**. Имена Unicode функций оканчивается символом "**W**" (**Wide char**), например **MessageBoxW**. Для некоторых стандартных DLL в пакете FASM имеются заполненные таблицы импорта, которые сохранены в файлах подключения ***.inc** каталога **\INCLUDE\API**, и позволяют импортировать ASCII и Unicode варианты API-функций. В каталоге **API** каждой библиотеке соответствует свой файл, имеющий одинаковое с библиотекой имя, и содержащий описание макро-инструкции **import** с таким же, как и название библиотеки в нижнем регистре именем таблицы импорта. Таким образом, необязательно описывать каждую импортируемую функцию, достаточно подключить файл, содержащий ее описание:

```
section '.idata' import data readable writeable
  library kernel32,'KERNEL32.DLL',\
          user32,'USER32.DLL'
  include 'api\kernel32.inc'
  include 'api\user32.inc'
```

Замечание. Далее будем обращаться к именам API-функций без постфикса **"A"** или **"W"**, так как подключаемые файлы могут определить и выбрать подходящую для системы кодирования функцию.

Секция данных имеет название **'.data'** и назначение **data**, в соответствии с которым в этой секции определяются данные программы, атрибуты указаны **readable writeable**, то есть эта область памяти будет

доступна для чтения и записи. В этой секции директивой **db** определяется две переменные **Caption** и **Text**, содержащие адреса первых символов строк заголовка и текста сообщения. Содержимое строк указывается в кавычках, ноль после запятой добавляет в конец строки нулевой байт – **null-terminator**.

Секция кода имеет название **'.code'** и назначение **code**, то есть исполняемая часть программы, атрибуты **readable executable** означают, что из секции кода мы можем читать данные (под данными подразумеваются коды инструкций и определенные в этой секции данные) и можем выполнять содержимое этой секции.

Первая команда в секции кода – это макро-инструкция **invoke**, которая вызывает функцию создания сообщения через указатель **MessageBox**, определенный в секции импорта. После имени функции через запятую следуют ее параметры. Команда **invoke** проверяет количество и тип параметров функции и передает их в стек от последнего к первому.

Первый параметр функции **MessageBox** должен содержать хэндл окна-владельца. Операционная система выдает каждому объекту (процессу, окну и др.) идентификационный номер или хэндл. Если у окна нет владельца, и оно не зависит ни от каких других окон, то его хэндл равен 0.

Второй параметр – указатель на адрес первой буквы текста сообщения, заканчивающегося нуль-терминатором.

Третий – указатель адреса первой буквы заголовка сообщения.

Четвертый – стиль сообщения. Список стилей находится в файле **INCLUDE\EQUATES\USER32.INC**. Все стили можно заменить числом, означающим тот, или иной стиль или их совокупность, например: **MB_OK+MB_ICONEXCLAMATION**. В **USER32.INC** указаны шестнадцатеричные значения стилей. Некоторые стили не могут использоваться одновременно, например **MB_OKCANCEL** и **MB_YESNO**. Причина в том, что сумма их числовых значений (1+4=5) будет соответствовать значению другого стиля – **MB_RETRYCANCEL**.

Макрос **invoke** задает генерацию кода эпилога, который очищает стек по завершении работы функции **MessageBox** от параметров функции и возвращает код нажатой пользователем кнопки в регистр EAX, или возвращает 0, если не хватило памяти для создания окна с сообщением.

Функция **MessageBox** выводит окно с сообщением и приостанавливает выполнение программы, ожидая реакцию пользователя.

Вызов функции **ExitProcess** завершает процесс исполняемой программы. Эта функция имеет лишь один параметр – код завершения, если программа нормально завершает свою работу, этот код равен нолю.

Структура каркасного оконного Windows – приложения в FASM

Используя функцию **MessageBox**, мы ограничены набором стандартных параметров. Кроме того, программа не может продолжать исполняться до тех пор, пока пользователь не прореагирует нажатием кнопки или закрытием сообщения. В случае использования окна, программа может исполняться, параллельно реагируя на действия пользователя. Программу, которая весь вывод на экран производит в графическом виде, будем называть оконным приложением.

Оконное приложение – строится на базе специального набора функций Win32 API, составляющих графический интерфейс пользователя (GUI, Graphic User Interface). Любое оконное Windows-приложение имеет типовую структуру, основу которой составляет так называемое **каркасное** приложение, содержащее минимально необходимый для функционирования полноценного Windows-приложения программный код. Минимальное приложение Windows состоит из трех частей:

- главной функции;
- цикла обработки сообщений;
- оконной процедуры.

