Норсеев Сергей

РАЗРАБОТКА ОКОННЫХ ПРИЛОЖЕНИЙ НА FASMe

Оглавление

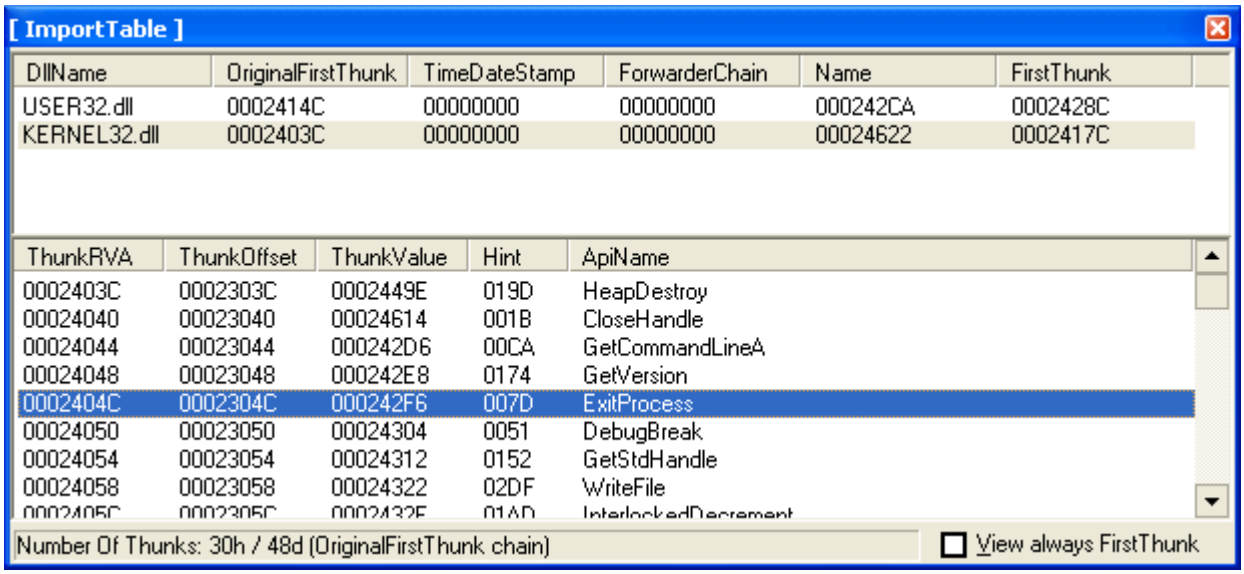
Глава 1. Первое знакомство с Flatasmom
Почему ассемблер

У читателя наверняка возник вопрос: почему я намерен описывать работу с функциями win32 API на ассемблере, а не на C, например. Основное преимущество языка ассемблера перед всеми языками высокого уровня состоит в том, что, программируя на ассемблере, вы должны знать, что и как работает (хотя многие считают это недостатком). Очень многие языки высокого уровня в отличие от языка ассемблер имеют неприятную особенность скрывать от программиста некоторые (порой весьма интересные и очень важные) особенности работы разрабатываемых им программ. Приведу несколько показательных примеров. Чтобы было более понятно, о чем это я.

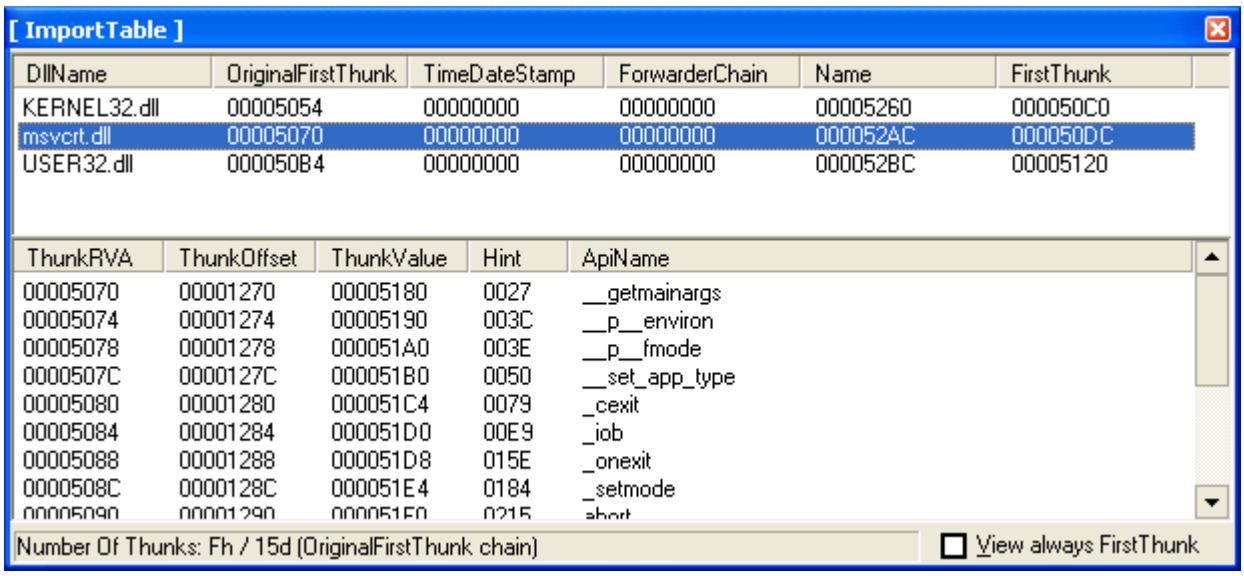
Предположим такую ситуацию: написал программист в Microsoft Visual Studio 6,0 небольшую программу выводящую сообщение:

```
#include <windows.h>
int main(){
 MessageBox(HWND_DESKTOP, "Hello", "Hello", MB_OK);
 return 0;
}
```

Вроде бы все просто всего одна функция (из библиотеки user32.dll). Что тут может быть не так. Сколько, по-вашему, различных API функций использует эта программа? По логике вещей она может обойтись всего двумя (MessageBox(A/W) и ExitProcess). Однако:



Зачем ей столько различных функций? Если вы думаете, что это просто причуды Microsoft Visual Studio, то вот таблица импорта той же самой программы, скомпилированной в Dev-C++:



А нам нужно всего только две функции. Неудивительно, что программы, написанные на ассемблере, опережают своих конкурентов по скорости работы и размеру.

Идем дальше. При разработке оконных приложений мы часто пишем код в так называемых обработчиках событий (OnCreate, OnInit, OnOk и другие). А вы знаете, как эти самые обработчики получают управление? Вы знаете, что такое оконная (диалоговая) процедура, цикл обработки сообщений? Многие программисты этого не знают. И что самое печальное и не хотят знать.

Вот и получается, что мы пишем программу, а как работает эта самая программа, мы и не знаем. Кто-то мне, наверное, возразит: я знаю, как работает моя программа: когда пользователь нажимает на такую-то кнопку на форме, она делает то-то. Хорошо, а как твоя программа узнает, что пользователь нажал именно на эту кнопку, а не на какую-нибудь другую?

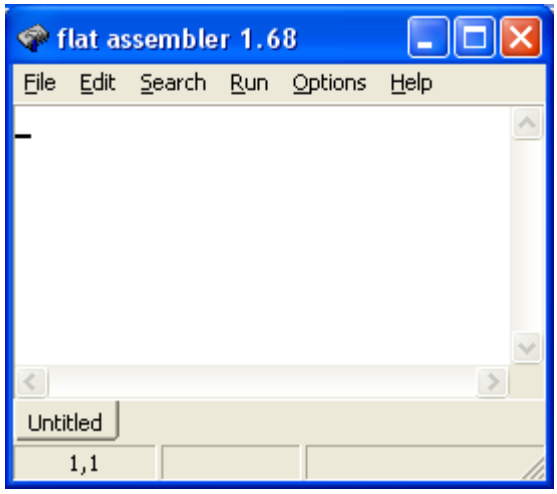
Увы, многие программисты хотят, чтобы за них думали среды разработки, в которых они работают. Они хотят получать готовые и, главное рабочие программы парой кнопкой мыши, не особенно вдаваясь в детали их функционирования. А если у них что-то не получается, они бегут на форумы, надеясь что кто-то подскажет им где и какую галочку надо поставить для того, чтобы программа наконец заработала.

Еще одним преимуществом ассемблера является то, что среды разработки программ на нем (если это конечно можно назвать средами), как правило, не требуют установки. Это позволяет вам скинуть такую среду себе на флешку (что я, кстати говоря, и сделал) и разрабатывать программы на ассемблере чуть ли не на коленке. Все что вам нужно это компьютер с любой ОС семейства Windows и USB порт для флеш-карты. Все, вам даже ничего устанавливать не надо.

Ладно, что-то я заговорился. Пора переходить к сути.

Что такое Flatasm

Flat assembler (FASM, Flatasm) – свободно распространяемый ассемблер[1], написанный Томашем Грыштаром (польск. Tomas Grysztar). По своему внешнему виду он очень похож на обычный блокнот:



По сути, единственная разница между ним и блокнотом состоит в том, что Flatasm умеет собирать (ассемблировать) набранный пользователем текст в исполняемый модуль (exe, dll, com).

В отличие от многих других ассемблеров Flatasm позволяет сразу получить исполняемый модуль без промежуточного этапа в виде объектного файла. Причем сделать все это можно прямо в окне программы (у некоторых ассемблеров, строго говоря, и своего рабочего окна нет, что сильно напрягает), без манипуляции с командной строкой. Многим это, конечно, может не понравиться. Но лично меня это очень радует.

Не лишен он, конечно, и недостатков. Во-первых, напрягает отсутствие в нем хоть какого-нибудь отладчика. Для отладки приложений написанных на Flatasm прихохдится прибегать к сторонним решениям. Во-вторых, малоинформативные сообщения об ошибках, по которым трудно сразу понять что случилось. А также некоторые другие замеченные мной нюансы, котрые будут упомянуты в разделе «замечания по работе с Flatasmom» этой главы.

На самом деле, я вас ни к чему не принуждаю. Вы вольны использовать тот ассемблер, который вам больше нравится. Лично я остановил свой выбор на Flatasme, и именно под него я и буду описывать синтаксис ассемблера и приводить примеры программ. Вы можете легко брать мои заготовки и использовать их на других ассемблерах (правда, для этого вам придется немного подправить их синтаксис). А функции win32API вообще не зависят от используемого вами ассемблера и даже языка программирования.

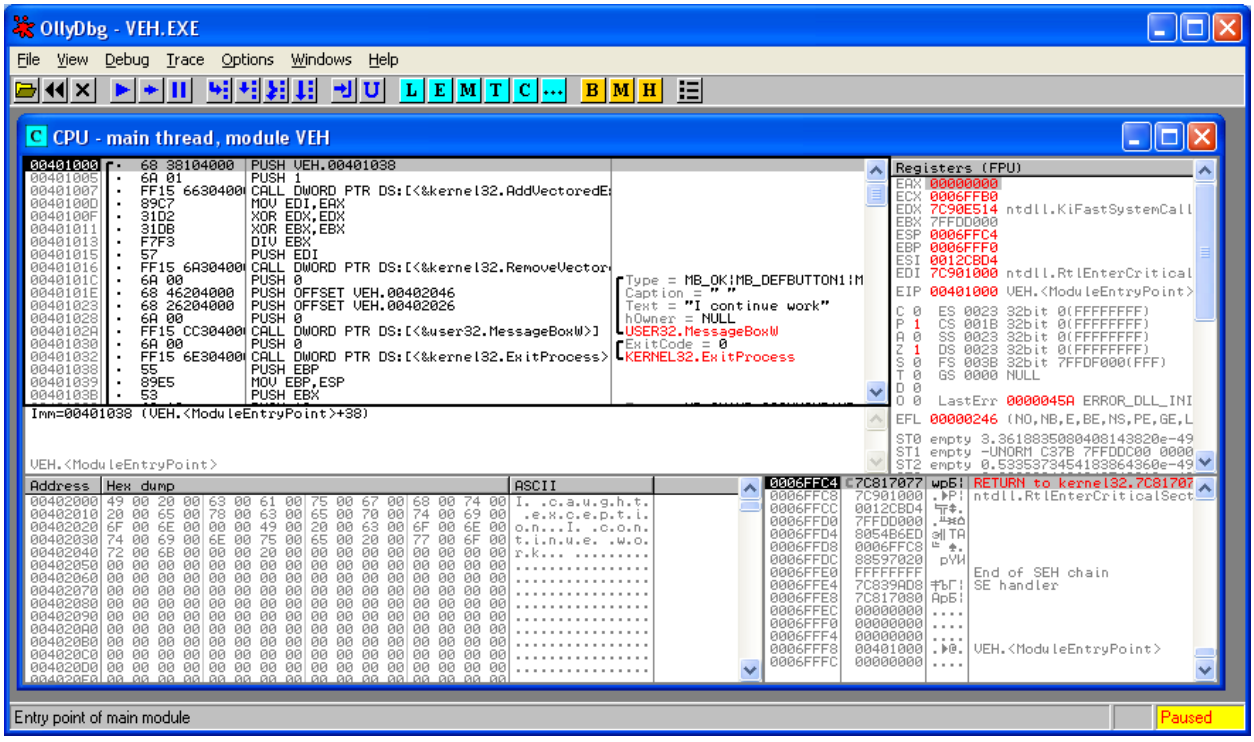
---

[1] Здесь не стоит путаться. Дело в том, что слово «ассемблер» имеет два значения: низкоуровневый язык программирования и компьютерная программа, осуществляющая преобразование программы написанной на языке ассемблер в машинный язык, то есть в исполняемый модуль.

Так что я очень надеюсь на то, что даже если вы не используете Flatasm и не планируете его использовать, вы найдете здесь что-то интересное для себя. Что-то я увлекся.

Чем отлаживать

Любой человек, который хоть раз писал в своей жизни что-нибудь сложнее Hello Worlda знает, что написание серьезных программ без отладки невозможно. Как я говорил выше во Flatasm не входит отладчик, поэтому для отладки программ, написанных на Flatasme приходится прибегать к сторонним решениям. Существует огромное количество отладчиков, что называется, на любой вкус и цвет. Но наибольшую популярность заслужил отладчик OllyDbg со своими многочисленными клонами (модами). Именно им мы и будем пользоваться. Выглядит он так:



Для того чтобы запустить в нем отладку той или иной программы в меню File выберите Open, и укажите исполняемый файл со своей программой. После этого программа откроется перед вами во всей своей красе. Будьте готовы к тому, текст программы будет несколько отличаться от того, что вы набирали во Flatasme. Связано это прежде всего с тем, что OllyDbg использует синтаксис MASMa, другого популярного ассемблера.

Как вы видите окно отладчика разделено на 4 части (кстати, такое разделение встречается во всех известных мне отладчиках ассемблерного кода). В центральной части располагается сам ассемблерный код, который мы и отлаживаем. В окне справа – регистры процессора (о них будет рассказано позднее). В левом нижнем окне – память, разделенная на три колонки: адреса ячеек, содержимое ячеек в шестнадцатеричной системе

счисления и ASCII представление (хотя может быть включено и Unicode представление, все зависит от настроек) содержимого этих ячеек. В правом нижнем окне – стек, также разделенный на три колонки.

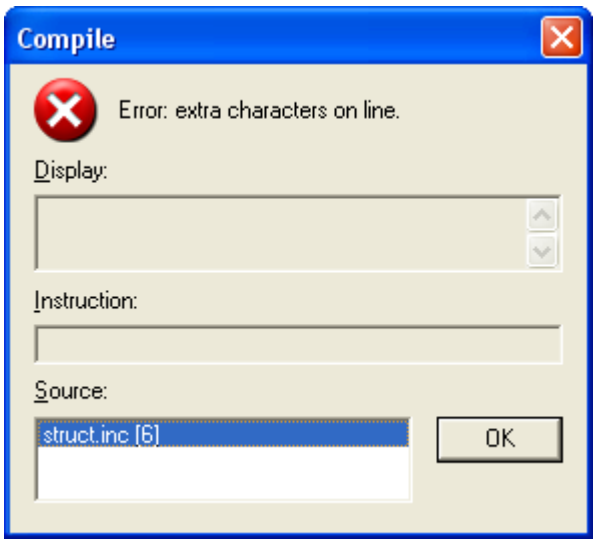
В таблице ниже представлены некоторые комбинации клавиш, которые могут быть вам полезны при работе с этим отладчиком:

Комбинация	Описание
F2	Установить/Снять точку останова
F7	Шаг трассировки с заходом в вызываемую процедуру
F8	Шаг трассировки без захода в вызываемую процедуру
F9	Запустить отлаживаемую программу
Ctrl+F2	Перезапустить отладку этого прилоюжения

Замечания по работе с Flatasmom

Здесь мне бы хотелось рассказать о двух тонкосях работы с этим ассемблером, которые поначалу меня очень сильно напрягали.

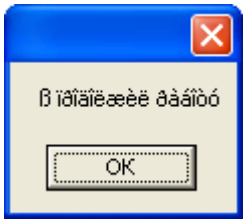
Первая тонкость связана с обработкой asm файлов ассоциированных с Flatasmom. Asm файл – это файл с текстом программы на языке ассемблер. Вроде бы кажется логичным установить ассоциацию этих файлов с Flatasmom, чтобы при двойном щелчке по ним, они сразу открывались в ассемблере. Ну так вот, подвох состоит в том, что если файл открыт таким нехитрым образом, то Flatasm откажется его компилировать, выдавая окно с ошибкой (даже если этот файл синтаксически верен):



А вот если сначала запустить Flatasm, а в нем открыть этот файл, то он скомпилируется на ура (если в нем, конечно, нет никаких ошибок).

Вторым важным моментом при работе с Flatasmom является обработка кириллических символов. Дело в том, что если в программе вы задаете

строку с кириллическими буквами, то при выводе на экран пользователю она может отобразиться в виде кракозябр:



Для того, чтобы этого не происходило нужно подключить файл win1251.inc, который находится в папке Include\Encoding в каталоге Flatasma, следующей командой:

```
include 'D:\FASM\INCLUDE\ENCODING\WIN1251.INC'
```

Глава 2. Из чего состоит программа на ассемблере

В этой главе я опишу оновные блоки, из которых состоит любая программа на ассемблере. Разумеется, это не единственные части и помимо них в программе могут использоваться другие элементы. Но это ключевые элементы.

Спецификаторы выходных файлов

Вначале мы должны сказать Flatasmy, что мы пишем: оконное или консольное приложение, библиотеку dll или что-то еще.

Flatasm позволяет создавать различные форматы выходных файлов. Тип файла, который мы хотим получить, указывается в самом начале программы после ключевого слова «Format». Основные значения этого поля представлены в таблице:

Заголовок программы	Получаемый файл
`Format MZ`	Создание приложения под MS-DOS
`Format PE`	Аналогично «`FormatPE GUI 4.0`»
`Format PE64`	64-разрядное приложение
`Format PE GUI 4.0`	графическое (оконное) приложение
`Format PE Console`	консольное[1] приложение
`Format PE Native`	драйвера
`Format PE DLL`	Создание DLL, тип DLL специфицируется с помощью префиксов Console, GUI 4.0, Native описанных ранее
`Format COFF`	OBJ-файл[2]
`Format MS COFF`	Аналогично предыдущему
`Format ELF`	OBJ-файл для gcc
`Format ELF64`	64-разрядный OBJ-файл для gcc

Разрядность выходных файлов можно специфицировать с помощью директивы use:

`Use16` — 16-разрядное приложение
`Use32` — 32-разрядное приложение
`Use64` — 64-разрядное приложение

[1] Разница между консольными и оконными приложениями чисто условная: предполагается, что консольное приложение будет взаимодействовать с пользователем с помощью консоли, а графичекое с помощью окна или окон. Хотя ничто и никто не мешает в консольном приложении использовать окно, а в оконном консоль

[2] Данные форматы используются при программировании в среде Linux. Поскольку мы будем рассматривать программирование под Windows, то мы рассматривать их не будем.

Точка входа

Точка входа – это адрес первой инструкции программы, то место, откуда программа начинает свое исполнение. Для ее задания используется ключевое слово entry, за которым следует метка, указывающая на самую первую инструкцию программы. Выглядит это примерно так:

```
Entry start
……
start:
        ;Отсюда начнется выполнение программы
```

Если точка входа не будет задана, то программа не сможет запуститься. Операционная система просто не будет знать откуда начать ее выполнение.

В случае же динамических библиотек точка входа может быть не задана. Такое послабление вызвано тем, что основная задача динамических библиотек состоит в описании процедур и функций, которые могут быть использованы в других программах. Если же у динамической библиотеки есть точка входа, то она должна представлять собой адрес процедуры dllmain, которая будет рассмотрена в главе посвященной разработке динамических библиотек.

Подключаемые файлы

Flatasm, как и язык C, позволяет использовать так называемые заголовочные файлы. Эти файлы могут быть использованы для описания в них констант и макросов. Но, в отличие от Сишных заголовочных файлов, в них не могут быть описаны прототипы используемых процедур и функций (макросы имеют несколько иной смысл).

Самым главным ключевым файлом я считаю файл win32a.inc, который в свою очередь подключает большое количество других заголовочных файлов. Именно в этих заголовочных файлах описаны такие макросы как section, invoke, proc (они будут рассмотрены нами несколько позднее) и многие другие. Поэтому я считаю написание программы без них достаточно трудоемким (но, наверное, возможным) делом.

Подключаются эти файлы так:

```
include 'win32a.inc'
```

Здесь в кавычках после слова include идет строка с именем подключаемого файла и путь до него.

Секции

Любая программа состоит из секции (сегментов). Это позволяет разбить программу на ряд логических частей: секция кода, секция данных. При работе во Flatasm вы сами создаете столько секций, сколько хотите и определяете необходимые флаги доступа к ним. Делается это так:

section ‘имя секции’ [<флаги секции>]

При выборе имени секции вы должны учитывать:

1) имя секции не должно быть больше 8 символов, это связано с форматом PE-заголовка (в частности с форматом структуры IMAGE_SECTION_HEADER).

2) Не удается создать пустую секцию (при этом программа нормально компилируется, но сам файл работать категорически отказывается). Если только это не последняя секция в файле. В случае же если это последняя секция файла, то она может быть пустой. Хотя, если подумать, кому они нужны эти самые пустые секции.

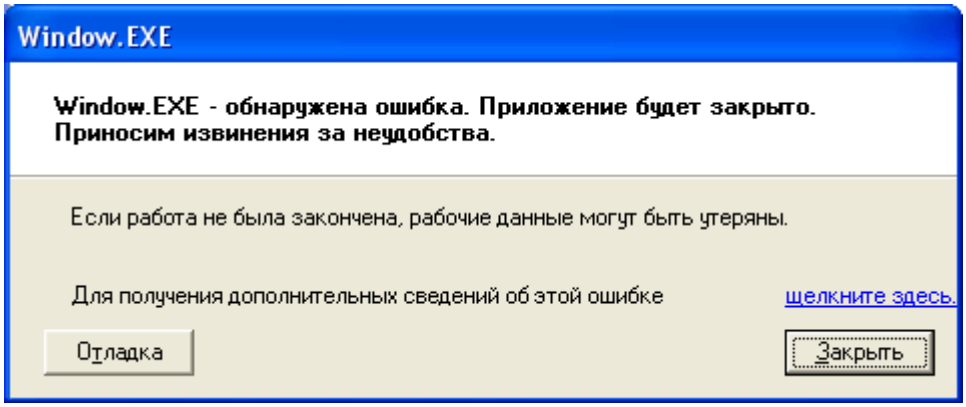
Перечислим основные флаги секции:

code – секция кода;
data – секция данных;
readable – секция, данные которой доступны для чтения;
writeable – секция, доступная на запись;
executable – секция, доступная на исполнение;
shareable – совместно используемая секция;
discardable – секция может быть выгружена из памяти;
notpageable – не подвергается страничному преобразованию;

Также для секции data возможно указание специальных флагов уточняющих вид данных:

export – таблица экспорта;
import – таблица импорта;
resource – ресурсы;
fixups – таблица перемещаемых элементов (см. главу разработка динамических библиотек);

В принципе компилятор Flatasm позволяет создавать секции без указания флагов. При этом поле Characteristics структуры IMAGE_SECTION_HEADER равно нулю. Но, при первом же обращении к этой секции (во время выполнения программы, разумеется) возникнет ошибка.



Пример описания секции кода:

```
section '.code' code readable executable
```

Пример описания секции таблицы импорта:

```
section '.idata' import data readable writeable
```

Таблица импорта

Таблица импорта определяет импортируемые из внешнего файла (как правило, библиотек dll) функции. Как правило, его размещают в отдельной секции (но это не обязательно). Приведем пример составления таблицы импорта с пояснениями:

```
;описываем секцию, в которой будет размещена таблица импорта
section '.idata' import data readable writeable
 ;перечисляем библиотеки, из которых импортируем функции
 ;и даем им (библиотекам) псевдонимы
 library kernel, 'KERNEL32.DLL',\
         user, 'USER32.DLL'
 ;перечисляем функции, импортируемые нами из библиотеки KERNEL32 (kernel)
 ;и даем им псевдонимы
 import kernel,\
        GetModuleHandle, 'GetModuleHandleA',\
        ExitProcess, 'ExitProcess'
 ;перечисляем функции, импортируемые нами из библиотеки USER32(user)
 ;и даем им псевдонимы
 import user,\
        DialogBoxParam, 'DialogBoxParamA',\
        EndDialog, 'EndDialog'
```

В данном случае приводится пример импортирования функции по наименованию, в случае импортирования функции по ординалу (числовой идентификатор функции в библиотеки) описание таблицы импорта сильно не изменится. Пример импорта функции из kernel32 представлен ниже (числа взяты произвольно исключительно для примера):

```
import kernel,\
        GetModuleHandle, 15,\
        ExitProcess, 16
```

Скелет программы на ассемблере

Обобщив все вышесказанное, можно сказать, что типичная программа на ассемблере выгладит приблизительно так:

```
//Спецификатор формата выходного файла
format PE GUI 4.0
//Точка входа
entry start
//Подключамые файлы (модули)
include 'win32a.inc'
//Секция кода
section '.code' code readable executable
start: //Код программы
//Секция данных
section '.data' data readable
       //Данные используемые программой
//Секция с таблицей импорта
section '.idata' import data readable writeable
       //Сама таблица импорта
```

Глава 3. Данные

Прежде чем начать программировать мы должны разобраться: с данными какого вида будет иметь дело наша программа. А для этого нам нужно разобраться, а какие данные вообще существуют в языке ассемблер. С этого мы и начнем.

Системы счисления

Все мы привыкли работать в десятичной системе счисления. Однако для представления данных в ЭВМ гораздо удобнее оказалась двоичная система. В двоичной системе используется всего две цифры ноль и единица (в десятичной десять: 0,1,2,3,4,5,6,7,8 и 9). Как это работает? Давайте рассмотрим на примере. В десятичной системе мы обычно представляем числа так:

$101 = 1*10^2+0*10^1+1*10^0$

А в двоичной системе числа представляются так:

$101 = 1*2^2+0*2^1+1*2^0$

Как видите, вся разница состоит в том, что в основании степеней вместо числа 10 используется число 2.

Для того чтобы отличать числа, записанные в двоичной системе счисления от чисел, записанных в десятичной системе счисления к первым в конце приписывают букву b. То есть, 101b это число пять записанное в двоичной системе счисления, а не число сто один в десятичной системе счисления.

Однако, при разработке программ работать с двоичными числами неудобно, поэтому в программировании на языке ассемблере обычно используют шестнадцатеричную систему. В ней шестнадцать цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и F. Буквами обозначают цифры, соответствующие числам 10 (A), 11 (B), 12 (C), 13 (D), 14 (E) и 15 (F). Числа записываются следующим образом:

$101 = 1*16^2+0*16^1+1*16^0$

Для того чтобы отличать числа написанные в шестнадцатеичной системе счисления от чисел, записанных в других системах счисления, к первым в конце приписывают букву h. То есть, 101h – это 257, а не сто один.

Помимо двоичной и шестнадцатеричной систем счисления иногда используют восьмеричную систему счисления (в ней всего 8 цифр от нуля до 7). Однако она не нашла широкого распространения, поэтомы мы ее не рассматриваем (хотя там все аналогично).

Биты, байты и слова

Единицей информации (и данных в том числе) является бит. Бит может принимать одно из двух значений: ноль или единицу. По своей сути бит – это один разряд двоичного числа, то есть числа, записанного в двоичной системе счисления.

Восемь бит образуют один байт. Один байт может задавать любое целое число от нуля (00h) до 255 (FFh).

Два байта (16 бит) образуют машинное слово (или просто слово). Машинное слово может задавать любое целое число от нуля (0000h) до 65535(FFFFh).

Два машинных слова (4 байта, 32 бита) образуют двойное машинное слово (или просто двойное слово). Двойное слово может задавать любое целое число от нуля (00000000h) до 2^{32}-1 (FFFFFFFFh).

Иногда в литературе встречается упоминание учетверенного машинного слова (4 слова, 8 байт, 64 бита), но оно не нашло широкого применения.

Где хранить данные

С размерами данных будем считать, что разобрались. Теперь перейдем к вопросу: а где их хранить? В ассемблере в отличие от языков высокого уровня нельзя просто объявить какую-то переменную (в ассемблере вообще нет такого понятия как переменная) и работать с ней. Любое значение нужно где-то хранить и программист должен сам решить где именно.

Регистры процессора

Регистры – это элементы центрального процессора ЭВМ, которые позволяют хранить в себе какие-то значения определенного размера. Регистры бывают нескольких типов:

Общего пользования – это те регистры, которые может использовать программист в своих программах, например, для хранения обрабатываемых данных.

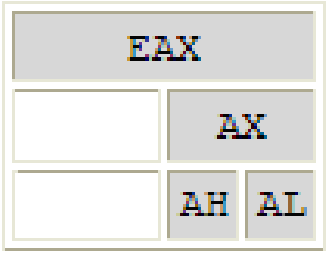
Сегментные регистры – в эпоху MS-DOS они использовались при адресации элементов памяти. Сейчас же практически не используются. Мы столкнемся с сегментным регистром только когда будем разбирать механизм обработки исключений SEH.

Служебного назначения – регистры используемые процессором и операционной системой при своей работе. Примером такого регистра может служить, например, регистр eip, в котором хранится адрес команды, которая будет выполнена после текущей команды.

Отладочные регистры – используются некоторыми отладчиками.

Разумеется, данная классификация не является полной и универсальной. Однако ее достаточно для того чтобы у вас сложилось верное представление о регистрах.

Мы будем рассматривать регистры общего назначения. К ним, как правило, относят регистры: eax/ax/ah/al, ebx/bx/bh/bl, edx/dx/dh/dl, ecx/cx/ch/cl, esi/si, edi/di. Наклонная черта в наименовании регистров означает то, что эти регистры связаны друг с другом. Для того чтобы понять как, взгляните на рисунок:



Итак: размер регистра eax составляет 4 байта (то есть, двойное слово). Младшее слово регистра eax представляет собой регистр ax. Регистр ax в свою очередь состоит из двух регистров ah и al, которые представляют собой соответственно старший и младший байты слова, хранящегося в ax. Поэтому нужно быть внимательными: если вы редактируете содержимое регистров ah, al или ax, вы тем самым изменяете содержимое регистра eax, и наоборот.

Точно такая же картина и с другими регистрами общего пользования.

Память

Регистры процессора это конечно хорошая штука, но их количество и размеры строго ограничены. В большинстве случаев их недостаточно для того, чтобы сохранить все данные, с которыми работает программа. Поэтому в большинстве случаев для хранения каких-либо данных вместо регистров используется память. Память является побайтно адресуемой. Это значит, что каждый байт памяти имеет свой уникальный адрес размером в двойное слово, по которому к данному конкретному байту можно обратиться.

Каждому процессу выделяется 4 гигабайта виртуальной памяти (именно гигабайта, это не опечатка). Виртуальная память – это память, которую гипотетически может использовать программа. Здесь думаю нужно пояснить.

Имеющаяся на вашем компьютере оперативная память – это физическая память. Размер физической памяти жестко ограничен. Предположим, у вас стоит 512 мегабайт оперативной памяти. Ну так вот, 512 мегабайт – это физическая память вашего компьютера. Для того чтобы предоставить программам и программистам, которые их пишут, больше свободы операционная система говорит каждому процессу: приятель, я могу дать тебе 4 гигабайта памяти, к которой ты можешь свободно обращаться во время своей работы. То есть, операционная система просто-напросто создает для процесса иллюзию существования 4 гигабайтов памяти (хотя на самом деле

этой памяти нет). Виртуальная память – это «иллюзорная» память, предоставляемая операционной системой каждому процессу. Процесс не знает, что он работает не с физической, а с виртуальной памятью. Его намеренно вводят в заблуждение.

Когда процесс обращается к той или иной ячейке вирутальной памяти, операционная система подхватывает это обращение, пересчитывает виртуальный (иллюзорный) адрес в физический (реально существующий) адрес, и выполняет с ней нужные программе действия. Процесс всего это не замечает. Весь процесс пересчета виртуальных адресов в физические осуществляется независимо от него операционной системой.

Поскольку суммарный объем виртуальных адресов много больше объема имеющейся физической памяти операционная система вынуждена задействовать так называемый файл подкачки. Это файл, в котором хранится часть оперативной памяти, а вместе с ней и данные некоторых процессов. Операционная система обращается к этому файлу по мере надобности тех или иных данных. Все это опять же осуществляется абсолютно прозрачно для процессов, запущенных в системе.

Из предоставленных процессу 4 гигабайт он может использовать только 2 гигабайта, то есть только половину, причем нижнюю. Вторую половину операционная система использует для своих нужд, храня в ней служебные структуры данных. То есть, каждый процесс может обращаться к памяти расположенной в диапазоне адресов 00000000h и 7FFFFFFFh? Тут опять не все так просто как хотелось бы.

В этом регионе располагаются не только данные программы, но и ее … код. А как вы хотели? Чтобы процессор мог выполнить программу он должен знать, из каких инструкций она состоит. И вот эти самые инструкции как раз и хранятся в памяти вместе со всей программой. Но погодите, скажете вы, данные и код ведь можно разместить в разных секциях. Так-то оно так. Но вот только все секции программы (кроме разве что секции ресурсов) загружаются в одну и ту же память процесса и требуют места для своего размещения.

Если программа использует какие-то динамические библиотеки, то они также загружаются в память этого процесса вместе со всеми своими секциями и библиотеками (динамические библиотеки могут использовать в своей работе другие динамические библиотеки).

Но это не главная сложность при работе с памятью. Главная сложность (точнее особенность) состоит в том, что 2 гигабайта это память, которая может быть предоставлена процессу, но она не предоставляется ему сразу при его старте. При старте процессу предоставляется тот объем памяти, который достаточен для размещения в ней данных, которые он задал в своих секциях. Если ему нужно еще памяти, то он должен использовать либо стек либо динамическую память. Изменять объем памяти, хранимой в секциях в процессе работы программы нельзя.

Задание данных в секциях

Будем считать с устройством памяти более-менее разобрались. В этом разделе поговорим о том, как описывать данные в секции данных.

Для того чтобы описать те или иные данные нужно для себя ответить на два вопроса: какого размера данные мы объявляем и хотим мы расположить в них какое-то значение или же мы просто хотим сказать системе, что в этой секции нам понадобится место для размещения данных такого-то размера. От ответов на эти вопросы зависит и то, как будут описываться данные.

В таблице ниже представлены основные ключевые слова объявления данных в зависимости от ответов на эти вопросы:

Размер данных	**Объявление данных**	**Резервирование места**
1	db	Rb
2	dw, du	Rw
4	dd	rd
6	dp, df	rp, rf
8	dq	rq

Как с ними работать? При использовании ключевых слов объявления каких-то инициализированных данных после этого ключевого слова должны через запятую следовать сами объявляемые значения. То есть если вы хотите объявить два байта со значениями ABh и CDh, то вы должны написать:

db ABh, CDh

Если же вы хотите объявить два слова со значениями 00ABh и 00CDh, то вы должны написать:

dw ABh, CDh

С остальными ключевыми словами аналогично. Как видно из предыдущего примера лидирующие нули можно опускать.

При резервировании данных после ключевого слова должно быть указано колиество элементов, под которые следует зарезервировать память. Что это значит? Поясню на примере, выражение:

rb 3

говорит о том, что необходимо зарезервировать память под три байта, а вот выражение:

rd 3

говорит о том, что необходимо зарезервировать память под три двойных слова (то есть 12 байт).

Обращаю ваше внимание на то, что в памяти ЭВМ числа хранятся задом наперед. То есть, если вы объявляете число 01020304h, то в памяти он будет храниться в виде 04030201h. Будьте готовы к этому при работе с отладчиком.

Директива file

Отдельного упоминания заслуживает ключевое слово (или, по документации Flatasm директива) file. Данная директива позволяет загрузить данные, хранящиеся в другом файле. Пример:

```
file 'text.txt'
```

Данная команда говорит о том, что компилятор Flatasm должен прочитать содержимое файла text.txt и поместить его в исполняемый модуль в качестве данных.

Ключевое слово file позволяет загрузить в исполняемый модуль не только весь файл, но и его определенную часть. В этом случае синтаксис это команды такой:

```
file ‘NameOfFile’:Offset, CountBytes
```

Здесь: NameOfFile – имя файла, из которого будут загружены данные;
Offset – номер байта, с которого седует начинать чтение (нумерация с нуля);
CountBytes – количество байт, которые следует прочитать.

Стек

Стек – это область памяти процесса, работающая по принципу LIFO (last in – first out, последним пришел – первым ушел). При работе со стеком вам не нужно беспокоиться о выделении памяти. Операционная система выделяет каждому процессу некоторое количество стековой памяти, которого достаточно для решения большинства задач.

Для работы со стеком определено всего несколько команд:

Команда	Описание
Pop	Извлечь из стека значение, лежащее на его вершине
popa или popad	Восстановить из стека регистры
popf или popfd	Восстановить из стека регистр флагов (о нем позже)
Push	Поместить какое-то значение в стек (при этом говорят, что оно кладется на вершину стека)
pusha или pushad	Поместить в стек значения регистров
pushf или pushfd	Поместить в стек значение регистра флагов

Все команды кроме pop и push вызываются без параметров. Команда pop требует указать место назначения извлекаемых данных, то есть куда следует их поместить. Например, команда:

pop ebx

извлекает из стека какое-то значение и помещает его в регистр ebx.

Команда push наоборот, требует указания данных, которые следует поместить в стек. Например:

push 5 – в стек кладется число 5,

push ebx – в стек кладется значение, хранящееся в регистре ebx.

Обращаю ваше внимание на то, что размер данных хранимых в стеке составляет 4 байта (двойное слово). Несмотря на это, при работе со стеком вполне допустимы команды вида push bx, pop ah. В этом случае в стек помещается (или извлекается) двойное слово, но перед размещением оно или раширяется (добавлением нулевых старших разрядов) или уменьшается (отбрасыванием старших разрядов). Поясню, на примере. Допустим, в регистре ebx содержится значение 01020304h, тогда после следующих команд:

```
push ebx
xor ebx, ebx ;Обнуление регистра ebx
pop bx
```

в регистре ebx будет содержаться значение 00000304h, то есть старшие два байта потеряны. Поэтому будьте внимательными.

Команда mov

Команда mov служит для пересылки (точнее копирования) данных. В общем, синтаксис этой команды выглядит следующим образом:

mov <приемник данных>, <источник данных или сами даные>.

Рассмотрим несколько вариантов использования этой команды. Наиболее простое:

mov ebx, 5

Этой командой в регистр ebx помещается значение 5. При использовании такого синтаксиса нужно следить за тем, чтобы приемник (в данном случае регистр) имел достаточный размер, для того чтобы вместить в себя данные. Например, команда

mov ah, 0102h

ошибочна, так как размер регистра ah составляет один байт, а мы пытаемся записать в него данные размером в два байта.

Идем дальше.

mov ebx, eax

Данная команда помещает в регистр ebx значение, хранящееся в регистре eax. При этом содержимое регистра eax остается неизменным. При

использовании такого синтаксиса нужно следить за тем, чтобы оба регистра имели одинавый размер, то есть команды вида

mov ebx, ax
mov bx, eax

ошибочны.

Теперь вновь поговорим о памяти.

mov [ebx], eax

Данная команда по адресу, хранящемуся в регистре ebx, записывает значение, хранящееся в регистре eax. То есть, если в регистре ebx хранится значение 01h, а в регистре eax значение ABCDh, то в память по адресу 01h, будет записано значение ABCDh. При этом содержимое регистров останется неизменным. Аналогично команда:

mov ebx, [eax]

помещает в регистр ebx значение, хранящееся в памяти по адресу, хранящемуся в регистре eax[1].

При работе с памятью следует помнить, что та ячейка памяти, к которой вы хотите обратиться должна быть выделена операционной системой. То есть, если вы, например, хотите прочитать содержимое памяти по адресу 00ABCDh, а эта ячейка памяти не выделена операцмионной системой, то произойдет ошибка доступа (access violation). Аналогично, если вы хотите записать какое-то значение в памяти, то соответствующая ячейка памяти должна не просто быть выделена операционной системой, но и у нее должны быть установлены права на запись (устанавливается на уровне секции, атрибут writeable). В противном случае при попытке записи в нее также возникнет ошибка доступа.

Очень часто при работе с памятью требуется явно указать размер данных, с которыми мы работаем. Например, в команде:

mov [eax], 05h

flatasm не может знать данные какого размера мы хотим записать по адресу, хранящемуся в регистре eax (байт, слово или двойное слово). Аналогично и команда:

mov ebx, [eax]

Flatasm также не может знать данные какого размера мы хотим прочитать из памяти и поместить их в регистр ebx. В таких ситуация нужно подсказать ему, путем явного указания размера данных с которорыми мы хотим работать. Делается это с помощью трех ключевых слов: byte (байт), word (слово) и dword (двойное слово). То есть, если в двух предыдущих командах мы подразумевали слова, то эти команды должны быть записаны в виде:

mov [eax], word 05h
mov ebx, word [eax]

Обращаю ваше внимание на то, что в языке ассемблер перемещение данных из памяти в память недопустимо. То есть команда вида

[1] В этом случае говорят, что регистр eax хранит указатель на данные

mov [eax], word [ebx]

ошибочна. Если вам все-таки нужно осуществить такое копирование вы должны использовать промежуточный регистр. Например, так:

mov dx, word [ebx]
mov [eax], word dx

Более сложные варианты команды mov

Как вы, наверное, уже поняли, команда mov только кажется простой. Помимо обращения по заранее известным адресам, в этой команде можно сразу вычислять нужный адрес, так сказать, не отходя от кассы. Взглянем на следующий вариант применения этой команды:

mov eax, dword [ebx+4*ecx]

Что здесь происходит? Здесь в регистр eax помещается двойное слово, адрес которого в памяти вычисляется прямо здесь. То есть, процессор сначала вычисляет значение выражения ebx+4*ecx, затем считывает по полученному адресу двойное слово и помещает его в реистр eax. И все это одна команда ассемблера.

Причем вычислять можно как адрес источника, так и адрес приемника данных.

Конечно, вы не можете использовать для вычисления нужного вам адреса произвольные выражения. Ниже приводится обобщенный вариант допустимой формулы расчета адреса в команде mov:

Регистр_или_метка+Число*Регистр ±Число

То есть вы можете использовать команды вида:

mov eax, dword [ebx+4]
mov eax, dword [ebx+4*ecx-6]
mov eax, dword [OurData+8*ecx+6]
mov eax, dword [8*ebx-4]
mov eax, dword [ebx+4*ecx]

и так далее.

Обращаю ваше внимание на то, что в качестве множителя может выступать только положительное число, являющееся степенью двойки, то есть 2, 4, 8 и так далее.

А вот команды вида:

mov eax, dword [ebx+3*ecx]
mov eax, dword [ebx+2*ecx+4*edx]
mov eax, dword [ebx-2*ecx]

являются ошибочными.

Команда обмена

Давайте рассмотрим такую типичную ситуацию: нужно обменять значениями два регистра процессора. Это можно сделать с помощью команды mov и еще одного регистра. Но лучше воспользоваться для этого специальной командой ассемблера xchg. Вот ее прототип:

xchg операнд1, операнд2

Эта команда как раз и осуществляет обмен значениями между операндами.

Операндами могут быть регистры процессора. А также одним из операндов может быть ячейка пмяти.

Метки

Неявно с метками мы уже сталкивались, когда говорили о точке входа. С помощью меток можно помечать места в коде или данные. Метки для кода выглядят приблизительно так:

```
;код
firstlabel:
    ;код
secondlabel:
    ;код
……
```

То есть, метки в языке ассемблер аналогичны меткам во многих языках программирования высокого уровня. Но, если все учебники по языкам высокого уровня в один голос твердят, что использование меток уродует код, и потому крайне нежелательно, то, как мы увидим далее в языке ассемблер без них практически невозможно обойтись.

С помощью меток помечаются также данные. Это позволяет не только дать им более-осмысленные имена, но и упростить к ним доступ. Пример объявления данных с меткой:

```
OurData: db 0ABh
```

При использовании меток данных символ двоеточие можно опускать.

Здесь мы объявили один байт со значение ABh, и обозначили его меткой OurData.

Теперь для того, чтобы обратиться к значению этого байта может быть использована команда вида:

mov ebx, byte [OurData]

При этом в регистр ebx будет помещено значение ABh. А вот команда:

mov ebx, OurData

поместит в регистр ebx адрес объявленного нами байта данных.

Аналогично для того, чтобы изменить значение этих данных может быть использована команда вида:

mov [OurData], byte 05h

При этом, значение ABh, хранящееся по адресу OurData будет заменено на значение 05h. Команда

mov OurData, byte 05h

ошибочна, так как сама метка изменена быть не может.

Указатели

По своей сути указатель представляет собой адрес в памяти, по которому хранятся те данные, на которые он указывает. Поэтому, когда мы говорим, что в каком-то регистре или по какому-то адресу хранится адрес каких-то данных, то это эквивалентно тому, что там же хранится указатель на те же самые данные.

Поскольку мы рассматриваем исключительно 32-рязрядные системы, то можно сказать, что указатель – это обычное целое число размером в двойное слово (4 байта).

Над указателями допустимы все операции, которые возможны над целыми числами. Однако следует быть осторожным, так как обращение по неверному адресу приводит к ошибке доступа со всеми вытекающими (как вариант крах всего приложения).

Единственная разница между указателем и меткой состоит в следующем: метки используются для помечения кода и данных в тексте программы, в процессе компиляции программы метки удаляются из программы, а в тех местах программы, где происходит обращение к данным или коду посредством этих меток, они заменяются указателями. В результате после компиляции программы в ней не остается меток, но есть указатели, полученные на основе этих самых меток. Примеры таких команд приводились в предыдущем разделе.

Вы можете и сами получать указатели на данные по соответствующим меткам на эти данные. Для этого используется команда lea. Вот ее прототип:

lea регистр, [метка]

пример использования

lea eax, [OurData]

В результате выполнения этой команды в регистр eax будет помещен адрес (читай: указатель) данных или кода, помеченных меткой OurData.

При работе с этой командой следует помнить, что, так как на 32-разрядных системах размер адреса составляет двойное слово, то для правильного получения адреса размер регистра, в который помещается адрес, также должен составлять двойное слово.

Еще один пример использования этой команды:

lea eax, [ebx]

В данном случае эта команда эквивалентна команде:

mov eax, ebx

Теперь поговорим о языках программирования высокого уровня.

Почему в них существует такое многообразие указателей? То есть указатели на символы и целые строки являются разными типами данными: char* и int* соответственно. На самом деле это «особенность» синтаксического анализатора кода, входящего в состав компилятора. Она нужна для того, чтобы контролировать правильность написаия программы и не позволить программисту допустить ошибку из-за неправильной интерпретации указателей. В ассемблере такого контроля нет. Вот почему все операции по приведению типов указателей нужны исключительно программисту (для повышения читаемости кода и его правильного понимания) и синтаксическому анализатору компилятора (для контроля за программистом и оказания ему помощи в виде указания возможных мест неправильной интерпретации указателей и всплывающих подсказок).

При работе на языке ассемблер вся ответственность по правильной интерпретации указателей целиком и полностью ложится на плечи программиста. Поэтому при работе с указателями нужно быть внимательным.

Что такое NULL? В языках высокого уровня ошибочные указатели принято обозначать ключевым словом NULL (или nil в случае языка Pascal). Что это? Если заглянуть в заголовочный файл WinDef.h, то там можно увидеть:

```
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
```

То есть, NULL – это нулевой указатель (или просто нуль). Дело в том, что обращение по нулевому указателю всегда приводит к ошибке. Это так называемая «сторожевая» зона в памяти приложений.

Вот вам и ответ, что такое NULL, это просто нуль, без палочки.

Константы

Константы также как и метки призваны облегчить жизнь программисту. Объявление константы выглядит следующим образом:

<имя константы> = <значение>

Пример:

Constant=05h

Такое объявление константы может быть размещено как в секции кода, так и вне какой-либо секции. После объявления константы допустимы команды вида:

mov eax, Constant

Данная команда эквивалента команде

mov eax, 05h

То есть константы во Flatasme работают аналогично директиве #define в языке Си. Для тех, кто незнаком с этим языком или подзабыл его, поясню: отдельно эта константа нигде не создается, просто при компиляции программы все ее вхождения в текст программы замещаются значением этой константы.

Разумеется команды вида

mov Constant, 0ABh

и

mov [Constant], 0ABh

ошибочны.

Глава 4. Целочисленная арифметика

Сложение чисел

Для сложения чисел предусмотрено несколько команд. Самая простая из них это команда инкремента inc, в случае работы с регистрами она записывается так:

inc eax

Данная команда увеличивает содержимое регистра eax на единицу. Команда inc может быть использована для работы со значениями, хранящимися в памяти (хоть это и не рекомендуется). В этом случае она записывается так:

inc byte [OurData]

Здесь слово byte специфирует размер операнда (говорит, что мы работаем с данными размером в один байт).

Вторая команда (add) складывает два числа и записывает результат сложения на место первого операнда. Рассмотрим несколько форм ее записи. Первая, наиболее простая:

add eax, 05

Данная команда к содержимому регистра eax жобавляет значение 5. Результат сложения при этом сохраняется в регистре eax. Команда

add 05, eax

является ошибочной.

Второй вариант записи:

add eax, ebx

Здесь к содердимому регистра eax прибавляется содержимое регистра ebx. Результат сложения сохраняется в регистре eax. При этом необходимо следить за тем, чтобы оба операнда имели одинаковый размер. То есть команды вида

add eax, bx

или

add ax, ebx

являются ошибочными.

И наконец, третий вариант записи:

add dword [OurData], ebx

или

add ebx, dword [OurData]

Здесь с помощью ключевого слова dword мы говорим, что размер данных, с которыми мы работаем, составляет 4 байта (двойное слово).

При работе с памятью также нужно следить за тем, чтобы операнды имели одинаковый размер.

Команды, в которых оба операнда это ячейки памяти недопустимы. А вот команда вида:

add byte [OurData], 05

вполне законна (если конечно память, на которую указывает OurData, доступна на запись).

И наконец, последняя команда сложения adc очень похожа на команду add. Обе команды имеют один и тот же синтаксис и правила применения. Единственная разница между ними состоит в том, что команда adc учитывает флаг переноса CF. Что это значит?

Флаг переноса является частью регистра флагов[1] (нулевой бит). Он взводится процессором в том случае, если результат арифметических действий вышел за пределы разрядной сетки числа. Поясню подробнее.

Как вы уже знаете, каждому числу отводится определенное число бит, в которых он может хранить свое значение. Напрмер, для байта 8 бит, для слова 16 бит, для двойного слова 32 бита. Ограниченность числа бит отводимых под хранение числа приводит к тому, что в нем можно хранить значения из определенного диапазона. Например, для байта от 00 до 255 (FFh), для слова от 00 до 65535 (FFFFh) и так далее.

А теперь представьте, что произойдет, если мы попытаемся сложить два таких числа, результат сложения которых выходит за определенные рамки. Давайте, для простоты ограничимся размером в один байт (для дугих размерностей ситуация будет аналогичная). Возьем, например числа FDh (253) и 06h:

FDh + 06h = 0103h

Как видите, у нас получился результат размером в два байта. Но у нас всего один байт под число. Это приводит к тому, что старшая часть числа, то есть старший байт будет отброшен. В результате мы получим:

FDh+06h=03h

Или в десятичной системе счисления:

253 + 6 = 3

Интересная получается арифметика, не правда ли?

Тут резонно возникает вопрос: может ли программист как-то отслеживать возникновение подобных ситуаций. Ответ: может. Дело в том, что когда в процессе работы программы возникают такие ситуации, процессор взводит (устанавливает в значение 1) флаг переноса CF. Программист путем проверки состояния этого флага[2] может отлавливать такие ситуации.

Это все конечно хорошо и очень интересно, но при чем тут команда adc? Отвечаю: единственное отличие команды adc от команды add состоит в том, что первая проверяет состояние флага переноса и прибавляет его к результату сложения. Для лучшего понимания рассмотрим два примера:

Первый:

---

[1] О нем мы еще поговорим

[2] О том, как проверять состояния тех или иных флагов мы поговорим в главе посвященной передаче управления

```
mov ah, 0FDh  ;1
mov bh, 06h   ;2
add ah, bh    ;3
add ah, 00    ;4
```

Здесь, в третьей строчке происходит переполнение разрядной сетки рассмотренное выше. В результате него в регистре ah[1] оказывается значение 03h и взводится флаг переноса. Четвертая строка никак не изменяет содержимое регистра ah, а вот флаг переноса она сбрасывает (устанавливает в значение ноль). А вы думали он так и останется взведенным до конца работы программы? Нет, на его значение влияют многие ассемблерные инструкции. Впрочем, сейчас не об этом.

Теперь взглянем на второй пример:

```
mov ah, 0FDh  ;1
mov bh, 06h   ;2
adc ah, bh    ;3
adc ah, 00    ;4
```

Здесь мы команды add заменили на команды adc. Что изменилось? Давайте посмотрим. В строке 3, как и прежде происходит переполнение разрадной сетки и в регистре ah оказывается значение 03h. Стоп, скажете вы, а как же флаг переноса, разве он не должен был прибавиться к этому значению? Дело в том, что на момент выполнения команды в строке 3 флаг переноса сброшен (то есть равен нулю), а вот уже в результате ее выполнения он взводится. Поэтому третьи строки в обоих примерах идентичны, между ними нет никакой разницы в плане получаемых результатов. А вот результаты четвертых строк будут различаться.

На момент выполнения команды в четвертой строке флаг переноса взведен. Поэтому команда adc добавит его вместе с нулем к значению, хранящемуся в регистре ah, и там вместо 3 окажется 4. При этом сам флаг переноса будет сброшен.

Как видите, за кажущейся простотой операции сложения скрываются многие не вполне очевидные, но весьма любопытные нюансы. Но на этом «чудеса» машинной арифметики не заканчиваются.

Знаковые и беззнаковые числа

Ответьте мне на вопрос: как компьютер отличает отрицательные числа от положительных. Кто-то, вспомнив базовый курс информатики, скажет: по старшему знаковому биту. Для тех, кто подзабыл, напомню.

При работе со знаковыми числами старший бит числа рассматривается как знаковый бит. Если он равен единице, то число признается отрицательным, если же он равен нулю, то число положительным. В случае с

[1] При этом содержимое других разрядов регистра eax, не являющихся разрядами регистра ah остается неизменным

числами размером в один байт это означает что: байты от 00h до 7Fh описывают положительные числа, а байты от 80h до FFh отрицательные числа. А теперь повнимательнее посмотрите на эти интервалы и ответьте мне на вопрос: какие числа больше: отрицательные или положительные? Правильный ответ: любое отрицательное число больше любого положительного числа.

Теперь поговорим о том, как представляются отрицательные числа. Они представляются в виде так называемого дополнения. Для того чтобы понять что это такое вспомним математическое тождество:

$$a + (-a) = 0$$

На основе этого тождества и формируются отрицательные числа. То есть они формируются таким образом, чтобы при сложении с этим же числом противоположного знака получался ноль[1]. Например, число -2 будет иметь вид FEh, так как:

02h + FEh = 00h

Аналогично число -23 (23 = 17h) будет иметь вид: E9h, так как:

E9h + 17h = 00h

Нужно понимать, что с точки зрения процессора разница между знаковыми и беззнаковыми числами невелика (операции сложения и тех и других выполнябтся одними и теми же командами). Это лишь вопрос интерпретации. Поэтому байт CDh может быть интерпретирован как 205 (в случае беззнакового числа), так и -51 (в случае числа со знаком). Поэтому при работе с числами нужно быть внимательными.

Команды сложения, рассмотренные в прошлом разделе, работают с обоими числами (беззнаковыми и знаковыми) одинаково. Однако, при сложении чисел одного знака может произойти неожиданная смена знака. Давайте для примера сложим два числа со знаком: 100 (64h) и 30 (1Eh). Оба числа положительные. В результате сложения:

64h + 1Eh = 82h

Но данное число (имеется ввиду результат) находится уже в области отрицательных чисел. И будет проинтерпретировано процессором как число -126. То есть мы получили, что:

100 + 30 = -126

Для того чтобы предоставить программисту возможность выявления и обработки подобного рода ситуаций, процессор при их возникновении взводит (устанавливает в значение 1) флаг переполнения OF (11 считая от нуля) регистра флагов. Данный флаг взводится не только при выполнении операции сложения, но и при выполнении операции вычитания, которая так или иначе ведет к нарушению знака.

Проверка значения этого флага позволяет обнаруживать такие казусы и по возможности исправлять их.

---

[1] Точнее, чтобы происходило такое переполнение разрядной сетки (см. предыдущий раздел), при котором после отсечение «лишних» разрядов в «сухом остатке» остался ноль.

Вычитание

Команды вычитания по своему синтаксису очень похожи на команды сложения, поэтому подробно останавливаться на их синтаксисе я не буду. Всего существует три команды вычитания: dec, sub и sbb. Причем эти команды кроме последней не делают различия между операндами со знаком и операндами без знака. Рассмотрим эти команды по отдельности.

Команда dec представляет собой команду декремента, то есть уменьшения значения операнда на единицу.

Команда sub имеет тот же синтаксис что и команда add:

sub операнд1, операнд2

Данная команда вычитает из значения операнд1 значение операнд2 и записывает его результат на место операнд1.

При этом, если в результате вычитания получается отрицательное число (то есть операнд1 меньше чем операнд2), то процессор взводит флаг переноса CF.

Команда sbb имеет тот же синтаксис:

sbb операнд1, операнд2

Она делает то же самое, что и команда sub, но, из результата дополнительно вычитает еще значение флага переноса.

Перемножение

Для перемножения чисел используется две команды mul и imul. Эти две команды в принципе идентичны. Единственное различие между ними сотстоит в том, что команда mul интерпретирует аргументы как числы без знака, а команда imul как числа со знаком. Синтаксис команд следующий:

mul операнд1

В качестве операнда может выступать регистр или ячейка памяти. Месторасположение второго операнда и результата определяется по таблице ниже в зависимости от размера операнда1:

Размер операндов	Второй операнд	Результат
Байт	Al	ax
Слово	Ax	dx:ax
Двойное слово	Eax	edx:eax

Запись dx:ax следует интерпретировать так: старшая часть результата находится в регистре dx, а младшая часть в регистре ax.

Обе команды в том случае, если результат полностью помещается в младшей половине отведенного места, сбрасывают флаги CF и OF. Если же результат не помещается в младшей половине, то оба этих флага устанавливаются в значение 1.

Для перемножения на -1 предусмотрена отдельная команда neg, вот ее синтаксис:

neg операнд

Данная команда считывает операнд, умножает его на -1 и записывает результат на место операнда. В качестве операнда может выступать регистр или ячейка памяти.

Деление

Для целочисленного деления чисел используется две команды div и idiv. Первая осущствляет деление чисел без знака, а вторая - со знаком. Их синтаксис таков:

div делитель

В качестве делителя может выступать регистр или ячейка памяти. Месторасположение делимого и результатов деления (остатка и частного) определяется по таблице в зависимости от размера делителя:

Размер делителя	**Делимое**	**Частное**	**Остаток**
Байт	Ax	al	ah
Слово	dx:ax	ax	dx
Двойное слово	edx:eax	eax	edx

Команды увеличения размеров чисел

В языке ассемблер предусмотрено несколько команд для увеличения размеров чисел. Причем данные команды одинаково хорошо работают как со знаковыми числами так и с числами без знака.

Достигается такое расширение путем распространения старшего бита расширяемого числа (знакового бита) на все добавляемые биты. Все эти команды не имеют операндов. Вот они (изменяемые регистры представлены в скобках):

cbw – расширение байта (al) до слова (ax);

cwd - расширение слова (ax) до двойного слова (dx:ax);

cwde – расширение слова (ax) до двойного слова (eax);

cdq – расширение двойного слова (eax) до учетверенного слова (edx:eax).

Глава 5. Команды сдвига и логические команды
Команды сдвига

Все команды сдвига имеют одинаковый синтаксис:

команда операнд, количествосдвигов

Здесь *операнд* задает месторасположение значения, к которому будет применяться операция сдвига. На этом же месте будет находиться результат выполнения операции

Операнд должен быть регистром, а вот количество сдвигов может указываться либо непосредственно в команде, либо в регистре, либо в ячейке памяти.

Количество сдвигов определяет количество бит, на которые следует сдвинуть операнд.

Направление сдвига и его механизм зависит от команды. В таблице ниже представлены основные команды сдвига и дано их краткое описание.

Команда	**Описание**
rcl	Циклический сдвиг влево через перенос. Похожа на команду shl. Отличается от нее тем, что справа вписывается не ноль, а значение флага CF
rcr	Циклический сдвиг вправо через перенос. Похожа на команду shr. Отличается от нее тем, что слева вписывается не ноль, а значение флага CF.
rol	Циклический сдвиг влево. Содержимое операнда сдвигается влево. Сдвигаемые влево биты записываются в этот же операнд справа.
ror	Циклический сдвиг вправо. Сдвигаемые вправо биты записываются в этот же операнд слева.
sal	Аналогична команде shl, но в отличие от нее в том случае, если операнд меняет знак, взводит флаг OF.
sar	Арифметический сдвиг вправо. Содержимое операнда сдвигается вправо. Слева в операнд вписываются нули. Команда сохраняет знак числа, восстанавливая его после сдвига каждого очередного бита. Выдвигаемый бит переходит во флаг CF.
shl	Логический сдвиг влево. Содержимое операнда сдвигается влево. Выдвигаемый бит переходит во флаг CF. Справа в позицию младших бит записываются нули
shr	Логический сдвиг вправо. Содержимое операнда сдвигается вправо. Выдвигаемый бит переходит во флаг CF. Слева в позицию старшего (знакового) бита вписывается нуль.

При ипользовании операции сдвига следует быть осторожным, так как при их неосторожном использовании можно потерять знак числа.

Можно легко убедиться в том, что сдвиг влево на один бит некоторого числа эквивалентен умножению этого числа на два. В самом деле, возьмем для примера число 3, в двоичной системе счисления оно запишется в виде 11b. Теперь сдвинем его на один бит влево, получим: 110b. Или в десятичной системе счисления 6. Сдвинем еще на один бит влево, получим 1100b, или 12 в десятичной системе счисления.

Аналогично, сдвиг вправо на один бит, эквивалентен делению на два.

Логический тип данных

Все мы привыкли использовать в языках программирования высокого уровня логический тип данных. Это такой тип, переменные которого могут принимать только одно из двух допустимых значений либо TRUE, либо FALSE, третьего как говорится не дано.

В языке ассемблер же типов нет, есть только числа, отличающиеся размером отводимой под них памяти. Однако Flatasm позволяет использовать ключевые слова TRUE и FALSE. Достигнуто это за счет введения соответствующих констант в заголовочном файле «INCLUDE\EQUATES\KERNEL32.INC». А указано в нем буквально следующее:

```
TRUE = 1
FALSE = 0
```

Получается, что логический тип данных представляется с помощью тех же самых чисел, причем, строго определенных чисел.

Более того, если почитать описание возвращаемых параметров функций, явно возвращающих значение типа bool (это следует из их прототипа), то мы увидим фразы вроде этой: функция возвращает ненулевое значение в случае успеха и ноль в случае ошибки.

Отсюда следует, что при необходимости интерпретировать какое-то число как логический тип, оно интерпретируетя по следующим правилам: если это число равно нулю, то оно признается значением FALSE, если же оно отлично от нуля, то признается значением TRUE.

Такая формулировка менее определенная, чем первая (ноль и единица), однако избегает неопределенностей, связанных с попыткой интерпретации чисел отличных от нуля и единицы.

Логические команды

Логические команды имеют следующий прототип:

команда операнд1, операнд2

При этом результат выполнения команды записывается на место операнд1.

Один из операндов может представлять собой ячейку памяти.

Все логические команды являются битовыми командами. Это значит, что они выполняются не над целыми числами, а над их битами. Нулевой бит первого числа с нулевым битом второго числа, первый бит первого числа с первым битом второго числа и так далее. Понятно, что как и во многих других командах, рассмотренных ранее операнды должны иметь одинаковый размер.

Ниже представлена таблица истинности основных логических команд ассемблера:

Операнд1	**Операнд2**	**or**	**and**	**xor**
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Если вы посмотрите на таблицу истинности, то увидите, что команда xor устанавливает очередной бит в единицу, только тогда когда соответствующие биты операндов отличаются друг от друга. Поэтому ее очень часто используют для обнуления регистров. То есть команда:

xor eax, eax

эквивалента команде:

mov eax, 00

Отдельного упоминания заслуживают команды test и not.

Первая команда очень похожа на команду and. Но, в отличие от нее команда test не изменяет сами операнды, а просто устанавливает соотвествующие флаги регистра флагов.

Команда not имеет прототип:

not операнд

Эта команда инвертирует биты операнда, то есть нули заменяет на единицы, а единицы соответственно на нули.

Глава 6. Команды передачи управления

В этой главе мы рассмотрим команды передачи управления как условные, передающие управление только в случае наступления какого-либо условия, так и безусловные, передающие управления вне зависимости от наступления каких-либо условий.

Команда jmp

Пожалуй, это самая простая из всех команд передачи управления. Вот ее синтаксис:

jmp метка

Здесь единственным операндом является метка, на которую следует передать управление. Вообще эта команда аналогична команде goto в языках высокого уровня. И работает точно так же.

Регистр флагов и команды условного перехода

Регистр флагов EFLAGS представляет собой 32-разрядный регистр процессора, в котором каждый бит выполняет строго определенную функцию. Как правило, эти флаги характеризуют определенные состояния процессора. С двумя такими флагами (OF и CF) мы уже сталкивались, когда говорили о целочисленной арифметике. Теперь рассмотрим некоторые другие флаги.

Мы не будем рассматривать все флаги этого процессора, а только наиболее часто используемые.

Порядковый номер бита флага	**Символьное обозначение флага**	**Название флага**	**Описание флага**	**Команда перехода по установленному флагу**	**Команда перехода по сброшенному флагу**
00	CF	переноса	Произошло переполнение разрядной сетки числа	jc	jnc
02	PF	четности	8 младших битов результата содержат четное	jp	jnp или jpo

			число единиц		
06	ZF	нуля	Результат равен нулю	jz	jnz
07	SF	знака	Результат меньше нуля	js	jns
11	OF	переполнения	Произошло непредвиденное изменение знака результата	jo	jno

Все команды в данной таблице имеют тот же синтаксис что и команда jmp. Но, в отличие от последней срабатывают только при выполнении определенного условия. Например, переход по команде jc срабатывает, только если установлен флаг переноса, а по команде jnc, наоборот, срабатывает только в том случае, если флаг переноса сброшен. Аналогично и другие команды из этой таблицы.

Для работы непосредственно с самим регистром флагом предусмотрно всего две команды: pushf – сохранить регистр флагов в стеке и popf – извлечь из стека значение регистра флагов.

Команда cmp

Команда cmp сравнивает два операнда и имеет следующий прототип:

cmp операнд1, операнд2

Обычно она используется совместно с командами условного перехода, которые позволяют выявить результат сравнения операндов. Все эти команды срабатывают только при выполнении определенного условия сравнения. Они представлены в таблице ниже:

Команда	**Тип операндов**	**Результат сравнения нужный для перехода**
ja	беззнаковый	операнд1>операнд2
jae	беззнаковый	операнд1>=операнд2
jb	беззнаковый	операнд1<операнд2
jbe	беззнаковый	операнд1<=операнд2
je	любой	операнд1=операнд2
jg	знаковый	операнд1>операнд2
jge	знаковый	операнд1>=операнд2

jl	знаковый	операнд1<операнд2
jle	знаковый	операнд1<=операнд2
jna	беззнаковый	то же что и jbe
jnae	беззнаковый	то же что и jb
jnb	беззнаковый	то же что и jae
jnbe	беззнаковый	то же что и ja
jne	любой	операнд1<>операнд2
jng	знаковый	то же что и jle
jnge	знаковый	то же что и jl
jnl	знаковый	то же что и jge
jnle	знаковый	то же что и jg

Присмотревшись к командам из этой таблицы можно заметить, что формируются они по определенным правилам: так буква n в названии команды означает отрицание. То есть jne противоположна команде je, команда jna команде ja и так далее. А буква e в названии команды добавляет к ней проверку на равенство операндов. Эти тонкости упрощают запоминание этих команд.

Все эти команды имеют тот же синтаксис что и команда jmp.

Команды call и ret

Команда call по своему назначению и синтаксису очень похожа на команду jmp. Единственное ее отличие состоит в том, что она (команда call) перед передачей управления сохраняет в стеке значение регистра eip. Как вы знаете, это регистр хранит в себе адрес команды, которая будет выполнена вслед за текущей командой. Этот адрес, сохраненный в стеке, принято называть адресом возврата.

Команда ret выполняет обратную операцию. Она извлекает из стека двойное слово и передает управление по адресу, указанному в этом двойном слове. Понятно, что эта команда ожидает увидеть на вершине стека адрес возврата, помещенный туда командой call.

Команда ret имеет два варианта синтаксиса с параметром и без параметра. В случае синтаксиса без параметра она делает то, что должна делать (передает управление на хранящийся на вершне стека адрес возврата). В случае же когда указан параметр, она извлекает с вершины стека адрес возврата, снимает со стека указанное в параметре количество байт, и только после этого передает управление на адрес возврата. Поясню на примере:

```
ret 8
```

В данном случае перед передачей управления на адрес возврата с вершины стека будет снято 8 байт.

Основное назначение команды call – это вызов процедур и функций прграммы, описанных в этом же модуле, либо в других подгружаемх модулях (например, динамических библиотеках dll). Соответсвенно команда ret используется для выхода из процедуры и возврата управления основной программе, вызвавшей эту процедуру.

Соглашения о вызовах

Команды call и ret это хорошо. Но возникает вопрос как вызываемой функции передать аргументы и как забрать результат ее работы. Для стандартизации этого было предложено несколько так называемых соглашений о вызовах, которые описывают, как функции следует передавать параметры, и как программе будет возвращен результат ее работы. Ниже представлены основные соглашения.

cdecl (объявление языка C). Параметры передаются функции через стек, причем в обратном порядке, то есть первый параметр функции помещается в стек последним. После работы функции стек должен быть очищен вызвавшей функцией. Из названия соглашения видно, что он широко используется в программах написанных на C.

stdcall (стандартный вызов). Как и в случае cdecl, параметры передаются через стек в обратном порядке, однако очищает стек вызванная функция перед возвращением управления в вызвавшую ею функцию (делается это командой ret с параметром). Практически все функции wi32 API вызываются именно по этому соглашению[1].

fastcall (быстрый вызов). Стек, как и в случае stdcall очищается вызываемой функцией. Отличие состоит в том, что првоначально предпринимается попытка передать пременные через регистры и только, если это оказывается невозможным, задействуется стек. Данное соглашение используется в программах, написанных на Delphi.

Результат выполнения функции возвращается в регистре eax.

В своих программах вы можете использовать любое из вышеперечисленных или какое-либо свое соглашение о вызове. Однако нужно помнить, что если с вашей программой будет работать другой программист, то использование малоизвестных или вообще никому кроме вас неизвестных соглашений о вызове сильно усложнит его работу. И вряд ли он скажет вам спасибо за это.

[1] Честно говоря, я не знаю ни одной функции win32 API, которая использовала бы какое-либо другое соглашение о вызове. Но это не значит, что таких функций нет.

Команды stdcall и invoke

Команды stdcall и invoke строго говоря не являются чистыми ассемблерными командами. Это скорее обертки, которые упрощают написание программ.

Поясню их работу на конкретном примере. Допустим, вы хотите вызвать некоторую процедуру, ее адрес находится в регистре eax. Этой процедуре вы передаете один единственный параметр нуль. На чистом ассемблере вызов такой процедуры будет состоять из двух команд:

```
push 0
call eax
```

Команда stdcall позволяет записать такой вызов процедуры в более коротком виде:

```
stdcall eax, 0
```

При этом в скомпилированном модуле мы получим тот же самый код, что и в предыдущем примере.

Вы можете передавать в процедуру и больше параметров:

stdcall eax, 0, 1, 2, ….

Синтаксис этой команды следующий:

stdcall Адрес [перечень параметров через запятую]

Команда invoke имеет такой же синтаксис но в плане упрощения программирования идет еще дальше. Команды call и stdcall требуют указания адреса или метки вызываемой функции, а как быть, если он нам неизвестен.

Например, мы хотим вызвать процедуру из внешнего модуля (динамической библиотеки), описанного в таблице импорта. Точный адрес этой функции нам неизвестен[1]. В таких условиях при использовании команд call или stdcall нам придется как-то самим его определять, а это весьма затратно. Команда invoke и позволяет нам решить эту проблему. Она имеет тот же синтаксис, что и команда stdcall, но вместо адреса вызываемой функции требует указания ее имени. Вот как, например, будет выглядеть вызов функции ExitProcess[2] с параметром нуль:

```
invoke ExitProcess, 0
```

[1] Он и не может быть нам известен, так как меняется от одной версии Windows к другой. А начиная с Windows Vista, он меняется при каждой загрузке системы. Этот адрес записывается в таблицу импорта (вот почему секция с ней должна быть доступна на запись), оттуда он и считывается при вызове функции.

[2] Предварительно эта функция должна быть описана в таблице импорта

При этом компилятор сам найдет адрес этой функции и сформирует необходимый код ее вызова.

При работе с функциями вызова процедур следует помнить, что Flatasm никак не контролирует правильность передачи параметров в вызываемую процедуру. Поэтому будьте внимательными.

Глава 7. Реализация конструкции языков высокого уровня

В этой главе мы поговорим о том, как на языке ассемблер могут быть реализованы различные конструкции языков высокого типа, такие как ветвления, циклы, процедуры и функции, а также некоторые сложные типы данных, такие как массивы и структуры.

Ветвления
Оператор if()

Рассмотрим такую конструкцию на языке С:

```
if(a<5){
    /* код по выполнению условия*/
}
/*продолжение программы*/
```

Как нечто подобное может быть реализовано на языке ассемблер? Очень просто. Нам на помощь приходят команды условного перехода и метки. Так, например вышеприведенный код может быть педставлен в виде (условимся, что в качестве переменной а у нас выступает регистр eax):

```
        cmp eax, 05
        jnl m1
        ;код по выполнению условия
m1:     ;продолжение программы
```

Хорошо, а как быть со сложными условиями? Например, такой вариант:

```
if((a<5)&&(a>2)){
    /*код по выполнению условия*/
}
/*продолжение программы*/
```

В таком случае сложные условия разбиваются на ряд простых. Конечная реализация для данного примера может выглядеть так:

```
        cmp eax, 5
        jnl m1
        cmp eax, 2
        jle m1
        ;код по выполнению условия
m1:     ;продолжение программы
```

В этом примере мы сначала проверяем первую часть условия, а потом вторую.

Оператор if()…else

Рассотрим такой фрагмент:

```
if(a<5){
   /*код по выполнению условия*/
}
else{
  /*код по невыполнению условия*/
}
/*продолжение программы*/
```

И опять на ассемблере это реализуется с помощью меток и команд условного перехода:

```
      cmp eax, 5
      jnl m1
      ;Код по выполнению условия
      jmp m2
m1: ;код по невыполнению условия
m2: ;продолжение программы
```

Оператор switch()… case

Оператор множественного выбора может быть реализован в виде нескольких команд if, которые мы уже реализовывали на ассемблере. Рассмотрим такой фрагмент кода:

```
switch (a){
    case 1: /*код если 1*/ break;
    case 2: /*код если 2*/ break;
    case 3: /*код если 3*/ break;
    default: /*код default*/
}
/*продолжение программы*/
```

На языке ассемблер этот код может быть реализован так:

```
        cmp eax, 1
        je m1
        cmp eax, 2
        je m2
        cmp eax, 3
        je m3
        ;код default
        jmp m4
m1: ;код если 1
        jmp m4
m2: ;код если 2
        jmp m4
m3: ;код если 3
m4: ;продолжение программы
```

Циклы
Команда loop

Команда loop очень похожа на обычные команды условного перехода и имеет такой же синтаксис. Однако, обычно ее используют при организации циклов. Вот ее синтаксис:

loop метка

Данная команда делает следующее:
1)Уменьшает на единицу значение регистра ecx
2) Если значение регистра ecx больше нуля, то управление передается на метку. В противном случае ничего не происходит

При этом содержимое регистра ecx рассматривается как безнаковое целое.

Похожей на команду loop является команда loope (она же loopz). Эта команда работает так же как и команда loop, но в отличие от последней проверяет не только равенство содержимого регистра ecx нулю, но и состояние флага нуля (ZF). То есть эти команды передают управление на метку только в том случае если значение регистра ecx отлично от нуля и взведен флаг нуля. Если хотя бы одно из этих условий не выполняется, управление не передается.

Ниже приводится пример цикла организованного с помощью команды loop:

```
        mov ecz, 05
m1:  loop m2
        jmp m3
m2: ;тело цикла
        jmp m1
```

```
m3: ;продолжение программы
```

Эквивалент этого кода на языке C (при условии, что переменной I у нас соответвует значение, хранящееся в регистре ecx) может быть записан в виде:

```
for(i=5;i>0;i--){
    /*тело цикла*/
}
/*продолжение программы*/
```

Как видите, команда loop позволяет легко создавать циклы for.

Оператор for()

Давайте посмотрим, как может быть реализован цикл for() без применения команды loop. Наример, такой цикл:

```
for(a=0;a<5;a++){
   /*тело цикла*/
}
/*продолжение программы*/
```

На языке ассемблер он может быть реализован так (напоминаю, что эквивалентом переменной a у нас является значение, хранящееся в регистре eax):

```
        xor eax, eax
m1:  cmp eax, 5
        jnb m2
        ;тело цикла
        jmp m1
m2: ;продолжение программы
```

Оператор while()

Рассмотрим фрагмент программы:

```
while(a<5){
    /*тело цикла*/
}
/*продолжение программы*/
```

Реализуется он похожим образом:

```
m1:  cmp eax, 5
     jnb m2
     ;тело цикла
     jmp m1
m2:  ;Продолжение программы
```

Оператор do…while

Данный оператор реализуется аналогичным образом, но с небольшим изменением. Рассмотрим пример:

```
do{
     /*тело цикла*/
}while(a<5);
/*продолжение программы*/
```

Возможная реализация на ассемблере данного фрагмента программы представлена ниже:

```
m1: ;тело цикла
     cmp eax, 5
     jb m1
     ;продолжение программы
```

Как видите ничего сложного. Операторы условного и безусловного перехода позволяют легко создавать на ассемблере конструкции языков высокого уровня.

Сложные типы данных
Перечисление

Спросите себя: что такое перечисление? Герберт Шилдт в своей книге «Полный справочник по С» дает такое определение: «Перечисление – это набор именованных целых констант».То есть перечисление – это несколько целых чисел, заранее предопределенных чисел. Главная особенность переменных типа перечисление заключается в том, что они могут принимать значения только из заранее составленного перечня.

Теперь другой вопрос: что нам мешает реализовать средствами языка ассемблер переменные такого типа? Задавать целые числа мы умеем. Определить для них уникальные наименования можно путем задания этих чисел в виде констант, это мы тоже умеем.

Единственная сложность при реализации перечислений и работе с ними состоит в том, что следить за правильностью операций присваивания и

сравнения мы должны будем сами. В языках высокого уровня за это отвечает компилятор. Это единственное препятствие на пути реализации и использовании перечислений в программах на ассемблере. Больше никаких преград нет.

Единственное, что требуется от программиста для решения этой прблемы это внимательность и аккуратность при написании программы. Ничего сверхъестественного.

Массив

Будем действовать по тому же принципу: что такое массив? Если говорить упрощенно, то массив – это последовательность значений определенного типа. В языке ассемблер понятие типа эквивалентно понятию размера, то есть объема памяти необходимого для хранения значения этого типа. То есть, каждый элемент массива имеет строго предопределенный размер.

Хорошо. Идем дальше: «как хранится в памяти эта последовательность?» Обычно в виде непрерывной последовательности значений. Поясню на примере.

Предположим у нас есть массив значений: 1, 2,3,4,5; каждый элемент массива занимает в памяти всего один байт. Тогда такой массив может иметь следующее представление в памяти:

01 02 03 04 05

Если размер одного элемента массива составляет слово (2 байта), тогда такое[1]:

0100 0200 0300 0400 0500

Если же размер элемента двойное слово (4 байта), тогда:

01000000 02000000 03000000 05000000

Надеюсь, идея понятна.

Хорошо, мы разобрались с тем как массив хранится в памяти, и что он из себя представляет, но как с ним работать? Для работы с массивом нам нужно научиться обращаться к его произвольному элементу по его индексу. Что нам для этого нужно?

Для того чтобы обратиться к произвольному элементу конкретного массива нам нужно знать три вещи:

1) адрес первого элемента массива (вы понимаете, что использовать для хранения массива регистры, по меньшей мере, неразумно), допустим, он у нас хранится в регистре ebx;

2) размер одного элемента массива

3) индекс элемента, к которому хотим обратиться, допустим, он у нас хранится в регистре ecx. Будем вести нумерацию элементов массива с нуля[2].

---

[1] Не забываем про обратный порядок байтов при представлении чисел в памяти ЭВМ

[2] При нумерации с единицы команды работы с элементами изменятся лишь незначительно

Предположим, что размер одного элемента массива составляет один байт. Тогда обратиться к элементу массива можно так:

mov ah, byte [ebx+ecx]

В результате выполнения этой команды в регистре ah окажется нужное нам значение. Как это работает?

Регистр ebx хранит адрес массива и соответственно его нулевого элемента. Значит, команда:

mov ah, byte [ebx]

поместит в регистр ah именно его. Так как размер одного элемента составляет один байт, то команда

mov ah, byte [ebx+1]

поместит в регистр ah первый элемент, а команда

mov ah, byte [ebx+2]

второй. И так далее.

Хорошо, а как быть, если размер элемента массива составляет не один, а четыре байта. В этом случае код изменится не сильно:

mov eax, dword [ebx+4*ecx]

логика та же самая что и с элементами массива размером в один байт.

Как видите, мы можем легко обращаться к любым элементам массива по их индексам, как и в случае языков высокого уровня.

Многомерный массив

Существует несколько способов реализации многомерных массивов на языке ассемблер. Мы опишем всего лишь один из них.

Начнем с простого двумерного массива. По своей сути двумерный массив представляет собой простую таблицу чисел. Каждое число в такой таблице адресуется номером колонки и номером строки, на пересечении которых оно и находится.

	0	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15

Мы уже знаем, как на ассемблере создаются одномерные массивы. А что если в качестве элементов таких массивов выступают не какие-то числа, а адреса других одномерных массивов. Например, так, как на рисунке:

0	1	2	3	4
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

В этом случае дочерние массивы (адреса, которых хранятся в основных массивах) представляют собой колонки таблицы.

Хорошо, как при такой организации двумерного массива добраться до нужной ячейки. Предположим, что адрес двумерного массива у нас хранится в регистре ebx, в регистре ecx хранится номер колонки, а в регистре edx номер строки. Колонки и строки нумеруются с нуля. Тогда возможный код чтения нужного элемента массива может выглядеть так:

```
push ebx  ;сохраняем адрес двумерного массива
mov ebx, [ebx+4*ecx];в ebx попадает адрес массива нужной нам колонки
mov eax, [ebx+4*edx] ;в eax нужный нам элемент двумерного массива
pop ebx   ;восстанавливаем адрес двумерного массива
```

Как видите, ничего архисложного в этом нет.

А теперь подумайте: что будет, если каждый элемент двумерного массива будет представлять собой адрес другого массива чисел. Правильно, мы получаем трехмерный массив. Если элементами трехмерного массива являются адреса других массивов, то это уже четырехмерный массив. И так далее.

Структура

Пойдем тем же путем: «что такое структура?»[1]. По сути это просто последовательность значений разного размера. Как она хранится в памяти? Точно так же как и массив. Рассмотрим структуру вида:

```
rec: record
   a1: integer; //размер 4 байта
   a2: byte;    //размер 1 байт
   a3: integer; //размер 4 байта
end;
```

Предположим, что поля этой структуры имеют значения 1, 2 и 3 соответственно. Тогда в памяти такая структура может храниться в виде:

[1] Здесь и далее под понятием струкутра мы подразумеваем тип record языка pascal, а не тип struc языка C, который, строго говоря, является классом.

01000000 02 03000000

Обращение к полям стуктуры может быть осуществлено точно так же как и при работе с массивом. Единственное различие состоит в том, что при работе со структурой немного усложняется расчет смещения нужного ее поля от начала структуры.

Предположим, что мы хотим обратиться к полю a3 выше приведенной структуры rec. Адрес начала этой структуры у нас хранится в решистре ebx. Тогда обращение может быть записано в виде:

```
mov eax, dword [ebx+5]     ;5=4+1
```

Здесь, в регистр eax будет помещено значение нужного нам поля.

Это пожалуй самый низкий уровень работы со структурами. К счастью, Flatasm может упростить нам работу со структурами за счет использования макрокоманды struct. Эта макрокоманда позволяет описывать структуру так же, как и в языках высокого уровня. Например, выше приводимая структура может быть описана с помощью этой макрокоманды следующим образом:

```
struct rec
  a1 dd ?
  a2 db ?
  a3 dd ?
ends
```

Здесь, команда ends говорит о завершении описания структуры, а символ знак вопроса о том, что данные остаются не проинициализированными[1].

Данное описание типа может быть размещено как в тексте самого исполнительного модуля приложения, так и в каком-либо подключенном к нему inc файле.

Хорошо мы описали тип структуры, а как описать данные этого типа? Двумя способами. Например, так:

```
varstruc rec
```

В этом случае поля структуры не инициализируются. Или так:

```
varstruc rec 01, 02, 03
```

В этом случае поля структуры будут проинициализированы значениями 1, 2 и 3 соответственно. А вот определение вида:

[1] Строго говоря, поля и не могут быть проинициализированы при описании типа. Они инициализируются при объявлении переменной соответствующего типа.

varstruc rec 01, 0203h, 04

Является ошибочным, так как размер поля a2 составляет всего один байт, а мы пытаемся его проинициализировать значением размером в 2 байта.

Следует понимать, что varstruc это не переменная типа struc, в ассемблере нет такого понятия как переменная, это метка (указатель) на экземпляр структуры rec в памяти.

После такого описания структуры работа с ней упрощается. И для получения значения поля a3 достаточно выпонить следующую команду:

mov eax, dword [varstruc.a3]

Как видите нам не нужно заботиться о расчете смещений. Однако следует иметь в виду, что Flatasm в процессе компиляции, сам подставит вместо этих конструкции обращения к полям структуры по заранее рассчитаннм смещениям. Поэтому код, отображаемый в отладчике, будет несколько отличаться от того, что вы написали в ассемблере. Хотя суть останется та же.

Класс

Да, ассемблер не явялется объектно-ориенттированным языком программирования, но это не значит, что на нем нельзя реализовать класс. Давайте подумаем вот над чем: «чем класс отличается от структуры, которую мы уже умеем реализовывать на ассемблере?»

Упрощенно говоря, класс – это набор свойств и методов этого класса. Свойство класса – это некоторое значение определенного типа (читай: размера). По своей сути свойство класса ничем не отличается от поля структуры.

Метод класса – это функция или процедура. Язык ассемблера позволяет вам работать с указателями на процедуры и функции точно так же и с указателями на данные. Исходя из этого, можно сказать, что метод класса – это свойство класса, которое хранит в себе адрес процедуры или функции. Получается, что метод класса также практически ничем не отличается от поля структуры и может быть реализован похожим образом.

Нужно просто написать процедуру, составляющую код этого метода, а затем поместить ее адрес в соответствующее поле экземпляра класса. При необходимости вызвать метод класса: из экземпляра этого класса извлекается адрес нужной процедуры, а затем она вызывается с параметрами любой командой передачи управления (call или stdcall). Вот и вся реализация.