Выполнение любого оконного Windows-приложения начинается с **главной функции**. Она содержит код, осуществляющий настройку (инициализацию) приложения в среде Windows. Видимым для пользователя результатом работы главной функции является появление на экране графического объекта в виде окна. Последним действием кода главной функции является создание цикла обработки сообщений. После его создания приложение становится пассивным и начинает взаимодействовать с внешним миром посредством специальным образом оформленных данных – **сообщений**. Обработка поступающих приложению сообщений осуществляется специальной процедурой, называемой оконной. **Оконная процедура** уникальна тем, что может быть вызвана только из операционной системы, а не из приложения, которое ее содержит.

Главная функция каркасного приложения

Главная функция начинается с выполнения стартового кода, который представляет собой последовательный вызов функций Win32 API. Первой в этой последовательности вызывается функция **`GetModuleHandle`**. Она предназначена для идентификации исполняемого модуля программы в адресном пространстве процесса. Идентификатор исполняемого модуля используется при создании главного окна приложения и его дочерних окон, а так же при вызове функций, загружающих иконки, курсоры, меню и т.д.

Основная задача главной функции оконного Windows-приложения состоит в правильной инициализации программы и корректном ее завершении. Правильная инициализация приложения предполагает выполнение ряда предопределенных шагов, включающих **регистрацию класса окна**, **создание окна**, **отображение окна**.

Каждое окно в системе принадлежит к какому-либо классу. Под **классом окна** понимается совокупность присущих ему характеристик, таких как стиль его границ, форма курсора мыши, иконка, цвет фона, наличие меню, адрес оконной процедуры, обрабатывающей сообщения этого окна. Все окна одного класса работают "одинаково", так как используют одну и ту же оконную процедуру. Существует три типа классов окна:

1. Системные глобальные классы: классы, которые регистрируются при загрузке операционной системы. К таким классам относятся классы элементов управления: **`button`** – кнопки, **`listbox`** – списки, **`combobox`** – комбинированные списки, **`edit`** – окна редактирования, **`static`** – статические элементы, **`scrollbar`** – полосы прокрутки.
2. Прикладные глобальные классы: классы, которые регистрируются динамическими связываемыми библиотеками, и доступны всем приложениям системы.
3. Прикладные локальные классы: классы, которые приложения регистрируют для своего внутреннего использования.

Характеристики собственного класса окна, создаваемого приложением, задаются с помощью специальной структуры **`WNDCLASS`**, которая описана в файле **`FASM\INCLUDE\EQUATES\USER32.INC`** и содержит 10 атрибутов класса окна:

1. **`style`** – стиль окна;

2. **lpfnWndProc** – указатель на процедуру обработки сообщений, посланных окну;
3. **cbClsExtra** – количество дополнительных байт в памяти для данной структуры;
4. **cbWndExtra** – количество дополнительных байт в памяти для данных присоединенных к окну;
5. **hInstance** – идентификатор исполняемого модуля программы;
6. **hIcon** – идентификатор иконки окна;
7. **hCursor** – идентификатор курсора окна;
8. **hbrBackground** – цвет фона окна;
9. **lpszMenuName** – указатель на имя меню окна (это так же может быть идентификатор меню);
10. **lpszClassName** – указатель на имя класса.

После инициализации структуры необходимо зарегистрировать класс окна в системе. Это действие выполняется с помощью функции **RegisterClassEx**, которой в качестве параметра передается указатель на экземпляр структуры **WNDCLASS**.

После того как класс окна описан и зарегистрирован в системе, приложение на его основе может создать множество различных окон. Создание окна выполняется функцией Win32 API **CreateWindowEx**, которой передаются следующие параметры:

1. **dwExStyle** – расширенный стиль окна;
2. **lpClassName** – указатель на зарегистрированное имя класса;
3. **lpWindowName** – указатель на имя окна, которое отображается в заголовке стандартного окна, или является текстом, написанным на кнопке и т.д.;
4. **dwStyle** – стандартный стиль окна;
5. **x** – X координата левого верхнего угла окна;
6. **y** – Y координата левого верхнего угла окна;
7. **nWidth** – ширина окна;
8. **nHeight** – высота окна;
9. **hWndParent** – идентификатор родительского окна или окна-владельца;
10. **hMenu** – идентификатор меню или дочернего окна;
11. **hInstance** – идентификатор исполняемого модуля программы;
12. **lpParam** – дополнительные данные.

В случае успешного выполнения функции **CreateWindowExA** требуемое окно будет создано и программа получит идентификатор созданного окна.

Каждое окно имеет стиль или несколько стилей. Стиль окна – это именованная константа, задающая дополнительные параметры окна внутри класса. Названия расширенных стилей окна находятся в файле **EQUATES\USER32.INC** в группе **Extended Window Styles**, названия стандартных стилей находятся в группах **Window Styles** и **Common Window Styles**. При создании окна его стиль передается в качестве одного из параметров функции **CreateWindowEx.** Существует три основных типа окон Windows:

- Перекрывающиеся окна – это окна верхнего уровня, имеющие заголовок, границу, и клиентскую область. Окна этого типа предназначены для использования в качестве главного окна приложения. Окно типа **WS_OVERLAPPED** имеет только заголовок и границу. Окно типа **WS_OVERLAPPEDWINDOW** дополнительно имеет системное меню (**WS_SYSMENU**), кнопки минимизации (**WS_MINIMIZEBOX**) и максимизации (**WS_MAXIMIZEBOX**).
- Вспомогательные окна – этот тип окон используется для создания окон диалога, окон сообщений и других временных окон, которые отображаются вне клиентской области главного окна приложения. Для порождения всплывающего окна используется стиль **WS_POPUP**.
- Дочерние окна – отображается в пределах клиентской области родительского. Для порождения дочернего окна используется стиль **WS_CHILD**. Дочернее окно по умолчанию имеет только клиентскую область и может содержать любые элементы кроме меню. Действия, производимые с родительским окном, влекут за собой действия, производимые с дочерним.

При порождении окна можно задать его начальное состояние. Для того чтобы созданное окно появилось на экране, используется стиль **WS_VISIBLE**. Окно, имеющее стиль **WS_VISIBLE**, порождается видимым. Иначе, для отображения окна необходимо вызвать функцию **ShowWindow** и функцию **UpdateWindow** для обновления содержимого окна.

Цикл обработки сообщений

Сообщение в Win32 представляет собой объект особой структуры, формируемый Windows. Формирование и доставка этого объекта в системе позволяют управлять работой как самой Windows, так и загруженных Windows-

приложений. Инициировать формирование сообщения могут несколько источников: пользователь, непосредственно приложение, система Windows, другие приложения.

Для каждого приложения Windows поддерживает очередь сообщений. Приложение просматривает свою очередь и обрабатывает сообщения в цикле. Простейший цикл обработки сообщений состоит из трех функций: **GetMessage**, **TranslateMessage**, **DispatchMessage**. Такой цикл, записанный на псевдокоде с использованием синтаксиса языка С, имеет вид:

```
while (GetMessage(&msg, (HWND) NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
```

Функция **GetMessage** получает сообщение из очереди сообщений приложения. Если сообщения отсутствуют – функция ожидает их и цикл приостанавливается до появления нового сообщения. Параметры функции следующие:

- указатель на экземпляр структуры **MSG**, в которую заносится информация о сообщении. **MSG** – структура сообщения описана в файле **USER32.INC** и состоит из шести элементов: 1. **hwnd** – идентификатор окна-получателя сообщения; 2. **message** – код сообщения; 3,4. **wParam**, **lParam** – дополнительная информация о сообщении; 5. **time** – время отправки сообщения; 6. **pt** – координаты курсора на время отправки сообщения (также является структурой, описанной в **USER32.INC**).
- идентификатор окна-получателя сообщения, если он равен **NULL**, то сообщения принимаются для любого окна данного приложения;
- минимальное значение сообщения;
- максимальное значение сообщения.

Последние два параметра исполняют роль фильтра сообщений. Так как все сообщения являются целочисленными значениями, можно установить фильтрацию типа "от… и до…". Если максимальное и минимальное значения равняются нолю, фильтрация не выполняется, и принимаются все сообщения без исключения.

Функция **GetMessage** возвращает нулевое значение (**FALSE**), если получено сообщение **WM_QUIT**, во всех остальных случаях возвращается ненулевое значение (**TRUE**). Следовательно, приложение может завершиться

посылкой сообщения **WM_QUIT**, для этого предназначена функция **PostQuitMessage**. Обычно она вызывается в ответ на получение главным окном приложения сообщения **WM_DESTROY**.

При нажатии клавиш Windows посылает активному окну сообщения **WM_KEYDOWN** и **WM_KEYUP**. Эти сообщения содержат некоторые виртуальные коды клавиш, а не символы. Для анализа нажатий и порождения сообщений **WM_CHAR**, которые содержат символьные коды клавиш, используется функция **TranslateMessage**.