«А как же части класса private и public?» - спросит читатель. На самом деле, контроль за (не) допустимостью обращения к свойствам и методам private части класса вне его осуществляется на этапе синтаксической проверки текста программы. Никакой принципиальной разницы между свойствами и методами private и public частей класса нет. Поэтому, если вы

хотите использовать эту возможность в программах на языке ассемблер, то вы должны сами следить за допустимостью соответствующих вызовов и обращений к свойствам.

А как насчет конструктора и деструктора? Я задам встречный вопрос: а чем конструктор и/или дестурктор отличается от метода класса. Единственное отличие между ними состоит в том, что если методы класса вызываются в процессе работы с этим классом и могут быть вызваны несколько раз, то конструктор и деструктор вызываются по одному разу только в строго определенных случаях: создание и уничтожение экземпляра класса. Поэтому реализация конструктора и деструктора практически ничем не отличается от реализации других методов этого же самого класса.

Единственное в чем состоит сложность при работе с конструктором и деструктором это необходимость вызывать их вручную по мере надобности. А также самостоятельный контроль за тем, чтобы конструктор или деструктор не могли быть вызваны в ненадлежащее для них время.

Хорошо, а как быть с указателем this? У Герберта Шилдта[1] читаем: «При вызове функции-члена[2] ей неявно передается указатель на вызывающий объект. Этот указатель называется this». Получается, что указатель this – это адрес экземпляра класса, метод которого вызывается в данный момент, который неявным образом передается в качестве параметра в процедуру, реализующую вызываемый метод. При реализации класса на ассемблере вы должны будете самостоятельно передать этот параметр в вызываемый метод. Вот и все.

Процедуры и функции
Общие сведения

В языке ассемблер нет такого понятия как процедура или функция. Но не стоит отчаиваться: не все так плохо.

Воспользуемся уже не раз проверенным методом: что такое процедура или функция?

Упрощенно говоря, процедура – это некий набор команд, который выполняет какие-то определенные действия над переданными ему данными и может быть вызван из любого места программы

Функция – это процедура[3], которая возвращает какое-то определенное значение предопределенного типа. Так как эти понятия очень похожи, то в дальнейшем мы будем использовать термин «процедура», подразумевая под ним и процедуру, и функцию.

Теперь давайте разбираться. «Набор команд» - с этим думаю все ясно, и никаких вопросов возникнуть недолжно.

---

[1] Полный справочник по C++

[2] То же самое что и метод класса

[3] В языке C, наоборот, процедура – это функция, которая возвращает значение типа void, т.е. никакого значения.

Под переданными данными подразумеваются параметры процедуры. Возможные варианты их передачи в вызываемый код мы уже обсуждали в разделе «соглашения о вызовах» предыдущей главы.

Различным вариантам вызова процедур была посвящена вся предыдущая глава.

В случае создания функции возвращаемое ею значение, как правило, помещается в регистр eax. Так, по крайней мере, делают практически все функции Windows API. Конечно, вам ничто не мешает реализовать какой-нибудь свой механизм возврата значений из функции, но этот наиболее популярен.

Возврат управления из процедуры в место ее вызова осуществляется командой ret, которая также обсуждалась в прошлой главе.

Получается, что у нас уже есть на руках вся необходимая информация. Дело за практикой. Ниже приводится пример простейшей процедуры на языке ассемблер:

```
OurProcedure:
    mov eax, 05
    ret
```

Данная процедура будет эквивалента процедуре на зыке C, представленной ниже:

```
int OurProcedure(){
    return 5;
}
```

А вот пример ее вызова на языке ассемблер:

```
    call OurProcedure
```

Как видите, в принципе ничего сложного. Но это если процедура вызывается без параметров. В случае применения параметров появляются некоторые подводные камни. О них и поговорим.

Чтение параметров из стека

Наиболее распротраненными соглашениями о вызоваз являются соглашения cdecl и stdcall. В них обоих параметры в вызываемую процедуру передаются через стек. Причем передаются в обратном порядке. То есть первый параметр функции кладется в стек последним.

Размер параметров состаляет 4 байта. Это не требование стандарта, а размер одного элемента, помещаемого в стек[1].

[1] Даже если размер операнда команды push меньше 4 байт, он расширяется до этого значения.

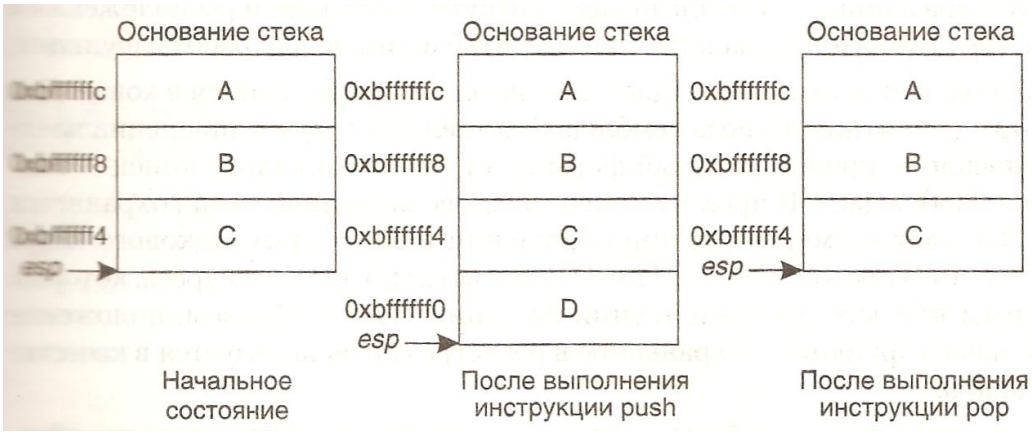
После того как параметры функции помещены в стек, вызывается команда call, которая сначала помещает в стек адрес возврата, а потом передает управление в вызываемую процедуру.

Теперь главный вопрос: как обратиться к параметрам процедуры?

Можно конечно использовать для этого команду pop. Но это не очень удобно. При этом подходе для того чтобы добраться для последнего параметра нам нужно извлечь из стека все предшествующие ему параметры и адрес возврата. При этом не потерять их, и проследить за тем, чобы к моменту вызова команды ret (выход из процедуры) адрес возврата лежал на вершине стека. Поэтому такой подход не годиться.

Как я уже говорил стек – это область памяти, для работы с которой предусмотрено несколько команд семейств pop и push. Регистр esp хранит адрес последнего элемента[1] в стеке. Почему бы нам не использовать его для обращения к параметрам процедуры. Это хорошая идея, но есть одно «но».

Дело в том, что стек растет не от младших адресов к старшим, а наоборот, от старшим к младшим. Это значит, что команда push уменьшает значение регистра esp на 4, а команда pop, наоборот увеличивает значение этого регистра на 4. Рисунок, представленный ниже, наглядно иллюстрирует это.



С учетом всего вышесказанного получаем, что обращение к первому параметру функции может быть выполнено следующим образом:

```
mov eax, dword [esp+4]
```

Добавляемые 4 байта нужны для того, чтобы пропустить хранящийся на вершине стека адрес возврата.

Обращение ко второму параметру может быть выполнено следующим образом:

```
mov eax, dword [esp+8]
```

[1] То есть элемента помещенного в стек последним

к третьему:

```
mov eax, [esp+0Ch]
```

и так далее.

Этот подход намного удобнее предыдущего, но тоже не лишен недостатков. При его использовании нужно помнить о том, что любая операция со стеком push или pop изменяет значение регистра esp, а это значит, что старые обращения к параметрам процедуры сановятся неправильными. Так же нужно самостоятельно следить за тем, чтобы к моменту выхода из процедуры регистр esp указывал на адрес возврата. В противном случае поведение программы станет непредсказуемым.

Пролог и эпилог

Для упрощения жизни программиста неплохо было бы сохранять первоначальное значение регистра esp в каком-то другом регистре, который больше нигде не используется. Таким регистром по умолчанию является регистр ebp[1]. Для реализации этого в начало процедуры вставляется следующий код, называемый прологом функции[2]:

```
push ebp        ;Сохраняем прежнее значение этого регистра
mov ebp, esp    ;Сохраняем адрес вершины стека
```

Иногда пролог оформляют в виде одной команды ассемблера:

```
enter 0, 0
```

Тогда к параметрам функции можно обращаться посредством регистра ebp, а значение регистра esp можно менять, не боясь при этом «потерять» параметры.

Обращение к первому параметру процедуры может быть выполнено следующим образом:

```
mov eax, [ebp+8]
```

Добавляя восьмерку мы «пропускаем» в стеке сохраненное значение регистра ebp и адрес возврата (на каждого по 4 байта).

Обращение ко второму параметру может быть записано так:

```
mov eax, [ebp+0Ch]
```

---

[1] Вообще говоря, вы можете использовать для этого любой понравившийся вам регистр, но использование этого является все-таки наиболее предпочтительным

[2] Упоминание термина «функция» не должно вас смущать: его можно использовать как в функциях так и в процедурах

К третьему:

```
mov eax, [ebp+10h]
```

и так далее.

В конец процедуры тогда вставляют следующий код, именуемый эпилогом функции:

```
mov esp, ebp
pop ebp
```

В нем происходит восстановление значений регистров esp и ebp. После его выполнения регистр esp вновь указывает на адрес возврата. А значит, команда ret может быть спокойно выполнена.

Иногда эпилог функции записывают в виде одной команды ассемблера:

```
leave
```

Использование пролога и эпилога функции значительно упрощает разработку процедуры, и помогает программисту избежать многих ошибок. Обнако их использование не является обязательным. Если вы пишете небольшую процедуру, то вы можете обойтись и без них, но тогда следить за правильностью регистра esp придется вам самим.

Макрокоманда proc

Flatasm позволяет упростить написание процедур посредством макрокоманды proc. Эта макрокоманда не является командой языка ассемблер. Она создана для упрощения написания процедур. Вот так, например, с помощью нее может быть описана процедура с двумя параметрами:

```
proc OurProcedure, par1, par2
    ;тело процедуры
    ret;
endp
```

При таком описании Flatasm сам создаст для нее эпилог и пролог. Внутри тела данной процедуры обращение к первому параметру может быть записано в виде:

```
mov eax, dword [par1]
```

а обращение ко второму в виде:

```
mov ax, dword [par2]
```

Как видите, обращение к параметрам функции значительно упростилось.

Здесь нужно помнить о том, что все эти упрощения отображаются при написании программы, при компиляции которой они будут заменены соответствующими командами ассемблера. Поэтому код, отображаемый в отладчике будет несколько отличаться от того, что вы набирали в окне Flatasma.

Вызов такой процедуры может быть осуществлен так:

```
stdcall OurProcedure, 5, 6
```

При этом числа 5 и 6 передаются ей в качестве параметров.

Глава 8. Начинаем программировать
Двойственность функции Windows API

Некоторые функции Windows API представлены в двух «немного различных видах». Например, существует две функции MessageBoxA и MessageBoxW, которые делают одно и то же (что именно описывается в следующих разделах этой главы). Но это две разные функции. Чем они отличаются друг от друга?

Разница между ними в формате строковых параметров. Функция MessageBoxA, как и другие функции оканчивающиеся символом A, ожидает на входе однобайтовую ASCII строку. А функция MessageBoxW, как и другие функции оканчивающиеся символом W, ожидает на входе двухбайтовую Unicode строку[1].

Конечно, далеко не все функции имеют различные варианты для обнобайтовых и двухбайтовых строк. Для того чтобы выделить функции Windows API имеющие такое разделение мы будем сопровождать их описание строкой «(A/W)».

Например, запись «FormatMessage(A/W)» должна говорить вам, что существует две функции FormatMessageA и FormatMessageW, а функции FormatMessage нет. В то же время простая запись «ExitProcess» должна говорить вам, что функция ExitProcess не имеет такого разделения.

Понятие дескриптора

Когда вы в своей программе работаете с какими-то сложными объектами такими, как файлы, окна, ключи реестра и многие другие, то для корректной работы с ними вы должны знать большое количество информации. К такой информации могут относиться права на доступ, с которыми был открыт файл, текущая позиция в файле, имя класса окна и различные его характеристики и многое другое. Это не только значительно усложняет работу с такими объектами, но и делает ее потенциально небезопасной. Так как в случае изменения этих характеристик в процессе работы программы их правильность будет сложно контролировать.

Для того чтобы избежать всех этих проблем было принято решение о том, что все характеристики используемых программой объектов будут храниться операционной ситсемой в специальной служебной таблице. Причем доступ к этой таблице будет, только у самой операционной системы. Приложение не может обращаться к ней и тем более вносить в нее какие-то изменения. Это позволяет избежать ряд трудностей, описанных выше. Но возникает вопрос: «как приложения смогут работать с этими объектами?».

Во множестве функций Windows API предусмотрены наборы функций для работы с конкретными объектами. Например, имеются функции для работы с файлами, с ключами реестра, окнами и т.д. То есть вся обработка

[1] Работа со строками различных видов будет подробно рассмотрена в соответствующей главе.

объектов ложится на плечи операционной системы, программист просто вызывает API функцию. Хорошо, но откуда эта функция узнает над каким именно объектом нужно выполнить то или иное действие.

Когда операционная система по запросу программы создает тот или иной объект, она не только формирует запись в соответствующей системной таблице об этом объекте, но и возвращает приложению определенное число, по которому в дальнейшем сможет понять о каком объекте идет речь. Данное число и называется дескриптором или хендлом соответствующего ему объекта.

Ошибочным дескриптором является дескриптор со значением нуль. Он возвращается, как правило, в случае ошибки создания того или иного объекта.

После окончания работы с объектом, он должен быть уничтожен, вернее должна быть стерта запись об этом объекте из специальной таблицы операционной системы. Делается это также специальными функциями Windows API, которые, так же как и функции создания объекта, зависят от вида уничтожаемого или создаваемого объекта.

Функция GetModuleHandle(A/W)

Данная функция имеет следующий прототип:

```
HMODULE WINAPI GetModuleHandle(
  __in_opt  LPCTSTR lpModuleName //Адрес строки с именем модуля
);
```

Под модулем подразумевается любой exe или dll файл, загруженный в адресное пространство текущего процесса.

Данная функция возвращает дескриптор этого модуля или NULL[1], в случае ошибки. Если в функцию в качестве параметра передается значение NULL, то она возвращает дескриптор модуля, из которого была вызвана данная функция.

Дескриптор модуля необходим функциям, работающим с ресурсами, описанными в этом модуле. Если такие функции не будут знать, откуда им брать ресурсы, то они не смогут правильно выполниться. Примерами таких функций являются: LoadImage(A/W), LoadString(A/W), DialogBoxParam(A/W) и другие.

Теперь о главном. Понятие дескриптора модуля хоть и является устоявшимся термином, является ошибочным, так как ни у какого модуля нет дескриптора. Для модуля, загруженного в адресное пространство процесса, не создается никакой записи в специальной таблице используемых объектов. Для хранения информации о загруженных модулях используется совершенно другой механизм, отличный от того, что используется для хранения

[1] В одной из предыдущих глав мы писали о том, что значение NULL и нуль являются эквивалентными.

информации о созданных окнах или открытых файлах. Так что же тогда возвращает эта функция?

Под понятием дескриптор модуля подразумевается базовый адрес загрузки этого самого модуля. Понятие базового адреса загрузки будет нами рассмотрено в одной и следующих глав при обсуждении вопроса разработки динамических библиотек.

Функция MessageBox(A/W)

Данная функция является, пожалуй, самой известной функцией Windows API. Она используется для вывода какого-либо сообщения в отдельном окне программы и определена в библиотеке user32.dll. Причем выполнение самой программы приостанавливается до тех пор, пока пользователь не закроет окно с этим сообщением.

Вот прототип этой функции:

```
int WINAPI MessageBox(
  __in_opt  HWND hWnd,          //Дескриптор родительского окна
  __in_opt  LPCTSTR lpText,     //Адрес строки с текстом сообщения
  __in_opt  LPCTSTR lpCaption, //Адрес строки с заголовком окна сообщения
  __in      UINT uType          //Тип окна
);
```

Параметр uType задает набор кнопок, отображаемых на окне с сообщением, и иконку, отображаемую в нем же. В таблице ниже представлены основные наборы кнопок и соответствующие им значения параметра uType.

Набор кнопок	Описание
MB_ABORTRETRYIGNORE	Окно с кнопками «Прервать», «Повтор», «Пропустить»
MB_CANCELTRYCONTINUE[1]	Окно с кнопками «Отмена», «Повторить», «Продолжить»
MB_HELP	Окно с кнопками «OK» и «Справка»
MB_OK	Окно с одной кнопкой «OK»
MB_OKCANCEL	С кнопками «OK» и «Отмена»
MB_RETRYCANCEL	С кнопками «Повтор» и «Отмена»
MB_YESNO	С кнопками «Да» и «Нет»
MB_YESNOCANCEL	С кнопками «Да», «Нет» и «Отмена»

В таблице ниже представлены основные иконки, которые могут отображаться в окне с сообщением и соответствующие им значения параметра uType.

[1] К сожалению, данная константа не определена заголовочных файлах Flatasm, потому вам придется определять ее самим или подставлять ее численное значение 6.

Иконка	Описание
MB_ICONEXCLAMATION	Восклицательный знак в желтом треугольнике
MB_ICONWARNING	То же самое
MB_ICONINFORMATION	Буква «i» на голубом фоне
MB_ICONASTERISK	То же самое
MB_ICONQUESTION	Знак вопроса на голубом фоне
MB_ICONSTOP	Красный круг с белым крестом внутри
MB_ICONERROR	То же самое
MB_ICONHAND	То же самое

Данная функция возвращает либо ноль в случае ошибки, либо номер нажатой пользователем кнопки. В таблице ниже представлены значения этих номеров для различных кнопок:

Номер кнопки	Надпись на кнопке
IDABORT	Прервать
IDCANCEL	Отмена
IDCONTINUE[1]	Продолжить
IDNO	Нет
IDOK	OK
IDRETRY	Повторить
IDYES	Да

Пишем Hello World

Ниже приводится исходный код программы выводящей сообщение «Hello world!»:

```
;Указываем формат выходного файла
Format PE GUI 4.0
;Указываем точку входа
entry start
;Подключаем библиотеку
include  'D:\FASM1\INCLUDE\win32a.inc'
;Описываем секцию кода
section '.code' code readable executable
   ;Текст, отображаемый на экране
   string: db 'Hello world!', 0

   start:
        ;Выводим диалоговое окно с текстом сообщения
        invoke MessageBox, 0, string, string, MB_OK
        ;Завершаем программу
        invoke ExitProcess, 0
;Описываем секцию с таблицей импорта
section '.idata' import data readable writeable
```

[1] Константа для нее также по умолчанию не опредлена, численное значение - 11

```
 library kernel, 'KERNEL32.DLL',\
         user, 'USER32.DLL'
import kernel,\
       ExitProcess, 'ExitProcess'
import user,\
       MessageBox, 'MessageBoxA'
```

Здесь для уменьшения размера программы выводимую строку мы разместили в секции кода (хотя нам ничто не мешало поместить ее в отдельной секции данных).

Дескриптор окна рабочего стола

Иногда функцию MessageBox вызывают с параметром hWnd равным HWND_DESKTOP. При этом в различных источниках утверждается, что это дескриптор окна рабочего стола. Но так ли это? Если взглянуть в заголовочный файл WinUser.h, то можно увидеть в нем следующее:

```
#define HWND_DESKTOP        ((HWND)0)
```

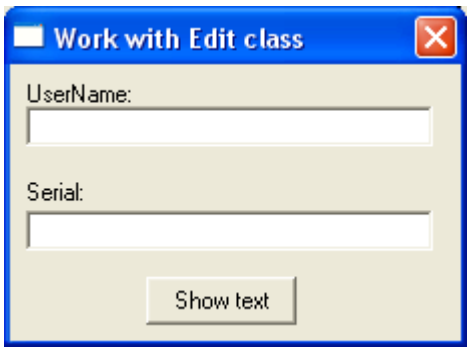
То есть, получается, что HWND_DESKTOP=0. В итоге имеем:

HWND_DESKTOP=NULL=0

Получается, что HWND_DESKTOP это никакой не дескриптор окна рабочего стола, это ошибочный дескриптор, то же самое что и NULL.

Глава 9. Окна и сообщения
Общие сведения

В Windows программах, окно – это прямоугольная область экрана, в которой приложение отображает выходные данные и в которую позволяет пользователю вводить свои данные. Вообще говоря, понятие окна является основополагающим понятием в ОС Windows[1]. Поэтому, первоочередной задачей программы, взаимодействующей с пользователем, является создание окна. Именно через него осуществляется взаимодействие пользователя с вашей программой. Для того чтобы было более понятно что же такое окно, предлагаю ответить на следующий вопрос: сколько окон изображено на следующем рисунке:



Правильный ответ: 6 окон. Давайте посчитаем: кнопка, два поля ввода, две надписи (да, все это отдельные окна) и главное окно, на котором расположены все предыдущие окна.

Окна могут образовывать иерахию родительских и дочерних окон. Закрытие родительского окна всегда приводит к закрытию вместе с ним всех его дочерних окон. В то время как закрытие дочернего окна никак не влияет на родительское. Это, пожалуй, единственное принципиальное различие между родительскими и дочерними окнами. Во всем остальном они похожи. На предыдущем рисунке представлено одно родительское окно (окно на котром расположены все другие окна), и пять дочерних по отношению к нему окон (все остальные окна).

Хорошо, будем считать, что с понятием окна мы разобрались. Теперь перейдем к сообщениям.

Сообщение – это уведомление приложения о наступлении того или иного события (нажатия на кнопку, сворачивании, разворачивании, перемещении окна, ввода в поле ввода и др.). По своей природе оно очень похоже на sms. По сути же, это определенный код, переданный окну приложению с дополнительными параметрами, которые зависят от кода сообщения, но в большинстве сообщений они отсутствуют. В ОС Windows имеется большое количество стандартных сообщений (при необходимости программист может создать свои сообщения). Вот лишь некоторые из них:

---

[1] Это, кстати говоря, отражено в названии ОС: слово «windows» в дословном переводе с английского означает «окна».

Сообщение	Описание
WM_CLOSE	Закрытие окна
WM_COMMAND	Уведомление от дочернего элемента управления
WM_CREATE	Создание окна
WM_DESTROY	Уничтожение окна
WM_INITDIALOG	Отправляется перед отображением диалогового окна[1]
WM_KILLFOCUS	Потерян фокус ввода
WM_MOVE	Отправляется окну после того как оно было перемещено
WM_PAINT	Перерисовка окна
WM_QUIT	Завершение прикладной программы
WM_SETFOCUS	Получен фокус ввода
WM_SHOWWINDOW	Посылается перед показом или сокрытием
WM_SIZE	Посылается после изменения размеров окна
WM_TIMER	Сообщение от таймера

Это далеко не полный перечень сообщений Windows. Некоторые другие сообщения будут рассмотрены при разборе элементов управления (у каждого из них свои специфические сообщения), и работы с мышью и клавиатурой.

Вообще говоря, с помощью сообщений можно из одной программы управлять поведением другого приложения. Например, мы можем сами отправить любому окну сообщение WM_CLOSE, требующее закрыть это окно. Для отправки сообщений существует две функции SendMessage (A/W) и PostMessage (A/W)[2]. Вот прототип функции SendMessage (A/W):

```
LRESULT WINAPI SendMessage(
  __in  HWND hWnd,      //handle окна, которому посылается сообщение
  __in  UINT Msg,       //код посылаемого сообщения
  __in  WPARAM wParam,  //дополнительные параметры сообщения
  __in  LPARAM lParam
);
```

Функция PostMessage (A/W) имеет такой же прототип. Различие между ними состоит в том, что функция SendMessage (A/W) отправляет сообщение и ждет, пока целевое окно это сообщение обработает, а функция PostMessage (A/W) просто помещает сообщение в очередь и программа работает дальше.

Все сообщения, за исключением сообщений WM_PAINT, WM_TIMER и WM_QUIT помещаются системой в конец очереди сообщений, которая работает по принципу FIFO (первым пришел, первым ушел – как в магазине).

Работа любой windows программы с графическим интерфейсом основана на обработке сообщений. В ОС Windows у каждого процесса[3] есть своя

[1] Подробнее диалоговые окна будут рассмотрены позже
[2] Это не единственные функции для отправки сообщений, но самые популярные
[3] Точнее говоря не у процесса, а у потока. Но многопоточные приложения мы рассматривать не будем так что при грубом упрощении процесс и поток можно считать синонимами (хотя это не так).

очередь сообщений (далее просто очередь), в нее попадают все сообщения, адресованные всем окнам данного процесса. Процесс (приложение) извлекает из этой очереди сообщение, обрабатывает его, потом берет следующее и так до бесконечности пока не будет получено сообщение WM_QUIT, требующее завершить данное приложение. Если процесс по какой-то причине не обрабатывает передаваемые ему сообщения, то он «зависает» и единственный способ его завершить – это «убить» его в диспетчере задач.

В основе каждого окна лежит так называемый класс окна, который определяет все основные свойства данного окна. С него мы и начнем.

Регистрация класса окна

Класс окна описывается одной из структур WNDCLASS или WNDCLASSEX, которая является расширенной версией первой. Начнем с WNDCLASS, вот ее описание:

```
typedef struct tagWNDCLASS {
  UINT      style;             //стиль класса
  WNDPROC   lpfnWndProc;       //адрес оконной процедуры
  int       cbClsExtra;        //объем дополнительной памяти резервируемой за
                               //структурой (обычно 0)
  int       cbWndExtra;        //объем дополнительной памяти за экземпляром
                               //окна оконной процедуры (обычно 0)
  HINSTANCE hInstance;         //дескриптор модуля, в котором описана оконная
                               //процедура
  HICON     hIcon;             //дескриптор иконки (если не используется
                               //иконка по умолчанию)
  HCURSOR   hCursor;           //дескриптор курсора (если не используется
                               //курсор по умолчанию)
  HBRUSH    hbrBackground;     //дескриптор кисти или системный цвет фона
  LPCTSTR   lpszMenuName;      //адрес строкового имени ресурса меню (NULL
                               //если меню нет) или его идентификатор
  LPCTSTR   lpszClassName;     //адрес строкового имени класса
} WNDCLASS, *PWNDCLASS;
```

Стиль класса – это один из предопределенных стилей классов окна или их сочетание. Существуют следующие стили классов:

Стиль класса	Описание
CS_BYTEALIGNCLIENT	(по горизонтали) выравнивание рабочей области окна по границе байта. Влияет на ширину окна и его горизонтальное положение на экране
CS_BYTEALIGNWINDOW	(по вертикали) выравнивает окна по границе байта
CS_CLASSDC	Контекст устройства, который будет разделяться всеми окнами класса. При нескольких потоках операционная система разрешит доступ только одному потоку
CS_DBLCLKS	Посылать сообщение от мыши при двойном щелчке в пределах окна
CS_GLOBALCLASS	Создавать глобальный класс, который можно поместить в динамическую библиотеку dll.
CS_HREDRAW	Перерисовывать все окно при изменении ширины
CS_NOCLOSE	Отключить команду закрыть
CS_OWNDC	У каждого окна уникальный контекст устройства
CS_PARENTDC	У дочернего окна будет область отсечки от родительского. Повышает производительность
CS_SAVEBITS	Позволяет сохранять область экрана в виде битовой матрицы закрытую в данный момент другим окном, используется для восстановления экрана
CS_VREDRAW	Перерисовывать окно при изменении вертикальных размеров

Существуют следующие системные цвета фона:

Символьное обозначение	Цвет
COLOR_ACTIVEBORDER	Белый
COLOR_ACTIVECAPTION	Черный
COLOR_APPWORKSPACE	Темно-серый
COLOR_BACKGROUND	Темно-серый
COLOR_BTNFACE	Белый
COLOR_BTNSHADOW	Светло-серый
COLOR_BTNTEXT	Темно-серый
COLOR_CAPTIONTEXT	Черный
COLOR_GRAYTEXT	Темно-серый
COLOR_HIGHLIGHT	Темно-серый
COLOR_HIGHLIGHTTEXT	Синий
COLOR_INACTIVEBORDER	Светло-серый

COLOR_INACTIVECAPTION	Синий
COLOR_MENU	Бледно-голубой
COLOR_MENUTEXT	Черный
COLOR_SCROLLBAR	На фон попадает все то, что собой закрыло окно
COLOR_WINDOW	Белый
COLOR_WINDOWFRAME	Белый
COLOR_WINDOWTEXT	Черный

Как я уже говорил, структура WNDCLASSEX очень похожа на структуру WNDCLASS. Отличаются они тем, что в первой структуре добавились два поля по сравнению со второй. Вот прототип описания структуры WNDCLASSEX:

```
typedef struct tagWNDCLASSEX {
  UINT      cbSize;      //Размер структуры
  UINT      style;
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCTSTR   lpszMenuName;
  LPCTSTR   lpszClassName;
  HICON     hIconSm;     //дескриптор малой иконки, ассоциированной с классом
} WNDCLASSEX, *PWNDCLASSEX;
```

Все остальные поля полностью идентичны соответствующим полям структуры WNDCLASS.

Для того, чтобы операционная система узнала о новом класса, его нужно зарегистрировать. Делается это с помощью функций RegisterClass (A/W) и RegisterClassEx (A/W). Посмотрим на их прототипы:

```
ATOM WINAPI RegisterClass(
  __in  const WNDCLASS *lpWndClass
);

ATOM WINAPI RegisterClassEx(
  __in  const WNDCLASSEX *lpwcx
);
```

Единственная разница между ними в том, что функция RegisterClass (A/W) принимает в качестве параметре указатель на структуру WNDCLASS, а функция RegisterClassEx (A/W) – указатель на структуру WNDCLASSEX.

В случае успеха данные функции возвращают уникальный идентификатор зарегистрированного класса. Если же функция по какой-то причине не может выполнить возложенную на нее задачу, то возвращаемое ею значение равно нулю.

Уничтожить или «снять с регистрации» класс можно функцией UnRegisterClass (A/W) (делать это только после того как уничтожены все окна построенные на этом классе), вот ее описание:

```
BOOL WINAPI UnregisterClass(
  __in      LPCTSTR lpClassName, //адрес строки с именем удаляемого класса
  __in_opt  HINSTANCE hInstance  //дескриптор модуля, в котором был создан
                                 //класс
);
```

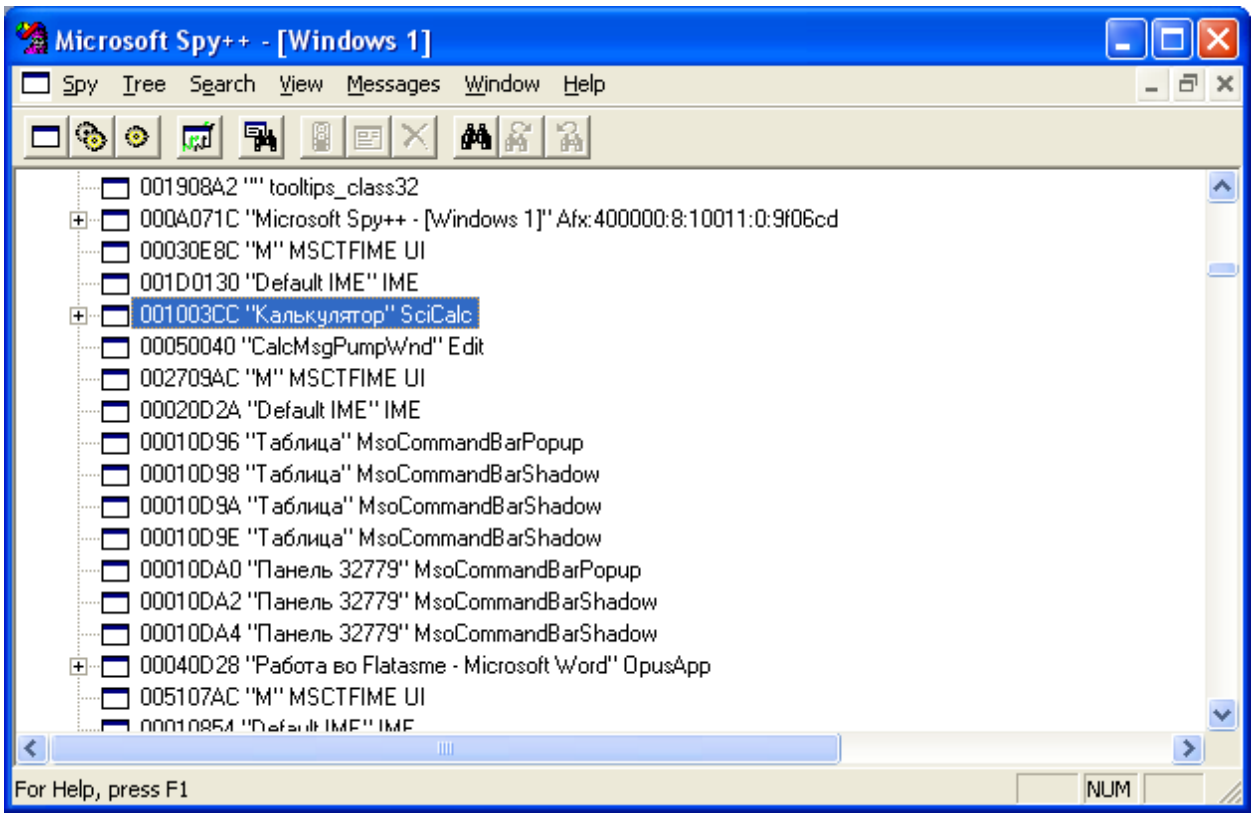
Если данная функция успешно отрабатывает, то возвращается не нулевое значение. Если же функция не может выполнить возложенную на нее задачу (не найден такой класс, существуют окна, основанные на этом классе и др.) тогда возвращается нуль.

В принципе, если в вашем приложении всего одно окно, которое уничтожается по завершении работы приложения тогда вызывать эту функцию необязательно. Но если в вашем приложении существует много окон, созданных на основе разных классов, то вызывать функцию UnregisterClass (A/W) необходимо, в противном случае есть все шансы получить утечку памяти.

Ниже приводится пример создания и регистрации класса окна:

```
;Получаем дескриптор модуля
invoke GetModuleHandle, 0
push eax
;Заполняем структуру WNDCLASS
mov [wc.style], CS_HREDRAW+CS_DBLCLKS
mov [wc.lpfnWndProc], WndProc
mov [wc.cbClsExtra], 0
mov [wc.cbWndExtra], 0
mov [wc.hInstance], eax
mov [wc.hIcon], NULL            ;мы используем иконку по умолчанию
mov [wc.hCursor], NULL          ;мы используем курсор по умолчанию
mov [wc.hbrBackground], COLOR_WINDOWTEXT
mov [wc.lpszMenuName], NULL     ;у нас нет меню
mov [wc.lpszClassName], _ClassName
;Регистрируем класс
invoke RegisterClass, wc
test eax, eax
jz FailtRegister    //перейти если не удалось зарегистрировать класс
```

Узнать класс произвольного окна можно с помощью утилиты Spy++, входящей в состав Microsoft Visual Studio. Вот как она выглядит:



Из данного примера мы видим (см. выделенную строку на рисунке выше), что класс окна стандартного калькулятора Windows SciCalc, заголовок окна: «Калькулятор», а дескриптор равен 001003CCh.

Создание окна. Стили окна

После того как мы задали и зарегистрировали класс окна, мы можем создать само окно. Создаются окна с помощью функции CreateWindowEx (A/W). Вот ее описание:

```
HWND WINAPI CreateWindowEx(
  __in      DWORD dwExStyle,        //Расширенные стили окна
  __in_opt  LPCTSTR lpClassName,    //Указатель на строку с именем
                                    //зарегистрированного класса
  __in_opt  LPCTSTR lpWindowName,   //Указатель на строку заголовка окна
  __in      DWORD dwStyle,          //Стили окна
  __in      int x,                  //Начальное положение окна по горизонтали
  __in      int y,                  //Начальное положение окна по вертикали
  __in      int nWidth,             //Ширина окна
  __in      int nHeight,            //Высота окна
  __in_opt  HWND hWndParent,        //Дескриптор родительского окна
  __in_opt  HMENU hMenu,            //Дескриптор меню (NULL если его нет)
  __in_opt  HINSTANCE hInstance,    //Дескриптор модуля, связанного с окном
  __in_opt  LPVOID lpParam          //Параметр lParam, который будет передан
                                    //окну с сообщением WM_CREATE
);
```

Как видите все достаточно просто. Осталось разобраться со стилями окна. Существуют следующие стили окна, которые обычно сочетаются друг с другом:

Стиль окна	**Описание**
WS_BORDER	Отображает границу окна
WS_CAPTION	Отображает заголовок окна (автоматически устанавливает флаг WS_BORDER)
WS_CHILD	Создаваемое окно является дочерним
WS_CHILDWINDOW	Аналогично WS_CHILD
WS_CLIPCHILDREN	Окно включает области занятые дочерними окнами, которые могут быть отрисованы в отсутствие родительского окна
WS_CLIPSIBLINGS	Отсекает дочерние окна друг от друга, то есть когда определенное дочернее окно получает сообщение WM_PAINT, стиль отсекает все другие дочерние окна, которые перекрывают данное дочернее окно, от той области дочернего окна, которая будет обновлена. Если стиль WS_CLIPSIBLINGS не определен и дочерние окна перекрывают друг друга, то можно при рисовании в клиентской области одного дочернего окна рисовать и в клиентской области соседнего дочернего окна.
WS_DISABLED	Недоступное окно
WS_DLGFRAME	Отображает двойную границу окна, но без заголовка (не работает при WS_CAPTION)
WS_GROUP	Первый элемент в группе элементов. При указании у формы отображает кнопку «свернуть окно»
WS_HSCROLL	Отображает горизонтальную полосу прокрутки (scroll)
WS_ICONIC	Отображает свернутое окно, то же что WS_MINIMIZE (почему-то не работает)
WS_MAXIMIZE	Создает окно максимальных размеров
WS_MAXIMIZEBOX	Добавляет в заголовок окна кнопку «развернуть»
WS_MINIMIZE	Создает окно минимальных размеров
WS_MINIMIZEBOX	Добавляет в заголовок окна кнопку «свернуть»
WS_OVERLAPPED	Перекрывающееся окно (может располагаться поверх других окон)
WS_OVERLAPPEDWINDOW	Перекрывающееся окно с одновременным установлением флагов: WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX,

	WS_MAXIMIZEBOX
WS_POPUP	Флаг, обратный к WS_OVERLAPPED. Создается окно, не имеющее ничего, кроме поверхности.
WS_POPUPWINDOW	Временное окно с одновременным включением флагом: WS_BORDER, WS_POPUP, WS_SYSMENU
WS_SIZEBOX	Окно, у которого можно изменять размер
WS_SYSMENU	Окно, имеющее системное меню (кнопки закрыть, свернуть, развернуть).
WS_TABSTOP	Включает обход этого элемента по TAB
WS_THICKFRAME	Тоже что WS_SIZEBOX
WS_TILED	Тоже что WS_OVERLAPPED
WS_TILEDWINDOW	Тоже что WS_OVERLAPPEDWINDOW
WS_VISIBLE	Видимое окно
WS_VSCROLL	Вертикальная полоса прокрутки

А вот перечень расширенных стилей окна, которые также сочетаются друг с другом:

Расширенный стиль окна	**Описание**
WS_EX_ACCEPTFILES	Определяет окно, способное принимать перетаскиваемые на него файлы из «Проводника Windows»
WS_EX_APPWINDOW	на панель задач выводится кнопка окна
WS_EX_CLIENTEDGE	окно с «утопленной» клиентской частью
WS_EX_CONTEXTHELP	включает кнопку помощи (знак вопроса) в заголовок окна
WS_EX_CONTROLPARENT	Включает возможность навигации пользователя по элементам формы с использованием клавиши TAB.
WS_EX_DLGMODALFRAME	окно с двойной рамкой может сочетаться с WS_CAPTION
WS_EX_LEFT	Окно имеет общеупотребительное свойство «выравнивания по левой границе»
WS_EX_LEFTSCROLLBAR	Размещает вертикальную полосу прокрутки (scrollbar) в левой части окна
WS_EX_LTRREADING	Текст окна отображается, используя свойство порядка чтения Слева-Направо.
WS_EX_MDICHILD	Создает MDI дочернее окно
WS_EX_NOPARENTNOTIFY	Дочернее окно, созданное с этим стилем не посылает сообщение WM_PARENTNOTIFY родительскому окну, когда оно создается или разрушается

WS_EX_OVERLAPPEDWINDOW	WS_EX_CLIENTEDGE+WS_EX_WINDOWEDGE
WS_EX_PALETTEWINDOW	WS_EX_WINDOWEDGE+WS_EX_TOOLWINDOW+WS_EX_TOPMOST
WS_EX_RIGHT	Окно имеет общеупотребительное свойство «выравнивание по правому краю»
WS_EX_RIGHTSCROLLBAR	Вертикальная полоса прокрутки (scrollbar) в правой части окна
WS_EX_RTLREADING	Текст окна отображается, используя свойство порядка чтения Справа-Налево
WS_EX_STATICEDGE	Окно с трехмерным стилем рамки
WS_EX_TOOLWINDOW	окно с тонким заголовком
WS_EX_TOPMOST	Определяет, что окно, созданное с этим стилем должно быть размещено выше всех, не самых верхних окон и должно стоять выше их, даже тогда, когда окно дезактивировано
WS_EX_TRANSPARENT	прозрачное окно
WS_EX_WINDOWEDGE	окно имеет рамку с выпуклым краем

Функция CeateWindowEx (A/W) возвращает дескриптор созданного окна в случае успешного выполнения, или ноль в том, случае если ей по какой-то причине не удалось создать окно.

Уничтожается окно функцией DestroyWindow, вот ее прототип:

```
BOOL WINAPI DestroyWindow(
  __in  HWND hWnd    //Дескриптор уничтожаемого окна
);
```

Если функции не удается уничтожить окно, тогда она возвращает ноль, в противном случае возвращается ненулевое значение.

Отлично. Мы задали, зарегистрировали класс окна и создали на его основе само окно. Теперь поговорим об обработке сообщений, второй важной составляющей любого оконного приложения.

Цикл обработки сообщений

Данный цикл выполняет следующие задачи: 1) извлекает из очереди сообщений очередное сообщение, требующее обработки; 2) Приводит полученное сообщение к аппаратно-независимому виду и 3) передает его соответствующей оконной процедуре, которая его и обрабатывает. Теперь по порядку.

Как я уже говорил все сообщения, посылаемые окнам приложения, помещаются в очередь сообщений приложения. Перво-наперво, нам нужно извлечь из этой очереди очередное обрабатываемое сообщение. Для этого

предусмотрено две функции GetMessage (A/W) и PeekMessage (A/W). Начнем с функции GetMessage (A/W) как наиболее известной. Вот ее прототип:

```
BOOL WINAPI GetMessage(
  __out     LPMSG lpMsg,        //адрес структуры MSG, в которую заносится
                                //информация о сообщении извлеченном из
                                 //очереди сообщений
  __in_opt  HWND hWnd,          //Дескриптор окна-адресата сообщения (если
                                //NULL то извлекаются все сообщения,
                                //адресованные всем окнам данного
                                //приложения)
  __in      UINT wMsgFilterMin,//Минимальный код извлекаемого сообщения
  __in      UINT wMsgFilterMax //Максимальный код извлекаемого сообщения
);
```

Данная функция не вернет управление до тех пор, пока не придет какое-либо сообщение. Она возвращает одно из трех возможных значений:

Ненулевое положительное число – Если из очереди извлечено сообщение.

0 – Если получено сообщение WM_QUIT, требующее закрыть приложение

-1 – Если в процессе работы функции произошла какая-то ошибка

Параметры wMsgFilterMin и wMsgFilterMax служат для отбора строго определенных сообщений, например, сообщений от клавиатуры.

А вот прототип функции PeekMessage (A/W):

```
BOOL WINAPI PeekMessage(
  __out     LPMSG lpMsg,
  __in_opt  HWND hWnd,
  __in      UINT wMsgFilterMin,
  __in      UINT wMsgFilterMax,
  __in      UINT wRemoveMsg
);
```

Как видно из описания в отличие от функции GetMessage (A/W) добавился всего один параметр: wRemoveMsg (остальные параметры полностью идентичны соответствующим параметрам функции GetMessage (A/W)). Данный параметр определяет, как и какие сообщения будут обработаны. Он может принимать одно из следующих значений или их сочетание:

Значение wRemoveMsg	Описание
PM_NOREMOVE	Сообщения не удаляются из очереди сообщений после их выборки
PM_REMOVE	Сообщения удаляются из очереди сообщений после их выборки
PM_NOYIELD	Говорит функции о том, что при отсутствии сообщений в очереди сообщений нельзя передавать управление другим приложениям. Обычно PeekMessage (A/W) при отсутствии сообщений в очереди передает управление другим приложениям и, только если их очереди тоже пусты, возвращает false.
PM_QS_INPUT	Выбираются сообщения мыши и клавиатуры
PM_QS_PAINT	Выбираются сообщения рисования
PM_QS_POSTMESSAGE	Выбираются все помещенные в очередь (синхронные) сообщения, включая сообщения таймера и горячих клавиш
PM_QS_SENDMESSAGE	Выбираются все отправленные (асинхронные) сообщения

Данная функция возвращает ноль в том случае если в очереди нет подходящих сообщений, в противном случае возвращается ненулевое значение.