Функция **DispatchMessage** предназначена для передачи сообщения оконной процедуре. Такая передача производится не напрямую, так как **DispatchMessage** ничего не знает о месторасположении оконной процедуры, а косвенно – посредством системы Windows. Это делается следующим образом:

1. Функция **DispatchMessage** возвращает сообщение операционной системе.
2. Операционная система, используя описание класса окна, передает сообщение соответствующей оконной процедуре приложения.
3. После обработки сообщения оконной процедурой управление возвращается операционной системе.
4. Операционная система передает управление функции **DispatchMessage**.
5. Функция **DispatchMessage** завершает свое выполнение.

После получения сообщения **WM_QUIT** функция **GetMessage** возвращает нулевое значение, и цикл обработки сообщений завершается. Выход из цикла означает завершение работы приложения.

Оконная процедура

Оконная процедура призвана организовать адекватную реакцию со стороны Windows-приложения на действия пользователя и поддерживать в актуальном состоянии то окно приложения, сообщения которого она обрабатывает. Приложение может иметь несколько таких процедур, их количество определяется количеством классов окон, зарегистрированных в системе функцией **RegisterClass**.

Каждый раз, когда окно Windows-приложения получает какое-либо сообщение, операционная система производит вызов соответствующей оконной процедуры и передает ей в качестве параметров элементы из структуры сообщения. Центральным блоком процедуры является синтаксическая

конструкция, в задачу которой входит распознавание поступившего сообщения по его коду и передача управления на ту ветвь подпрограммы, которая предназначена для его обработки.

После того как обработаны все предусмотренные сообщения, вызывается функция **`DefWindowProc`**, для того чтобы операционная система своими средствами произвела стандартную обработку сообщения.

Пример 2. Базовый вывод окна

Создайте оконное приложение, которое имеет системное меню и завершается, получив сообщение закрытия окна.

```
format PE GUI 4.0
entry start
include 'win32a.inc'
section '.data' data readable writeable
  _class db 'Simple Window',0                    ; Класс окна.
  _title db 'Простое Окно',0                     ; Заголовок окна.
  _error db 'Ошибка',0                           ; Сообщение об ошибке.

; Структура, описывающая класс окна.
wc        WNDCLASS 0,WindowProc,0,0,0,0,COLOR_BTNFACE+1,0,_class
; wc – указатель на первый байт структуры данных, созданной в соответствии с шаблоном WNDCLASS.
; Элементам структуры wc присваиваются соответствующие значения, разделенные запятыми.

; Структура, в которую сохраняются элементы сообщения.
msg       MSG
; msg – указатель на первый байт структуры данных, созданной в соответствии с шаблоном MSG.
; Значения элементов структуры msg считаются неопределенными.

section '.code' code readable executable
; Главная функция.
 start:
      invoke  GetModuleHandle,0                  ; Функция GetModuleHandle с параметром равным 0
                                                 ; возвращает в eax идентификатор вызвавшего ее модуля.

; Заполнить элементы структуры wc необходимыми данными.
      mov     [wc.hInstance],eax                 ; Идентификатор модуля поместить в  wc.hInstance.

     invoke  LoadIcon,0,IDI_APPLICATION          ; Функция LoadIcon с первым параметром равным 0
                                                 ; и вторым параметром, равным константе IDI_APPLICATION
                                                 ; загружает в класс окна стандартную иконку приложения.
     mov     [wc.hIcon],eax                      ; Идентификатор иконки поместить в wc.hIcon.

     invoke  LoadCursor,0,IDC_ARROW              ; Функция LoadCursor с первым параметром равным 0
                                                 ; и вторым параметром, равным константе IDC_ARROW
                                                 ; загружает в класс окна стандартный курсор в форме стрелки.
     mov     [wc.hCursor],eax                    ; Идентификатор курсора поместить в wc.hCursor.
; Зарегистрировать класс.
     invoke  RegisterClass,wc                    ; Вызвать функцию RegisterClass,
                                                 ; передав в качестве параметра указатель на структуру wc,
                                                 ; содержащую описание класса окна.
; Если возвращаемое значение отлично от 0,
; то класс успешно зарегистрирован.
      cmp     eax,0                              ; Сравнить eax и 0,
      je      error                              ; в случае равенства перейти на метку ошибки.
 ; Создать окно.
     invoke  CreateWindowEx,0,_class,_title,\ ;
```

```
        WS_VISIBLE+WS_SYSMENU,\
        128,128,256,192,0,0,[wc.hInstance],0
;  Если возвращаемое значение отлично от 0,
;  то окно успешно создано.
      cmp    eax,0                                ;  Сравнить eax и 0,
      je     error                                ;  в случае равенства перейти на метку ошибки.

;Цикл обработки сообщений – проверка сообщений и выход по WM_QUIT.
msg_loop:
      invoke  GetMessage,msg,NULL,0,0             ;  Получить сообщение.
      cmp    eax,0                                ;  Если получено WM_QUIT,
      je     end_loop                             ;  завершить программу.
      invoke  TranslateMessage,msg                ;  Преобразовать сообщения типа WM_KEYUP
                                                   ;  в сообщения типа WM_CHAR.
      invoke  DispatchMessage,msg                 ;  Передать сообщение оконной процедуре.
      jmp    msg_loop                             ;  Продолжить цикл.

 error:
      invoke  MessageBox,0,_error,0,MB_ICONERROR+MB_OK        ;  Вывести сообщение об ошибке.

 end_loop:
      invoke  ExitProcess,[msg.wParam]                        ;  Завершить программу,
                                                              ;  возвратив  операционной системе
                                                              ;  результат работы оконной процедуры.
;  Оконная процедура WindowProc.
;  Вызывается каждый раз при получении окном нового сообщения и передаче его через DispatchMessage.
;  Процедура обработки сообщений WindowProc вызывается со следующими параметрами:
;  hwnd – идентификатор окна-получателя сообщения,
;  wmsg – код сообщения,
;  wparam – первый параметр, определяющий дополнительные данные, связанные с сообщением.
;  lparam – второй параметр, определяющий дополнительные данные, связанные с сообщением.
proc WindowProc hwnd,wmsg,wparam,lparam
      push    ebx esi edi                         ;  Поместить значения регистров ebx esi edi в стек,
                                                  ;  так как при выполнении действий по умолчанию
                                                  ;  по отношению к переданным сообщениям
                                                  ;  эти значения могут быть изменены.
;  Если получили сообщение WM_DESTROY, которое означает, что окно удалили
;  с экрана, нажав Alt-F4 или кнопку в верхнем правом углу,
;  то переходим на метку .wmdestroy, где будет вызвана функция PostQuitMessage.
;  Параметром функции PostQuitMessage является код завершения,
;  ноль означает, что программа самостоятельно завершает работу.
      cmp    [wmsg],WM_DESTROY
      je     .wmdestroy
 .defwndproc:
;  Если получено другое сообщение,
;  вызвать его обработчик по умолчанию с параметрами,
;  которые передавались WindowProc.
      invoke  DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]
      jmp    .finish
 .wmdestroy:
      invoke  PostQuitMessage,0               ;  Отправляем главной программе сообщение WM_QUIT,
                                              ;  после получения которого функция GetMessage вернет 0,
                                              ;  и цикл обработки сообщений прервется переходом на end_loop.
      mov    eax,0
 .finish:
      pop    edi esi ebx                      ;  Извлечь из стека значения в регистры edi esi ebx.
      ret                                     ;  Вернуться в цикл обработки сообщений.
endp
section '.idata' import data readable writeable
 library kernel32,'KERNEL32.DLL',\
       user32,'USER32.DLL'
 include 'api\kernel32.inc'
```

```
include 'api\user32.inc'
```

Рассмотренный в примере 2 код, представляет собой шаблонную программу для написания в FASM всех последующих оконных приложений. Например, для того чтобы добавить на окно кнопку, во-первых, укажем в секции данных класс кнопки и ее имя:

```
_classb db 'BUTTON',0                  ; Стандартный класс BUTTON не требует регистрации.
_textb  db 'КНОПКА',0                  ; Текст, написанный на кнопке.
```