Главное различие между функциями GetMessage (A/W) и PeekMessage (A/W) состоит в том, что вторая (PeekMessage (A/W)) не блокирует работу приложения до тех пор, пока не будет получено сообщение.

Сама структура MSG имеет следующий вид:

```
typedef struct tagMSG {
  HWND   hwnd;     //Дескриптор окна, которому адресовано сообщение
  UINT   message;  //Код сообщения
  WPARAM wParam;   //Параметр сообщения (зависит от кода)
  LPARAM lParam;   //Параметр сообщения (зависит от кода)
  DWORD  time;     //Время, когда сообщение было помещено в очередь
  POINT  pt;       //Позиция курсора мыши в момент, когда сообщение было
                   //помещено в очередь
} MSG, *PMSG, *LPMSG;
```

Хорошо, с извлечением сообщений из очереди будем считать, что разобрались, теперь переходим к приведению их к аппаратно-независимому виду. Делается это с помощью функции TranslateMessage, вот ее описание:

```
BOOL WINAPI TranslateMessage(
  __in  const MSG *lpMsg       //Адрес структуры MSG
);
```

Данная функция позволяет отделить нажатия функциональных клавиш от символьных. Она преобразовывает виртуальный код клавиши в

символьный, а так же передает его окну[1]. В принципе, данная функция нужна только для корректной обработки сообщений клавиатуры, но обычно ее используют и для других сообщений.

Данная функция возвращает ненулевое значение в одном из двух случаев: 1) сообщение было переведено; 2) полученное сообщение является одним из сообщений: WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, WM_SYSKEYUP. И нулевое значение если сообщение не было переведено.

Теперь нам осталось только передать полученное сообщение в оконную процедуру. Осуществляется это функцией DispatchMessage (A/W), вот как она описывается:

```
LRESULT WINAPI DispatchMessage(
  __in  const MSG *lpmsg  //адрес структуры MSG с передаваемым сообщением
);
```

Возвращает данная функция значение, возвращенное оконной процедурой при обработке переданной ей сообщения.
Сам цикл обработки сообщения может выглядеть следующим образом:

```
StartLoop:
//Извлекаем из очереди сообщений очередное сообщение
invoke GetMessage, msg, NULL, 0, 0
cmp eax, 1
//перейти, на ExitProgramm если получено сообщение WM_QUIT
jb ExitProgramm
//Переводим сообщение
invoke TranslateMessage, msg
//Передаем сообщение в оконную процедуру
invoke DispatchMessage, msg
//Возвращаемся в начало цикла за следующим сообщением
jmp StartLoop
```

Оконная процедура

Оконная процедура осуществляет непосредственную обработку полученных сообщений. Когда мы в цикле обработки сообщений вызываем функцию DuspatchMessage (A/W), операционная система находит окно, которому адресовано сообщение, находит оконную процедуру этого окна и передает ей управление. Вот прототип оконной процедуры:

```
LRESULT CALLBACK WindowProc(
  __in  HWND hwnd,      //дескриптор окна, которому адресовано сообщение
  __in  UINT uMsg,      //код сообщения
  __in  WPARAM wParam,  //параметр сообщения
  __in  LPARAM lParam   //параметр сообщения
);
```

Возвращаемое значение зависит от обрабатываемого сообщения.

По идее данная процедура должна обрабатывать все сообщения, которые получает окно (причем неважно ждет оно их или нет). Но это крайне

[1] Обработка событий клавиатуры будет рассмотрена отдельно

трудоемко и превращает саму процедуру в огромного монстра. Поэтому в Microsoft пошли навстречу разработчикам и придумали такую функцию как DefWindowProc. Она имеет тоже самое описание, что и оконная процедура (единственное она не является функцией обратного вызова). Данная функция осуществляет обработку сообщения «по умолчанию». Это означает, что Windows берет на себя корректную обработку практически всех сообщений (программе нужно будет только вызывать функцию DefWindowProc). Это позволяет программисту реализовывать в своих программах только «нестандартную» обработку. Правда тут нужно понимать, что сама Windows не удалит все созданные пользователем объекты. Это программист должен прописывать сам. Ниже приводится пример оконной процедуры:

```
proc WndProc, hwnd, uMsg, wParam, lParam
     cmp [uMsg], WM_CLOSE
     jz msgCloseWindow
     cmp [uMsg], WM_DESTROY
     jz msgDestroyWindow
     jmp ExitWndProc
msgCloseWindow:
     invoke DestroyWindow, [hwnd]
     jmp ExitWndProc
msgDestroyWindow:
     invoke PostQuitMessage, 0
ExitWndProc:
     invoke DefWindowProc, [hwnd], [uMsg], [wParam], [lParam]
     ret
endp
```

Здесь только одна незнакомая вам функция – PostQuitMessage. Данная функция отправляет приложению сообщение WM_QUIT.

Пример оконного приложения

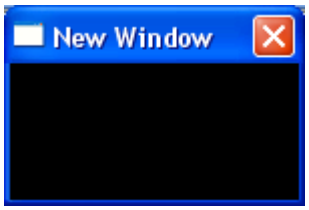
Ну что ж. Теперь я думаю пришло время собрать все наши знания воедино и написать небольшое оконное приложение для закрепления материала. Ниже приводится исходный код такого приложения:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
        push eax
        ;Заполняем структуру WNDCLASS
        mov [wc.style], CS_HREDRAW+CS_DBLCLKS
        mov [wc.lpfnWndProc], WndProc
        mov [wc.cbClsExtra], 0
        mov [wc.cbWndExtra], 0
        mov [wc.hInstance], eax
        mov [wc.hIcon], NULL
        mov [wc.hCursor], NULL
        mov [wc.hbrBackground], COLOR_WINDOWTEXT
        mov [wc.lpszMenuName], NULL
        mov [wc.lpszClassName], _ClassName
```

```
            ;Регистрируем класс
            invoke RegisterClass, wc
            test eax, eax
            jz FailtRegister
            ;Создаем окно
            pop eax
            invoke      CreateWindowEx,0,       _ClassName,       _WindowName,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+WS_OVERLAPPED,    100,    100,    150,    100,
HWND_DESKTOP, NULL, eax, NULL
            test eax, eax
            jz FailtCreate
            ;Цикл обработки сообщений
    StartLoop:
            invoke GetMessage, msg, NULL, 0, 0
            cmp eax, 1
            jb ExitProgramm

            invoke TranslateMessage, msg
            invoke DispatchMessage, msg
            jmp StartLoop

    FailtRegister:
            invoke MessageBox, HWND_DESKTOP,_FailtRegister, NULL, MB_OK
            jmp ExitProgramm
    FailtCreate:
            invoke MessageBox, HWND_DESKTOP, _FailtCreate, NULL, MB_OK
    ExitProgramm:
            invoke ExitProcess, [msg.wParam]
    ;Оконная процедура
    proc WndProc, hwnd, uMsg, wParam, lParam
         cmp [uMsg], WM_CLOSE
         jz msgCloseWindow
         cmp [uMsg], WM_DESTROY
         jz msgDestroyWindow
         jmp ExitWndProc
    msgCloseWindow:
         invoke DestroyWindow, [hwnd]
         jmp ExitWndProc
    msgDestroyWindow:
         invoke PostQuitMessage, 0
    ExitWndProc:
         invoke DefWindowProc, [hwnd], [uMsg], [wParam], [lParam]
         ret
    endp

    section '.data' data readable
    _ClassName    : db 'ClassName', 0
    _WindowName   : db 'New Window', 0
    _FailtRegister: db 'Не удалось зарегистрировать класс', 0
    _FailtCreate  : db 'Не удалось создать окно', 0

    section '.bss' data readable writeable
    wc WNDCLASS
    msg MSG

    section '.idata' import data readable writeable
    library kernel, 'KERNEL32.DLL',\
            user  , 'USER32.DLL'

    import kernel,\
           ExitProcess     , 'ExitProcess',\
           GetModuleHandle , 'GetModuleHandleA'

    import user,\
```

```
DefWindowProc   , 'DefWindowProcA'  ,\
MessageBox      , 'MessageBoxA'     ,\
RegisterClass   , 'RegisterClassA'  ,\
CreateWindowEx  , 'CreateWindowExA' ,\
DestroyWindow   , 'DestroyWindow'   ,\
GetMessage      , 'GetMessageA'     ,\
TranslateMessage, 'TranslateMessage',\
DispatchMessage , 'DispatchMessageA',\
PostQuitMessage , 'PostQuitMessage'
```

Данная программа создает простое окно и выводит его на экран. Само окно представлено на рисунке ниже:



Функции работы с заголовком окна

Выше мы рассмотрели, как изменить атрибуты отображения и месторасположения окна. Теперь поговорим о том, как можно работать с заголовком окна. Это тем более важно, что заголовком окна класса «EDIT» является текст, введенный туда пользователем, а заголовком окна класса «STATIC» (подробнее про окна этих классов смотри в «Элементы управления») является текст, отображаемый этим самым окном.

Изменение заголовка окна

Для того чтобы изменить заголовок окна используется функция SetWindowText (A/W) из библиотеки user32.dll. Вот ее прототип:

```
BOOL WINAPI SetWindowText(
  __in      HWND hWnd,       //Дескриптор окна
  __in_opt  LPCTSTR lpString //Адрес строки с новым заголовком
);
```

Данная функция возвращает ненулевое значение если заголовок окна успешно изменен, и ноль если изменить его не удалось.

Определение длины заголовка

Для того чтобы узнать длину заголовка окна используется функция GetWindowTextLength (A/W) из той же библиотеки. Вот ее прототип:

```
int WINAPI GetWindowTextLength(
  __in  HWND hWnd //Дескриптор окна
);
```

Кто-то может спросить: а зачем нам знать длину заголовка. Это необходимо при работе с данными переменной длины. Например,

пользователь вводит строку, которую мы помещаем во временный буфер. Для того, чтобы определить буфер какой длины нам нужен мы должны знать строку какой длины ввел пользователь. В частности мы воспользуемся этой функцией при рассмотрении динамической памяти.

Длина заголовка возвращается в символах.

Чтение заголовка окна

Для того, чтобы получить заголовок окна используется функция GetWindowText (A/W) из библиотеки user32.dll. Вот как она выглядит:

```
int WINAPI GetWindowText(
  __in   HWND hWnd,       //Дескриптор окна
  __out  LPTSTR lpString, //Адрес буфера, в который следует поместить
                          //заголовок
  __in   int nMaxCount    //Максимальное количество копируемых символов
);
```

Данная функция возвращает количество скопированных символов.

Глава 10. Диалоговые окна

Что такое диалоговое окно

Из примера выше может показаться, что разработка оконных приложений достаточно сложная задача. Действительно, по приведенному выше примеру: для каждого окна нужно определять класс, создавать это самое окно, описывать оконную процедуру (а в любом приложении таких окон несколько). Все это превращает разработку оконного приложения в трудоемкую и неинтересную задачу, при которой разработчик больше занимается логикой создания и отображения окон, нежели логикой самой программы (то зачем он ее разрабатывает). К счастью, тут на помощь приходят диалоговые окна[1], которые значительно упрощают эту задачу.

Диалоговое окно – это окно, созданное на базе предопределенного класса (поэтому нам не нужно описывать и регистрировать класс окна), предназначенное для вывода информации и (или) получения ответа от пользователя. По сути это окна, осуществляющие диалог с пользователем (отсюда и название – диалоговое окно). Диалоговые окна бывают трех видов: немодальные, модальные относительно приложения (или просто модальные), модальные относительно системы (системно-модальные). Теперь по порядку:

Немодальное окно – это диалоговое окно, которое при своем отображении на экране сразу же возвращает управление, создавшему ему процессу.

Модальное окно – это окно, которое приостанавливает работу создавшего его процесса. Примером такого окна служит окно, создаваемое функцией MessageBox(A/W). Программа не сможет продолжить свое исполнение до тех пор, пока пользователь не закроет это окно.

Системное модальное окно – это модальное окно, которое даже будучи в неактивном состоянии перекрывает все другие окна, не являющиеся системно-модальными.

В таблице ниже представлены основные стили диалоговых окон:

Флаг	Описание
DS_3DLOOK	Предоставляет диалоговому окну не полужирный шрифт и чертит трехмерные рамки вокруг окон органов управления в блоке диалога. Этот стиль требуется только прикладным программам, базирующимся на Win32, откомпилированным для версий Windows более ранних, чем Windows 95 или Windows NT 4.0 (Может быть именно поэтому Flatasm и не прописывает этот стиль в самом файле, хотя позволяет задать его для окна). Система автоматически применяет трехмерный вид к диалоговым окнам, созданным

[1] Диалоговое окно иногда называют блок диалога

	прикладными программами, откомпилированными для текущих версий Windows.
DS_ABSALIGN	Указывает, что координаты диалогового окна – экранные координаты. Если этот флаг не установлен, система обрабатывает их как координаты пользователя[1].
DS_CENTER	Центрирует диалоговое окно на экране.
DS_CENTERMOUSE	Центрирует окно по курсору мыши, так, чтобы курсор располагался в центре окна
DS_CONTEXTHELP	Включает вопросительный знак в область заголовка диалогового окна. Когда пользователь щелкает по вопросительному знаку, курсор изменяется в вопросительный знак с указателем. Если пользователь затем щелкает по органу управления в блоке диалога, элементу управления посылается сообщение WM_HELP.
DS_CONTROL	Окно является дочерним
DS_FIXEDSYS	Устанавливает шрифт окна совместимый с системным шрифтом в версиях Windows ранее, чем 3.0.
DS_LOCALEDIT	Применяется только в 16-разрядных прикладных программах (Flatasm его опускает). Этот флаг предписывает полям редактирования в диалоговом окне назначать память в сегменте данных приложения. Иначе, поле редактирования назначает память в объекте глобальной памяти.
DS_MODALFRAME	Создает модальное диалоговое окно
DS_NOFAILCREATE	Windows 95 только: создает диалоговое окно, даже если происходят ошибки – например, если дочернее окно не может быть создано или если система не может создать специальный сегмент данных для поля редактирования.
DS_NOIDLEMSG	Подавляет сообщения WM_ENTERIDLE[2], которые Windows, иначе отправил бы владельцу диалогового окна, в то время когда блок диалога показывается на экране.

[1] Для экранных координат началом отсчета является левый верхний угол экрана. Началом отсчета клиентских координат является верхний левый угол клиентской области элемента управления или формы. Применение клиентских координат гарантирует, что приложение может использовать согласованные значения координат во время рисования в форме или элементе управления, независимо от положения формы или элемента управления на экране. Более подробно про клиентские, экранные и другие координаты смотри в литературе по компьютерной графике

[2] Сообщение WM_ENTERIDLE отправляется окну владельцу модального окна или меню, которое вводит состояние «не занято». Данное состояние вводится когда в очереди окна нет никаких сообщений, которые ожидают обработки после того, как оно обработало одно или несколько предыдущих сообщений.

DS_SETFONT	Указывает, что заголовок расширенного шаблона диалогового окна DLGTEMPLATEEX[1] содержит четыре дополнительных члена (pointsize, weight, bltalic, font), которые описывают шрифт, использующийся для текста в рабочей области и органах управления диалога. Если возможно, система создает шрифт согласно значениям, заданным в этих членах. Затем система передает дескриптор шрифта в диалоговое окно и в каждый орган управления, отправляя им сообщение WM_SETFONT.
DS_SETFOREGROUND	Вынуждает систему использовать функцию SetForegroundWindow[2], чтобы привести диалоговое окно в режим переднего плана.
DS_SYSMODAL	Создает системно-модальное окно.

Все что нужно для создания диалогового окна это описать его шаблон (в ресурсах или в памяти) и создать на основе этого шаблона само диалоговое окно.

Взаимодействие с пользователем диалоговое окно осуществляет посредством элементов управления.

Элемент управление – это окно, являющееся дочерним по отношению к диалоговому окну, построенное на предопределенном оконном классе. Примерами элементов управления являются: кнопка, поле вода, статический текст и др. Элементы управления имеют следующие преимущества перед обычными окнами:

1) Для элемента управления не нужно определять оконную процедуру, она уже имеется в недрах Windows. Всю обработку сообщений, поступающих элементу управления, осуществляет ОС, передавая диалоговому окну соответствующее уведомительное сообщение.
2) Все элементы управления создаются вместе с диалоговым окном. Их не нужно создавать отдельно

Одно из преимуществ диалоговых окон перед обычными окнами состоит в том, что для первых не нужно задавать цикл обработки сообщений. Вся обработка сообщений осуществляется диалоговой процедурой.

[1] Данная структура используется при создании диалогового окна с использованием динамически определяемого шаблона.

[2] Эта функция переводит поток, который создал определяемое окно в приоритетный режим и активизирует окно. Ввод с клавиатуры направлен в окно, а различные визуальные ориентиры изменяются для пользователя.

Диалоговая процедура

По своему назначению диалоговая процедура идентична оконной процедуре, но все же, между ними существует ряд отличий. Вот некоторые из них:

1) Диалоговая процедура может возвращать либо 1 (TRUE: сообщение обработано), либо 0 (FALSE: сообщение не обработано).
2) Если оконная процедура не обрабатывает какое-либо сообщение, то она должна вызвать функцию DefWindowProc и передать в нее это сообщение. Если диалоговая процедура не обрабатывает какое-либо сообщение, она просто возвращает ноль (FALSE): система сама обработает это сообщение так, как оно обрабатывается по умолчанию.
3) В диалоговую процедуру не передается сообщение WM_CREATE, вместо него передается WM_INITDIALOG, которое предлагает проинициализировать (то есть заполнить начальными значениями) все необходимые данные.

Сама диалоговая процедура описывается следующим образом:

```
INT_PTR CALLBACK DialogProc(
  __in  HWND hwndDlg,        //Дескриптор окна, которому адресовано сообщение
  __in  UINT uMsg,           //Код сообщения
  __in  WPARAM wParam,       //Параметр сообщения
  __in  LPARAM lParam        //Параметр сообщения
);
```

Ниже приводится пример диалоговой процедуры:

```
proc DialogProc hwnddlg, msg, wparam, lparam
     ;Мы не обрабатываем сообщения, которые к нам приходят
     xor eax, eax
     ;Это сообщение закрытия окна?
     cmp [msg], WM_CLOSE
     ;Если да то переходим к освобождению окна
     je FreeDialog
     ;Выход из оконной процедуры
     ret
     ;Участок кода ответственный за освобождение окна
FreeDialog:
     ;Уничтожаем диалоговое окно
     invoke EndDialog, [hwnddlg], 0
     ;Выход из оконной процедуры
     xor eax,eax
     ret
endp
```

Функция EndDialog уничтожает модальное диалоговое окно. Вот ее описание:

```
BOOL WINAPI EndDialog(
  __in  HWND hDlg,        //Дескриптор уничтожаемого окна
  __in  INT_PTR nResult   //Значение, которое будет возвращено в приложение,
                          //создавшее уничтожаемое окно
);
```

Фактически nResult – это то значение, которое вернет функция создания уничтожаемого окна.

Выше мы упоминали, что диалоговая процедура осуществляет обработку сообщений не только самого диалогового окна, но и элементов управления этого окна. Осуществляется это с помощью уведомлений (не путать с сообщениями). Когда элемент управление получает какое-либо сообщение (то есть в системе произошло какое-то событие, связанное с ним: например, пользователь нажал на копку, являющуюся элементом управления), операционная система формирует уведомление и передает его в диалоговую процедуру сообщением WM_COMMAND. Вот об этом сообщении и поговорим:

Сообщение WM_COMMAND

Итак, сообщение WM_COMMAND передает в диалоговую процедуру уведомление от элемента управления данного диалогового окна. Уведомление передается в параметре wParam данного сообщения. Его старшее слово содержит код уведомления (что произошло), а младшее слово идентификатор элемента управления (с кем произошло).

Поясню на примере. Допустим, у нас есть элемент управления кнопка с идентификатором 2. Тогда, получить значение параметра wParam при одинарном клике на эту кнопку (уведомление BN_CLICKED) можно следующим образом:

```
;Помещаем в регистр EAX код уведомления
mov eax, BN_CLICKED
;Передвигаем его в старшее слово
shl eax, 16
;Добавляем идентификатор элемента
add eax, 2
```

В результате выполнения этого кода в регистре EAX будет искомое значение параметра wParam, которое будет передано с сообщением WM_COMMAND.

Описание шаблона диалогового окна в ресурсах

Описание шаблона диалогового окна в ресурсах имеет вид:

```
dialog demonstration, 'Window', 70, 80, 190, 175, WS_CAPTION +
                                                  WS_POPUP+WS_SYSMENU+DS_CENTER
        ;описание элементов управления диалогового окна
enddialog
```

Здесь раздел (dialog … enddialog) задает область описания шаблона (их может быть несколько). Теперь разберем параметры:

“demonstration” — псевдоним шаблона, присвоенный ему на этапе задания языка;

‘Window’ — заголовок окна;

70 — положение окна по горизонтали;

80 — положение окна по вертикали;

190—ширина окна;

175—высота окна;

`WS_CAPTION+WS_POPUP+WS_SYSMENU+DS_CENTER` – набор стилей окна.

Создание диалогового окна на основе ресурсов

Практически все диалоговые окна (исключение составляет разве что окно создаваемое функцией MessageBox (A/W)) создаются на основе так называемого шаблона диалогового окна. Данный шаблон может быть статическим (строго фиксированным и неизменным) иди динамическим (создаваться самой программой в процессе своего исполнения). Статический шаблон обычно хранится в ресурсах приложения. О том, как он описывается мы говорили выше, теперь поговорим о том, как на его основе создать диалоговое окно.

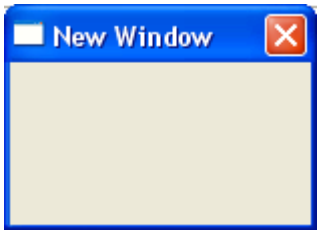
Для создания модального диалогового окна из ресурсов используется функция DialogBoxParam (A/W). Выглядит она следующим образом:

```
INT_PTR WINAPI DialogBoxParam(
  __in_opt  HINSTANCE hInstance,    //Дескриптор модуля, в котором описан
                                    //шаблон
  __in      LPCTSTR lpTemplateName,//Идентификатор ресурса, задающего шаблон
  __in_opt  HWND hWndParent,        //Дескриптор родительского окна
  __in_opt  DLGPROC lpDialogFunc,   //Адрес диалоговой процедуры
  __in      LPARAM dwInitParam      //Значение, которое будет передано в
                                    //диалоговую процедуру с сообщением
                                    //WM_INITDIALOG
);
```

В случае успеха данная функция возвращает значение, которое было передано в функцию EndDialog при закрытии созданного ею диалогового окна. Ниже приводится пример использования этой функции:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable

start:
        invoke GetModuleHandle, 0
        ;Создаем модальное диалоговое окно на основе шаблона
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0
;Диалоговая процедура
proc DialogProc hwnddlg, msg, wparam, lparam
     xor eax, eax
     cmp [msg], WM_CLOSE
     je FreeDialog
     ret
FreeDialog:
     invoke EndDialog, [hwnddlg], 0
     xor eax,eax
     ret
endp

section '.idata' import data readable writeable
 library kernel, 'KERNEL32.DLL',\
         user  , 'USER32.DLL'

 import kernel,\
        GetModuleHandle, 'GetModuleHandleA',\
        ExitProcess    , 'ExitProcess'

 import user,\
        DialogBoxParam, 'DialogBoxParamA',\
        EndDialog     , 'EndDialog'
;Шаблон диалогового окна
section '.rsrc' resource data readable
 directory RT_DIALOG, dialogs

 resource dialogs,\
        1,LANG_ENGLISH, form1

 dialog     form1,     'New     Window',     100,     100,     100,     50,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU
 enddialog
```

Данная программа создает простое диалоговое окно без элементов управления:



Если сравнить эту программу с программой создания похожего не диалогового окна, то мы увидим, насколько диалоговые окна упрощают разработку оконных приложений.

Создание диалогового окна на основе шаблона в памяти

Для создания модального диалогового окна, на основе динамически определяемого шаблона используют функцию DialogBoxIndirectParam (A/W). Вот как она выглядит:

```
INT_PTR WINAPI DialogBoxIndirectParam(
  __in_opt  HINSTANCE hInstance,             //Дескриптор модуля
  __in      LPCDLGTEMPLATE hDialogTemplate, //Адрес шаблона в памяти
  __in_opt  HWND hWndParent,                 //Дескриптор родительского окна
  __in_opt  DLGPROC lpDialogFunc,            //Адрес диалоговой процедуры
  __in      LPARAM dwInitParam               //Значение, которое будет
                                             //передано в диалоговую процедуру
                                             //с сообщением WM_INITDIALOG
);
```

Сам шаблон диалогового окна представляет собой буфер (ограниченная последовательность байт) определенного формата. Начинается он со структуры DLGTEMPLATE, которая имеет следующий прототип:

```
typedef struct {
  DWORD style;           //Набор стилей окна (используются как стили обычных
                         //окон так и стили диалоговых окон)
  DWORD dwExtendedStyle; //Набор расширенных стилей окна
  WORD  cdit;            //Количество элементов управления диалогового окна
  short x;               //Положение окна по горизонтали
  short y;               //Положение окна по вертикали
  short cx;              //Ширина окна
  short cy;              //Высота окна
} DLGTEMPLATE, *LPDLGTEMPLATE;
```

Сразу за ней идет массив меню, который определяет меню диалогового окна. Если первый элемент этого массива равен 0000h, значит, у диалогового окна нет меню. Если он равен FFFFh, то следом за ним идет идентификатор меню в ресурсах. Если же первый элемент отличен и от 0000h и FFFFh система интерпретирует данный массив как Unicode строку, задающую имя ресурса меню. Признаком конца строки служит последовательность 0000h (NULL байт в кодировке Unicode).

Следом идет массив класса, который определяет класс окна. Если массив равен 0000h, то система использует предопределенный класс диалогового окна, и массив не имеет больше элементов. Если первый элемент этого массива равен FFFFh, то следом за ним идет код предопределенного класса. Если же первый элемент массива отличен и от 0000h и от FFFFh то система интерпретирует этот строку как Unicode строку, представляющую собой имя используемого класса, с NULL-символом на конце.

Внимательный читатель, наверное, заметит: какой класс, ведь ты говорил, что при создании диалогового окна не нужно регистрировать никакого класса. Действительно диалоговые окна в подавляющем большинстве случаев строятся на стандартных оконных классах, но это не значит, что для построения диалогового нельзя использовать какой-либо

собственный класс. Использовать нестандартный класс окна можно в том случае, если стандартный класс вас не устраивает по какой-либо причине.

После массива класса идет описание заголовка окна, которое представляет собой Unicode строку, содержащую заголовок окна. Если заголовок окна равен 0000h, то у окна нет заголовка.

Если у диалогового окна установлен стиль DS_SETFONT то следом идет 16-разрядное значение размера шрифта в пунктах и сам шрифт в виде Unicode строки.

После описания шрифта идет массив элементов управления. Количество элементов этого массива определяется параметром cdit структуры DLGTEMPLATE рассмотренной выше. Каждый элемент этого массива определяет один элемент управления диалогового окна и представляет собой структуру DLGITEMTEMPLATE. Выглядит она следующим образом:

```
typedef struct {
  DWORD style;           //Набор стилей элемента управления (могут
                         //использоваться как оконные стили, так и стили
                         //определенного класса элемента управления)
  DWORD dwExtendedStyle; //Набор расширенных стилей окна
  short x;               //Положение элемента по горизонтали
  short y;               //Положение элемента по вертикали
  short cx;              //Ширина
  short cy;              //Высота
  WORD  id;              //Идентификатор элемента управления
} DLGITEMTEMPLATE, *PDLGITEMTEMPLATE;
```

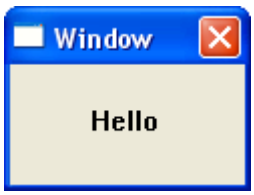
За каждой структурой DLGITEMTEMPLATE идет массив класса, который определяет тип (класс) элемента управления. Если первый элемент этого массива равен FFFFh, то следующее слово интерпретируется как код предопределенного класса, в противном случае система интерпретирует массив как Unicode строку, представляющую собой имя класса[1] элемента управления.

Следом за ним находится массив заголовка, который содержит в себе заголовок элемента управления. Если первый элемент этого массива равен FFFFh, то следом за ним идет идентификатор ресурса, такого как иконка. Если первый элемент массива отличен от FFFFh, то система интерпретирует его как Unicode строку заголовка окна элемента управления.

Массивы классов и заголовка должны быть выровнены по границам слова. Элементы массива структур DLGITEMTEMPLATE должны быть выровнены по границам двойного слова.

Для того чтобы стало более понятно, что к чему, рассмотрим небольшое приложение, создающее диалоговое окно на основе шаблона в памяти. А также разберем по косточкам используемый им шаблон. Предположим мы хотим создать из шаблона диалоговое окно, имеющее вид:

[1] Подробнее про элементы управления и соответствующие им классы смотри в разделе «Элементы управления»



Тогда программа, создающая такое диалоговое окно из шаблона, выгляди так:

```
format PE GUI 4.0
entry start
include 'F:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
        ;Создаем диалоговое окно
 invoke DialogBoxIndirectParam, eax, _DialogTemplate, NULL, DialogProc, 0
        invoke ExitProcess, 0
        ;Диалоговая процедура
        proc DialogProc hwnddlg, msg, wparam, lparam
             xor eax, eax
             cmp [msg], WM_CLOSE
             je FreeDialog
             ret
FreeDialog:
             invoke EndDialog, [hwnddlg], 0
             xor eax,eax
             ret
        endp

section '.data' data readable
_DialogTemplate:
;Описываем шаблон диалогового окна
;Структура DLGTEMPLATE
dd WS_VISIBLE+DS_CENTER+WS_CAPTION+WS_SYSMENU  ;style
dd 00h                    ;dwExtendedStyle
dw 01h                    ;cdit
dw 100                    ;x
dw 100                    ;y
dw  30                    ;cx
dw  30                    ;cy
;Массив меню
dw 00h    ;у нас нет меню
;Массив класса
dw 00h   ;используем стандартный класс
;Массив заголовка диалогового окна
dw 0057h,\ ;W
   0069h,\ ;i
   006Eh,\ ;n
   0064h,\ ;d
   006Fh,\ ;o
   0077h,\ ;w
   0000h
;Описывать массив шрифта нам не нужно (не установлен стиль DS_SETFONT)
;Начинается массив элементов управления
;Структура DLGITEMTEMPLATE первого (и единственного) элемента управления
dd WS_VISIBLE ;style
dd 00h        ;dwExtendedStyle
dw 20         ;x
dw 10         ;y
dw 50         ;cx
```

```
    dw 20          ;cy
    dw 01          ;id
    ;Массив класса элемента управления
    dw 0FFFFh
    dw  0082h ;STATIC
    ;Массив заголовка элемента управления
    dw  0048h ,\ ;H
        0065h ,\ ;e
        006Ch ,\ ;l
        006Ch ,\ ;l
        006Fh ,\ ;o
        0000h
    ;По сути дальше идет описание следующего элемента, но у нас их больше
    ;нет, поэтому шаблон окончен

    section '.idata' import data readable writeable
    library kernel, 'KERNEL32.DLL',\
            user,   'USER32.DLL'

    import kernel,\
           ExitProcess, 'ExitProcess',\
           GetModuleHandle, 'GetModuleHandleA'

    import user,\
           DialogBoxIndirectParam, 'DialogBoxIndirectParamA',\
           EndDialog, 'EndDialog'
```

Немодальные диалоговые окна

Выше мы обсуждали только модальные диалоговые окна. Теперь поговорим о немодальных диалоговых окнах. В отличие от своих «собратьев» они возвращают управление в создавшую их процедуру сразу после своего создания, не дожидаясь закрытия окна.

Создать немодальное диалоговое окно из шаблона в ресурсах можно функцией CreateDialogParam (A/W). Выглядит она так:

```
HWND WINAPI CreateDialogParam(
  __in_opt  HINSTANCE hInstance,     //Дескриптор модуля
  __in      LPCTSTR lpTemplateName, //Идентификатор ресурса шаблона
  __in_opt  HWND hWndParent,        //Дескриптор родительского окна
  __in_opt  DLGPROC lpDialogFunc,   //Адрес диалоговой процедуры
  __in      LPARAM dwInitParam      //Значение, которое будет передано
                                    //диалоговую процедуру с сообщением
                                    //WM_INITDIALOG
);
```

Как видно из описания данная функция принимает те же параметры, что и функция DialogBoxParam (A/M). CreateDialogParam (A/W) возвращает дескриптор созданного диалогового окна или ноль, если создать окно не удалось.

Для создания немодального диалогового окна на основе шаблона в памяти используется функция CreateDialogIndirectParam(A/W), которая выглядит так:

```
HWND WINAPI CreateDialogIndirectParam(
  __in_opt  HINSTANCE hInstance,        //Дескриптор модуля
  __in      LPCDLGTEMPLATE lpTemplate,  //Адрес шаблона
  __in_opt  HWND hWndParent,            //Дескриптор родительского окна
  __in_opt  DLGPROC lpDialogFunc,       //Адрес оконной процедуры
  __in      LPARAM lParamInit           //Значение, которое будет передано в
                                        //диалоговую процедуру
                                        //вместе с сообщением WM_INITDIALOG
);
```

Данная функция возвращает дескриптор созданного диалогового окна или ноль, если создать окно не удалось.

Теперь давайте взглянем на простое приложение, работающее с немодальным диалоговым окном:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable

start:
        invoke GetModuleHandle, 0
        ;Создаем немодальное диалоговое окно на основе ресурсов
        invoke CreateDialogParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        mov dword [_hDialog], eax
        ;Цикл обработки сообщений
StartLoop:
        invoke GetMessage, _msg, NULL, 0, 0
        cmp eax, 1
        jb ExitProgramm

        invoke IsDialogMessage, [_hDialog], _msg
        jmp StartLoop

ExitProgramm:
        invoke ExitProcess, 0
;Оконная процедура
proc DialogProc hwnddlg, msg, wparam, lparam
     xor eax, eax
     cmp [msg], WM_CLOSE
     je DestroyDialog
     ret
DestroyDialog:
     invoke DestroyWindow, [hwnddlg]
     invoke PostQuitMessage, 0
     xor eax,eax
     ret
endp

section '.bss' data readable writeable
_msg MSG
_hDialog: dd 00h

section '.idata' import data readable writeable

 library kernel, 'KERNEL32.DLL',\
         user, 'USER32.DLL'

 import kernel,\
        GetModuleHandle, 'GetModuleHandleA',\
        ExitProcess, 'ExitProcess' ,\
```

```
            Sleep, 'Sleep'

     import user,\
            CreateDialogParam, 'CreateDialogParamA',\
            DestroyWindow, 'DestroyWindow',\
            GetMessage, 'GetMessageA',\
            IsDialogMessage, 'IsDialogMessageA',\
            PostQuitMessage, 'PostQuitMessage'

    section '.rsrc' resource data readable
     directory RT_DIALOG, dialogs

     resource dialogs,\
            1,LANG_ENGLISH, form1

     dialog     form1,     'New     Window',     100,     100,     100,     50,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+DS_CENTER
     enddialog
```

Первым, что бросается в глаза при просмотре этого листинга это цикл обработки сообщений. Но он отличается от того цикла, который мы создавали при работе с обычными окнами. Здесь не вызывается ни функция TranslateMessage ни DispatchMessage(A/W), но вызывается функция IsDialogMessage(A/W). Вот как она выглядит:

```
BOOL WINAPI IsDialogMessage(
  __in  HWND hDlg,   //Дескриптор окна, которому адресовано сообщение
  __in  LPMSG lpMsg  //Адрес структуры MSG с полученным сообщением
);
```

Данная функция проверяет: если сообщение адресовано диалоговому окну, то оно передается в соответствующую диалоговую процедуру, при этом функция возвращает значение TRUE. Если же сообщение адресовано не диалоговому окну она просто возвращает значение FALSE, при этом само сообщение не претерпевает никаких изменений.

Также можно заметить, что в диалоговой процедуре не вызывается функция EndDialog, вместо нее вызывается DestroyWindow как в оконной процедуре.

Глава 11. Элементы управления

Типы элементов управления

Ранее мы говорили о том, что элементы управления представляют собой окна, дочерние по отношению к диалоговому окну, построенные на предопределенных оконных классах. Для элементов управления не нужно задавать оконную процедуру, так как всю работу по обработке сообщений берет на себя Windows, отправляя в диалоговую процедуру лишь уведомления о наступлении того или иного события, связанного с данным элементом управления.

Для элементов управления доступны следующие предопределенные оконные классы:

Код класса	Строковое представление класса	Описание
0080h	Button	Кнопка
0081h	Edit	Поле ввода
0082h	Static	Статический элемент управления (надпись или рисунок)
0083h	List box	Список
0084h	Scroll bar	Полоса прокрутки
0085h	Combo box	Комбинированный список

Позже мы рассмотрим некоторые из этих классов более подробно.

Описание элемента управления в ресурсах

Описание элемента управления в ресурсах имеет следующий вид:

```
dialogitem 'CLASS', 'Caption', 1,10,15,70,8,WS_VISIBLE
```

Здесь: dialoitem — зарезервированное слово;

“CLASS” — класс создаваемого элемента управления

‘Caption’ — заголовок элемента управления;

1 — идентификатор элемента;

10 — расположение элемента по горизонтали;

15— расположение элемента по вертикали;

70 — ширина элемента;

8 — высота элемента;

`WS_VISIBLE` – Набор стилей данного элемента (представляет собой смесь оконных стилей со стилями специфичными для данного элемента управления);

При задании шаблона диалогового окна в памяти его элементы управления описываются структурой DLGITEMTEMPLATE. Она уже рассматривалась нами в разделе «Создание диалогового окна на основе шаблона в памяти». Повторяться я не буду.

Идентификатор и дескриптор

Пришло время обсудить два очень похожих понятия: идентификатор и дескриптор.

Идентификатор – это числовое обозначение элемента управления размером в слово, назначаемое программистом на стадии описания этого элемента управления. Уникальность идентификаторов обеспечивается только в пределах одного диалогового окна. Это значит, что вы можете иметь два и более диалоговых окна, у каждого из которых будет элемент управления с идентификатором равным, например, единице.

Дескриптор же в отличие от идентификатора назначается системой в момент создания элемента управления. В этом состоит их главное (и пожалуй единственное) отличие друг от друга.

Они оба выполняют одну и ту же функцию: позволяют программе обращаться к элементу управления, управлять им. Но почему нельзя обойтись одним дескриптором, спросите вы? А как мы его узнаем, спрошу я вас?

Рассмотрим, такую ситуацию: мы создаем диалоговое окно, на котором расположено поле ввода. Система сообщит нам лишь дескриптор диалогового окна, а вот дескриптор поля ввода останется нам неизвестным. И как тогда нам обратиться к этому полю ввода? Да, конечно, можно поискать его среди окон дочерних по отношению к диалоговому окну, но: во-первых, это не красиво, а во-вторых, полей ввода может быть несколько: как тогда определить какое из них нам нужно?

Вот тут и приходят на выручку идентификаторы. Ведь мы их знаем для всех элементов управления (мы сами их задали при описании шаблона диалогового окна). Как будет показано ниже, целый ряд функций работы с элементами управления в качестве параметров воспринимают не дескриптор, а идентификатор нужного элемента управления. Это значительно облегчает труд программиста.

Если же вам все равно нужно знать дескриптор элемента управления, то выяснить его можно с помощью функции GetDlgItem:

```
HWND WINAPI GetDlgItem(
  __in_opt  HWND hDlg,      //Дескриптор диалогового окна
  __in      int nIDDlgItem  //Идентификатор элемента управления
);
```

Данная функция возвращает дескриптор элемента управления по его идентификатору. Если определить дескриптор не удалось, возвращается ноль.

Функции работы с элементами управления

Данные функции предназначены для взаимодействия с элементами управления. Наиболее известной из них является функция GetDlgItemText(A/W). Вот как она выглядит:

```
UINT WINAPI GetDlgItemText(
  __in   HWND hDlg,        //Дескриптор диалогового окна
  __in   int nIDDlgItem,   //Идентификатор элемента управления
  __out  LPTSTR lpString, //Адрес буфера, в который следует скопировать
                           //заголовок
  __in   int nMaxCount     //Максимальное количество копируемых символов
                           //считая завершающий нуль
);
```

Как вы уже, наверное, догадались из описания, данная функция предназначена для копирования заголовка элемента управления в строковый буфер, расположенный по адресу lpString. Данная функция возвращает количество скопированных символов.

Противоположной ей функцией является функция SetDlgItemText(A/W). Вот она:

```
BOOL WINAPI SetDlgItemText(
  __in  HWND hDlg,         //Дескриптор диалогового окна
  __in  int nIDDlgItem,    //Идентификатор элемента управления
  __in  LPCTSTR lpString   //Адрес строки с новым заголовком
);
```

Внимательный читатель наверняка уже заметил, что приведенные функции очень похожи на функции SetWindowText(A/W) и GetWindowText (A/W) рассмотренные нами при разговоре об обычных окнах. Действительно, все эти функции делают одно и то же. Единственное различие между ними состоит в адресации целевого окна. Функции SetWindowText(A/W), GetWindowText(A/W) требуют указания дескриптора окна. А функции SetDlgItemText(A/W) и GetDlgItemText(A/W) ожидают от нас идентификатор окна (разницу между этими двумя понятиями мы уже объясняли в предыдущем разделе).

Из заголовка поля ввода можно читать не только строки, но и числа. Осуществляется это функцией GetDlgItemInt:

```
UINT WINAPI GetDlgItemInt(
  __in       HWND hDlg,          //Дескриптор диалогового окна
  __in       int nIDDlgItem,     //Идентификатор элемента
  __out_opt  BOOL *lpTranslated,//Признак успешности преобразования
  __in       BOOL bSigned        //Признак знака
);
```

Параметр lpTranslated является адресом, по которому будет записан признак успешности преобразования. Если после вызова функции он равен FALSE, значит, введенное пользователем значение не удалось преобразовать

к числу (сама функция возвращает при этом ноль). Если же этот параметр равен TRUE, то преобразование выполнено успешно.

Параметр bSigned определяет, может ли пользователь вводить отрицательные числа.

Функция возвращает прочитанное число.

Противоположной ей по назначению является функция SetDlgItemInt:

```
BOOL WINAPI SetDlgItemInt(
  __in  HWND hDlg,        //Дескриптор диалогового окна
  __in  int nIDDlgItem,   //Идентификатор элемента управления
  __in  UINT uValue,      //Устанавливаемое значение
  __in  BOOL bSigned      //Мы устанавливаем значение со знаком или без знака
);
```

В случае успеха функция возвращает ненулевое значение. Если же установить значение не удалось, возвращается ноль.

В отношении элементов управления вы можете не только читать и изменять их заголовок, но и отправлять им сообщения. Осуществляется это функцией SendDlgItemMessage(A/W):

```
LRESULT WINAPI SendDlgItemMessage(
  __in  HWND hDlg,       //Дескриптор диалогового окна
  __in  int nIDDlgItem, //Идентификатор элемента управления
  __in  UINT Msg,        //Код сообщения
  __in  WPARAM wParam,   //Параметр сообщения
  __in  LPARAM lParam    //Параметр сообщения
);
```

Отправлять элементу управления вы можете как обычные оконные сообщения, общие для всех окон, так и специальные, зависящие от класса элемента управления.

Как видно из выше приведенных функций вся работа с элементами управления осуществляется посредством их идентификаторов. Дескрипторы нам не нужны.

Ну что ж, с общей теорией будем считать, что разобрались. Теперь пришло время познакомиться с различными классами элементов управления поближе.

Статический элемент управления
Теория

Это самый простой элемент управления из всех. В основе лежит класс Static. По сути, это просто текст или изображение.

В таблице ниже представлены стили данного элемента управления

Стиль	Описание
SS_BITMAP	Определяет, что в статическом элементе управления должен отобразиться рисунок. Элемент управления автоматически устанавливает собственные размеры, чтобы поместить точечный рисунок
SS_BLACKFRAME	Определяет окно с рамкой черного цвета
SS_BLACKRECT	Определяет прямоугольник, заполненный черным цветом
SS_CENTER	Определяет простой прямоугольник и выравнивает по центру текст. Текст форматируется перед отображением его на экране. Слова, которые выходят за пределы конца строки автоматически переносятся на следующую строку и там выравниваются
SS_CENTERIMAGE	Располагает изображение в центре статического элемента управления
SS_ENHMETAFILE	Стиль для расширенных метафайлов. То есть изображений сохраненный в виде последовательности команд и параметров рисования
SS_ETCHEDFRAME	Рисует прямоугольник с «выдавленными» границами
SS_GRAYFRAME	Определяет поле окна с рамкой, того же серого цвета.
SS_GRAYRECT	Определяет прямоугольник, заполненный серым цветом
SS_ICON	В этом элементе будет размещаться иконка
SS_LEFT	Определяет простой прямоугольник и выравнивание текста по левому краю. Текст форматируется перед его отображением. Слова, которые выходят за границы строки автоматически переносятся в начало следующей строки.
SS_LEFTNOWORDWRAP	Определяет простой прямоугольник и выравнивание текста по левому краю. Текст, который выходит за пределы конца строки

	отсекается.
SS_NOTIFY	Разрешает посылку родительскому окну уведомлений STN_CLICKED, STN_DBLCLK, STN_DISABLE и STN_ENABLE с сообщением WM_COMMAND
SS_OWNERDRAW	Создается элемент управления, внешний вид которого должна контролировать программа. То есть Windows самостоятельно его никак не отображает
SS_REALSIZEIMAGE	Графическое изображение изменяет размер в соответствии с заданной для статического окна областью
SS_RIGHT	Определяет простой прямоугольник и выравнивает по правому краю текст. Слова, которые выходят за пределы строки автоматически переносятся в начало следующей выровненной по правой границе строки.
SS_RIGHTJUST	Определяет, что при автоматическом изменении размера элемента управления в соответствии с размером загружаемого графического изображения фиксируется правый нижний угол статического окна (по умолчанию – верхний левый)
SS_SIMPLE	Определяет простой прямоугольник и отображает одиночную строку выровненного по левому краю текста.
SS_SUNKEN	Статическое окно отображается в виде «утопленного» прямоугольника
SS_WHITEFRAME	Определяет поле окна с рамкой белого цвета
SS_WHITERECT	Определяет прямоугольник, заполненный белым цветом.

В таблице ниже представлены уведомления, которые может отсылать статический элемент управления:

Уведомление	**Описание**
STN_CLICKED	Одинарный щелчок мышью по элементу
STN_DBLCLK	Двойной щелчок мышью по элементу
STN_DISABLE	Отправляется, когда элемент становится недоступным
STN_ENABLE	Отправляется, когда элемент становится доступным

Данные уведомления будут отправляться только в том случае, если у статического элемента управления установлен стиль SS_NOTIFY.

Статический элемент управления способен обрабатывать следующие сообщения:

Сообщение	**Описание**
STM_GETICON	Позволяет получить дескриптор иконки, связанной с элементом управления
STM_GETIMAGE	Позволяет получить дескриптор изображения связанного с элементом управления
STM_SETICON	Устанавливает иконку
STM_SETIMAGE	Устанавливает изображение

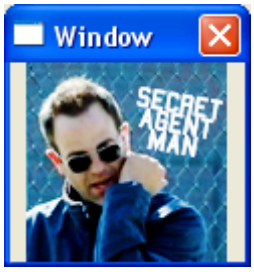
Практика

Напишем небольшое приложение, демонстрирующее работу с этим элементом управления. В этом приложении при создании диалогового окна в статический элемент управления будет загружаться картинка из ресурсов. Вот само приложение:

```
    format PE GUI 4.0
    entry start
    include 'D:\FASM1\INCLUDE\win32a.inc'
    ;Перечень идентификаторов
    ID_DIALOG = 1  ;шаблона диалогового окна
    ID_STATIC = 2  ;Статического элемента управления
    ID_BITMAP = 3  ;изображения

    section '.code' code readable executable
    start:
            invoke GetModuleHandle, 0
            mov dword [_hModule], eax
            invoke DialogBoxParam, eax, ID_DIALOG,HWND_DESKTOP, DialogProc, 0
            invoke ExitProcess, 0
    ;Диалоговая процедура
    proc DialogProc hwnddlg, msg, wParam, lParam
         cmp [msg], WM_CLOSE
         je FreeDialog
         cmp [msg], WM_INITDIALOG
;если получено сообщение отличное от WM_INITDIALOG, то выходим из процедуры
         jne ExitProc
         ;Загружаем изображение из ресурсов
         invoke LoadImage, [_hModule], ID_BITMAP, IMAGE_BITMAP, 0, 0,
LR_DEFAULTCOLOR+LR_DEFAULTSIZE
         ;Сохраняем дескриптор загруженного изображения в памяти
         mov dword [_hBitmap], eax
         ;С помощью сообщения STM_SETIMAGE устанавливаем изображение в
         ;статическом элементе управления
         invoke SendDlgItemMessage, [hwnddlg], ID_STATIC, STM_SETIMAGE,
IMAGE_BITMAP, eax
         jmp ExitProc
    FreeDialog:
         ;Уничтожаем диалоговое окно
         invoke EndDialog,[hwnddlg], 0
         ;Уничтожаем загруженное ранее изображение
```

```
        invoke DeleteObject, [_hBitmap]
    ExitProc:
        xor eax, eax
        ret
    endp

    section '.bss' data readable writeable
    ;Здесь будут храниться дескрипторы
    _hModule: dd 00h ;модуля
    _hBitmap: dd 00h ;изображения
    ;Таблица импорта
    section '.idata' data import readable writeable
    library kernel, 'KERNEL32.DLL',\
            user  , 'USER32.DLL'  ,\
            gdi   , 'GDI32.DLL'

    import kernel,\
           GetModuleHandle, 'GetModuleHandleA',\
           ExitProcess    , 'ExitProcess'

    import user,\
           DialogBoxParam, 'DialogBoxParamA',\
           EndDialog     , 'EndDialog'      ,\
           LoadImage     , 'LoadImageA'     ,\
           SendDlgItemMessage, 'SendDlgItemMessageA'

    import gdi,\
           DeleteObject  , 'DeleteObject'
    ;Ресурсы
    section '.rsrc' resource data readable
    directory RT_DIALOG, dialogs,\
              RT_BITMAP, bitmaps

    resource dialogs,\
            ID_DIALOG, LANG_NEUTRAL, MainForm

    resource bitmaps,\
            ID_BITMAP,LANG_NEUTRAL, main_bitmap

    dialog MainForm, 'Window', 0, 0, 0, 61,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+DS_CENTER
           dialogitem 'STATIC', '', ID_STATIC, 5, 0, 0, 50,
WS_VISIBLE+SS_BITMAP;+SS_CENTERIMAGE
    enddialog

    bitmap main_bitmap, 'picture.bmp'
```

В итоге мы получим приблизительно следующее диалоговое окно:



У вас рисунок может быть другой.

Кнопка
Теория

Думаю никому не нужно объяснять, что такое кнопка и зачем она нужна. В основе нее лежит класс Button. На этом же классе основаны флажки и переключатели, являющиеся разновидностями кнопок. Вот возможные стили кнопки:

Флаг	Описание
BS_AUTOCHECKBOX	Отличается от BS_CHECKBOX тем, что переключением состояния занимается Windows
BS_AUTORADIOBUTTON	Отличается от BS_RADIOBUTTON тем, что переключением состояний занимается Windows.
BS_BITMAP	Создается кнопка с растровым изображением. Для изменения/получения изображений следует использовать сообщения BM_SETIMAGE и BM_GETIMAGE
BS_BOTTOM	Помещает текст кнопки в нижней части ограничивающего прямоугольника
BS_CENTER	Центрирует текст кнопки в ограничивающем прямоугольнике
BS_CHECKBOX	Создается кнопка переключатель (флажок). Состояние кнопки должна отслеживать и переключать программа.
BS_DEFPUSHBUTTON	Отличается от BS_PUSHBUTTON тем, что имеет более толстую границу
BS_GROUPBOX	Создается контейнер с заголовком, который может содержать другие элементы. Никаких сообщений окну-родителю не посылает.
BS_ICON	Создается кнопка с пиктограммой. Для изменения/получения изображений следует использовать сообщения BM_SETIMAGE и BM_GETIMAGE
BS_LEFT	Помещает текст кнопки в левой части ограничивающего прямоугольника
BS_LEFTTEXT	Помещает подпись к радио-кнопке или флажку слева
BS_MULTILINE	Создает кнопку с многострочным текстом
BS_NOTIFY	Дает кнопке возможность посылать родительскому окну уведомления BN_KILLFOCUS и BN_SETFOCUS в составе сообщения WM_COMMAND
BS_OWNERDRAW	Создается кнопка, внешний вид которой

	должна контролировать программа. То есть Windows самостоятельно никак не отображает эту кнопку
BS_PUSHBUTTON	Создается обычная «нажимная» кнопка
BS_PUSHLIKE	Делает флажки или радио-кнопки похожими на «нажимаемые» кнопки
BS_RADIOBUTTON	Создается радио-кнопка
BS_RIGHT	Помещает текст кнопки в правой части ограничивающего прямоугольника
BS_RIGHTBUTTON	То же что и BS_LEFTTEXT
BS_TOP	Помещает текст кнопки в верхней части ограничивающего прямоугольника
BS_VCENTER	Помещает текст кнопки в центральной (по вертикали) части ограничивающего прямоугольника.

В таблице ниже представлены уведомления кнопки:

Уведомление	**Описание**
BN_CLICKED	Щелчок мыши по кнопке
BN_DBLCLK или BN_DOUBLECLICKED	Двойной щелчок мыши по кнопке (для работы данного уведомления нужно чтобы у кнопки был определен стиль BS_NOTIFY)
BN_DISABLE	Кнопка недоступна
BN_PUSHED или BN_HILITE	Поддерживается для совместимости с ранними версиями Windows
BN_KILLFOCUS	Кнопка потеряла фокус (у кнопки должен быть установлен стиль BS_NOTIFY)
BN_PAINT	Поддерживается для совместимости со старыми версиями Windows
BN_SETFOCUS	Кнопка получила фокус (у кнопки должен быть установлен стиль BS_NOTIFY)
BN_INHILITE или BN_UNPUSHED	Поддерживается для совместимости со старыми версиями Windows

Сама кнопка обрабатывать следующие сообщения:

Сообщение	Описание
BM_CLICK	Имитирует нажатие на кнопку пользователем
BM_GETCHECK	Получить состояние флажка или радиокнопки (установлен или нет)
BM_GETIMAGE	Получить дескриптор изображения, размещенного на кнопке
BM_GETSTATE	Проверяет, нажата кнопка или нет
BM_SETCHECK	Устанавливает состояние флажка или радио-кнопки
BM_SETDONTCLICK	После обработки этого сообщения радио-кнопка сгенерирует уведомление BN_CLICKED при получении фокуса ввода
BM_SETIMAGE	Устанавливает изображение на кнопке
BM_SETSTATE	Позволяет перевести кнопку в нажатое или не нажатое состояние
BM_SETSTYLE	Устанавливает стили кнопки

Практика

Давайте для закрепления материала напишем небольшое приложение, которое будет подсчитывать количество нажатий на кнопку и отображать это количество на самой кнопке. Исходный код такого приложения приведен ниже:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32A.inc'
;Перечень идентификаторов
ID_DIALOG = 1 ;Диалогового окна
ID_BUTTON = 2 ;Кнопки

section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
       invoke DialogBoxParam, eax, ID_DIALOG, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

proc DialogProc, hwnddlg, msg, wParam, lParam
        cmp [msg], WM_CLOSE
       ;Если получено сообщение WM_CLOSE, то переходим к уничтожению окна
        je FreeDialog
        ;Если это не «нажатие на кнопку», то выходим из процедуры
        cmp [msg], WM_COMMAND
        jne ExitProc
        mov eax, BN_CLICKED
        shl eax, 10h
        add eax, ID_BUTTON
        cmp [wParam], eax
        jne ExitProc
        ;Определяем текущее количество нажатий на кнопку
        invoke GetDlgItemInt, [hwnddlg], ID_BUTTON, _Translated, FALSE
        ;Увеличиваем это количество на единицу
        inc eax
        ;Записываем новое количество в заголовок кнопки
```

```
            invoke SetDlgItemInt, [hwnddlg], ID_BUTTON, eax, FALSE
            jmp ExitProc
    FreeDialog:
            invoke EndDialog, [hwnddlg], 0
    ExitProc:
            xor eax, eax
            ret
    endp

    section '.idata' data  readable writeable
    _Translated: db TRUE

    data import
    library kernel, 'KERNEL32.DLL',\
            user  , 'USER32.DLL'

    import kernel,\
           ExitProcess     , 'ExitProcess',\
           GetModuleHandle , 'GetModuleHandleA'

     import user,\
           DialogBoxParam  , 'DialogBoxParamA',\
           EndDialog       , 'EndDialog'      ,\
           GetDlgItemInt   , 'GetDlgItemInt'  ,\
           SetDlgItemInt   , 'SetDlgItemInt'
    end data

    section '.rsrc' data resource readable
    directory RT_DIALOG, dlg
    resource dlg,\
             ID_DIALOG, LANG_NEUTRAL, MainWindow
    dialog       MainWindow,       '',       100,       100,       100,       50,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+DS_CENTER
            dialogitem 'BUTTON', '0', ID_BUTTON, 25, 10, 50, 30, WS_VISIBLE
    enddialog
```

Поле ввода
Теория

Элемент управления поле ввода предназначен для предоставления пользователю возможности вводить какую-либо информацию текстового характера[1]. В его основе лежит предопределенный оконный класс «Edit».

Основные стили данного элемента:

Стиль	Описание
ES_AUTOHSCROLL	Создается поле, текст в котором будет автоматически прокручиваться по горизонтали, когда точка ввода текста приблизится к краю окна. Если этот флаг не задавать, то длина вводимого текста будет ограничена шириной поля.
ES_AUTOVSCROLL	Аналогично предыдущему, только для вертикальной прокрутки
ES_CENTER	Выравнивает текст в окне по центру (по горизонтали)
ES_LEFT	Выравнивает текст по левому краю
ES_LOWERCASE	Создается поле, текст в которое будет всегда вводиться в нижнем регистре
ES_MULTILINE	Создается многострочное текстовое поле
ES_NOHIDESEL	Создается поле, в котором выделенный текст остается выделенным даже когда поле ввода не имеет фокуса
ES_NUMBER	Создается поле, в которое можно вводить только цифры
ES_OEMCONVERT	Вводимый текст будет преобразовываться в oem-формат[2]
ES_PASSWORD	Создается поле для ввода пароля (в нем все вводимые символы отображаются одним символом (по умолчанию звездочка)).
ES_READONLY	Создается поле, предназначенное только для чтения
ES_RIGHT	Выравнивает текст по правому краю
ES_UPPERCASE	Создается поле, текст в котором всегда вводиться в верхнем регистре
ES_WANTRETURN	Заставляет Windows переводить строку, когда пользователь нажимает Enter (актуально только для многострочных полей)

Поле ввода может отправлять следующие уведомления:

[1] Поэтому поле ввода иногда называют текстовым полем ввода

[2] OEM-формат – кодировка символов, использующаяся в MS DOS, в Windows на смену ей пришла кодировка ANSI

Уведомление	Описание
EN_CHANGE	При изменении содержимого поля ввода
EN_ERRSPACE	Превышен размер буфера выделенного операционной системой под этот элемент (по умолчанию размер буфера составляет 32 KB для однострочных и 64 KB для многострочных полей)
EN_HSCROLL	При прокручивании текста по горизонтали
EN_KILLFOCUS	При потере фокуса ввода
EN_MAXTEXT	Когда вводимый пользователем текст не умещается в выделенных размерах окна при отключенной прокрутке (не установлены стили ES_AUTOHSCROLL ES_AUTOVSCROLL).
EN_SETFOCUS	При получении фокуса ввода
EN_UPDATE	Отправляется, когда поле ввода перерисовывает само себя. Оно отправляется после того как элемент сформировал текст, но до того как этот текст отобразился на экране
EN_VSCROLL	При прокручивании текста по вертикали

Оконная процедура поля ввода может обрабатывать сообщения:

Сообщение	Описание
EM_GETFIRSTVISIBLELINE	Получить индекс (считается с нуля) самой верхней отображаемой строки в многострочном поле ввода
EM_GETLIMITTEXT	Получить максимально возможную длину текста в поле ввода
EM_GETLINE	Скопировать строку из многострочного поля ввода. При этом wParam должен содержать индекс копируемой строки, а lParam – адрес буфера в который следует скопировать эту строку.
EM_GETLINECOUNT	Получить количество строк в многострочном поле ввода
EM_GETMODIFY	Получить флаг модифицированности поля ввода. Данный флаг взводится когда содержимое поля было изменено
EM_GETPASSWORDCHAR	Получить символ, который отображается в поле вода при вводе пароля
EM_LIMITTEXT	Установить максимально возможную длину текста вводимого в поле ввода
EM_LINELENGTH	Получить длину в символах строки в поле ввода. При этом wParam должен содержать

	в себе индекс строки длину, которой требуется определить
EM_REPLACESEL	Заменить выделенный текст другим текстом. При этом wParam указывает, может ли эта операция быть отменена (TRUE – операция не может быть отменена; FALSE – может быть отменена); lParam – адрес строки, на которую следует заменить
EM_SETLIMITTEXT	Аналогично EM_LIMITTEXT
EM_SETMODIFY	Установить значение флага модифицированности поля ввода. При этом wParam содержит новое значение этого флага
EM_SETPASSWORDCHAR	Установить символ, который будет отображаться при вводе пароля
EM_SETREADONLY	Установить для поля ввода стиль ES_READONLY

Практика

В качестве примера работы с данным элементом управления рассмотрим программу, по нажатию на кнопку, выводящую через MessageBox (A/W) текст, введенный пользователем в поле ввода. Вот исходный код это программы:

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
  start:invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

        proc DialogProc hwnddlg, msg, wparam, lparam
             xor eax, eax
             cmp [msg], WM_CLOSE
             je FreeDialog
             cmp [msg], WM_COMMAND
             jne ExitProc
             mov eax, BN_CLICKED
             shl eax, 10h
             add eax, 3
             cmp [wparam], eax
             jne ExitProc
             ;Считать текст введенный пользователем
             invoke GetDlgItemText, [hwnddlg], 2, text, 20h
             ;Выводим его
             invoke MessageBox, [hwnddlg], text, text, MB_OK
             ret
  FreeDialog:
             invoke EndDialog, [hwnddlg], 0
  ExitProc:  ret
        endp
```

```
    section '.bss' readable writeable
            text rb 20h

    section '.idata' import data readable writeable
            library kernel, 'KERNEL32.DLL' ,\
                    user  , 'USER32.DLL'

            import kernel,\
                   GetModuleHandle, 'GetModuleHandleA',\
                   ExitProcess    , 'ExitProcess'

            import user,\
                   DialogBoxParam, 'DialogBoxParamA',\
                   EndDialog     , 'EndDialog'      ,\
                   MessageBox    , 'MessageBoxA'    ,\
                   GetDlgItemText, 'GetDlgItemTextA'
    ;ресурсы
    section '.rsrc' resource data readable
            directory RT_DIALOG, dialogs
            resource dialogs,\
                     1, LANG_NEUTRAL, WorkWithEdit
            ;Шаблон диалогового окна
            dialog WorkWithEdit, 'Work with Edit class', 0, 0, 150, 50,
WS_CAPTION+WS_SYSMENU+DS_CENTER
                   ;Элемент управления поле ввода
                   dialogitem 'Edit',    '',            2, 10, 10, 130, 12,
WS_VISIBLE+WS_BORDER
                   ;Элемент управления кнопка
                   dialogitem 'Button', 'Show text', 3, 45, 25, 50,   15,
WS_VISIBLE
            enddialog
```

Флажок
Теория

Флажок является кнопкой с установленным стилем BS_AUTOCHECKBOX или BS_CHECKBOX. Во втором случае программе придется самой переключать его состояния.

Флажок может находиться в одном из двух[1] состояний: установлен или не установлен.

Проверить состояние флажка можно помощью функции IsDlgButtonChecked, выглядит она так:

```
UINT IsDlgButtonChecked(
  __in  HWND hDlg, //Дескриптор диалогового окна, которому принадлежит флажок
  __in  int nIDButton //Идентификатор элемента управления флажка, состояние
                      //которого нужно узнать
);
```

Данная функция возвращает одно из трех значений:
BST_CHECKED – флажок установлен;
BST_INDETERMINATE - он находится в промежуточном состоянии;
BST_UNCHECKED – не установлен.

Также узнать состояние флажка можно путем отправки ему сообщения BM_GETCHECK.

Изменить состояние флажка программно можно путем отправки ему сообщения BM_SETCHECK. При этом wParam содержит устанавливаемое состояние.

Каждый раз, когда пользователь изменяет состояние флажка, в диалоговую процедуру отправляется сообщение WM_COMMAND с уведомлением BN_CLICKED.

Практика

Для закрепления всего вышесказанного напишем небольшую программу, которая по нажатию на кнопку будет проверять состояние флажка и извещать об этом пользователя. Исходный код такой программы представлен ниже:

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'

ID_CHECKBOX = 2 ; Идентификатор флажка
ID_BUTTON = 3   ; Идентификатор кнопки

section '.code' code readable executable
```

[1] При установленном стиле BS_3STATE к этим двум возможным состояниям добавляется третье (промежуточное): галочка стоит, но сам флажок выделен серым цветом

```
  start:invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

        proc DialogProc hwnddlg, msg, wparam, lparam
             xor eax, eax
             cmp [msg], WM_CLOSE
             je FreeDialog
             cmp [msg], WM_COMMAND
             jne ExitProc
             mov eax, BN_CLICKED
             shl eax, 10h
             add eax, ID_BUTTON
             cmp [wparam], eax
             jne ExitProc
             ;Выясняем состояние флажка
             invoke IsDlgButtonChecked, [hwnddlg], ID_CHECKBOX
             cmp eax, BST_CHECKED
             jz Checked
             mov eax, _TextUnChecked
             jmp ShowMessage
   Checked:  mov eax, _TextChecked
   ShowMessage:
             ;Сообщаем пользователю
             invoke MessageBox, [hwnddlg], eax, NULL, MB_OK
             ret
  FreeDialog:
             invoke EndDialog, [hwnddlg], 0
  ExitProc:  ret
        endp
;Сообщения пользователю
section '.data' readable
 _TextChecked db 'Checked', 0
 _TextUnChecked db 'UnChecked', 0

section '.idata' import data readable writeable
        library kernel, 'KERNEL32.DLL' ,\
                user  , 'USER32.DLL'

        import kernel,\
               GetModuleHandle, 'GetModuleHandleA',\
               ExitProcess    , 'ExitProcess'

        import user,\
               DialogBoxParam, 'DialogBoxParamA',\
               EndDialog     , 'EndDialog'      ,\
               MessageBox    , 'MessageBoxA'    ,\
               IsDlgButtonChecked, 'IsDlgButtonChecked'
               ;GetDlgItemText, 'GetDlgItemTextA'

section '.rsrc' resource data readable
        directory RT_DIALOG, dialogs
        resource dialogs,\
                 1, LANG_NEUTRAL, WorkWithCheckBox
        dialog   WorkWithCheckBox,   'Checkbox',   0,   0,   50,   50,
WS_CAPTION+WS_SYSMENU+DS_CENTER
                 dialogitem 'Button',    'Checkbox', ID_CHECKBOX, 10, 10,
50, 12, WS_VISIBLE+BS_AUTOCHECKBOX
                 dialogitem 'Button', 'Check' , ID_BUTTON   , 10, 25, 50,
15, WS_VISIBLE
        enddialog
```

Рамка группы
Теория

Рамка группы представляет собой обычную рамку с заголовком, внутри которой могут размещаться другие элементы управления. Сама по себе рамка группы является кнопкой с установленным стилем BS_GROUPBOX.

Практика

Приводить пример работы с данным элементом управления я не буду. Нет, не потому что мне жалко. Просто пример работы с ним вы увидите в следующем разделе при рассмотрении группы переключателей.

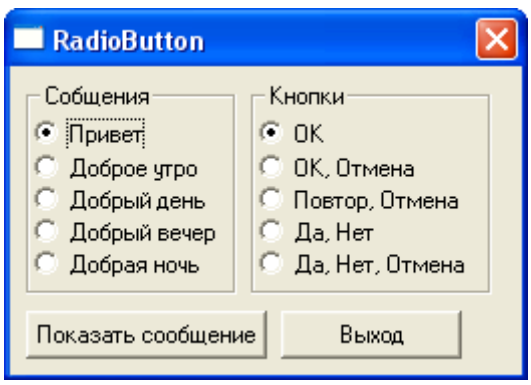
Группа переключателей
Теория

Группа переключателей – это несколько элементов управления (переключателей), объединенных в одну группу. Переключатель (или радио-кнопка) является кнопкой с установленным стилем BS_RADIOBUTTON или BS_AUTORADIOBUTTON. Проверка установленности того или иного переключателя осуществляется аналогично проверке установленности флажка.

Первый переключатель группы должен иметь стиль WS_GROUP. Без этого стиля система объединит все переключатели в одну группу. Несмотря на то, что вы хотели разместить в разных группах.

Практика

В качестве практического упражнения напишем небольшую программу, демонстрирующую работу с группой переключателей. Главное окно приложения представлено ниже:



В данном приложении в зависимости от параметров, выбранных пользователем, выводится сообщение с различным текстом и набором кнопок. Ниже приводится исходный код данного приложения:

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'
;Идентификаторы элементов управления
ID_BUTTONSHOWMESSAGE = 3
ID_BUTTONCANCEL = 10h
ID_MESSAGETEXT = 4

ID_HELLO  = 5
ID_GOODMORNING = 6
ID_GOODDAY = 7
ID_GOODEVENING = 8
ID_GOODNIGHT = 9

ID_BUTTONS = 0Ah
ID_OK = 0Bh
ID_OKCANCEL = 0Ch
ID_RETRYCANCEL = 0Dh
ID_YESNO = 0Eh
ID_YESNOCANCEL = 0Fh

section '.code' code readable executable
  start:invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

        proc DialogProc hwnddlg, msg, wparam, lparam
             xor eax, eax
             cmp [msg], WM_CLOSE
             je FreeDialog
             cmp [msg], WM_INITDIALOG
             je InitDialog
             cmp [msg], WM_COMMAND
             jne ExitProc
             mov eax, BN_CLICKED
             shl eax, 10h
             add eax, ID_BUTTONCANCEL
             cmp [wparam], eax
             je FreeDialog
             sub eax, ID_BUTTONCANCEL
             add eax, ID_BUTTONSHOWMESSAGE
             cmp [wparam], eax
             jne ExitProc
             ;Путем последовательной проверки состояний переключателей
             ;определяем выбранный текст

             invoke IsDlgButtonChecked, [hwnddlg], ID_HELLO
             cmp eax, BST_CHECKED
             jne CheckGoodMorning
             mov eax, _TextHello
             mov dword [_TextMessage], eax
             jmp FindButtons
   CheckGoodMorning:
             invoke IsDlgButtonChecked, [hwnddlg], ID_GOODMORNING
             cmp eax, BST_CHECKED
             jne CheckGoodDay
             mov eax, _TextGoodMorning
             mov dword [_TextMessage], eax
             jmp FindButtons
```

```
 CheckGoodDay:
           invoke IsDlgButtonChecked, [hwnddlg], ID_GOODDAY
           cmp eax, BST_CHECKED
           jne CheckGoodEvening
           mov eax, _TextGoodDay
           mov dword [_TextMessage], eax
           jmp FindButtons
 CheckGoodEvening:
           invoke IsDlgButtonChecked, [hwnddlg], ID_GOODEVENING
           cmp eax, BST_CHECKED
           jne CheckGoodNight
           mov eax, _TextGoodEvening
           mov dword [_TextMessage], eax
           jmp FindButtons
 CheckGoodNight:
           mov eax, _TextGoodNight
           mov dword [_TextMessage], eax
           jmp FindButtons
 FindButtons:
           ;Аналогично определяем выбранный набор кнопок
           invoke IsDlgButtonChecked, [hwnddlg], ID_OK
           cmp eax, BST_CHECKED
           jne CheckOkCancel
           mov dword [_Buttons], MB_OK
           jmp ShowMessage
 CheckOkCancel:
           invoke IsDlgButtonChecked, [hwnddlg], ID_OKCANCEL
           cmp eax, BST_CHECKED
           jne CheckRetryCancel
           mov dword [_Buttons], MB_OKCANCEL
           jmp ShowMessage
 CheckRetryCancel:
           invoke IsDlgButtonChecked, [hwnddlg], ID_RETRYCANCEL
           cmp eax, BST_CHECKED
           jne CheckYesNo
           mov dword [_Buttons], MB_RETRYCANCEL
           jmp ShowMessage
 CheckYesNo:
           invoke IsDlgButtonChecked, [hwnddlg], ID_YESNO
           cmp eax, BST_CHECKED
           jne CheckYesNoCancel
           mov dword [_Buttons], MB_YESNO
           jmp ShowMessage
  CheckYesNoCancel:
           mov dword [_Buttons], MB_YESNOCANCEL
 ShowMessage:
           ;Выводим сообщение в соответствии с настройками
           invoke MessageBox, [hwnddlg], dword [_TextMessage], dword
                                  [_TextMessage], dword [_Buttons]
           ret
FreeDialog:
           invoke EndDialog, [hwnddlg], 0
ExitProc:  ret
InitDialog:
          ;Устанавливаем заголовки элементов управления
          invoke SetDlgItemText, [hwnddlg], ID_BUTTONSHOWMESSAGE,
                                            _TextButtonShowMessage
         invoke SetDlgItemText, [hwnddlg], ID_BUTTONCANCEL,
                                            _TextButtonCancel

         invoke SetDlgItemText, [hwnddlg], ID_HELLO, _TextHello
         invoke SetDlgItemText, [hwnddlg], ID_GOODMORNING,
                                             _TextGoodMorning
         invoke SetDlgItemText, [hwnddlg], ID_GOODDAY, _TextGoodDay
```

```
            invoke SetDlgItemText, [hwnddlg], ID_GOODEVENING,
                                              _TextGoodEvening
            invoke SetDlgItemText, [hwnddlg], ID_GOODNIGHT,
                                            _TextGoodNight

            invoke SetDlgItemText, [hwnddlg], ID_BUTTONS, _TextButtons
            invoke SetDlgItemText, [hwnddlg], ID_MESSAGETEXT,
                                                 _TextMessages

            invoke SetDlgItemText, [hwnddlg], ID_OK, _TextOk
            invoke SetDlgItemText, [hwnddlg], ID_OKCANCEL, _TextOkCancel
            invoke SetDlgItemText, [hwnddlg], ID_RETRYCANCEL,
                                                      _TextRetryCancel
            invoke SetDlgItemText, [hwnddlg], ID_YESNO, _TextYesNo
            invoke SetDlgItemText, [hwnddlg], ID_YESNOCANCEL,
                                                      _TextYesNoCancel
            ;Устанавливаем переключатели, «выбранные» по умолчанию
            invoke SendDlgItemMessage, [hwnddlg], ID_HELLO, BM_SETCHECK,
                                                    BST_CHECKED, 0
            invoke SendDlgItemMessage, [hwnddlg], ID_OK, BM_SETCHECK,
                                                       BST_CHECKED, 0
            xor eax, eax
            ret
        endp

section '.data' readable
 _TextButtonShowMessage db 'Показать сообщение', 0
 _TextButtonCancel db 'Выход', 0

 _TextHello db 'Привет', 0
 _TextGoodMorning db 'Доброе утро', 0
 _TextGoodDay db 'Добрый день', 0
 _TextGoodEvening db 'Добрый вечер', 0
 _TextGoodNight db 'Добрая ночь', 0

_TextButtons db 'Кнопки', 0
_TextMessages db 'Сообщения', 0

_TextOk db 'OK', 0
_TextOkCancel db 'OK, Отмена', 0h
_TextRetryCancel db 'Повтор, Отмена', 0h
_TextYesNo  db 'Да, Нет', 0h
_TextYesNoCancel db 'Да, Нет, Отмена', 0h

section '.bss' data readable writeable
;Сюда будет записан выбранный набор кнопок
_Buttons dd 00h
;Сюда будет записан адрес выбранного сообщения
_TextMessage dd 00h

section '.idata' import data readable writeable
        library kernel, 'KERNEL32.DLL' ,\
                user  , 'USER32.DLL'

        import kernel,\
               GetModuleHandle, 'GetModuleHandleA',\
               ExitProcess    , 'ExitProcess'

        import user,\
               DialogBoxParam, 'DialogBoxParamA',\
               EndDialog     , 'EndDialog'      ,\
               MessageBox    , 'MessageBoxA'    ,\
               IsDlgButtonChecked, 'IsDlgButtonChecked',\
               SetDlgItemText, 'SetDlgItemTextA',\
```

```
                SendDlgItemMessage, 'SendDlgItemMessageA'

section '.rsrc' resource data readable
        directory RT_DIALOG, dialogs
        resource dialogs,\
                 1, LANG_NEUTRAL, WorkWithEdit
        dialog WorkWithEdit, 'Checkbox', 0, 0, 170, 95,
                                         WS_CAPTION+WS_SYSMENU+DS_CENTER
               ;Рамка группы переключателей выбора текста сообщения
               dialogitem 'Button', 'Message text', ID_MESSAGETEXT, 5, 5,
                                         70, 65, WS_VISIBLE+BS_GROUPBOX
               ;Группа переключателей выбора текста сообщения
               dialogitem 'Button', '', ID_HELLO, 7, 15, 60, 12,
                                  WS_VISIBLE+BS_AUTORADIOBUTTON+WS_GROUP
               dialogitem 'Button', '', ID_GOODMORNING, 7, 25, 60, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               dialogitem 'Button', '', ID_GOODDAY, 7, 35, 60, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               dialogitem 'Button', '', ID_GOODEVENING, 7, 45, 65, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               dialogitem 'Button', '', ID_GOODNIGHT, 7, 55, 60, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               ;Рамка группы переключателей выбора набора кнопок
               dialogitem 'Button', '', ID_BUTTONS, 80, 5, 80, 65,
                                          WS_VISIBLE+BS_GROUPBOX
               ;Группа переключателей выбора набора кнопок
               dialogitem 'Button', '', ID_OK, 82, 15, 30, 12,
                                  WS_VISIBLE+BS_AUTORADIOBUTTON+WS_GROUP
               dialogitem 'Button', '', ID_OKCANCEL, 82, 25, 60, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               dialogitem 'Button', '', ID_RETRYCANCEL, 82, 35, 70, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               dialogitem 'Button', '', ID_YESNO, 82, 45, 60, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON
               dialogitem 'Button', '', ID_YESNOCANCEL, 82, 55, 70, 12,
                                          WS_VISIBLE+BS_AUTORADIOBUTTON

               dialogitem 'Button', '', ID_BUTTONSHOWMESSAGE , 5, 75, 80,
                                                        15, WS_VISIBLE
               dialogitem 'Button', '',  ID_BUTTONCANCEL  , 90, 75, 60,
                                                        15, WS_VISIBLE
        enddialog
```

Список
Теория

Список строится на предопределенном оконном классе «ListBox». Он представляет собой список определенных значений, из которого пользователь может выбрать одно или несколько значений. В таблице ниже представлены основные стили данного элемента управления:

Стиль	Описание
LBS_EXTENDEDSEL	Допускается использование клавиши Shift при множественном выборе элементов, если он разрешен
LBS_MULTICOLUMN	Создается список с несколькими столбцами. При этом список становится похож на таблицу
LBS_MULTIPLESEL	Создается список с возможностью одновременного выбора нескольких элементов
LBS_NODATA	Определяет отсутствие данных в окне со списком. Этот стиль задается когда число элементов в нем может превысить тысячу.
LBS_NOINTEGRALHEIGHT	Создается список без автоматической корректировки высоты, которая применяется по умолчанию. Корректировка происходит для того, чтобы в области списка помещались строки целиком по высоте, например, если в список вмещается полностью пять элементов, а шестой не полностью, то вертикальный размер списка будет уменьшен, чтобы вместить только пять
LBS_NOTIFY	Создается список, отсылающий уведомляющие сообщения окну-владельцу
LBS_NOSEL	Создается список, элементы которого не выделяются цветом
LBS_NOREDRAW	Создается список, который не перерисовывается автоматически при каких-либо изменениях
LBS_OWNERDRAWFIXED	Создается список, отображаемый программой, при этом все элементы имеют переменную высоту
LBS_OWNERDRAWVARIABLE	Говорит о том, что программа сама будет

	обрабатывать сообщения WM_DRAWITEM и WM_MEASUREITEM
LBS_SORT	Создается отсортированный список. При добавлении нового элемента, он будет автоматически помещен в нужную позицию
LBS_WANTKEYBOARDINPUT	Определяет, что окно-владелец получает сообщение WM_VKEYTOITEM, когда пользователь нажимает клавишу на списке, когда тот имеет фокус ввода

Список может отправлять следующие уведомления:

Уведомление	Описание
LBN_DBLCLK	Пользователь дважды щелкнул на элементе списка
LBN_ERRSPACE	Посылается при переполнении буфера текста
LBN_KILLFOCUS	Окно списка потеряло фокус ввода
LBN_SELCANCEL	Пользователь сбросил выделение в списке
LBN_SELCHANGE	Выбранные элементы изменились
LBN_SETFOCUS	Список получил фокус ввода

Также окно списка может принимать следующие сообщения:

Сообщение	Описание
LB_ADDFILE	Добавить в список строковое имя файла, при этом lParam содержит адрес строки с именем файла
LB_ADDSTRING	Добавить строку в список. При этом параметр lParam содержит адрес добавляемой строки. При этом возвращается индекс добавленной строки
LB_DELETESTRING	Удалить сроку из списка. wParam содержит индекс удаляемой строки
LB_DIR	Добавить в список имена файлов из текущего каталога. wParam содержит набор атрибутов файлов[1], которые следует добавить. lParam содержит адрес строки с путем к файлу или каталогу, а также маской файлов.
LB_FINDSTRING	Найти в списке строку, начинающуюся с заданных символов. wParam содержит индекс, с которого следует начать поиск (или -1, если

[1] К таким атрибутам относятся: DDL_ARCHIVE – включать архивы, DDL_DIRECTORY – включать поддиректории (они обрамляются в списке квадратными скобками), DDL_HIDDEN – включать скрытые файлы и некоторые другие атрибуты

	начинать надо с начала списка). lParam – адрес строки с префиксом строки, которую нужно найти. Возвращается либо индекс найденной строки, либо LB_ERR, в случае если поиск не дал результатов
LB_FINDSTRINGEXACT	Найти первую строку в списке, которая соответствует образцу. wParam – Индекс строки, предыдущей по отношению к той, с которой следует начать поиск (или -1 если поиск следует начать с начала списка). lParam – адрес строки по которой ищется. Поиск регистронезависимый. Возвращается либо индекс найденной строки, либо LB_ERR, если ничего не найдено
LB_GETANCHORINDEX	Получить индекс первого элемента в группе выделенных элементов
LB_GETCARETINDEX	Получить индекс строки, которой принадлежит фокус ввода
LB_GETCOUNT	Получить число элементов в списке
LB_GETCURSEL	Получить индекс выбранного элемента
LB_GETHORIZONTALEXTENT	Получить ширину, на которую список может быть прокручен по горизонтали
LB_GETITEMDATA	Получить 32 битное целое, связанное с определенным элементом списка. wParam – индекс строки.
LB_GETITEMHEIGHT	Получить высоту строки. wParam - индекс строки, высоту которой нужно определить
LB_GETITEMRECT	Получить прямоугольник, ограничивающий элемент списка. wParam – индекс элемента списка
LB_GETLOCALE	Получить текущие языковые настройки (локаль) элемента управления
LB_GETSEL	Проверить, выделен ли элемент. wParam – индекс проверяемого элемента. Если элемент выделен, то возвращается ненулевое значение, иначе ноль
LB_GETSELCOUNT	Получить общее число выделенных элементов в списке с возможностью множественного выбора
LB_GETSELITEMS	Заполнить буфер массивом выделенных элементов. wParam – максимальное количество выбранных элементов, которые будут размещены в буфере. lParam – адрес буфера, в который следует записать выбранные элементы
LB_GETTEXT	Получить текст заданного элемента. wParam –

	индекс элемента строковое представление, которого следует получить. lParam – адрес буфера, в который следует записать требуемую строку
LB_GETTEXTLEN	Получить длину строки заданного элемента. wParam – индекс элемента списка, длину строкового представления которого следует получить
LB_GETTOPINDEX	Получить индекс первого видимого элемента списка
LB_INITSTORAGE	Выделить память для хранения элементов списка. wParam – количество элементов, для которых следует выделить память. lParam – размер выделяемой памяти в байтах.
LB_INSERTSTRING	Вставить строку в заданное место списка. wParam – индекс, по которому следует разместить, вставляемую строку. lParam – адрес вставляемой строки. Возвращает индекс вставленной строки.
LB_ITEMFROMPOINT	Получить индекс элемента списка ближайшего к данной точке. Младшее слово lParam определяет координату точки по горизонтали, старшее слово – по вертикали.
LB_RESETCONTENT	Стереть содержимое списка
LB_SELECTSTRING	Найти элемент списка, строка которого начинается с заданных символов. wParam – индекс элемента предыдущего по отношению к тому, с которого следует начать поиск (-1 если поиск нужно начать с начала списка). lParam – адрес строки, с началом строки, которую нужно найти. Возвращает индекс найденной сроки, или LB_ERR если ее не удалось найти. Не используется со списками у которых установлены стили LBS_MULTIPLESEL или LBS_EXTENDEDSEL
LB_SELITEMRANGE	Выбрать один или более элементов списка из списка допускающего множественный выбор. Если wParam не равен нулю, тогда выбранные элементы выделяются, если же он равен нулю, то они никак не выделяются. Младшее слово lParam содержит индекс элемента списка, с которого начинается выделение, старшее слово – индекс последнего выделенного элемента.
LB_SELITEMRANGEEX	Аналогично предыдущему отличаются только

	параметры: wParam – индекс первого выделенного элемента; lParam – индекс последнего выделенного элемента
LB_SETANCHORINDEX	Установить позицию начала выделения. wParam – индекс начала выделения
LB_SETCARETINDEX	Установить фокус ввода на элемент списка. wParam – индекс элемента списка, на котором нужно установить фокус. Старшее слово lParam указывает должен ли список быть прокручен до полного отображения нужного элемента. (если оно равно нулю, то должно, если же оно отлично от нуля, то достаточно частичного отображения элемента)
LB_SETCOLUMWIDTH	Установить ширину колонки в списке с несколькими колонками.
LB_SETCOUNT	Установить количество элементов в списке со стилем LBS_NODATA. wParam – устанавливаемое количество элементов.
LB_SETCURSEL	Выбрать новую строку. При необходимости список прокручивается. wParam – индекс выделяемой строки
LB_SETHORIZONTALEXTENT	Установить в пикселях ширину списка, на которую он может быть прокручен по горизонтали.
LB_SETITEMDATA	Установить значение 32 битного целого, связанного с конкретным элементом. wParam – индекс элемента, для которого нужно установить значение. Если он равен -1, то значение устанавливается всем элементам списка. lParam – устанавливаемое значение.
LB_SETITEMHEIGHT	Установить высоту в пикселях элементов списка. wParam – индекс элемента списка (если у списка установлен стиль LBS_OWNERDRAWVARIABLE, в противном случае ноль). lParam – устанавливаемая высота
LB_SETLOCALE	Установить параметры языка (локаль). wParam – идентификатор устанавливаемой локали.
LB_SETSEL	Установить или отменить выделение в списке с возможностью множественного выбора. Если wParam равен TRUE, то элемент списка выделяется, иначе с него снимается выделение. lParam – индекс элемента списка, для которого устанавливается (снимается) выделение.
LB_SETTABSTOPS	Установить позиции табуляции списка

LB_SETTOPINDEX	Установить первый видимый элемент списка. wParam – индекс элемента, который устанавливается в качестве первого видимого элемента списка.

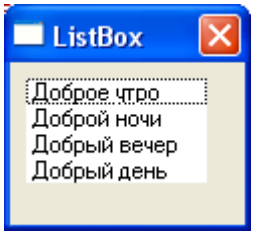
Практика

Ниже приводится исходный код небольшого приложения, демонстрирующего принцип работы со списком. В этом приложении формируется список из возможных текстов сообщений, по двойному щелчку мыши по любому из элементов списка выводится сообщение с таким текстом.

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'

ID_LISTBOX = 2

section '.code' code readable executable
   string: db 'Push a Button',0
start:
        invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
     cmp [msg], WM_CLOSE
     je FreeDialog

     cmp [msg], WM_INITDIALOG
     je InitDialog

     cmp [msg], WM_COMMAND
     jne exitproc

     mov eax, LBN_DBLCLK
     shl eax, 16
     add eax, ID_LISTBOX
     cmp [wparam], eax
     jne exitproc
     ; Получаем индекс выделенной строки списка
     invoke SendDlgItemMessage, [hwnddlg], ID_LISTBOX, LB_GETCURSEL,
NULL, NULL
     ; Получаем связанное с ней значение
     invoke SendDlgItemMessage, [hwnddlg], ID_LISTBOX, LB_GETITEMDATA,
eax, NULL

     ; Выводим сообщение пользователю
     invoke MessageBox, 0, eax, _Caption, MB_OK
     ; Выход из диалоговой  процедуры
exitproc:
     xor eax, eax
     ret
     ;Участок кода ответственный за освобождение окна
FreeDialog:
     invoke EndDialog, [hwnddlg], 0
     xor eax,eax
```

```
          ret
     InitDialog:
          ;Заполняем список строками и связываем с ними их адреса
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_ADDSTRING,
NULL, _GoodMorning
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_SETITEMDATA,
eax, _GoodMorning

          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_ADDSTRING,
NULL, _GoodDay
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_SETITEMDATA,
eax, _GoodDay

          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_ADDSTRING,
NULL, _GoodEvening
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_SETITEMDATA,
eax, _GoodEvening

          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_ADDSTRING,
NULL, _GoodNight
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_LISTBOX,  LB_SETITEMDATA,
eax, _GoodNight

          xor eax, eax
          ret
     endp

     section '.data' data readable
     _GoodMorning db 'Доброе утро', 0
     _GoodDay db 'Добрый день', 0
     _GoodEvening db 'Добрый вечер', 0
     _GoodNight db 'Доброй ночи', 0
     _Caption db ' ', 0

     section '.idata' import data readable writeable

      library kernel, 'KERNEL32.DLL',\
              user, 'USER32.DLL'

      import kernel,\
             GetModuleHandle, 'GetModuleHandleA',\
             ExitProcess, 'ExitProcess'

      import user,\
             DialogBoxParam, 'DialogBoxParamA',\
             EndDialog, 'EndDialog' ,\
             MessageBox, 'MessageBoxA',\
             SendDlgItemMessage, 'SendDlgItemMessageA'


     section '.rsrc' resource data readable
      directory RT_DIALOG, dialogs

      resource dialogs,\
             1,LANG_ENGLISH, form1

      dialog form1, 'ListBox', 70, 70, 65, 50,\
      WS_CAPTION+WS_SYSMENU+DS_CENTER
            dialogitem     'ListBox',   '',   ID_LISTBOX,   5,   5,   60,   45,
WS_VISIBLE+LBS_SORT+LBS_NOTIFY
      enddialog
```

На рисунке ниже представлен внешний вид приложения:



Комбинированный список
Теория

Комбинированный список очень похож на обычный список. У них одно и тоже назначение: предоставить пользователю возможность выбрать значение из предлагаемого списка. Разница в том, что если все варианты, предлагаемые обычным списком видны сразу, то комбинированный список отображает только текущий выбранный элемент, остальные элементы находятся в свернутом виде. Комбинированный список логически состоит из двух частей: поле ввода, в котором отображается текущий выбранный элемент и раскрывающийся список, который разворачивается при нажатии на кнопку рядом с полем ввода.