Во-вторых, для того чтобы создать соответствующее кнопке дочернее окно, в секции кода добавим обработку сообщения **WM_CREATE**, которое приходит окну при его создании. Также добавим обработку сообщения **WM_COMMAND**, которое приходит окну, когда пользователь выбирает пункт меню или совершает действие с другим дочерним элементом окна (нажатие на кнопку):

```
proc WindowProc hwnd,wmsg,wparam,lparam
      push    ebx esi edi
;  При создании окно получает сообщение WM_CREATE.
      cmp     [wmsg],WM_CREATE
      je      .wmcreate
;  При нажатии на кнопку окно получает сообщение WM_COMMAND.
      cmp     [wmsg],WM_COMMAND
      je      .wmcommand
      cmp     [wmsg],WM_DESTROY
      je      .wmdestroy
 .defwndproc:
      invoke  DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]
      jmp     .finish
 .wmcreate:
      invoke  CreateWindowEx,0,_classb,_textb,\                  ; Создать на главном окне
         WS_VISIBLE+WS_CHILD+BS_PUSHBUTTON,\                     ; кнопку с идентификатором 1001.
        10,10,100,50,[hwnd],1001,[wc.hInstance],NULL
      jmp     .finish
;  Если получили WM_COMMAND от кнопки, то wparam содержит
;  в старших двух байтах константу BN_CLICKED=0 (кликнута кнопка)
;  и в младших двух байтах идентификатор кнопки 1001,
;  можно считать, что весь wparam содержит идентификатор кнопки.
 .wmcommand:
;  Определить нажатие кнопки,
;  сравнивая значение wparam с 1001.
      cmp     [wparam],1001                                      ; Если кнопка не нажата,
      jne     .finish                                            ; то выйти из процедуры,
      invoke  MessageBox,[hwnd],_textb,_title,0                  ; иначе показать MessageBox.
      jmp     .finish
 .wmdestroy:
      invoke  PostQuitMessage,0
      mov     eax,0
 .finish:
      pop     edi esi ebx
      ret
endp
```

Ресурсы

Практически в каждом приложении Windows имеются ресурсы – данные, которые определяются до начала работы программы и особым образом добавляются в исполняемый файл. Примерами ресурсов, определенных в Win32 API, являются иконки, курсоры, меню, диалоговые окна, используемые в программе растровые изображения, шрифты и т.п.

Компилятор FASM позволяет размещать описание ресурсов в исходном коде в отдельной **секции ресурсов**, которая определяется директивой:

```
section '.rsrc' resource data readable
```

Ресурсы, описанные в отдельном файле ***.rc** и скомпилированные в файл ***.res** специальным компилятором ресурсов, можно подключить в FASM к тексту программы с помощью директивы:

```
section '.rsrc' resource from 'resfile.res' data readable
```

Рассмотрим общие принципы описания ресурсов в исходном коде на примере загрузки в класс окна собственной иконки приложения. Для этого добавим после секции импорта в программе из примера 2 секцию ресурсов вида:

```
section '.rsrc' resource data readable
IDI_MAIN=10
; Определение типов ресурсов.

  directory   RT_ICON,icons,\                ; Иконкам выделено два типа ресурсов:
              RT_GROUP_ICON,group_icons      ; RT_ICON – тип отдельной иконки,
                                             ; RT_GROUP_ICON – тип ресурса, связанного с одним
                                             ; или несколькими ресурсами типа RT_ICON.

; Объявление поддиректорий.
  resource group_icons,\                     ; Группа иконок.
           IDI_MAIN,LANG_NEUTRAL,main_icon
  resource icons,\                           ; Отдельная иконка.
           1,LANG_NEUTRAL,main_icon_data
; Объявление ресурсов.
; Группа иконок содержит единственную иконку,
; поэтому задаем только одну пару параметров.
  icon main_icon, main_icon_data,'1.ico'
```

В самом начале секции ресурсов должна быть макро-инструкция **directory**, которая определяет типы ресурсов содержащихся в секции. После нее парами следуют значения, разделенные запятыми: первое в каждой паре – идентификатор типа ресурса, второе – имя поддиректории содержащей ресурсы указанного типа.

В случае с иконкой из нашего примера необходимо выделить ресурсы типа **RT_ICON** и **RT_GROUP_ICON**, так как приложение в разных ситуациях может отображать наиболее подходящую по размеру и глубине цвета иконку из

имеющегося у него набора. Таким образом, отдельные иконки типа **RT_ICON** организованы в один ресурс типа **RT_GROUP_ICON**.

Далее каждая поддиректория объявляется макро-инструкцией **resource**, в которой после имени, указанном в макросе **directory**, тройками идут параметры ресурсов – первый параметр является идентификатором ресурса, второй параметр определяет язык, третий – имя ресурса. Для более удобного доступа к ресурсу из программы вместо идентификатора используется именованная константа (**IDI_MAIN=10**). Для ресурсов, не имеющих языковой принадлежности, в качестве второго параметра следует использовать константу **LANG_NEUTRAL**.

Ресурсы различных типов объявляются соответствующими макро-инструкциями, например, для иконки используется инструкция **icon** с именем ресурса группы иконок, за которым через запятую следуют пары параметров: имя ресурса отдельной иконки и, заключенный в кавычки, путь к файлу с иконкой.