Комбинированный список строится на основе предопределенного оконного класса «ComboBox». В таблице ниже представлены основные стили данного элемента управления:

Стиль	Описание
CBS_AUTOHSCROLL	Автоматически прокручивать текст в поле ввода, когда точка ввода переместится вплотную к правой границе
CBS_HASSTRINGS	Определяет, что комбинированный список, отображаемый пользователем, содержит строки текста
CBS_DROPDOWN	Создается комбинированный список, состоящий из поля ввода и выпадающего блока
CBS_DROPDOWNLIST	Создается комбинированный список, состоящий из выпадающего блока, пользователь может делать выбор только из предопределенных элементов
CBS_LOWERCASE	Автоматически конвертирует текст во всем списке в нижний регистр
CBS_NOINTEGRALHEIGHT	Создается список без автоматической корректировки высоты, которая применяется по умолчанию. Корректировка происходит для того,

	чтобы в области списка помещались строки целиком по высоте, например, если в список вмещается полностью пять элементов, а шестой не полностью, то вертикальный размер списка будет уменьшен, чтобы вместить только пять элементов
CBS_OEMCONVERT	Автоматически конвертирует текст во всем списке в формат OEM
CBS_OWNERDRAWFIXED	Создается список с элементами одинаковой высоты, элементы которого отображать должна программа
CBS_OWNERDRAWVARIABLE	Создается список с элементами переменной высоты, элементы которого должна отображать сама программа
CBS_SIMPLE	Создается комбинированный список, состоящий из поля ввода и обычного (не выпадающего) списка
CBS_SORT	Создается автоматически сортирующийся список
CBS_UPPERCASE	Автоматически конвертирует текст во всем списке в верхний регистр

В таблице ниже представлены основные уведомления, отправляемые комбинированным списком родительскому окну.

Уведомление	**Описание**
CBN_CLOSEUP	При закрытии выпадающего списка
CBN_DBLCLK	При двойном щелчке по элементу списка
CBN_DROPDOWN	При открытии выпадающего списка
CBN_EDITCHANGE	Пользователь изменил текст в поле ввода или выбрал элемент списка. Изменения отобразились на экране
CBN_EDITUPDATE	Пользователь изменил текст в поле ввода или выбрал элемент списка, изменения еще не отобразились на экране
CBN_ERRSPACE	Произошло переполнение буфера
CBN_KILLFOCUS	При потере фокуса ввода
CBN_SELCHANGE	Изменен выбранный элемент
CBN_SELENDCANCEL	Пользователь закрыл выпадающий блок, ничего не выбрав
CBN_SELENDOK	Пользователь закрыл выпадающий блок, что-то выбрав
CBN_SETFOCUS	Получен фокус ввода

Также окно комбинированного списка способно принимать и обрабатывать сообщения, основные из них представлены в таблице ниже:

Сообщение	Описание
CB_ADDSTRING	Добавить строку в список. При этом параметр lParam содержит адрес добавляемой строки. При этом возвращается индекс добавленной строки
CB_DELETESTRING	Удалить строку из списка. wParam содержит индекс удаляемой строки
CB_DIR	Добавить в список имена файлов из текущего каталога. wParam содержит набор атрибутов файлов[1], которые следует добавить. lParam содержит адрес строки с путем к файлу или каталогу, а также маской файлов.
CB_FINDSTRING	Найти в списке строку, начинающуюся с заданных символов. wParam содержит индекс, с которого следует начать поиск (или -1, если начинать надо с начала списка). lParam – адрес строки с префиксом строки, которую нужно найти. Возвращается либо индекс найденной строки, либо LB_ERR, в случае если поиск не дал результатов
CB_FINDSTRINGEXACT	Найти первую строку в списке, которая соответствует образцу. wParam – Индекс строки, предыдущей по отношению к той, с которой следует начать поиск (или -1 если поиск следует начать с начала списка). lParam – адрес строки по которой ищется. Поиск регистронезависимый. Возвращается либо индекс найденной строки, либо LB_ERR, если ничего не найдено
CB_GETCOUNT	Получить число элементов в списке
CB_GETCURSEL	Получить индекс выбранной строки
CB_GETDROPPEDCONTROLRECT	Получить экранные координаты комбинированного списка. lParam – адрес структуры RECT, в которую будут записаны запрашиваемые координаты
CB_GETDROPPEDSTATE	Проверить видимость раскрывающегося списка. Если этот список видим, то

[1] К таким атрибутам относятся: DDL_ARCHIVE – включать архивы, DDL_DIRECTORY – включать поддиректории (они обрамляются в списке квадратными скобками), DDL_HIDDEN – включать скрытые файлы и некоторые другие атрибуты

	возвращается TRUE иначе FALSE.
CB_GETDROPPEDWIDTH	Получить минимально возможную ширину раскрывающегося списка в пикселях.
CB_GETEDITSEL	Получить начальную и конечную позицию выделенного фрагмента в поле ввода комбинированного списка. wParam – адрес двойного слова, по которому будет записан индекс начала выделения. lParam - адрес двойного слова, в которое будет записан индекс последнего выделенного символа
CB_GETEXTENDEDUI	Проверить использует ли комбинированный список расширенный пользовательский интерфейс. Если использует, то возвращается значение TRUE, иначе значение FALSE
CB_GETHORIZONTALEXTENT	Получить ширину в пикселях, на которую список может быть прокручен по горизонтали
CB_GETITEMDATA	Получить 32 битное целое, связанное с определенным элементом списка. wParam – индекс строки.
CB_GETITEMHEIGHT	Получить высоту строки. wParam - индекс строки, высоту которой нужно определить
CB_GETLBTEXT	Получить строку из раскрывающегося списка. wParam – индекс возвращаемой строки. lParam – адрес буфера, в который будет скопирована строка. При этом возвращается длина скопированной строки
CB_GETLBTEXTLEN	Получить длину строки в символах. wParam – индекс строки, длину которой нужно определить
CB_GETLOCALE	Получить текущие языковые настройки (локаль) элемента управления
CB_GETTOPINDEX	Получить индекс первого видимого элемента списка
CB_INITSTORAGE	Выделить память для хранения элементов списка. wParam – количество элементов, для которых следует выделить память. lParam – размер выделяемой памяти в байтах.
CB_INSERTSTRING	Вставить строку в заданное место списка. wParam – индекс, по которому следует разместить, вставляемую строку. lParam – адрес вставляемой строки. Возвращает индекс вставленной строки.
CB_LIMITTEXT	Задать максимально допустимую длину

	текста. wParam – устанавливаемая длина в символах
CB_RESETCONTENT	Очистить содержимое списка
CB_SELECTSTRING	Найти элемент списка, строка которого начинается с заданных символов. wParam – индекс элемента предыдущего по отношению к тому, с которого следует начать поиск (-1 если поиск нужно начать с начала списка). lParam – адрес строки, с началом строки, которую нужно найти. Возвращает индекс найденной сроки, или LB_ERR если ее не удалось найти.
CB_SETCURSEL	Выбрать строку. При необходимости список прокручивается. wParam – индекс выделяемой строки
CB_SETDROPPEDWIDTH	Установить минимально возможную ширину раскрывающегося списка в пикселях. wParam – устанавливаемая ширина
CB_SETEDITSEL	Установить начальную и конечную позицию выделенного фрагмента в поле ввода комбинированного списка. Младшее слово lParam задает начальную позиция выделения, старшее слово – конечную позицию
CB_SETEXTENDEDUI	Установить расширенный или обычный пользовательский интерфейс. Если wParam равен TRUE устанавливается расширенный интерфейс, иначе устанавливается обычный интерфейс.
CB_SETHORIZONTALEXTENT	Установить в пикселях ширину списка, на которую он может быть прокручен по горизонтали. wParam – устанавливаемое значение.
CB_SETITEMDATA	Установить значение 32 битного целого, связанного с конкретным элементом. wParam – индекс элемента, для которого нужно установить значение. Если он равен -1, то значение устанавливается всем элементам списка. lParam – устанавливаемое значение.
CB_SETITEMHEIGHT	Установить высоту в пикселях элементов списка. wParam – индекс элемента списка (если у списка установлен стиль CBS_OWNERDRAWVARIABLE, в противном случае ноль). lParam – устанавливаемая высота

CB_SETLOCALE	Установить параметры языка (локаль). wParam – идентификатор устанавливаемой локали.
CB_SETTOPINDEX	Установить первый видимый элемент списка. wParam – индекс элемента, который устанавливается в качестве первого видимого элемента списка.
CB_SHOWDROPDOWN	Показать или скрыть раскрывающийся список. Если wParam равен TRUE то показать список, иначе скрыть его

Из таблицы видно, что работа с комбинированным списком очень похожа на работу с обычным списком.

Практика

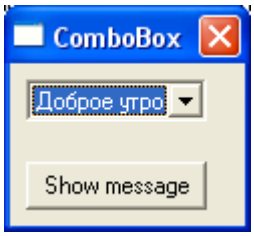
В качестве практического примера давайте немного улучшим программу, которую мы писали для обычного списка. Сделаем ее с использованием комбинированного списка, а само сообщение будем выводить по нажатию на кнопку. Исходный код такого приложения с комментариями приводится ниже:

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'

ID_COMBOBOX = 2
ID_BUTTON = 3

section '.code' code readable executable
   string: db 'Push a Button',0
start:
        invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
     cmp [msg], WM_CLOSE
     je FreeDialog

     cmp [msg], WM_INITDIALOG
     je InitDialog

     cmp [msg], WM_COMMAND
     jne exitproc

     mov eax, BN_CLICKED
     shl eax, 16
     add eax, ID_BUTTON
     cmp [wparam], eax
     jne exitproc
     ; Получаем индекс выбранной сроки
     invoke SendDlgItemMessage, [hwnddlg], ID_COMBOBOX, CB_GETCURSEL,
NULL, NULL
     ; Получаем связанное с ней значение
```

```
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_GETITEMDATA,
eax, NULL

          ; Выводим сообщение пользователю
          invoke MessageBox, 0, eax, _Caption, MB_OK
          ; Выход из диалоговой процедуры
      exitproc:
          xor eax, eax
          ret
          ; Участок кода ответственный за освобождение окна
     FreeDialog:
          invoke EndDialog, [hwnddlg], 0
          xor eax,eax
          ret
     InitDialog:
          ; Заполняем список строками и связываем с ними их адреса
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_ADDSTRING,
NULL, _GoodMorning
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_SETITEMDATA,
eax, _GoodMorning

          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_ADDSTRING,
NULL, _GoodDay
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_SETITEMDATA,
eax, _GoodDay

          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_ADDSTRING,
NULL, _GoodEvening
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_SETITEMDATA,
eax, _GoodEvening

          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_ADDSTRING,
NULL, _GoodNight
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_SETITEMDATA,
eax, _GoodNight
          ; Устанавливаем строку, выбранную по умолчанию
          invoke  SendDlgItemMessage,  [hwnddlg],  ID_COMBOBOX,  CB_SETCURSEL,  0,
0
          xor eax, eax
          ret
     endp

     section '.data' data readable
     _GoodMorning db 'Доброе утро', 0
     _GoodDay db 'Добрый день', 0
     _GoodEvening db 'Добрый вечер', 0
     _GoodNight db 'Доброй ночи', 0
     _Caption db ' ', 0

     section '.idata' import data readable writeable

      library kernel, 'KERNEL32.DLL',\
              user, 'USER32.DLL'

      import kernel,\
             GetModuleHandle, 'GetModuleHandleA',\
             ExitProcess, 'ExitProcess'

      import user,\
             DialogBoxParam, 'DialogBoxParamA',\
             EndDialog, 'EndDialog' ,\
             MessageBox, 'MessageBoxA',\
             SendDlgItemMessage, 'SendDlgItemMessageA'
```

```
     section '.rsrc' resource data readable
      directory RT_DIALOG, dialogs

      resource dialogs,\
             1,LANG_ENGLISH, form1

      dialog form1, 'ComboBox', 70, 70, 65, 50,\
      WS_CAPTION+WS_SYSMENU+DS_CENTER
            dialogitem     'ComboBox',   '',   ID_COMBOBOX,   5,   5,   60,   45,
WS_VISIBLE+LBS_SORT+LBS_NOTIFY
            dialogitem 'Button', 'Show message', ID_BUTTON, 5, 30, 60, 15,
WS_VISIBLE
      enddialog
```

На рисунке ниже представлено главное окно данной программы.



Глава 12. Меню

Практически в каждом приложении под заголовком главного окна отображается полоса меню, содержащего набор пунктов. Отдельные пункты меню могут быть двух видов: подменю, команда.

При выборе подменю открывается новый список пунктов меню более низкого уровня по отношению к тому, из которого выбрано подменю.

При выборе команды осуществляется какое-либо действие.

Описание меню в ресурсах

Для описания меню в ресурсах предопределен тип ресурса RT_MENU. В общем виде, задание ресурса меню выглядит следующим образом:

```
;Задаем тип ресурса
directory RT_MENU, menus
;Задаем параметры языка
resource menus,\
           37,LANG_ENGLISH+SUBLANG_DEFAULT,main_menu
;Задаем само меню
;Перечень пунктов первого уровня
       menuitem '&File',0,MFR_POPUP
                ;Перечень пунктов меню второго уровня
                menuitem '&New',IDM_NEW
                menuseparator
                menuitem 'E&xit',IDM_EXIT,MFR_END
       menuitem '&Help',0,MFR_POPUP + MFR_END
                ;Перечень пунктов меню второго уровня
          menuitem '&About...',IDM_ABOUT,MFR_END, MFS_ENABLED, MF_ENABLED
```

Здесь: «File», «New», «Exit», «Help» и «About…» заголовки пунктов меню.

IDM_NEW, IDM_EXIT и IDM_ABOUT – идентификаторы пунктов меню.

Menuitem и menuseparator – зарезервированные слова.

Зарезервированное слово «Menuitem» говорит о том, что создается обычный пункт меню (или подменю)

Ключевое слово «menuseparator» задает разделитель пунктов меню, который представляет собой горизонтальную черту, разбивающую пункты на логические блоки. На рисунке ниже представлено меню описанное данным образом в ресурсах.

В этом примере для всех пунктов кроме последнего («About…») мы указывали только необходимые параметры. Для последнего пункта мы указали все параметры (в том числе и необязательные). В общем виде, описание отдельно взятого пункта меню выглядит следующим образом:

```
menuitem 'Заголовок пункта меню', ИдентификаторПункта, ФлагиТипаМеню, ПервыйНаборФлагов, ВторойНаборФлагов
```

Теперь поговорим о флагах. Всего существует два флага типа меню, которые могут быть использованы одновременно:

MFR_POPUP – говорит о том, что данный пункт меню является подменю. После пункта меню с таким флагом начинается цепочка пунктов меню более низкого уровня.

MFR_END – говорит о том, что данный пункт является последним в цепочке пунктов подменю. После пункта меню с таким флагом продолжается цепочка пунктов меню более высокого уровня.

Первый и второй наборы флагов очень похожи друг на друга единственное отличие между ними состоит в том, что в первый набор входят флаги начинающиеся с MFS_, а во второй с MF_ или MFT_. В таблице ниже представлены некоторые флаги.

Флаг	**Описание**
MF_CHECKED, MFS_CHECKED	При выводе меню на экран строка меню отмечается галочкой.
MF_DEFAULT, MFS_DEFAULT	Так помечается пункт меню по умолчанию (он выделяется жирным шрифтом)
MF_DISABLED	Строка меню отображается в нормальном виде (не серым цветом), но не может быть выбрана пользователем
MF_ENABLED, MFS_ENABLED	Строка меню может быть использована пользователем
MF_GRAYED, MFS_DISABLED	Строка меню отображается серым цветом и не может быть выбрана пользователем
MF_HILITE, MFS_HILITE	При первом открытии меню пункт меню отмеченный этим флагом будет выделен
MF_MENUBREAK, MFT_MENUBREAK	Если описывается меню верхнего уровня, то элемент выводится с новой строки. Если описывается меню второго и более уровня, то он выводится в новом столбце

MF_MENUBARBREAK, MFT_MENUBARBREAK	Аналогично MF_MENUBREAK, но дополнительно новый столбец отделяется вертикальной линией
MF_OWNERDRAW, MFT_OWNERDRAW	Строка меню рисуется окном, создавшем меню
MF_SEPARATOR, MFT_SEPARATOR	Создается горизонтальная разделительная линия
MF_STRING, MFT_STRING	Элемент меню является строкой символов
MF_UNCHECKED, MFS_UNCHECKED	При выводе меню на экран строка не отмечается галочкой
MF_UNHILITE, MFS_UNHILITE	Пункт меню никак не выделяется цветом при открытии.

Для получения дескриптора меню, описанного в ресурсах используется функция LoadMenu (A/W) из библиотеки User32.dll. Вот ее прототип:

```
HMENU WINAPI LoadMenu(
  __in_opt  HINSTANCE hInstance, //Дескриптор модуля, в котором описано меню
  __in      LPCTSTR lpMenuName   //Имя ресурса, задающего меню или его
                                 //идентификатор
  );
```

Данная функция возвращает дескриптор загруженного меню или ноль, если загрузить последний по какой-то причине не удалось.

Ручное создание меню

Помимо описания меню в ресурсах, существует возможность программно создать меню. Это необходимо в том случае, если состав меню зависит от каких-либо параметров и может меняться.

Для создания меню используется функция CreateMenu из библиотеки User32.dll. Вот ее прототип:

```
HMENU WINAPI CreateMenu(void);
```

Данная функция не имеет входных параметров. Она возвращает дескриптор созданного меню или ноль, если создать меню не удалось. Необходимо учитывать, что данная функция создает абсолютно пустое меню, в котором нет ни одного пункта.

Для того чтобы в меню добавить новый пункт используется функция AppendMenu (A/W) из библиотеки User32.dll. Вот ее описание:

```
BOOL WINAPI AppendMenu(
  __in      HMENU hMenu,          //Дескриптор меню, в которое добавляется
                                  //пункт
  __in      UINT uFlags,          //Набор флагов с префиксом MF_
  __in      UINT_PTR uIDNewItem, //Зависит от флагов
  __in_opt  LPCTSTR lpNewItem    //зависит от флагов
);
```

Если установлен флаг MF_POPUP (то есть пункт меню раскрывает подменю), то параметр uIDNewItem задает дескриптор меню, раскрывающегося данным пунктом меню. В противном случае (если флаг MF_POPUP не установлен) данный параметр задает идентификатор нового пункта меню.

Если установлен флаг MF_BITMAP, то параметр lpNewItem задает дескриптор изображения, которое будет отображаться в этом пункте меню. Если установлен флаг MF_STRING, то параметр lpNewItem задает адрес строки, которая будет выведена в пункте меню.

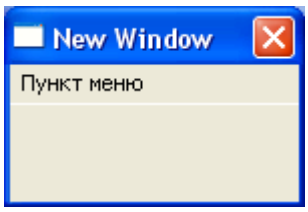
Ниже приводится исходный код программы создающей меню с одним единственным пунктом:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
        mov [hInstance], eax
        ;Заполняем структуру WNDCLASS
        mov [wc.style], CS_HREDRAW+CS_DBLCLKS
        mov [wc.lpfnWndProc], WndProc
        mov [wc.cbClsExtra], 0
        mov [wc.cbWndExtra], 0
        mov [wc.hInstance], eax
        mov [wc.hIcon], NULL
        mov [wc.hCursor], NULL
        mov [wc.hbrBackground], COLOR_BTNSHADOW
        mov [wc.lpszMenuName], NULL
        mov [wc.lpszClassName], _ClassName
        ;Регистрируем класс
        invoke RegisterClass, wc
        test eax, eax
        jz FailtRegister
        ;Создаем окно
        invoke      CreateWindowEx,0,       _ClassName,       _WindowName,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU,   0,   0,   150,   100,   HWND_DESKTOP,   NULL,
[hInstance], NULL
        test eax, eax
        jz FailtCreate
        mov [hWnd], eax
        ;Создаем меню
        invoke CreateMenu
        mov [hMenu], eax
        ;Добавляем в меню новый пункт
        invoke AppendMenu, [hMenu],MF_ENABLED+MF_STRING, 100, _ItemMenu
        ;Устанавливаем меню в созданном ранее окне
        invoke SetMenu, [hWnd], [hMenu]
        ;Запускаем цикл обработки сообщений
```

```
StartLoop:
        invoke GetMessage, msg, NULL, 0, 0
        cmp eax, 1
        jb ExitProgramm
        jne StartLoop

        invoke TranslateMessage, msg
        invoke DispatchMessage, msg
        jmp StartLoop

FailtRegister:
        invoke MessageBox, HWND_DESKTOP,_FailtRegister, NULL, MB_OK
        jmp ExitProgramm
FailtCreate:
        invoke MessageBox, HWND_DESKTOP, _FailtCreate, NULL, MB_OK
ExitProgramm:
        invoke ExitProcess, [msg.wParam]
;Оконная процедура
proc WndProc, hwnd, uMsg, wParam, lParam
     cmp [uMsg], WM_CLOSE
     jz msgCloseWindow
     cmp [uMsg], WM_DESTROY
     jz msgDestroyWindow
     jmp ExitWndProc
msgCloseWindow:
     invoke DestroyWindow, [hwnd]
     jmp ExitWndProc
msgDestroyWindow:
     invoke PostQuitMessage, 0
ExitWndProc:
     invoke DefWindowProc, [hwnd], [uMsg], [wParam], [lParam]
     ret
endp

section '.data' data readable
_ClassName    : db 'ClassName', 0
_WindowName   : db 'New Window', 0
_FailtRegister: db 'Не удалось зарегистрировать класс', 0
_FailtCreate  : db 'Не удалось создать окно', 0
_ItemMenu     : db 'Пункт меню', 0

section '.bss' data readable writeable
hInstance dd ?
hMenu dd ?
hWnd dd ?
wc WNDCLASS
msg MSG


section '.idata' import data readable writeable
library kernel, 'KERNEL32.DLL',\
        user  , 'USER32.DLL'

import kernel,\
       ExitProcess     , 'ExitProcess',\
       GetModuleHandle , 'GetModuleHandleA'

import user,\
       DefWindowProc   , 'DefWindowProcA'  ,\
       MessageBox      , 'MessageBoxA'     ,\
       RegisterClass   , 'RegisterClassA'  ,\
       CreateWindowEx  , 'CreateWindowExA' ,\
       DestroyWindow   , 'DestroyWindow'   ,\
       GetMessage      , 'GetMessageA'     ,\
```

```
TranslateMessage, 'TranslateMessage',\
DispatchMessage , 'DispatchMessageA',\
PostQuitMessage , 'PostQuitMessage' ,\
CreateMenu      , 'CreateMenu'      ,\
AppendMenu      , 'AppendMenuA'     ,\
SetMenu         , 'SetMenu'
```

Главное окно данной программы представлено ниже:



Как видно из рисунка здесь создается простое меню с одним-единственным пунктом «Пункт меню».

Для уничтожения ранее созданного меню используется функция DestroyMenu из библиотеки user32.dll. Вот ее прототип:

```
BOOL WINAPI DestroyMenu(
  __in  HMENU hMenu //Дескриптор уничтожаемого меню
);
```

Данная функция возвращает True если меню успешно уничтожено, или False если уничтожить его по какой-либо причине не удалось (например, был передан неверный дескриптор).

В идеале следует уничтожать все когда-либо создаваемые программой меню (во избежание утечек памяти). Но, так как мы будем создавать только одно меню, которое будет использоваться на всем протяжении работы программы и автоматически уничтожаться при ее завершении, мы не будем вызывать эту функцию.

Присоединение меню к окну

Хорошо. Будем считать, что мы научились создавать меню. Теперь нам надо определить, а как это самое меню присоединить к окну, чтобы пользователь мог с ним работать. Осуществить это можно различными способами в зависимости от того каким образом создается окно.

Способ 1 (с помощью структуры WNDCLASS)

Если вы еще раз взглянете на описание структуры WNDCLASS (подробнее смотри раздел «Регистрация класса окна»), задающей класс окна, то увидите в ней такое поле как lpszMenuName. В данном хранится или адрес строки с наименованием ресурса меню, или идентификатор этого самого ресурса, или NULL, если окно создается без меню.

Достаточно записать в это поле идентификатор ресурса меню и при вызове функции CreateWindowEx будет создано окно с уже присоединенным меню. При этом нам даже не нужно вызывать LoadMenu для создания меню из ресурсов.

Данный способ подходит, если меню создается из ресурсов, а окно создается вызовом функции CreateWindowEx.

Способ 2 (с помощью функции CreateWindowEx)

Если посмотреть на описание этой функции, то можно увидеть, что третьим считая с конца параметром является hMenu, в котором хранится дескриптор меню, которое должно быть присоединено к создаваемому окну.

Данный способ подходит, если окно создается вызовом функции CreateWindowEx, а меню создается или вручную или на основе ресурсов (при этом нужно вызывать LoadMenu).

Способ 3 (с помощью шаблона диалогового окна в памяти)

Если мы вспомним структуру такого шаблона (см. раздел «Создание диалогового окна на основе шаблона в памяти»), то увидим, что в ней следом за структурой DLGTEMPLATE идет так называемый массив меню, в котором задается меню создаваемого диалогового окна. Если первый элемент этого массива равен 0000h, значит, у диалогового окна нет меню. Если он равен FFFFh, то следом за ним идет идентификатор меню в ресурсах. Если же первый элемент отличен и от 0000h и FFFFh система интерпретирует данный массив как Unicode строку, задающую имя ресурса меню. Признаком конца строки служит последовательность 0000h (NULL байт в кодировке Unicode).

Данный способ подходит, если создается диалоговое окно на основе шаблона в памяти, а меню создается на основе ресурсов.

Способ 4 (с помощью функции SetMenu)

Это универсальный способ. Им мы воспользовались в предыдущем разделе при рассмотрении ручного создания меню.

Функция SetMenu присоединяет созданное ранее меню к созданному ранее окну. Вот ее прототип:

```
BOOL WINAPI SetMenu(
  __in      HWND hWnd,  //Дескриптор окна, к которому присоединяется меню
  __in_opt  HMENU hMenu //Дескриптор присоединяемого меню
);
```

Данная функция возвращает True, если меню успешно присоединено и False если выполнить операцию по какой-то причине не удалось.

Ниже приводится исходный текст программы, в которой на основе шаблона создается модальное диалоговое окно, к которому присоединяется меню, также описанное в ресурсах.

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
;Идентификаторы пунктов меню
IDM_NEW   = 101
IDM_EXIT  = 102
IDM_ABOUT = 901

section '.code' code readable executable

start:
        invoke GetModuleHandle, 0
        mov [hInstance], eax

        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0
;описываем диалоговую процедуру
proc DialogProc hwnddlg, msg, wparam, lparam

     cmp [msg], WM_CLOSE
     je FreeDialog

     cmp [msg], WM_INITDIALOG
     je InitDialog

     jmp ExitProcedure
     ;При инициализации диалогового окна присоединяем к нему меню
InitDialog:
     invoke LoadMenu, [hInstance], 37
     invoke SetMenu , [hwnddlg], eax
     jmp ExitProcedure
FreeDialog:
      invoke EndDialog, [hwnddlg], 0
ExitProcedure:
     xor eax,eax
     ret
endp

section '.bss' data readable writeable
hInstance dd ?

;Определяем таблицу импорта
section '.idata' import data readable writeable

 library kernel, 'KERNEL32.DLL',\
         user, 'USER32.DLL'

 import kernel,\
        GetModuleHandle,'GetModuleHandleA',\
        ExitProcess    ,'ExitProcess'

 import user,\
        DialogBoxParam, 'DialogBoxParamA',\
        EndDialog     , 'EndDialog'      ,\
        LoadMenu      , 'LoadMenuA'      ,\
        SetMenu       , 'SetMenu'
;Определяем ресурсы
section '.rsrc' resource data readable
 directory RT_DIALOG, dialogs ,\
           RT_MENU  , menus
 ;Задаем шаблон диалогового окна
 resource dialogs,\
        1,LANG_ENGLISH, form1
```

```
      dialog      form1,      'Window',      70,      70,      100,      100,
WS_CAPTION+WS_SYSMENU+DS_CENTER+DS_SYSMODAL
      enddialog

     ;Задаем шаблон меню
     resource menus,\
                37,LANG_ENGLISH+SUBLANG_DEFAULT,main_menu

       menu main_menu
            ;Перечень пунктов первого уровня
            menuitem '&File',0,MFR_POPUP,,
                     ;Перечень пунктов меню второго уровня
                     menuitem '&New',IDM_NEW
                     menuseparator
                     menuitem 'E&xit',IDM_EXIT,MFR_END , MFS_HILITE
            menuitem '&Help',0,MFR_POPUP + MFR_END
                     ;Перечень пунктов меню второго уровня
                     menuitem    '&About...',IDM_ABOUT,MFR_END,    MF_ENABLED;,
MFS_DEFAULT
```

В данной программе меню присоединяется к диалоговому окну при получении сообщения WM_INITDIALOG (инициализировать диалоговое окно). Само диалоговое окно с подключенным меню представлено на рисунке ниже:

Обработка событий меню

Мы разобрались с тем как создать меню и присоединить его к нужному окну. Теперь поговорим о том, как мы узнаем, что пользователь выбрал тот или иной пункт меню. Узнаем мы это благодаря полученным сообщениям. При работе с меню мы можем получать несколько различных сообщений. В таблице ниже представлены наиболее интересные сообщения:

Сообщение	Описание
WM_COMMAND	Отправляется при выборе доступного пункта меню. Не отправляется при раскрытии подменю. При этом младшее слово параметра wParam содержит идентификатор выбранного пункта, старшее слово равно нулю.
WM_INITMENU	Отправляется когда пользователь щелкает левой кнопкой мыши по пункту меню, являющемся подменю перед отображением этого подменю на экране. При этом wParam задает дескриптор раскрывающегося меню.
WM_INITMENUPOPUP	Аналогично сообщению WM_INITMENU но отправляется при любом раскрытии подменю, даже если пользователь не щелкал левой кнопкой мыши на соответствующем пункте.
WM_MENUSELECT	Похоже на WM_COMMAND, но отправляется не только при выборе доступного пункта меню, но и при выборе подменю, недоступного пункта меню, а также при наведении на пункт меню курсора мыши.

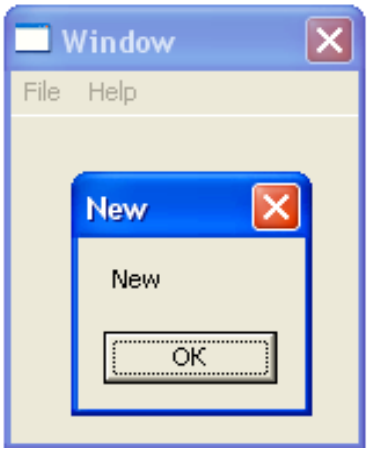
Ниже приводится исходный текст программы, которая определяет какой пункт меню был выбран и в соответствии с этим выводит сообщение на экран.

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
;Идентификаторы пунктов меню
IDM_NEW   = 101
IDM_EXIT  = 102
IDM_ABOUT = 103

section '.code' code readable executable

start:
        invoke GetModuleHandle, 0
        mov [hInstance], eax

        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam

     cmp [msg], WM_CLOSE
     je FreeDialog

     cmp [msg], WM_INITDIALOG
     je InitDialog
```

```
     cmp [msg], WM_COMMAND
     je CheckSelectNew

     jmp ExitProcedure

InitDialog:
     invoke LoadMenu, [hInstance], 37
     invoke SetMenu , [hwnddlg], eax
     jmp ExitProcedure
;Проверяем был выбран пункт New?
CheckSelectNew:
     xor ebx, ebx
     mov eax, IDM_NEW
     cmp eax, [wparam]
     jne CheckSelectExit
     mov ebx, _New
     jmp Message
;Проверяем был выбран пункт Exit?
CheckSelectExit:
     mov eax, IDM_EXIT
     cmp eax, [wparam]
     jne CheckSelectAbout
     mov ebx, _Exit
     jmp Message
;Проверяем был выбран пункт About?
CheckSelectAbout:
     mov eax, IDM_ABOUT
     cmp eax, [wparam]
     jne ExitProcedure
     mov ebx, _About

Message:
     invoke MessageBox, [hwnddlg], ebx, ebx, MB_OK
     jmp ExitProcedure

FreeDialog:
     invoke EndDialog, [hwnddlg], 0

ExitProcedure:
     xor eax,eax
     ret
endp

section '.bss' data readable writeable
hInstance dd ?
_New db 'New', 0
_Exit db 'Exit', 0
_About db 'About', 0

section '.idata' import data readable writeable

 library kernel, 'KERNEL32.DLL',\
         user, 'USER32.DLL'

 import kernel,\
        GetModuleHandle,'GetModuleHandleA',\
        ExitProcess    ,'ExitProcess'

 import user,\
        DialogBoxParam, 'DialogBoxParamA',\
        EndDialog     , 'EndDialog'      ,\
        LoadMenu      , 'LoadMenuA'      ,\
        SetMenu       , 'SetMenu'        ,\
        MessageBox    , 'MessageBoxA'
```

```
    section '.rsrc' resource data readable
     directory RT_DIALOG, dialogs ,\
               RT_MENU  , menus

     resource dialogs,\
            1,LANG_ENGLISH, form1

     dialog      form1,      'Window',      70,      70,      100,      100,
WS_CAPTION+WS_SYSMENU+DS_CENTER+DS_SYSMODAL
     enddialog


    resource menus,\
               37,LANG_ENGLISH+SUBLANG_DEFAULT,main_menu

      menu main_menu
            menuitem '&File',0,MFR_POPUP,,
                     menuitem '&New',IDM_NEW
                     menuseparator
                     menuitem 'E&xit',IDM_EXIT,MFR_END , MFS_HILITE
           menuitem '&Help',0,MFR_POPUP + MFR_END
                     menuitem     '&About...',IDM_ABOUT,MFR_END,     MF_ENABLED;,
MFS_DEFAULT
```

На рисунке ниже представлен внешний вид данного диалогового окна с примером выводимого сообщения:



Редактирование меню

Выше мы научились создавать меню и добавлять в него новые пункты с помощью функции AppendMenu. Теперь поговорим о других способах изменения меню.

Функция DeleteMenu из библиотеки user32.dll позволяет удалить пункт меню. Вот ее прототип:

```
BOOL WINAPI DeleteMenu(
  __in  HMENU hMenu,      //Дескриптор меню, из которого нужно удалить пункт
  __in  UINT uPosition,   //Позиция удаляемого пункта. Зависит от флага
  __in  UINT uFlags       //Флаги
);
```

Параметр uFlags определяет содержимое параметра uPosition. uFlags может принимать одно из следующих значений:

MF_BYCOMMAND – в этом случае uPosition должен содержать идентификатор удаляемого пункта меню

MF_BYPOSITION – в этом случае uPosition должен содержать порядковый номер пункта в меню (отсчет идет от нуля).

Функция ModifyMenu из библиотеки user32.dll позволяет изменить пункт меню. Вот как она выглядит:

```
BOOL WINAPI ModifyMenu(
  __in      HMENU hMenu,          //Дескриптор меню, пункт которого меняется
  __in      UINT uPosition,       //Позиция редактируемого пункта.
  __in      UINT uFlags,          //Флаги
  __in      UINT_PTR uIDNewItem, //Новый идентификатор пункта меню
  __in_opt  LPCTSTR lpNewItem     //Новое содержание пункта меню
);
```

Здесь набор флагов uFlags определяет не только то как будет интерпретироваться значение параметра uPosition (точно так же как и в функции DeleteMenu рассмотренной ранее) но и то, как будет интерпретироваться значение параметра lpNewItem (точно так же как и в функции AppendMenu[1]).

[1] Подробнее смотри раздел «Ручное создание меню»

Глава 13. Взаимодействие с окнами других приложений
Поиск окон

Выше мы рассмотрели создание и уничтожение окна. Теперь поговорим о том, как управлять окнами, манипулировать ими.

Прежде всего, для управления нам нужно знать дескриптор окна. Если окно создано нами то, мы, как правило, уже знаем его дескриптор, но, если окно принадлежит другому приложению то, дескриптор нам не известен. Определить его можно с помощью функций FindWindow (A/W) или FindWindowEx (A/W). Начнем с первой, вот как она выглядит:

```
HWND WINAPI FindWindow(
  __in_opt  LPCTSTR lpClassName, //указатель на строку с именем класса окна
  __in_opt  LPCTSTR lpWindowName //указатель на строку с заголовком окна
);
```

Данная функция возвращает дескриптор найденного окна, или ноль, если окно не найдено. Функция FindWindowEx очень похожа на нее:

```
HWND WINAPI FindWindowEx(
  __in_opt  HWND hwndParent,     //Дескриптор родительского окна
  __in_opt  HWND hwndChildAfter, //Дескриптор окна, после которого следует
                                 //искать
  __in_opt  LPCTSTR lpszClass,   //указатель на строку с именем класса окна
  __in_opt  LPCTSTR lpszWindow   //указатель на строку с заголовком окна
);
```

Эта функция, так же как и функция FindWindow (A/W) возвращает дескриптор найденного окна, иди ноль, если окно не найдено.

Функция FindWindowEx (A/W) осуществляет поиск окна, являющегося дочерним по отношению к окну с дескриптором hwndParent. Причем поиск может быть начат не с первого окна, а с определенного окна среди всех дочерних окон. Указать на окно, с которого следует начать поиск можно с помощью параметра hwndChilAfter (дескриптор предыдущего окна, по отношению к тому, с которого следует начать поиск).

Что ж, с получением дескрипторов будем считать, что разобрались. Теперь переходим непосредственно к управлению. Управлять окном можно двумя способами: посредством отправки ему определенных сообщений и с помощью специальных функций.

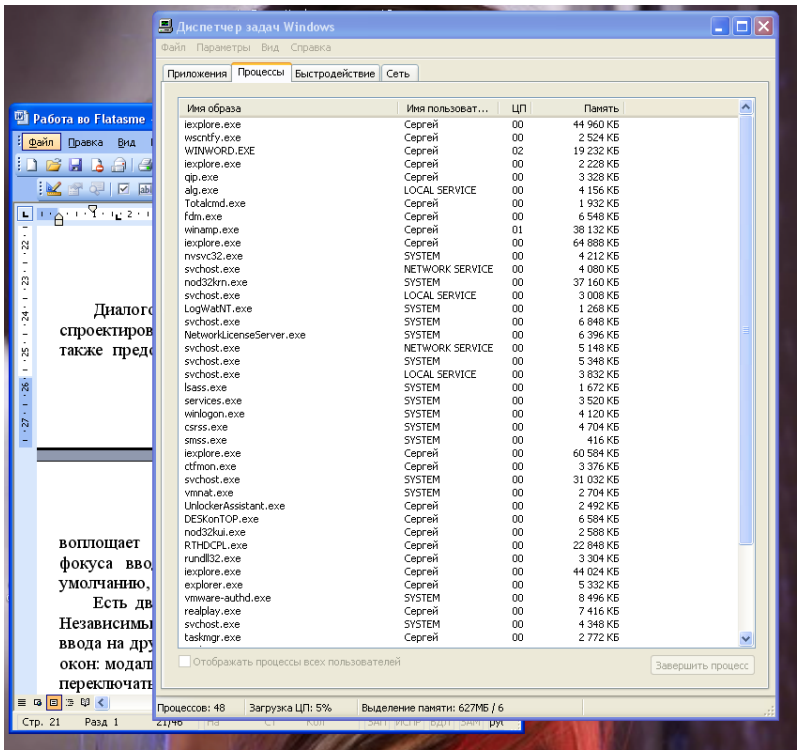
Посредством сообщений можно закрыть (WM_CLOSE), уничтожить окно (WM_DESTROY), К сожалению (или к счастью), сообщения носят уведомительный а не манипулятивный характер. Это значит, что управление другими окнами сообщениями несколько ограничено, но возможно.

Функция SetWindowPos

Функция SetWindowPos позволяет изменить размеры окна, положение окна на экране, а также в Z-последовательности. Вот как выглядит эта функция:

```
BOOL WINAPI SetWindowPos(
  __in      HWND hWnd,             //Дескриптор окна, которым мы управляем
  __in_opt  HWND hWndInsertAfter,  //Дескриптор порядка размещения
  __in      int X,                 //Новая позиция по горизонтали
  __in      int Y,                 //Новая позиция по вертикали
  __in      int cx,                //Новая ширина окна
  __in      int cy,                //Новая высота окна
  __in      UINT uFlags            //Флаги
    );
```

Для того, чтобы понять назначение параметра hWndInsertAfter следует вспомнить назначение стиля WS_EX_TOPMOST. Окна с этим стилем даже находясь в неактивном состоянии, перекрывают другие окна, не обладающими им[1]. Хорошим примером такого окна является диспетчер задач Windows на рисунке ниже. Несмотря на то, что активным приложением является приложение Word, диспетчер задач все равно перекрывает его.



Ну так вот, параметр hWndInsertAfter позволяет установить или снять этот флаг с подопытного окна вне зависимости от того является оно диалоговым или не является.