После добавления в программу из примера 2 секции ресурсов, изменим вызов функции **LoadIcon**, указав в первом параметре идентификатор исполняемого модуля, из ресурсов которого будет загружена иконка, и во втором параметре – идентификатор группы иконок, из которой функция сама выберет наиболее подходящую:

```
... ... ... ...
invoke  LoadIcon,[wc.hInstance],IDI_MAIN
... ... ... ...
```

В нашем случае с единственной иконкой в группе у функции **LoadIcon** нет выбора, и в качестве иконки приложения из ресурсов всегда будет загружаться только одна иконка.

Упражнения для самостоятельной работы

1. Напишите программу, которая выводит на экран окно типа MessageBox с сообщением "Нажмите Да", заголовком "Первая программа" и кнопками "Да" и "Нет".

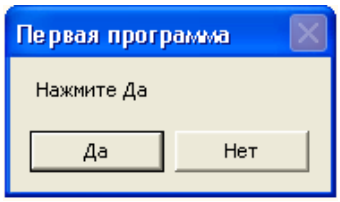


Рис. 2. MessageBox из упражнения 1.

2. Переделайте программу из примера 2 так, чтобы окно содержало четыре кнопки. При нажатии на каждую кнопку должен выводиться MessageBox с именем кнопки в качестве текста сообщения. Исполняемый файл программы должен иметь собственную иконку.

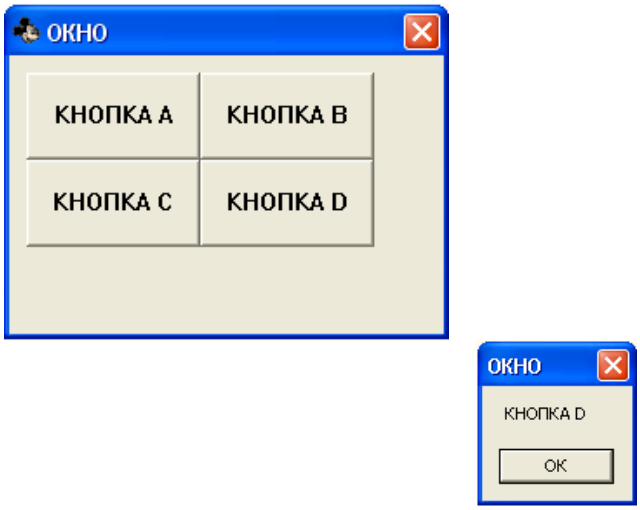


Рис. 3. Окно программы из упражнения 2.

3. Напишите программу, которая выводит на экран диалоговое окно, содержащее кнопку "OK" и статические элементы в виде надписей, иконки и горизонтальной линии.

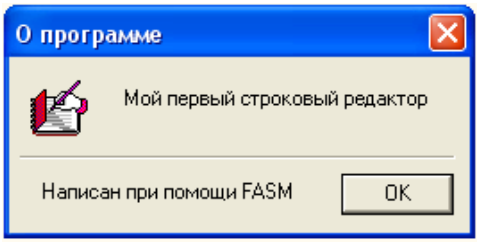


Рис. 4. Окно диалога из упражнения 3.

4. Напишите приложение "Строковый редактор", которое содержит три компонента:

- список (listbox) – область, где показывается текст, разбитый по строкам, редактирование текста запрещено.