[1] Аналогичным образом различаются модальные и системно-модальные диалоговые окна

hWndInsertAfter	Описание
HWND_NOTOPMOST	Снять флаг WS_EX_TOPMOST и сделать «обычным» окном
HWND_TOPMOST	Установить флаг WS_EX_TOPMOST если у окна нет такого стиля

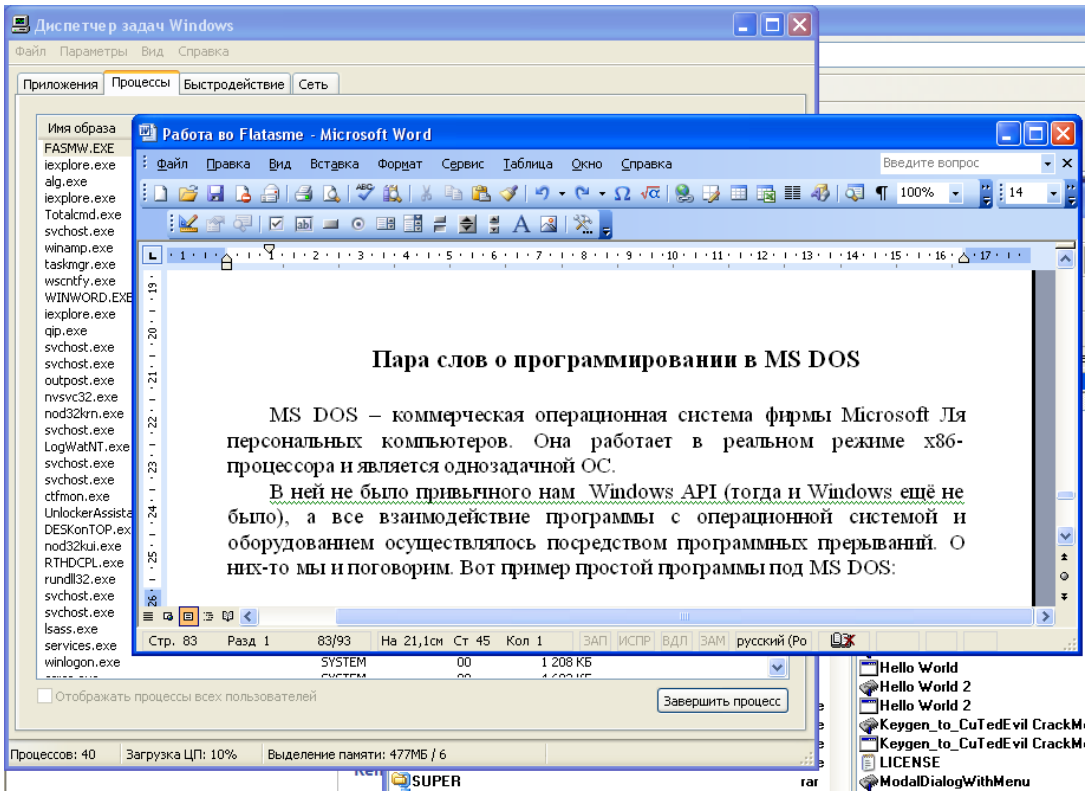
Параметр uFlags представляет комбинацию флагов. Некоторые из которых представлены в таблице ниже:

Флаг	Описание
SWP_HIDEWINDOW	Скрыть окно (сделать его невидимым)
SWP_NOMOVE	Не перемещать окно. При этом параметры X и Y игнорируются
SWP_NOSIZE	Сохранить текущие размеры окна. При этом параметры cx и cy игнорируются.
SWP_SHOWWINDOW	Показать окно (сделать его видимым)

Ниже приводится исходный текст программы, которая находит окно диспетчера задач Windows, перемещает его в левый верхний угол экрана и «уравнивает в правах» с другими окнами.

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\WIN32A.INC'

section '.code' code readable executable
start:
        ;Ищем окно диспетчера задач по его заголовку
        invoke FindWindow, NULL, _WindowCaption
        ;Убеждаемся в том, что окно найдено
        test eax, eax
        jz NotFound
        ;Изменяем его атрибуты
        invoke SetWindowPos, eax, HWND_NOTOPMOST, 0, 0, 0, 0, SWP_NOSIZE
        jmp Exit
NotFound:
        invoke MessageBox, HWND_DESKTOP, _TextNotFound, NULL, MB_OK
Exit:
        invoke ExitProcess, 0

section '.data' data readable
_WindowCaption db 'Диспетчер задач Windows', 0
_TextNotFound db 'Не удалось найти окно', 0

section '.idata' data import readable writeable
library user,   'USER32.DLL',\
         kernel, 'KERNEL32.DLL'

import user,\
        FindWindow, 'FindWindowA',\
        MessageBox, 'MessageBoxA',\
        SetWindowPos, 'SetWindowPos'

import kernel,\
        ExitProcess, 'ExitProcess'
```

Результат работы этой программы представлен на рисунке ниже:



Как видно из этого рисунка теперь активный Word перекрывает собой неактивный диспетчер задач.

Функция MoveWindow

Данная функция похожа на рассмотренную ранее SetWindowPos. Она так же позволяет изменить расположение и размеры окна, но имеет и ряд существенных отличий. Функция MoveWindow экспортируется библиотекой User32.dll. Вот ее прототип:

```
BOOL WINAPI MoveWindow(
  __in  HWND hWnd,      //Дескриптор окна
  __in  int X,          //Новая координата окна по горизонтали
  __in  int Y,          //Новая координата окна по вертикали
  __in  int nWidth,     //Новая ширина окна в пикселях
  __in  int nHeight,    //Новая высота окна в пикселях
  __in  BOOL bRepaint   //Нужно ли перерисовывать окно
);
```

Думаю, назначение параметров здесь понятно без лишних слов, за исключением последнего параметра. Если bRepaint установлен в значение TRUE, то окну, размеры и положение которого мы меняли будет отправлено сообщение WM_PAINT. То есть, данное окно будет перерисовано. В

противном случае (если параметр bRepaint установлен в значение FALSE) окну никакого дополнительного сообщения отправляться не будет.

Как видно из прототипа данной функции в ней нельзя изменить что-то одно, то есть либо размеры окна, либо его расположение, как это было возможно в функции SetWindowPos.

В случае успеха она вовзращает ненулевое значение и нуль в случае ошибки.

Приводить пример использования данной функции я не буду, так как ее использование мало чем отличается от использования функции SetWindowPos, работа с которой обсуждалась в предыдущем разделе.

Функция ShowWindow

Данная функция позволяет изменить режим отображения окна. Вот ее прототип:

```
BOOL WINAPI ShowWindow(
  __in  HWND hWnd,    //Дескриптор окна
  __in  int nCmdShow  //Набор флагов
);
```

Параметр nCmdShow задает новый режим отображения окна. Он представляет комбинацию флагов. Некоторые доступные флаги представлены в таблице:

Значение	Описание
SW_HIDE	Сделать окно невидимым
SW_MAXIMIZE	Максимизировать окно, то есть расширить на весь экран
SW_MINIMIZE	Минимизировать окно, то есть свернуть его
SW_RESTORE	Если окно было свернуто или растянуто на весь экран, то восстанавливает его предыдущее состояние
SW_SHOW	Показать окно (сделать видимым) и сделать его активным
SW_SHOWMAXIMIZED	Показать окно, растянуть его на весь экран и сделать активным
SW_SHOWMINIMIZED	Сделать окно видимым и свернуть его
SW_SHOWNA	Аналогично SW_SHOW но окно не делается активным
SW_SHOWNORMAL	Сделать окно таким, каким оно является при открытии

Глава 14. Работа с динамической памятью

Функции работы с динамической памятью

Во всех предыдущих примерах мы работали исключительно со статической памятью, то есть с памятью, которая резервируется под нужды программы на этапе ее загрузки и не может быть изменена (увеличена или уменьшена в размере или перемещена) во время ее выполнения. Это не всегда удобно, по той простой причине, что очень часто мы не можем заранее предугадать, сколько конкретно памяти нам потребуется для выполнения той или иной задачи.

При использовании статической памяти мы вынуждены заранее закладываться на какой-либо ограниченный объем памяти. Если программе требуется меньше памяти, чем мы предусмотрели то это полбеды: максимум, что происходит это напрасное выделение памяти, которая никогда не будет использована. Но вот если предусмотренного нами количества памяти оказывается недостаточно, то при использовании статической памяти мы вынуждены или полностью отказаться от решения поставленной задачи, либо обрезать данные до нужного размера (а это может привести к трудно предсказуемому результату, кто знает что именно мы «отрежем»).

Динамическая память позволяет избавиться от этих ограничений. При работе с ней мы выделяем память тогда, когда она нам нужна и в том количестве, в котором она нам нужна. Если в процессе выполнения программы станет ясно, что выделенной ранее памяти недостаточно, мы можем расширить выделенный ранее регион памяти до нужного нам размера.

Для того чтобы выделить необходимый объем памяти используется функция LocalAlloc из библиотеки kernel32.dll. Вот ее прототип:

```
HLOCAL WINAPI LocalAlloc(
  __in  UINT uFlags,   //Флаги
  __in  SIZE_T uBytes  //Размер выделяемой области в байтах
);
```

Параметр uFlags представляет собой комбинацию флагов, некоторые из которых представлены в таблице ниже:

Флаг	Описание
LHND	LMEM_MOVEABLE+LMEM_ZEROINIT
LMEM_FIXED	Функция возвращает адрес выделенного блока памяти
LMEM_MOVEABLE	Функция возвращает дескриптор выделенного блока памяти
LMEM_ZEROINIT	Заполняет выделенный блок памяти нулями
LPTR	LMEM_FIXED+LMEM_ZEROINIT
NONZEROLHND	Тоже что и LMEM_MOVEABLE
NONZEROLPTR	Тоже что и LMEM_FIXED

В случае ошибки данная функция вовзращает NULL.

Для того чтобы получить адрес выделенного блока памяти по его дескриптору используется функция LocalLock из той же библиотеки. Вот ее прототип:

```
LPVOID WINAPI LocalLock(
  __in  HLOCAL hMem  //Дескриптор блока памяти
);
```

В случае успеха данная функция возвращает адрес выделенного блока памяти, а в случае ошибки значение NULL.

Для обратной операции (получение дескриптора блока памяти по его адресу) используется функция LocalHandle. Вот ее прототип:

```
HLOCAL WINAPI LocalHandle(
  __in  LPCVOID pMem  //Адрес блока памяти
);
```

В случае ошибки данная функция возвращает значение NULL.

После окончания работы с выделенным ранее блоком памяти его нужно освободить. Для этого используется функция LocalFree. Вот ее прототип:

```
HLOCAL WINAPI LocalFree(
  __in  HLOCAL hMem  //Дескриптор освобождаемого блока памяти
);
```

В случае успеха данная функция возвращает значение NULL, а в случае ошибки она вернет значение, переданное ей в параметре hMem.

Для того чтобы изменить размер ранее выделенного блока памяти используется функция LocalReAlloc. Вот ее прототип:

```
HLOCAL WINAPI LocalReAlloc(
  __in  HLOCAL hMem,  //Дескриптор блока памяти, размер которого меняется
  __in  SIZE_T uBytes,//Новый размер блока памяти
  __in  UINT uFlags   //Набор флагов
);
```

Параметр uFlags представляет собо значение заданное одним или двумя флагами.

Флаг LMEM_MODIFY говорит о том, что размер области меняться не будет (параметр uBytes игнорируется). Совместно с ним может быть определен флаг LMEM_MOVEABLE, который говорит о том, что блок памяти заданный параметром hMem следует разместить по иному адресу.

Если флаг LMEN_MODIFY не задан, то может быть задан флаг LMEM_ZEROINIT, который говорит о том, что дополнительную память, присоединяемую к уже имевшейся следует заполнить нулями.

Для того чтобы определить размер имеющегося буфера памяти используется функция LocalSize. Вот ее протип:

```
UINT WINAPI LocalSize(
  __in  HLOCAL hMem //Дескриптор блока памяти
);
```

Данная функция возвращает размер выделенного блока памяти, или ноль в случае ошибки.

Пример программы

При рассмотрении элемента управления поле ввода мы писали программу, считывающую из поля ввода введенную пользователем строку и выводящую ее на экран через MessageBox. Тогда для хранения введенной строки мы использовали статическую память и не позволяли вводить строки длиннее 32 символов. Давайте сейчас улучшим эту программу и предоставим пользователю возможность вводить строку произвольной длины, используя динамическую память. Исходный код этой программы представлен ниже:

```
format PE GUI 4.0
entry start
include 'D:\FASM\INCLUDE\win32a.inc'
ID_EDIT = 2

section '.code' code readable executable
  start:invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

        proc DialogProc hwnddlg, msg, wparam, lparam
             cmp [msg], WM_CLOSE
             je FreeDialog

             cmp [msg], WM_INITDIALOG
             je InitDialog

             cmp [msg], WM_COMMAND
             jne ExitProc
             mov eax, BN_CLICKED
             shl eax, 10h
             add eax, 3
             cmp [wparam], eax
             jne ExitProc
             ;Получаем длину строки введенной в поле ввода
             invoke GetWindowTextLength, [hwndEdit]
             ;Увеличиваем длину на единицу (под завершающий нуль)
             inc eax
             push eax
             invoke LocalAlloc, LPTR, eax
             ;в ebx у нас попадает длина введенной строки
             pop ebx
             ;Сохраняем адрес выделенного блока памяти
             push eax
             invoke GetDlgItemText, [hwnddlg], ID_EDIT, eax, ebx
             mov eax, [esp]
             invoke MessageBox, [hwnddlg], eax, eax, MB_OK
             pop eax
              ;Получаем дескриптор выделенного блока памяти
             invoke LocalHandle, eax
             ;Освобождаем память
             invoke LocalFree, eax
```

```
                ret
        InitDialog:
                invoke GetDlgItem, [hwnddlg],  ID_EDIT
                mov [hwndEdit], eax
                jmp ExitProc

        FreeDialog:
                invoke EndDialog, [hwnddlg], 0
        ExitProc:
                xor eax, eax
                ret
              endp

    section '.bss' readable writeable
    hwndEdit dd NULL

    section '.idata' import data readable writeable
            library kernel, 'KERNEL32.DLL' ,\
                    user  , 'USER32.DLL'

            import kernel,\
                   GetModuleHandle, 'GetModuleHandleA',\
                   ExitProcess    , 'ExitProcess'     ,\
                   LocalAlloc     , 'LocalAlloc'      ,\
                   LocalHandle    , 'LocalHandle'     ,\
                   LocalFree      , 'LocalFree'

            import user,\
                   DialogBoxParam, 'DialogBoxParamA',\
                   EndDialog     , 'EndDialog'      ,\
                   MessageBox    , 'MessageBoxA'    ,\
                   GetDlgItemText, 'GetDlgItemTextA',\
                   GetDlgItem    , 'GetDlgItem'     ,\
                   GetWindowTextLength, 'GetWindowTextLengthA'


    section '.rsrc' resource data readable
            directory RT_DIALOG, dialogs
            resource dialogs,\
                     1, LANG_NEUTRAL, WorkWithEdit
            dialog  WorkWithEdit,  'Work  with  Edit  class',  0,  0,  150,  50,
WS_CAPTION+WS_SYSMENU+DS_CENTER
                   dialogitem 'Edit',    '',           ID_EDIT, 10, 10, 130,
12, WS_VISIBLE+WS_BORDER+ES_AUTOHSCROLL
                   dialogitem  'Button',  'Show  text',  3,  45,  25,  50,   15,
WS_VISIBLE
            enddialog
```

В msdn по поводу данных функций говорится примерно следующее: несмотря на то, что данные функции предоставляют больше возможностей по работе, чем функции по работе с кучей, они значительно медленнее последних. Поэтому новые приложения должны использовать функции работы с кучей.

Это говорит о том, что использовать эти функции в своих программах нежелательно, однако некоторые функции windows API, например FormatMessage в некоторых случаях вынуждают нас прибегать к этим функциям. Поэтому я и включил их описание.

Что такое куча

В очень многих источниках понятия куча и динамическая память являются синонимами. Действительно, куча – это регион зарезервированного адресного пространства, из которого программе может быть выделен какой-либо участок памяти или несколько участков различного размера.

Тут важно понимать зарезервированная память – это память, которая потенциально может быть выделена программе. Это как забронированный номер в отеле до того как вы въехали в этот отель. Вы знаете, что у вас забронированный номер, которым никто не сможет воспользоваться кроме вас (официально и фактически), но вы не можете воспользоваться этим номером пока не въедете в отель. Выделенная память – это память предоставленная программе в непосредственное пользование, то есть это номер отеля, в который мы въехали.

Но куча – это не просто какой-то абстрактный регион памяти, это полноценный объект операционной системы со своим дескриптором. По умолчанию, у каждого процесса уже есть своя, так называемая системная куча, размером в 1 мегабайт. Это динамическая память, предоставляемая процессу по умолчанию. Однако каждая программа может создавать сколько угодно своих куч (точнее столько, на сколько хватит системных ресурсов), различного размера.

Для получения дескриптора системной кучи используется функция GetProcessHeap из библиотеки kernel32.dll. Вот ее прототип:

```
HANDLE WINAPI GetProcessHeap(void);
```

Данная функция не имеет входных параметров. Все что она делает, это возвращает дескриптор кучи, предоставленной системой текущему процессу, или NULL, если выполнить это по какой-либо причине не удалось.

Для создания своей собственной кучи используется функция HeapCreate из той же библиотеки. Вот как она выглядит:

```
HANDLE WINAPI HeapCreate(
  __in  DWORD flOptions,      //Набор атрибутов создаваемой кучи
  __in  SIZE_T dwInitialSize, //Начальный размер кучи в байтах
  __in  SIZE_T dwMaximumSize  //Максимальный размер кучи в байтах. Если
                              //равен нулю, то максимальный размер
                             //ограничен только размером доступной памяти
);
```

Данная функция возвращает дескриптор созданной кучи или NULL, если создать кучу не удалось.

В таблице ниже представлены доступные атрибуты кучи, различное сочетание которых может быть указано в параметре flOptions:

Атрибут	Описание
HEAPE_CREATE_ENABLE _EXECUTE	Все участки памяти, выделенные из данной кучи, могут содержать исполняемый код, который может быть выполнен
HEAP_GENERATE_EXCEP TIONS	Генерировать исключение в случае какой-либо ошибки при работе с данной кучей. Обычно при возникновении ошибок возвращается значение NULL
HEAP_NO_SERIALIZE	Куча не является сериализуемой. То есть к ней могут одновременно обращаться более одного потока

Для уничтожения кучи используется функция HeapDestroy из библиотеки kernel32.dll. Вот ее описание:

```
BOOL WINAPI HeapDestroy(
  __in  HANDLE hHeap  //Дескриптор уничтожаемой кучи
);
```

Обращаю внимание, что программа должна сама уничтожать все созданные ею кучи, во избежание утечек системных ресурсов.

Функции работы с кучей

После того как мы разобрались с терминами и научились создавать (уничтожать) кучи. Можем перейти непосредственно к функциям работы с памятью.

Выделение памяти

Для выделения памяти из кучи используется функция HeapAlloc из библиотеки kernel32.dll. Вот как она выглядит:

```
LPVOID WINAPI HeapAlloc(
  __in  HANDLE hHeap,  //Дескриптор кучи, из которой выделяется память
  __in  DWORD dwFlags, //Набор флагов
  __in  SIZE_T dwBytes //Размер выделяемой области в байтах
);
```

В случае успешного завершения данная функция вернет адрес выделенной из кучи области. Если же она не сможет выполниться успешно, то будет возвращено значение NULL.

В таблице ниже представлены основные флаги, сочетанием которых и является параметр dwFlags.

Флаг	Описание
HEAP_GENERATE_EXCEPTIONS	В случае ошибки выделения памяти из кучи сгенерировать исключение
HEAP_NO_SERIALIZE	Выделенный блок памяти не является сериализуемым, то есть к нему может одновременно обращаться более одного потока
HEAP_ZERO_MEMORY	Выделенный блок памяти инициализируется нулями.

Освобождение памяти

После того, как выделенный ранее блок памяти стал ненужным, его следует освободить, для этого используется функция HeapFree из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI HeapFree(
  __in  HANDLE hHeap,  //Дескриптор кучи, память из которой освобождается
  __in  DWORD dwFlags, //Набор флагов
  __in  LPVOID lpMem   //Адрес ранее выделенной области, подлежащей
                       //освобождению
);
```

В случае успешного освобождения памяти данная функция возвращает ненулевое значение и возвращает ноль, если освободить область памяти не удалось.

Для параметра dwFlags доступен всего один флаг HEAP_NO_SERIALIZE, назначение которого тоже, что и для функции HeapAlloc рассмотренной ранее.

Изменение размера памяти

Для изменения размера памяти используется функция HeapReAlloc из библиотеки kernel32.dll. Вот ее описание:

```
LPVOID WINAPI HeapReAlloc(
  __in  HANDLE hHeap,   //Дескриптор кучи
  __in  DWORD dwFlags,  //Набор флагов
  __in  LPVOID lpMem,   //адрес перераспределяемого блока памяти
  __in  SIZE_T dwBytes  //новый размер блока памяти в байтах
);
```

Данная функция возвращает адрес нового блока памяти или NULL, если перераспределить память не удалось.

В таблице ниже представлены флаги, комбинацией которых является поле dwFlags.

Флаг	Описание
HEAP_GENERATE_EXCEPTIONS	В случае ошибки выделения памяти из кучи сгенерировать исключение
HEAP_NO_SERIALIZE	Выделенный блок памяти не является сериализуемым, то есть к нему может одновременно обращаться более одного потока
HEAP_REALLOC_IN_PLACE_ONLY	Перераспределяемый блок памяти нельзя перемещать на новое место. По умолчанию он может быть перемещен.
HEAP_ZERO_MEMORY	Присоединенная к блоку память инициализируется нулями.

Пример программы

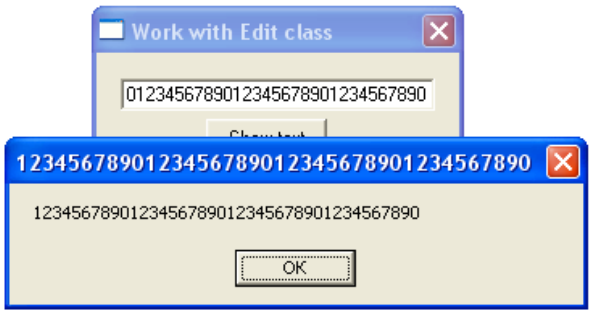
Давайте перепишем программу, которую мы писали для демонстрации работы с динамической памятью так, чтобы она использовала функции работы с кучей. Вот ее исходный код:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
;Идентификатор поля ввода
ID_EDIT = 2

section '.code' code readable executable
  start:invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

        proc DialogProc hwnddlg, msg, wparam, lparam
             cmp [msg], WM_CLOSE
             je DestroyHeap

             cmp [msg], WM_INITDIALOG
             je InitDialog

             cmp [msg], WM_COMMAND
             jne ExitProc
             mov eax, BN_CLICKED
             shl eax, 10h
             add eax, 3
             cmp [wparam], eax
             jne ExitProc
             ;Получаем длину строки введенной в поле ввода
             invoke GetWindowTextLength, [hwndEdit]
             ;Увеличиваем длину на 1 (под завершающий ноль)
             inc eax
             push eax
             ;Выделяем память под строку
             invoke HeapAlloc, [hHeap], HEAP_ZERO_MEMORY, eax
             ;в ebx у нас попадает длина введенной строки
             pop ebx
             ;Сохраняем адрес выделенного блока памяти
```

```
                push eax
                ;Считываем строку из поля ввода и заносим ее в выделенную
                ;область памяти
                invoke GetDlgItemText, [hwnddlg], ID_EDIT, eax, ebx
                ;Вновь заносим в eax адрес выделенной области
                mov eax, [esp]
                ;Выводим строку пользователю
                invoke MessageBox, [hwnddlg], eax, eax, MB_OK
                ;Освобождаем память
                pop eax
                invoke HeapFree, [hHeap], 0, eax
                ret
      InitDialog:
                invoke GetDlgItem, [hwnddlg],  ID_EDIT
                mov [hwndEdit], eax
                ;Создаем кучу
                invoke HeapCreate, 0, 0, 0
                ;Убеждаемся в том, что куча успешно создана
                cmp eax, NULL
                jne GoodCreateHeap
                invoke    MessageBox,    [hwnddlg],_FailtCreateHeap,    NULL,
MB_ICONERROR
                jmp FreeDialog
      GoodCreateHeap:
                ;Сохраняем дескриптор созданной кучи
                mov [hHeap], eax
                jmp ExitProc


      DestroyHeap:
                ;Уничтожаем кучу
                invoke HeapDestroy, [hHeap]
      FreeDialog:
                invoke EndDialog, [hwnddlg], 0
      ExitProc:
                xor eax, eax
                ret
            endp

    section '.bss' readable writeable
    hwndEdit dd NULL
    hHeap dd NULL

    _FailtCreateHeap db 'Не удалось создать кучу', 0


    section '.idata' import data readable writeable
            library kernel, 'KERNEL32.DLL' ,\
                    user  , 'USER32.DLL'

            import kernel,\
                   GetModuleHandle, 'GetModuleHandleA',\
                   ExitProcess    , 'ExitProcess'     ,\
                   HeapCreate     , 'HeapCreate'      ,\
                   HeapDestroy    , 'HeapDestroy'     ,\
                   HeapAlloc      , 'HeapAlloc'       ,\
                   HeapFree       , 'HeapFree'

            import user,\
                   DialogBoxParam, 'DialogBoxParamA',\
                   EndDialog     , 'EndDialog'      ,\
                   MessageBox    , 'MessageBoxA'    ,\
                   GetDlgItemText, 'GetDlgItemTextA',\
                   GetDlgItem    , 'GetDlgItem'     ,\
```

```
                    GetWindowTextLength, 'GetWindowTextLengthA'


    section '.rsrc' resource data readable
            directory RT_DIALOG, dialogs
            resource dialogs,\
                     1, LANG_NEUTRAL, WorkWithEdit
            dialog WorkWithEdit, 'Work with Edit class', 0, 0, 150, 50,
WS_CAPTION+WS_SYSMENU+DS_CENTER
                    dialogitem 'Edit',    '',           ID_EDIT, 10, 10, 130,
12, WS_VISIBLE+WS_BORDER+ES_AUTOHSCROLL
                    dialogitem 'Button', 'Show text', 3, 45, 25, 50,   15,
WS_VISIBLE
            enddialog
```

Результат работы этой программы представлен на рисунке ниже:



Глава 15. Работа со строками

Что такое строка на самом деле

Внимательный читатель, прочитав заголовок этого раздела, сразу воскликнет: погоди, погоди, какие строки. Ты же сам говорил, что в ассемблере нет никаких типов и строк в том числе.

Ну что ж, пришло время разобраться в терминах. В ассемблере под термином строка я понимаю любую последовательность символов, заканчивающуюся нулевым байтом (00h). В языке Си, кстати говоря, используется точно такое же определение. Как вы знаете, любой символ является ничем иным как числом размером в байт из специальной таблицы (в частности, таблицы ASCII), которое специальным образом интерпретируется компьютером. Это позволяет обрабатывать символы как числа. Строка - это последовательность символом (читай: чисел). А признаком конца служит нулевой символ (байт 00h).

В принципе все типовые строковые операции могут быть реализованы средствами языка Ассемблер без привлечения средств Win32 API. Однако, чтобы в сотый раз не изобретать велосипед в этой главе мы рассмотрим средства предоставляемые нам Windows для работы со строками. Вот о них мы и поговорим.

Функции работы со строками

Определение длины строки

Для определения длины строки используется функция lstrlen (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
int WINAPI lstrlen(
  __in  LPCTSTR lpString //Адрес строки, длину которой измеряем
);
```

Как нетрудно догадаться данная функция просто ищет нулевой байт (или слово в случае Unicode строк) и возвращает подсчитанную таким нехитрым образом длину строки.

Сравнение строк

Для сравнения двух строк используется функция lstrcmp (A/W) из библиотеки kernel32.dll. Вот как она выглядит:

```
int WINAPI lstrcmp(
  __in  LPCTSTR lpString1,  //адрес первой строки
  __in  LPCTSTR lpString2   //адрес второй строки
);
```

Данная функция последовательно побайтно сравнивает две строки до тех пор пока либо одна из строк не закончится, либо не наткнется на различный

символ. В том случае если строки совпадают, возвращается ноль, в противном случае возвращается значение отличное от нуля.

Функция lstrcmp сравнивает строки с учетом регистра, то есть для нее символы «б» и «Б» два разных символа. Если нужно сравнить две строки без учета регистра, то используется функция lstrcmpi (A/W) из той же библиотеки:

```
    int WINAPI lstrcmpi(
  __in  LPCTSTR lpString1,  //адрес первой строки
  __in  LPCTSTR lpString2   //адрес второй строки
);
```

Эта функция идентична lstrcmp. Единственное отличие состоит в том. что при использовании первой регистр не учитывается, то есть символы «б» и «Б» считаются одинаковыми.

Копирование строки

Для того чтобы скопировать строку в какой-либо буфер используется функция lstrcpy (A/W) из библиотеки kernel32.dll. Вот как она выглядит:

```
LPTSTR WINAPI lstrcpy(
  __out  LPTSTR lpString1, //адрес целевого буфера (строки)
  __in   LPTSTR lpString2  //адрес копируемой строки
);
```

Если функция завершается успехом то она возвращает адрес целевого буфера, в который была скопирована строка. Если же функции не удалось сделать то, что нужно она вернет NULL.

При работе с этой функцией следует быть предельно осторожным. Так как операционная система не проверяет размер целевого буфера. То есть перед началом копирования она не убеждается в том, что размер целевого буфера достаточен для того, чтобы вместить в себя копируемую строку. Если размера буфера недостаточно для этого происходит затирание данных (а это может быть что угодно) расположенных за этим буфером, со всеми вытекающими.

Соединение строк

Для того, чтобы одну строку присоединить к концу другой используется функция lstrcat (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
LPTSTR WINAPI lstrcat(
  __inout  LPTSTR lpString1, //адрес первой строки, к которой
                             //присоединяем
  __in     LPTSTR lpString2  //адрес присоединяемой строки
);
```

При работе с этой функцией также следует быть осторожными. Важно помнить, что буфер, к которому вы присоединяете строку должен иметь

достаточный размер для того чтобы вместить две строки (та, что там уже была и ту что мы присоединили) и завершающий нуль.

Пример программы

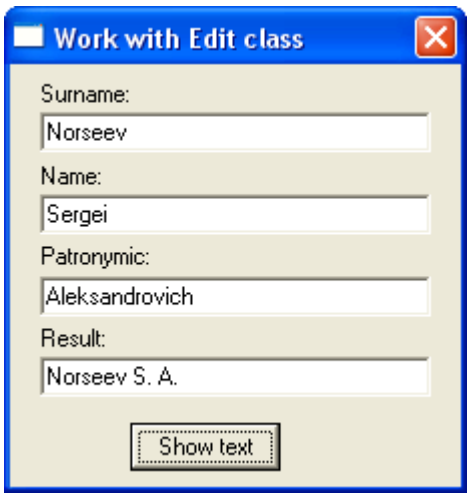
Давайте в качестве примера напишем небольшую программу, которая по введенным пользователем фамилией, имени и отчеству будет выводить строку с фамилией и инициалами этого человека. Для упрощения программы мы будем размещать обрабатываемые данные в статической памяти (хотя для этого вполне можно было использовать динамическую память). Длину фамилии ограничим 32 символами.

Хватит слов. Вот исходный код программы:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
;Идентификаторы элементов управления
ID_EDITSurname = 3
ID_EDITName = 5
ID_EDITPatronymic = 7
ID_EDITResult = 9
ID_BUTTON = 0Ah

section '.code' code readable executable
  start:invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, 1, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0

        proc DialogProc hwnddlg, msg, wparam, lparam
             xor eax, eax
             cmp [msg], WM_CLOSE
             je FreeDialog
             cmp [msg], WM_COMMAND
             jne ExitProc
             mov eax, BN_CLICKED
             shl eax, 10h
             add eax, ID_BUTTON
             cmp [wparam], eax
             jne ExitProc
             ;Читаем фамилию и заносим ее в  ResultString
             invoke    GetDlgItemText,    [hwnddlg],    ID_EDITSurname,
ResultString, 20h
             ;Присоединяем к ней символ пробела
             mov eax, PointTabString
             inc eax
             invoke lstrcat, ResultString, eax
             ;Читаем первую букву имени
             invoke GetDlgItemText, [hwnddlg], ID_EDITName, Intermediate,
2
             ;Присоединяем ее к результирующей строке
             invoke lstrcat, ResultString, Intermediate
             ;Присоединяем к результату точку и пробел
             invoke lstrcat, ResultString, PointTabString
             ;Читаем первую букву отчества
             invoke    GetDlgItemText,    [hwnddlg],    ID_EDITPatronymic,
Intermediate, 2
             ;Присоединяем ее к результату
             invoke lstrcat, ResultString, Intermediate
             ;Присоединяем к результату точку и пробел
```

```
                invoke lstrcat, ResultString, PointTabString
                ;Выводим результат пользователю
                invoke     SetDlgItemText,     [hwnddlg],     ID_EDITResult,
ResultString
                ret
      FreeDialog:
                invoke EndDialog, [hwnddlg], 0
      ExitProc:  ret
            endp

    section '.bss' readable writeable
    PointTabString db '. ', 00h
    Intermediate db  00h, 00h
    ResultString rb 28h


    section '.idata' import data readable writeable
            library kernel, 'KERNEL32.DLL' ,\
                    user  , 'USER32.DLL'

            import kernel,\
                   GetModuleHandle, 'GetModuleHandleA',\
                   ExitProcess    , 'ExitProcess'     ,\
                   lstrcat        , 'lstrcatA'

            import user,\
                   DialogBoxParam, 'DialogBoxParamA',\
                   EndDialog     , 'EndDialog'      ,\
                   GetDlgItemText, 'GetDlgItemTextA',\
                   SetDlgItemText, 'SetDlgItemTextA'

    section '.rsrc' resource data readable
            directory RT_DIALOG, dialogs
            resource dialogs,\
                     1, LANG_NEUTRAL, WorkWithEdit
            dialog    WorkWithEdit,    'Strings',    0,    0,    150,    130,
WS_CAPTION+WS_SYSMENU+DS_CENTER
                    dialogitem 'STATIC', 'Surname: '    , 2, 10,  5, 100, 12,
WS_VISIBLE
                    dialogitem 'EDIT'   , ''             , ID_EDITSurname, 10,
15, 130, 12, WS_VISIBLE+WS_BORDER+WS_TABSTOP
                    dialogitem 'STATIC', 'Name: '       , 4, 10, 30, 100, 12,
WS_VISIBLE
                    dialogitem 'EDIT'   , ''            , ID_EDITName, 10, 40,
130, 12, WS_VISIBLE+WS_BORDER+WS_TABSTOP
                    dialogitem 'STATIC', 'Patronymic: ', 6, 10, 55, 100, 12,
WS_VISIBLE
                    dialogitem 'EDIT'   , ''             , ID_EDITPatronymic,
10, 65, 130, 12, WS_VISIBLE+WS_BORDER+WS_TABSTOP
                    dialogitem 'STATIC', 'Result: '     , 8, 10, 80, 100, 12,
WS_VISIBLE
                    dialogitem 'EDIT'   , ''             , ID_EDITResult, 10,
90, 130, 12, WS_VISIBLE+WS_BORDER
                    dialogitem 'BUTTON', 'Show text'    , ID_BUTTON, 40, 110,
50, 15, WS_VISIBLE+WS_TABSTOP
            enddialog
```

Результат работы этой программы представлен на рисунке ниже:



Unicode строки

Выше мы говорили, что каждый символ строки представляет собой один байт. Это значит что всего доступно 256 символов. Да, для латиницы и кириллицы этого вполне достаточно, но ведь существуют и другие языки, для которых этого катастрофически недостаточно. Взять, к примеру, японские иероглифы.

Именно для расширения количества допустимых символов были придуманы двухбайтовые строки. В них каждый символ кодируется не одним, а двумя байтами (соответственно признаком конца строки служит на байт 00h, а слово 0000h). Это позволяет увеличить число возможных символов с 256 до 65 536, что достаточно для работы с любым современным языком.

Unicode – это стандарт кодирования символов, поддерживаемый специальным консорциумом, в состав которого вошли такие гиганты как: Apple, Hewlett-Packard, IBM, Microsoft, Oracle и многие другие организации. Данный стандарт определяет соответствие между символом и его кодом (точно так же как ASCII, только теперь символ кодируется не одним, а двумя байтами).

Зачем нам использовать Unicode, когда символов ASCII нам хватает, да и потом Unicode-строки занимают в два раза больше памяти по сравнению с теми же ANSI-строками (которыми мы пользовались все это время)? По двум причинам. Во-первых, работа с Unicode значительно упрощает локализацию и разработку многоязыкового приложения. Да, знаю, что вы не собираетесь со своим творением покорять китайский рынок. Но все-таки лучше подстраховаться: чем черт не шутит. Во-вторых, ОС Windows, начиная с Windows 2000, построена так, что Unicode-строки ей ближе, «роднее». Объясню в чем тут дело.

Как вы уже знаете большинство все функции Windows, работающие со строками имеют две реализации (для работы с ANSI и Unicode строками). На самом деле это не так (каюсь, в начале книги врал). В «новых»[1] версиях

[1] Под «новыми» я подразумеваю операционные системы Windows 2000 и старше.

Windows функции работы с ANSI строками представляют собой не что иное, как «обертки» вокруг соответствующих функций для работы с Unicode строками. Поясню на примере. Допустим, ваша программа вызывает функцию MessageBoxA, передавая ей в качестве параметра ANSI-строку. Данная функция (MessageBoxA) конвертирует полученную строку в формат Unicode и вызывает MessageBoxW, передавай ей в качестве параметра конвертированную строку. Если ваша программа изначально работает с Unicode-строками, вы можете избежать лишних преобразований, сразу вызывая функцию MessageBoxW.

Это в «новых» операционных системах. А вот в старых (Windows 98 и младше) поддержка Unicode реализована крайне плохо. Поэтому при работе с Unicode будьте готовы к тому, что многие ваши приложения не будут корректно работать в этих операционных системах.

Flatasm поддерживает работу с Unicode-строками. Для задания такой строки нужно перед ее описанием указать префикс «du». И Flatasm будет обрабатывать ее как Unicode-строку.

Ниже приводится исходный текст программы, работающей с Unicode-строкой.