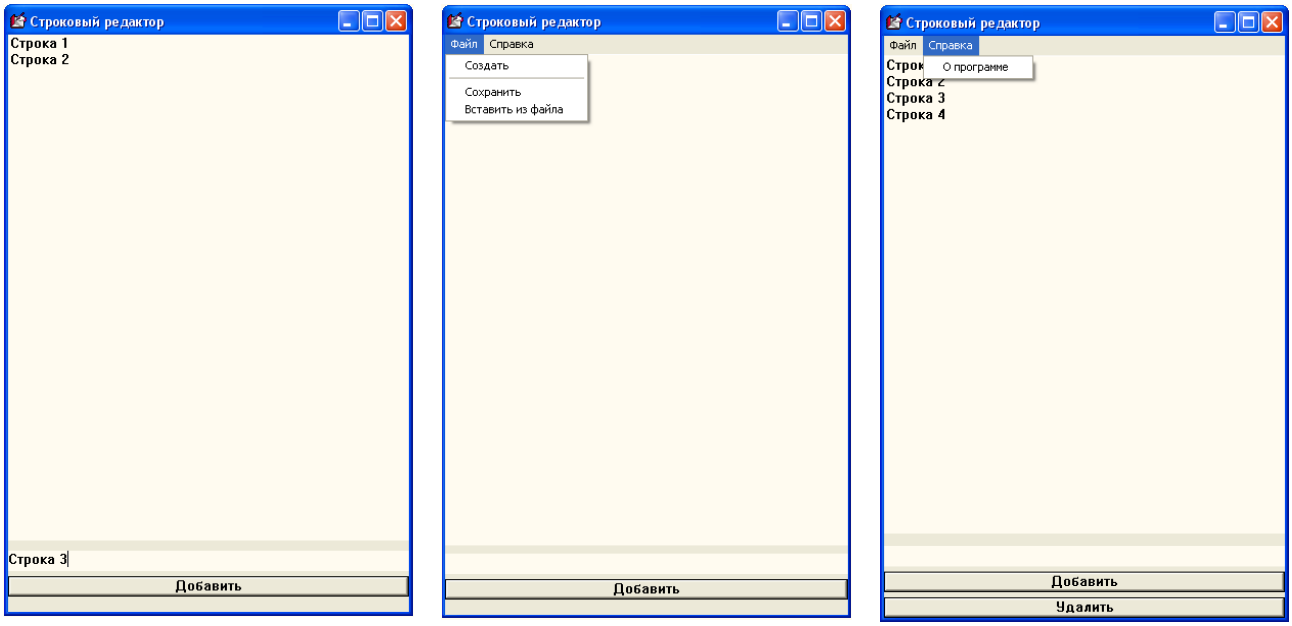
- окно редактирования (editbox) – область для ввода одной строки текста.
- кнопка (button) с текстом "Добавить", при нажатии которой строка, находящаяся в editbox'е добавляется к строкам listbox, и editbox очищается.

5. Добавьте в программу "Строковый редактор" главное меню с двумя пунктами "Файл" и "Справка". Пункт меню "Файл" содержит три подпункта:

- "Создать". Очищается listbox и editbox.
- "Сохранить". Все строки из listbox сохраняются в файл "NoteLine.txt".
- "Вставить из файла". Все строки из файла "NoteLine.txt" отображаются в listbox.

Пункт меню "Справка" содержит подпункт "О программе", при выборе которого отображается диалоговое окно из задания 3.

6. Добавьте в программу "Строковый редактор" кнопку "Удалить", при нажатии на которую из списка строк удаляется одна выделенная.



а) *б)* *в)*

Рис. 5. Окно программы "Строковый редактор" после выполнения а) упражнения 4; б) упражнения 5; в) упражнения 6.

7. Добавьте функцию помещения иконки на панель System Tray при минимизации окна программы "Строковый редактор".

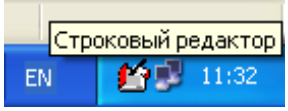


Рис. 6. Окно программы "Строковый редактор" после минимизации.

По клику на иконке окно программы должно восстанавливаться, а иконка удаляться с панели System Tray.

Заключение

Представленные методические указания знакомят студентов с разработкой приложений для платформы Win32 на языке ассемблера.

Для удобства использования в программах API-функций Win32 в качестве компилятора выбран имеющий упрощенный синтаксис компилятор FASM и несколько первых разделов методических указаний посвящены его описанию. Далее рассматриваются вопросы, связанные с программированием оконного интерфейса и создания полноценного приложения Windows, включающего в себя меню, диалоги и т.д.

Таким образом, для разработки собственного приложения в ходе выполнения упражнений, студенты используют в качестве шаблона примеры из методических указаний, расширенные за счет добавления новых ресурсов и обработчиков для различных событий меню и диалогов. Такой подход способствует успешному изучению студентами программирования для Windows и эффективному использованию на базе полученных знаний обширной справочной документации для разработки приложений.

Список использованной литературы

1. Tomasz Grysztar. Flat Assembler Programmer’s Manual [Электронный ресурс]. – Официальный сайт FASM. Режим доступа: http://flatassembler.net/docs.php?article=manual
2. SASecurity gr., BarMentaLisk. Цикл статей «Ассемблер под Windows для Ч.» [Электронный ресурс]. – Портал SASecurity Information Box. Режим доступа: http://sa-sec.org/?cat=11&paged=3
3. Цикл статей «Уроки по Win32 API» [Электронный ресурс]. – Портал WASM.ru. Режим доступа: http://www.wasm.ru/series.php?sid=1
4. Зубков С.В. Assembler для DOS, Windows и UNIX [Текст]/ Зубков С.В. – 3-е издание, стер. – М.: ДМК Пресс; Спб.: Питер, 2005. – 608 с.