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.text' code readable executable
start:
        invoke MessageBox, HWND_DESKTOP, _HelloWorld, _HelloWorld, MB_OK
        invoke ExitProcess, 0

section '.data' data readable
;Выводимая Unicode-строка
_HelloWorld du 'Hello World!', 0

section '.idata' data import readable writeable
library kernel, 'KERNEL32.DLL',\
        user  , 'USER32.DLL'
import kernel,\
       ExitProcess, 'ExitProcess'

import user,\
       MessageBox, 'MessageBoxW'
```

Данная программа просто выводит приветственное сообщение.

Как можно убедиться из примера, работа с Unicode-строками не сложнее чем с привычными ANSI-строками.

Функции конвертирования

Здесь мы поговорим о том, как можно ANSI строку преобразовать в формат Unicode и наоборот. Для преобразования ANSI строки в формат Unicode используется функция MultiByteToWideChar из библиотеки kernel32.dll, вот как она выглядит:

```
int MultiByteToWideChar(
  __in   UINT CodePage,          //Идентификатор кодовой страницы
  __in   DWORD dwFlags,          //Набор флагов преобразования (обычно 0)
  __in   LPCSTR lpMultiByteStr, //Адрес конвертируемой строки
  __in   int cbMultiByte,        //Размер конвертируемой строки в байтах
  __out  LPWSTR lpWideCharStr,   //Адрес, по которому будет записана
                                 //сконвертированная строка
  __in   int cchWideChar         //Размер в символах буфера, в который
                                 //будет записана сконвертированная строка
);
```

В таблице ниже представлены основные стандартные идентификаторы кодовых страниц:

Кодовая страница	Описание
CP_ACP	Стандартная кодовая страница ANSI Windows
CP_MACCP	Текущая кодовая страница на компьютере Macintosh
CP_OEMCP	Текущая системная кодовая страница OEM (данная кодировка использовалась в эпоху MS DOS)
CP_THREAD_ACP	Кодовая страница ANSI текущего потока
CP_UTF7	Кодовая страница UTF-7 (разновидность Unicode). Предпочтительнее использовать UTF-8
CP_UTF8	Кодовая страница UTF-8 (разновидность Unicode).

Если верить Джеффри Рихтеру (а не верить ему у нас причин нет), то флаги параметра dwFlags влияют на символы с диакритическими знаками[1] и на символы, которые система не может преобразовать. Как бы то ни было для всех кодовых страниц указанных в вышеприведенной таблице он должен быть равен нулю.

Данная функция возвращает количество символов записанных в буфер по адресу lpwideCharStr или ноль, если в процессе ее выполнения произошла какая-то ошибка.

Для обратного преобразования из Unicode в ANSI используется функция WideCharToMultiByte из той же библиотеки. Вот как она выглядит:

[1] Диакритическими знаками называют различные надстрочные, подстрочные, реже внутристрочные знаки, применяемые в письменности не как самостоятельные обозначения звуков, а для изменения или уточнения значения других знаков. Пример символов с такими знаками: **ā**, **á**, **ǎ**, **à**, **a.**

```
int WideCharToMultiByte(
  __in   UINT CodePage,            //Идентификатор кодовой страницы
  __in   DWORD dwFlags,            //Набор флагов (аналогично функции
                                   //MultibyteToWideChar)
  __in   LPCWSTR lpWideCharStr,    //Адрес конвертируемой Unicode строки
  __in   int cchWideChar,          //Количество символов в Unicode строке
  __out  LPSTR lpMultiByteStr,     //Адрес, по которому будет записана
                                   //сконвертированная ANSI строка
  __in   int cbMultiByte,          //Размер в байтах буфера, на который
                                   //указывает параметр lpMultiByteStr
  __in   LPCSTR lpDefaultChar,
  __out  LPBOOL lpUsedDefaultChar
);
```

lpDefaultChar – адрес символа, который будет подставлен вместо символов, преобразовать которые не удалось. Для кодовых страниц UTF-7 и UTF-8 этот параметр должен быть установлен в NULL. В противном случае функция не сможет выполнить свою работу и вернет ошибку.

lpUsedDefaultChar – адрес флага, который будет установлен в TRUE если будет использован символ, на который указывает параметр lpDefaultChar, то есть если попадется символ, который не удастся преобразовать. Для кодовых страниц UTF-7 и UTF-8 данный параметр должен быть установлен в значение NULL, в противном случае функция вернет ошибку.

Ниже приводится полный исходный код программы, демонстрирующий работу с этими функциями:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
SizeSymbols = 13 ;Размер строки в символах (включая завершающий нуль)
SizeBytesANSI = SizeSymbols ;Размер ANSI строки в байтах
SizeBytesUnicode =SizeBytesANSI*2 ;Размер Unicode строки в байтах

section '.text' code readable executable
start:
        ;Получаем дескриптор системной кучи процесса
        invoke GetProcessHeap
        ;Сохраняем его
        mov [_hHeap], eax
        ;Выделяем память под ANSI строку
        invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, SizeBytesANSI
        ;Сохраняем адрес выделенного буфера
        mov [_lpBuferANSI], eax
        ;Конвертируем Unicode строку в ANSI строку
        invoke WideCharToMultiByte, CP_UTF8, 0, _HelloWorld, SizeSymbols,
eax, SizeBytesANSI, NULL, NULL
        ;Выделяем память под Unicode строку
        invoke HeapAlloc, [_hHeap],HEAP_ZERO_MEMORY,SizeBytesUnicode
        ;Сохраняем адрес выделенного буфера
        mov [_lpBuferUnicode], eax
        ;Конвертируем ANSI строку обратно в Unicode строку
        invoke    MultiByteToWideChar,    CP_ACP,    0,    [_lpBuferANSI],
SizeBytesANSI, [_lpBuferUnicode], SizeSymbols
        ;Освобождаем память, которую мы выделяли под ANSI строку
        invoke HeapFree, [_hHeap], 0, [_lpBuferANSI]
        mov [_lpBuferANSI], NULL
        ;Выводим сообщение с полученной Unicode строкой
```

```
            invoke      MessageBox,      HWND_DESKTOP,      [_lpBuferUnicode],
[_lpBuferUnicode], MB_OK
            ;Освобождаем память, которую мы выделяли под Unicode строку
            invoke HeapFree, [_hHeap], 0, [_lpBuferUnicode]
            mov [_lpBuferUnicode], NULL
            mov [_hHeap], NULL
            invoke ExitProcess, 0

    section '.data' data readable
    ;Исходная Unicode строка, над которой мы и будем издеваться
    _HelloWorld du 'Hello World!', 0

    section '.rdata' data readable writeable
    _hHeap dd NULL          //Дескриптор кучи
    _lpBuferANSI dd NULL    //Адрес буфера под ANSI строку
    _lpBuferUnicode dd NULL //Адрес буфера под Unicode строку

    section '.idata' data import readable writeable
    library kernel, 'KERNEL32.DLL',\
            user  , 'USER32.DLL'

    import kernel,\
           GetProcessHeap, 'GetProcessHeap',\
           HeapAlloc     , 'HeapAlloc'    ,\
           HeapFree      , 'HeapFree',\
           WideCharToMultiByte, 'WideCharToMultiByte',\
           MultiByteToWideChar, 'MultiByteToWideChar',\
           ExitProcess   , 'ExitProcess'

    import user,\
           MessageBox, 'MessageBoxW'
```

В данной программе происходит следующее: исходную Unicode строку конвертируют в ANSI, затем, полученную ANSI строку конвертируют обратно в Unicode и, полученную таким образом Unicode строку через MessageBoxW выводят пользователю. При этом для хранения сконвертированных строк используется динамическая память. Результат работы программы представлен на рисунке ниже:

Глава 16. Работа с файлами
Создание и открытие файла

Открытие и создание осуществляется одной функцией CreateFile (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
HANDLE WINAPI CreateFile(
  __in      LPCTSTR lpFileName,     //Адрес строки с именем файла
  __in      DWORD dwDesiredAccess,  //Способ доступа
  __in      DWORD dwShareMode,      //Режимы совместного использования
  __in_opt  LPSECURITY_ATTRIBUTES lpSecurityAttributes, //Атрибуты защиты
  __in      DWORD dwCreationDisposition, //Создание или открытие файла
  __in      DWORD dwFlagsAndAttributes,  //Флаги и атрибуты
  __in_opt  HANDLE hTemplateFile         //Дескриптор файла шаблона
);
```

Выглядит довольно сложно. Давайте разбираться.

В параметре lpFileName указывается адрес строки с полным наименованием открываемого (создаваемого) файла. Если в ней указано сокращенное наименование, то файл с таким наименованием ищется в текущей директории (из которой запущено приложение).

Параметр dwDesiredAccess задает, с какими правами (с какой целью) мы открываем (создаем файл). Оно может принимать либо одно из трех значений, либо их произвольное сочетание. Допустимые значения представлены в таблице:

Значение	Описание
0	Сам файл не открывается, мы можем только менять его атрибуты
GENERIC_READ	Файл открывается для чтения
GENERIC_WRITE	Мы можем писать в файл, но не читать

Параметр dwShareMode определяет, могут ли другие программы одновременно с нами работать с этим файлом. В таблице ниже возможные значения этого параметра.

Значение	Описание
0	Программы не могут работать с этим файлом до тех пор пока мы не закроем его
FILE_SHARE_READ	Другие программы могут читать этот файл
FILE_SHARE_WRITE	Другие программы могут писать в этот файл

Параметр lpSecurityAttributes используется для настройки прав доступа к файлу для разных пользователей и групп пользователей. Поскольку вопросов разграничения прав в системах Windows мы касаться не будем, то этот параметр будем задавать равным NULL, то есть назначить права по умолчанию.

Параметр dwCreationDisposition задает действие, производимое над файлом (создание или удаление). В таблице ниже представлены допустимые действия:

Значение	Описание
CREATE_NEW	Создать новый файл, если файл с заданным именем уже существует, то функция заканчивается неудачей
CREATE_ALWAYS	Создать новый файл, если файл с таким именем уже существует, то он уничтожается и создается новый файл
OPEN_EXISTING	Открыть существующий файл, если файла с заданным именем не существует, то функция заканчивается неудачей
OPEN_ALWAYS	Открыть файл, если файл с заданным именем не существует, то создается новый файл
TRUNCATE_EXISTING	Открыть файл и уничтожить его содержимое, если файл с таким именем не существует, то функция завершается неудачей.

Параметр dwFlagsAndAttributes задает атрибуты и флаги открываемого (создаваемого) файла. В таблице ниже представлены основные атрибуты файла:

Атрибут	Описание
FILE_ATTRIBUTE_ARCHIVE	Файл является архивом
FILE_ATTRIBUTE_ENCRYPTED	Зашифрованный файл
FILE_ATTRIBUTE_HIDDEN	Скрытый файл
FILE_ATTRIBUTE_NORMAL	Обычный файл
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	Содержимое файла не индексировано
FILE_ATTRIBUTE_READONLY	Файл можно только читать
FILE_ATTRIBUTE_SYSTEM	Системный файл, используемый операционной системой
FILE_ATTRIBUTE_TEMPORARY	Временный файл

В таблице ниже представлены основные флаги, сочетание которых также может быть установлено в параметре dwFlagsAndAttributes.

Флаг	Описание
FILE_FLAG_WRITE_THROUGH	Запись данных осуществлять непосредственно на диск.
FILE_FLAG_OVERLAPPED	Используется асинхронное[1] выполнение операций чтения и записи.
FILE_FLAG_NO_BUFFERING	Не использовать буферизацию при доступе к файлу, то есть читать файл напрямую с диска
FILE_FLAG_RANDOM_ACCESS	Программа предполагает выбирать записи из файла случайным образом
FILE_FLAG_SEQUENTIAL_SCAN	Программа предполагает читать файл последовательно
FILE_FLAG_DELETE_ON_CLOSE	Файл должен быть удален по закрытии всех дескрипторов этого файла

Параметр hTemplateFile представляет дескриптор файла, атрибуты которого должны быть переданы создаваемому файлу.

Уф… С параметрами вроде как разобрались.

Данная функция возвращает дескриптор открытого (созданного) файла или INVALID_HANDLE_VALUE в случае ошибки. Для закрытия файла используется функция CloseHandle из библиотеки kernel32.dll. Вот как она выглядит:

```
BOOL WINAPI CloseHandle(
  __in  HANDLE hObject //Дескриптор открытого файла
);
```

Необходимо отметить, что данная функция позволяет открывать не только файлы, но и устройства. Ниже приводится исходный код программы, открывающей лоток CD-ROMa (на моем компьютере он имеет букву E).

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
IOCTL_STORAGE_EJECT_MEDIA = 74808h
section '.text' code readable executable
start:
      ;«Открываем» драйвер привода
      invoke  CreateFile,_Driver,  GENERIC_READ,  FILE_SHARE_READ,  NULL,
OPEN_EXISTING, 0, NULL
      push eax
      ;Убеждаемся в том, что мы его открыли
      cmp eax, INVALID_HANDLE_VALUE
```

[1] При синхронном вводе/выводе программа дожидается, когда требуемая операция будет выполнена и только потом продолжает исполнение. При асинхронном вводе/выводе программа продолжает исполнение, не дожидаясь окончания выполнения затребованной операции.

```
        je FailtOpen
        ;CD-ROM, откройся
        invoke  DeviceIOControl,eax,  IOCTL_STORAGE_EJECT_MEDIA,  NULL,  0,
NULL, 0, _r, NULL
        pop eax
        ;«Закрываем» драйвер привода
        invoke CloseHandle, eax
        jmp Exit
    FailtOpen:
        invoke MessageBox, HWND_DESKTOP, _FailtOpen, NULL, MB_ICONERROR
    Exit:
        invoke ExitProcess, 0

    section '.data' data readable writeable
    _Driver du '\\.\E:', 0
    _FailtOpen du 'CD-ROM not find', 0
    _r dd 0

    section '.idata' data import readable
    library kernel, 'kernel32.dll',\
            user  , 'user32.dll'

    import kernel,\
           CreateFile     , 'CreateFileW',\
           CloseHandle    , 'CloseHandle',\
           DeviceIOControl, 'DeviceIoControl',\
           ExitProcess    , 'ExitProcess'
    import user,\
           MessageBox     , 'MessageBoxW'
```

В данной программе мы взаимодействовали с приводом CD-ROM с помощью функции DeviceIoControl, которая позволяет отправить соответствующему драйверу какую-либо инструкцию, определяемую специальным IOCTL кодом.

Чтение из файла, запись в файл

Для чтения из файла (с любым расширением) используется функция ReadFile из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI ReadFile(
  __in         HANDLE hFile,      //Дескриптор файла, из которого читаем
  __out        LPVOID lpBuffer,   //Адрес буфера, в который будут
                                  //записаны прочитанные данные
  __in         DWORD nNumberOfBytesToRead, //Максимальное количество
                                           //читаемых байт
  __out_opt    LPDWORD lpNumberOfBytesRead,//Адрес, по которому будет
                                  //записано число фактически прочитанных байт
  __inout_opt  LPOVERLAPPED lpOverlapped //Используется при асинхронном
                                         //чтении
);
```

В случае успеха данная функция возвращает ненулевое значение (TRUE), в случае же неудачи она возвращает ноль.

Для записи данных в файл используется функция WriteFile из той же библиотеки. Вот как она выглядит:

```
BOOL WINAPI WriteFile(
  __in         HANDLE hFile,      //Дескриптор файла, в который мы пишем
  __in         LPCVOID lpBuffer,  //Адрес буфера с записываемыми данными
  __in         DWORD nNumberOfBytesToWrite, //Количество байт, которые
                                            //следует записать в файл
  __out_opt    LPDWORD lpNumberOfBytesWritten, //Адрес, по которому будет
                                               //записано фактическое
                                              //количество записанных байтов
  __inout_opt  LPOVERLAPPED lpOverlapped //Используется при асинхронной
                                         //записи
);
```

В качестве примера рассмотрим небольшую программу. В ней при открытии диалогового окна содержимое файла MyFile.txt выводится в текстовое поле ввода. При закрытии диалогового окна содержимое этого поля записывается в файл. Исходный код этой программы с комментариями приведен ниже (проверки на успешность выполнения для упрощения понимания опущены).

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'

ID_MainWindow = 1 ;Идентификатор главного окна
ID_EDIT = 2       ;Идентификатор поля ввода

section '.text' code readable writeable
start:
      invoke GetModuleHandle, 0
      invoke    DialogBoxParam,    eax,    ID_MainWindow,    HWND_DESKTOP,
DialogProc, 0
      invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
     cmp [msg], WM_CLOSE
     je WMClose

     cmp [msg], WM_INITDIALOG
     je InitDialog

     jmp ExitProc

InitDialog:
     ;При получении сообщения WM_INITDIALOG читаем файл и выводим его
     ;содержимое в поле ввода

     ;Получаем дескриптор системной кучи процесса и сохраняем его
     invoke GetProcessHeap
     mov [_hHeap], eax

     ;Открываем файл и сохраняем его дескриптор
     invoke  CreateFile,  _FileName,  GENERIC_READ,  0,  NULL,  OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL
     mov [_hFile], eax
     ;Определяем размер файла и убеждаемся в том, что в нем что-то есть
     invoke GetFileSize, eax, _HiSizeFile
     cmp eax, 0
     je CloseFile
```

```
        push eax
        add eax, 2
        ;Выделяем память под содержимое файла и сохраняем адрес буфера
        invoke HeapAlloc, [_hHeap], HEAP_ZERO_MEMORY, eax
        mov [_lpBuffer], eax
        pop eax
        ;Читаем содержимое файла
        invoke ReadFile, [_hFile],[_lpBuffer], eax, _r, NULL
        ;Выводим прочитанное в поле ввода
        invoke SetDlgItemText, [hwnddlg], ID_EDIT, [_lpBuffer]
        ;Освобождаем память
        invoke HeapFree, [_hHeap], 0, [_lpBuffer]
        mov [_lpBuffer], NULL
    CloseFile:
        ;Закрываем файл и выходим из диалоговой процедуры
        invoke CloseHandle, [_hFile]
        mov [_hFile], NULL
        jmp ExitProc


    WMClose:
        ;При получении сообщения WM_CLOSE записываем содержимое поля ввода в
        ;файл

        ;Открываем файл и сохраняем его дескриптор
        invoke CreateFile, _FileName, GENERIC_WRITE, 0, NULL, OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL
        mov [_hFile], eax
        ;Определяем дескриптор окна поля ввода
        invoke GetDlgItem, [hwnddlg], ID_EDIT
        ;Определяем длину текста в поле ввода
        invoke GetWindowTextLength, eax
        ;Длину текста умножаем на два: получаем длину текста в байтах для
        ;Unicode строк. Умножение осуществляем сдвигом. Сохраняем размер в
        ;байтах в стеке
        shl eax, 1
        push eax
        ;Выделяем необходимую память и сохраняем адрес буфера
        invoke HeapAlloc, [_hHeap], HEAP_ZERO_MEMORY, eax
        mov [_lpBuffer], eax
        ;Вновь пересчитываем размер строки к символам
        mov eax, [esp]
        shr eax, 1
        ;Читаем данные из текстового поля ввода
        invoke GetDlgItemText, [hwnddlg], ID_EDIT, [_lpBuffer], eax
        pop eax
        ;Записываем их в файл
        invoke WriteFile, [_hFile], [_lpBuffer], eax, _r, NULL
        ;Освобождаем память
        invoke HeapFree, [_hHeap],0, [_lpBuffer]
        mov [_lpBuffer], NULL
        ;Закрываем файл и освобождаем диалоговое окно
        invoke CloseHandle, [_hFile]
        mov [_hFile], NULL

        invoke EndDialog, [hwnddlg], 0

    ExitProc:
        xor eax, eax
        ret
    endp
    section '.data' data readable
    _FileName du 'MyFile.txt', 0
```

```
    section '.rdata'data readable writeable
    _hHeap dd NULL      ;Дескриптор системной кучи
    _hFile dd NULL      ;Дескриптор файла
    _lpBuffer dd NULL   ;Адрес буфера в динамической памяти
    _HiSizeFile dd 00   ;Сюда будет записываться старшее двойное слово размера
                        ;файла
    _r dd 00            ;Сюда будет записываться фактическое число прочитанных
                        ;(записанных) байтов

    section '.idata' data import readable writeable
    library kernel, 'kernel32.dll' ,\
            user  , 'user32.dll'

    import kernel,\
           CreateFile     , 'CreateFileW',\
           GetFileSize    , 'GetFileSize',\
           ReadFile       , 'ReadFile'   ,\
           WriteFile      , 'WriteFile'  ,\
           CloseHandle    , 'CloseHandle',\
           GetProcessHeap , 'GetProcessHeap',\
           HeapAlloc      , 'HeapAlloc',\
           HeapFree       , 'HeapFree' ,\
           GetModuleHandle, 'GetModuleHandleW',\
           ExitProcess    , 'ExitProcess'

    import user,\
           DialogBoxParam, 'DialogBoxParamW',\
           GetDlgItem    , 'GetDlgItem'     ,\
           GetWindowTextLength, 'GetWindowTextLengthW',\
           EndDialog     , 'EndDialog'      ,\
           MessageBox    , 'MessageBoxW'    ,\
           SetDlgItemText, 'SetDlgItemTextW',\
           GetDlgItemText, 'GetDlgItemTextW'

    section '.rsrc' resource data readable
    directory RT_DIALOG, dialogs

    resource dialogs,\
           ID_MainWindow, LANG_NEUTRAL, MainWindow

    dialog    MainWindow,    'Read/Write    file',    0,    0,    300,    300,
WS_CAPTION+WS_SYSMENU+DS_CENTER
           dialogitem    'EDIT',    '',    ID_EDIT,    0,    0,    300,    300,
WS_VISIBLE+ES_MULTILINE+ES_WANTRETURN
    enddialog
```

Для определения размера файла используется функция GetFileSize из библиотеки kernel32.dll. Вот ее прототип:

```
DWORD WINAPI GetFileSize(
  __in       HANDLE hFile,           //Дескриптор файла
  __out_opt  LPDWORD lpFileSizeHigh  //Адрес по которому следует записать
                                     //старшее двойное слово размера файла
);
```

В случае успешного выполнения данная функция возвращает младшее двойное слово размера файла (старшее записывается по адресу указанному в параметре lpFileSizeHigh). В случае же ошибки она возвращает -1 (или FFFFFFFFh в шестнадцатеричном виде).

Навигация в файле

В примере выше мы читали и перезаписывали файл целиком. Но иногда требуется прочитать или перезаписать не весь фал, а какую-то его часть. Причем эта часть может находиться в начале, в середине или даже в конце файла. Функции ReadFile и WriteFile позволяют задать размер читаемых или записываемых данных, но не позволяют указать: а откуда эти самые данные начинаются в файле. Да, можно конечно прочитать весь файл, изменить в буфере нужные данные, а потом целиком перезаписать весь файл. Но, во-первых, это не слишком удобно, а во-вторых, ведет к лишнему перерасходу памяти (нам придется резервировать память под данные, которые нам не нужны).

Тут нам на выручку приходит указатель позиции файла. Что это такое? Указатель позиции файла как раз определяет начало читаемого (записываемого) буфера в файл. По своей сути он похож на курсор при редактировании текста и выполняет приблизительно ту же функцию. Для того чтобы изменить позицию этого указателя используется функция. SetFilePoint из библиотеки kernel вот ее прототип:

```
DWORD SetFilePointer(
  HANDLE hFile,                //Дескриптор файла
  LONG lDistanceToMove,        //младшее двойное слово величины на которую
                               //следует переместить указатель
  PLONG lpDistanceToMoveHigh, //Адрес, по которому записан старшее
                               //двойное слово величины, на которую
                               //следует переместить указатель
  DWORD dwMoveMethod           //Откуда отсчитывать перемещение
);
```

Возможные значения параметра dwMoveMethod представлены в таблице ниже:

Значение параметра	Описание
FILE_BEGIN	От начала файла
FILE_CURRENT	От текущей позиции указателя
FILE_END	От конца файла

В случае удачного завершения данная функция возвращает младшую часть новой позиции указателя файла, а по адресу, заданному параметром lpDistanceToMoveHigh, записывает старшую часть новой позиции указателя файла. При этом сам указатель отсчитывается от начала файла.

Ниже приводится исходный текст программы, читающей файл с середины.

```
format PE GUI 4.0
entry  start
include 'E:\FASM1\INCLUDE\win32a.inc'

section '.code' code readable executable
start:
        ;Открываем файл и убеждаемся в том, что мы его открыли
        invoke    CreateFile,    _FileName,    GENERIC_READ,    0,    NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
        cmp eax, INVALID_HANDLE_VALUE
        je   FileNotOpen

        mov [_hFile], eax
        ;Перемещаем указатель файла
        invoke SetFilePointer, eax, 20, _r, FILE_BEGIN
        ;Читаем файл
        invoke ReadFile, [_hFile], _Buffer, 20, _r, NULL
        ;Выводим прочитанное пользователю
        invoke MessageBox, HWND_DESKTOP, _Buffer, _Probel, MB_OK
        ;Закрываем файл
        invoke CloseHandle, [_hFile]
        mov [_hFile], NULL
        jmp Exit

FileNotOpen:
       invoke MessageBox,HWND_DESKTOP, _FileErrorOpen, NULL, MB_ICONERROR

Exit:
       invoke ExitProcess, 0

section '.data' data readable
_FileName du 'MyFile.txt', 0
_FileErrorOpen du 'File opening failt', 0
_Probel du ' ', 0

section '.rdata' data readable writeable
_hFile dd NULL
_Buffer rb 22
_r dd 00

section '.idata' data import readable writeable
library kernel, 'kernel32.dll',\
        user  , 'user32.dll'

import kernel,\
       CreateFile    , 'CreateFileW',\
       SetFilePointer, 'SetFilePointer',\
       ReadFile      , 'ReadFile',\
       CloseHandle   , 'CloseHandle',\
       ExitProcess   , 'ExitProcess'

import user,\
       MessageBox, 'MessageBoxW'
```

Операции над файлами

Удаление файла

Удалить файл можно с помощью функции DeleteFile (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI DeleteFile(
  __in  LPCTSTR lpFileName //Адрес строки с именем удаляемого файла
);
```

В случае успеха данная функция возвращает ненулевое значение, и ноль, если удалить файл по какой-то причине не удалось (например, файл с таким наименованием не найден).

Копирование файла

Скопировать файл можно функцией CopyFile (A/W) из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI CopyFile(
  __in  LPCTSTR lpExistingFileName, //Адрес строки с именем копируемого
                                    //файла
  __in  LPCTSTR lpNewFileName,      //Адрес строки с месторасположением и
                                    //наименованием скопированного файла
  __in  BOOL bFailIfExists          //Определяет что делать, если файл
                                    //заданный параметром lpNewFileName
                                    //уже существует
);
```

Если параметр bFailIfExists равен TRUE и файл с наименованием заданным в параметре lpNewFileName уже существует, то функция вернет ошибку. Если же параметр bFailIfExists равен FALSE, то существующий файл будет перезаписан.

В случае успешного копирования данная функция возвращает ненулевое значение. И ноль в случае ошибки.

Перемещение файла

Для перемещения файла используется функция MoveFile (A/W) из той же библиотеки.

```
BOOL WINAPI MoveFile(
  __in  LPCTSTR lpExistingFileName, //Адрес строки с именем перемещаемого
                                    //файла
  __in  LPCTSTR lpNewFileName       //Адрес строки с новым
                                    //месторасположением и наименованием
                                    //файла
);
```

При этом, если по указанному новому месторасположению файл уже существует, функция перемещения возвращает ошибку.

В случае успеха данная функция возвращает ненулевое значение. И ноль в том случае если произошла какая-то ошибка.

Необходимо заметить, что при использовании Unicode версии всех этих функций имя файла должно начинаться с символов «\\?\»[1].

Ниже приводится исходный код программы демонстрирующей работу с этими функциями. Программа создает в текущей директории файл, перемещает его в корень диска C, а затем удаляет его. Для упрощения проверки успешности выполнения опущены. Для того чтобы пользователь мог убедиться в успешности (не успешности) той или иной операции после каждой выводится диалоговое окно с сообщением, до закрытия которого он может проверить это с помощью проводника Windows или любого файлового менеджер. Хватит слов, смотрим код:

```
    format PE GUI
    entry start
    include 'E:\FASM1\INCLUDE\win32a.inc'

    section '.code' code readable executable
    start:
            ;Создаем файл, над которым будем издеваться
            invoke CreateFile, _FileName, GENERIC_READ, 0, NULL, CREATE_NEW,
FILE_ATTRIBUTE_NORMAL, NULL
            invoke CloseHandle, eax
            invoke  MessageBox,  HWND_DESKTOP,  _FileCreateSuccess,  _Probel,
MB_OK
            ;Перемещаем файл
            invoke MoveFile, _FileName, _FileMoveTo
            invoke MessageBox, HWND_DESKTOP, _FileMoveSuccess, _Probel, MB_OK
            ;Удаляем файл
            invoke DeleteFile, _FileMoveTo
            invoke  MessageBox,  HWND_DESKTOP,  _FileDeleteSuccess,  _Probel,
MB_OK

            invoke ExitProcess, 0

    section '.data' data readable
    ;Здесь мы используем сокращенный путь к файлу, поэтому префикс \\?\ не
    ;нужен
    _FileName du 'MyFile.dat', 0
    _FileMoveTo du '\\?\C:\MyFileMoved.dat', 0
    _FileCreateSuccess du 'File created',0
    _FileMoveSuccess du 'File moved', 0
    _FileDeleteSuccess du 'File deleted', 0
    _Probel du ' ', 0

    section '.idata' import data readable writeable
    library kernel, 'kernel32.dll',\
            user  , 'user32.dll'

    import kernel,\
           ExitProcess, 'ExitProcess',\
           CreateFile , 'CreateFileW',\
           CloseHandle, 'CloseHandle',\
           MoveFile   , 'MoveFileW'  ,\
```

[1] Данный префикс говорит о том, что путь указывает на текущий компьютер, а не на какой-то другой компьютер в сети. При использовании ANSI функций данный префикс добавляется самой при конвертировании ANSI строки (подробнее об этом см. раздел посвященный Unicode строкам)

```
        DeleteFile , 'DeleteFileW'

import user,\
        MessageBox, 'MessageBoxW'
```

Чтение и изменение атрибутов

Для того что узнать атрибуты файла используется функция GetFileAttributes (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
DWORD WINAPI GetFileAttributes(
  __in  LPCTSTR lpFileName //Адрес строки с именем файла
);
```

В случае успеха данная функция возвращает атрибуты файла, а в случае неудачи значение -1.

Сами атрибуты файла мы описывали в разделе «Создание и открытие файлов» в начале этой главы. Поэтому здесь мы их описывать не будем.

Для установки тех или иных атрибутов используется функция SetFileAttributes (A/W) из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI SetFileAttributes(
  __in  LPCTSTR lpFileName,     //Адрес строки с именем файла
  __in  DWORD dwFileAttributes  //Устанавливаемые атрибуты
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки ноль.

Следует заметить, что при использовании Unicode версии функций полное имя файла должно предваряться префиксом «\\?\».

Ниже приводится исходный код программы, которая создает файл, читает его атрибуты и делает его скрытым.

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'

section '.code' code readable executable
start:
        ;Создаем файл, над которым будем издеваться
        invoke    CreateFile,    _FileName,    GENERIC_READ,    0,    NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
        invoke CloseHandle , eax
        ;Получаем атрибуты созданного файла
        invoke GetFileAttributes, _FileName
        ;добавляем атрибут "скрытый"
        or eax, FILE_ATTRIBUTE_HIDDEN
        ;Устанавливаем новые атрибуты
        invoke SetFileAttributes, _FileName, eax
        invoke ExitProcess, 0

section '.data' data readable
_FileName du 'MyFile.dat', 0
```

```
section '.idata' data import readable writeable
library kernel, 'kernel32.dll'

import kernel,\
       CreateFile , 'CreateFileW',\
       CloseHandle, 'CloseHandle',\
       GetFileAttributes, 'GetFileAttributesW',\
       SetFileAttributes, 'SetFileAttributesW',\
       ExitProcess, 'ExitProcess'
```

Директории

Директория (она же папка, она же каталог) с точки зрения файловой системы является специальным файлом. Конкретная реализация директорий и описание хранимых в ней файлов и поддиректорий зависит от файловой системы. Операционная система Windows предоставляет нам несколько функций для работы с директориями. О них поговорим.

Создание директории

Для создания директории используется функция CreateDirectory (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI CreateDirectory(
  __in      LPCTSTR lpPathName, //Адрес строки  с наименованием
                                //создаваемой директории
  __in_opt  LPSECURITY_ATTRIBUTES lpSecurityAttributes //Арибуты защиты
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки возвращает нуль.

Удаление директории

Для удаления директории используется функция RemoveDirectory (A/W) из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI RemoveDirectory(
  __in  LPCTSTR lpPathName //Адрес строки с именем директории
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки возвращает нуль.

Перемещение директории

Для перемещения директорий используется функция MoveFile (A/W), которую мы уже рассматривали.

Необходимо заметить, что Unicode версии этих функций, как впрочем, и любых других функций, работающих с файловой системой, в началале полного пути к файлу ожидают увидеть префикс «\\?\».

Ниже приводится исходный код программы, который сначала создает, а затем удаляет директорию в той же папке, из которой запущена программа. Для того чтобы пользователь смог убедиться в том, что директория успешно создана, программа выводит сообщение об успешном создании директории.

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'

section '.code' code readable executable
start:
       ;Создаем директорию
       invoke CreateDirectory, _DirectoryName, NULL
       ;Извещаем пользователя о создании директории
       invoke MessageBox, HWND_DESKTOP, _CreationSuccess, _Probel, MB_OK
       ;Удаляем созданную ранее директорию
       invoke RemoveDirectory, _DirectoryName
       invoke ExitProcess, 0

section '.data' data readable
_DirectoryName du 'MyDirectory', 0
_CreationSuccess du 'Directory was created', 0
_Probel du ' ', 0

section '.idata' data import readable writeable
library kernel, 'kernel32.dll',\
        user  , 'user32.dll'

import kernel,\
       CreateDirectory, 'CreateDirectoryW',\
       RemoveDirectory, 'RemoveDirectoryW',\
       ExitProcess    , 'ExitProcess'

import user,\
       MessageBox     , 'MessageBoxW'
```

Текущая директория

В программах рассмотренных выше мы редко указывали полный путь к файлу (директории), ограничиваясь коротким путем. Это говорило том, что требуемый файл находится в текущей директории приложения, то есть директории по умолчанию. При запуске приложения такой директорией является директория, из которой запущено это приложение. Однако в дальнейшем она может быть изменена.

Для того чтобы узнать текущую директорию приложения используется функция GetCurrentDirectory (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
DWORD WINAPI GetCurrentDirectory(
  __in   DWORD nBufferLength, //Размер буфера под строку с наименованием
                              //текущей директорией в символах
  __out  LPTSTR lpBuffer      //Адрес буфера, по которому следует
                              //записать строку с наименованием текущей
                              //директорией
);
```

Если при выполнении этой функции происходит ошибка, она возвращает нуль.

Если функция выполняется успешно, она возвращает число фактически записанных в буфер символов.

Если же в процессе выполнения этой функции окажется, что размер буфера недостаточен для того, чтобы вместить полное наименование текущей директории, то возвращается требуемый размер буфера в символах, включая завершающий нуль.

Для изменения текущей директории используется функция SetCurrentDirectory (A/W) из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI SetCurrentDirectory(
  __in  LPCTSTR lpPathName //Адрес строки с наименованием новой текущей
                           //директории
);
```

В случае успешного изменения текущей директории возвращается ненулевое значение, а в случае ошибки возвращается ноль.

Ниже приводится исходный текст программы определяющей текущую директорию и выводящую ее пользователю.

```
format PE GUI 4.0
entry start
include 'E:\FASM1\INCLUDE\win32a.inc'

section '.code' code readable executable
start:
        ;Получаем дескриптор системной кучи
        invoke GetProcessHeap
        mov [_hHeap], eax
        ;Выделяем память на один символ
        invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, 2
        mov [_lpBuffer], eax
        ;Пытаемся получить текущую директорию
        ;Функция вернет необходимый размер буфера в символах
        invoke GetCurrentDirectory, 1, eax
        mov [_SizeBufferSymbols], eax
        ;Пересчитываем размер буфера в байты
        shl eax, 1
        ;Расширяем выделенный ранее буфер до необходимого размера
        invoke HeapReAlloc, [_hHeap], HEAP_ZERO_MEMORY, [_lpBuffer], eax
        mov [_lpBuffer], eax
        ;Получаем текущую директорию
        invoke GetCurrentDirectory, [_SizeBufferSymbols], eax
        ;Выводим ее пользователю
        invoke MessageBox, HWND_DESKTOP, [_lpBuffer], _Title, MB_OK
        ;Освобождаем память
        invoke HeapFree,[_hHeap], 0, [_lpBuffer]
        mov [_lpBuffer], NULL

        invoke ExitProcess, 0

section '.rdata' data readable writeable
_hHeap dd NULL
_lpBuffer dd NULL
_SizeBufferSymbols dd 00
_Title du 'Current directory', 00
```

```
section '.idata' data import readable writeable
library kernel, 'kernel32.dll',\
        user  , 'user32.dll'

import kernel,\
       GetProcessHeap, 'GetProcessHeap',\
       HeapAlloc     , 'HeapAlloc',\
       HeapReAlloc   , 'HeapReAlloc',\
       HeapFree      , 'HeapFree',\
       GetCurrentDirectory, 'GetCurrentDirectoryW',\
       ExitProcess   , 'ExitProcess'

import user,\
       MessageBox, 'MessageBoxW'
```

Результат работы программы представлен на рисунке ниже:

Окно открытия файла

В своих программах вы можете использовать стандартное диалоговое окно «Открытие файла». Для его создания и отображения пользователю предусмотрена функция GetOpenFileName (A/W) из библиотеки Comdlg32.dll. Вот ее прототип:

```
BOOL WINAPI GetOpenFileName(
  __inout  LPOPENFILENAME lpofn //адрес структуры OPENFILENAME
);
```

Если пользователь выбрал какой-то файл, то возвращается ненулевое значение. Если же пользователь ничего не выбрал или произошла какая-то ошибка, то возвращается нуль.

Структура OPENFILENAME используется как для открытия файлов, так и для их сохранения. Выглядит она так:

```
typedef struct tagOFN {
  DWORD         lStructSize;        //Размер структуры в байтах
  HWND          hwndOwner;          //Дескриптор родительского окна
  HINSTANCE     hInstance;          //Дескриптор модуля, в котором
                                    //хранится шаблон окна (если он
                                    //создается из шаблона)
  LPCTSTR       lpstrFilter;        //Адрес фильтра
  LPTSTR        lpstrCustomFilter;  //Адрес пользовательского фильтра
  DWORD         nMaxCustFilter;     //Размер в байтах буфера по адресу
                                  //lpstrCustomFilter (минимум 40 символов)
  DWORD         nFilterIndex;       //Индекс выбранного по умолчанию
                                    //фильтра (lpstrFilter) если их
                                    //несколько нумерация с нуля
```

```
  LPTSTR        lpstrFile;          //Адрес строки с полем «Имя файла»
                                    //при открытии окна. Сюда же будет
                                    //записано имя выбранного файла и
                                    //путь до него
  DWORD         nMaxFile;           //Размер буфера lpstrFile в символах
  LPTSTR        lpstrFileTitle;     //Адрес буфера куда будет записано
                                    //имя выбранного файла (без пути)
  DWORD         nMaxFileTitle;      //Размер буфера lpstrFileTitle в
                                    //байтах
  LPCTSTR       lpstrInitialDir;    //Адрес строки с начальной
                                    //директорией
  LPCTSTR       lpstrTitle;         //Адрес строки для заголовка окна
  DWORD         Flags;              //Набор флагов
  WORD          nFileOffset;      //Определяет смещение от начала строки
                                  //lpstrFile до начала имени файла
  WORD          nFileExtension;   //Определяет смещение от начала строки
                                  //lpstrFile до расширения
  LPCTSTR       lpstrDefExt;      //Адрес буфера с расширением по
                                  //умолчанию без точки (будет
                                  //присоединено к файлу если
                                //пользователь явно не указал расширение)
  LPARAM        lCustData;        //Данные передающиеся перехватывающей
                                  //процедуре
  LPOFNHOOKPROC lpfnHook;         //Адрес перехватывающей процедуры
  LPCTSTR       lpTemplateName;   //Адрес строки с именем ресурса, на
                                //основе которого создается диалоговое окно
#if (_WIN32_WINNT >= 0x0500)
  void          *pvReserved;
  DWORD         dwReserved;
  DWORD         FlagsEx;
#endif
} OPENFILENAME, *LPOPENFILENAME;
```

Фильтр представляет собой буфер, в котором находится две строки. Вторая строка при этом должна заканчиваться двумя NULL символами. Первая строка задает имя фильтра (например, «Text Files»), которое отображается в поле «Тип файлов». Вторая строка задает допустимые расширения выбираемых файлов (пример: “*.txt”). Для того чтобы указать несколько допустимых расширений они должны быть разделены символом точка запятая (пример: “*.txt;*.doc;*.bak”). Для того, чтобы указать все расширения нужно написать так: “*.*”

Пользовательский фильтр имеет такое же строение, что и обычный фильтр. Единственное различие состоит в том, что пользовательский фильтр может быть изменен пользователем по своему усмотрению.

В таблице ниже представлены некоторые допустимые значения флагов, сочетанием которых является значение поля Flags.

Флаг	Описание
OFN_ALLOWMULTISELECT	Разрешает выбирать несколько файлов сразу. Если пользователь выбирает несколько файлов, то параметр lpstrFile будет содержать путь к каталогу в котором находятся выбранные файлы, за которым будет следовать перечисление имен файлов
OFN_CREATEPROMPT	Если пользователь указал файл, которого не существует, то диалог запросит разрешение создать новый файл с указанным именем. Ели пользователь выберет создание нового файла, то диалог закроется и вернет указанное имя, в противном случае диалог останется открытым
OFN_ENABLEHOOK	Разрешает применение перехватывающей процедуры
OFN_ENABLETEMPLATE	Указывает, что диалоговое окно создается из шаблона
OFN_ENABLETEMPLATEHANDLE	Указывает, что параметр hInstance определяет блок данных, который является шаблоном
OFN_FILEMUSTEXIST	Указывает, что пользователь может ввести только имена существующих файлов
OFN_LONGNAMES	Для диалогов старого образца. Определяет использование длинных имен файлов
OFN_CHANGEDIR	Запрещает изменение директории
OFN_NODEREFERENCELINKS	Возвращать путь и имя файла ярлыка. Если флаг не установлен, то диалог возвращает путь и имя файла, на который указывает ярлык.
OFN_NONETWORKBUTTON	Убирает кнопку «Сеть» диалога
OFN_OVERWRITEPROMPT	Указывает, что диалог «Сохранить как» выдаст предупреждающее сообщение, если файл уже существует. Пользователь должен подтвердить перезапись файла
OFN_PATHMUSTEXIST	Аналогично OFN_FILEMUSTEXIST

Ниже приводится исходный текст программы предоставляющей пользователю выбрать произвольный wordовский файл, а затем

открывающей этот файл. В этом примере мы полностью заполним структуру OPENFILENAME.

```
    format PE GUI 4.0
    entry start
    include 'D:\FASM1\INCLUDE\win32a.inc'

    section '.code' code readable executable
    start:
            ;Заполянем структуру OPENFILENAME
            ;Размер всей структуры 76 байт
            mov [_OpenFile.lStructSize], 76
            ;Родительского окна у нас нет
            mov [_OpenFile.hwndOwner], NULL
            ;Мы не будем использовать шаблона диалогового окна, поэтому
            ;модуль нам не нужен
            mov [_OpenFile.hInstance], NULL
            ;Фильтр
            mov [_OpenFile.lpstrFilter], _Filter
            ;Пользовательского фильтра у нас нет
            mov [_OpenFile.lpstrCustomFilter], NULL
            ;Длина пользовательского фильтра
            mov [_OpenFile.nMaxCustFilter], 0
            ;Фильтр у нас всего один, его индекс - 0
            mov [_OpenFile.nFilterIndex], 0
            ;Куда записать имя выбранного файла и путь до него
            mov [_OpenFile.lpstrFile], _lpstrFile
            ;Размер выделенного буфера 255 символа
            mov [_OpenFile.nMaxFile], 255
            ;Инициализировать поле «Файл» мы не будем
            ;Записывать имя файла отдельно не нужно
            mov [_OpenFile.lpstrFileTitle], NULL
            ;Длина выделенного буфера отдельно под имя файла(без пути)
            mov [_OpenFile.nMaxFileTitle], 0
            ;Инициализировать каталог мы не будем
            mov [_OpenFile.lpstrInitialDir], NULL
            ;Заголовок окна оставим по умолчанию
            mov [_OpenFile.lpstrTitle], NULL
            ;Указываем, что выбранный файл должен существовать
            mov [_OpenFile.Flags], OFN_FILEMUSTEXIST
            ;Эти два поля будут заполнены операционной системой
            mov [_OpenFile.nFileOffset], 0
            mov [_OpenFile.nFileExtension], 0
            ;Расширения по умолчанию у нас нет
            mov [_OpenFile.lpstrDefExt], NULL
            ;Функции перехватчика у нас нет и передавать что-либо некому
            mov [_OpenFile.lCustData], NULL
            mov [_OpenFile.lpfnHook], NULL
            ;Мы не используем шаблон диалогового окна
            mov [_OpenFile.lpTemplateName], NULL

            ;Создаем окно выбора файла
            invoke GetOpenFileName, _OpenFile
            ;Убеждаемся что пользователь что-то выбрал
            test eax, eax
            jz Exit
            ;Открываем выбранный файл
            invoke  ShellExecute,  NULL,  _Operation,  [_OpenFile.lpstrFile],
NULL, NULL, SW_SHOWNORMAL
    Exit:
            invoke ExitProcess, 0

    section '.data' data readable
```

```
;Фильтр
_Filter du 'Word files', 00, '*.doc;*.docx;*.rtf', 00, 00
_Operation du 'open', 00

section '.rdata' data readable writeable
_OpenFile OPENFILENAME
_lpstrFile rf 512

section '.idata' data import readable writeable
library kernel, 'kernel32.dll',\
        comdlg, 'comdlg32.dll',\
        shell , 'shell32.dll'

import kernel,\
       ExitProcess, 'ExitProcess'

import comdlg,\
       GetOpenFileName, 'GetOpenFileNameW'

import shell,\
       ShellExecute, 'ShellExecuteW'
```

В данном примере для открытия выбранного файла мы использовали функцию ShellExecute (A/W) из библиотеки shell32.dll. Данная функция находит приложение, чей файл мы открываем, запускает это приложение и открывает в нем выбранный нами файл. Ниже приводится прототип этой функции:

```
HINSTANCE ShellExecute(
  __in_opt  HWND hwnd,             //Дескриптор окна, которое будет
                                   //родительским по отношению  к окну
                                   //запущенного приложения
  __in_opt  LPCTSTR lpOperation,   //Адрес строки с выполняемым действием
  __in      LPCTSTR lpFile,        //Адрес строки с именем открываемого
                                   //файла
  __in_opt  LPCTSTR lpParameters,  //Если открывается файл-приложение то
                                   //адрес строки с параметрами запуска
                                   //этого приложения, иначе NULL
  __in_opt  LPCTSTR lpDirectory,   //Адрес строки с директорией
                                   //открываемого файла (если в имени
                                   //файла не указан полный путь до него)
  __in      INT nShowCmd           //Параметры открываемого окна
);
```

В таблице ниже представлены основные допустимые строки, на которые может указывать параметр lpOperation.

Строка-операция	Описание
edit	Открыть файл на редактирование
explore	Открыть в проводнике директорию, на которую указывает параметр lpFile
find	Начать поиск с директории, на которую указывает параметр lpDirectory
open	Открыть файл
print	Распечатать файл

Параметр nShowWindow задает режим отображения окна. Он имеет то же самое значение что и параметр nCmdShow в функции ShowWindow (более подробно см. в разделе «функция ShowWindow»).

Окно сохранения файла

Для создания и отображения окна «Сохранение файла» используется функция GetSaveFileName (A/W) из библиотеки comdlg32.dll. Вот ее прототип:

```
BOOL WINAPI GetSaveFileName(
  __inout  LPOPENFILENAME lpofn //Адрес структуры OPENFILENAME
);
```

Структуру OPENFILENAME мы рассматривали в предыдущем разделе. Здесь описывать ее не будем.

Если пользователь выбрал какой-то файл, то данная функция возвращает ненулевое значение. Если же пользователь нажал кнопку «Отмена» или произошла какая-то ошибка, то возвращается нуль.

Ниже приводится исходный код программы, демонстрирующей работу с данной функцией. В ней создается диалоговое окно с полем ввода. По закрытии этого окна если поле ввода не пусто, вызывается окно сохранения файла, в которое будет записано все содержимое поля ввода.

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'

ID_MainWindow = 1
ID_EDIT = 2

section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
        invoke   DialogBoxParam,   eax,   ID_MainWindow,   HWND_DESKTOP,
DialogProc, 0
        invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
        cmp [msg], WM_CLOSE
        je OnClose
        jmp ExitProc

OnClose:
        invoke GetDlgItem, [hwnddlg], ID_EDIT
        mov [_hEdit], eax
        ;Убеждаемся в том, что в поле ввода что-то есть
        invoke GetWindowTextLength, eax
        test eax, eax
        jz _EndDialog
        inc eax
        mov [_SizeTextSymbols], eax
        ;Заполняем структуру OPENFILENAME
```

```
            mov [_OpenFile.lStructSize], 76
            mov [_OpenFile.lpstrFilter], _Filter
            mov [_OpenFile.lpstrFile], _FileName
            mov [_OpenFile.nMaxFile], 255
            mov [_OpenFile.Flags], OFN_OVERWRITEPROMPT+OFN_CREATEPROMPT
            mov [_OpenFile.lpstrDefExt], _DefExt
            ;Создаем окно «Сохранение файла» и убеждаемся в том, что
            ;пользователь выбрал файл
            invoke GetSaveFileName, _OpenFile
            test eax, eax
            jz _EndDialog
            ;Получаем дескриптор системной кучи
            invoke GetProcessHeap
            mov [_hHeap], eax
            ;Пересчитываем размер текста в поле ввода из символов в байты
            mov eax, [_SizeTextSymbols]
            shl eax, 1
            mov [_SizeTextBytes], eax
           ;Выделяем небходимое для размещения этого текста количество памяти
            invoke HeapAlloc,[_hHeap], HEAP_ZERO_MEMORY, eax
            mov [_lpBuffer], eax
            ;Читаем текст из поля ввода
            invoke GetWindowText, [_hEdit], eax, [_SizeTextSymbols]
            ;Открываем выбранный пользователем файл
            invoke    CreateFile,    _FileName,GENERIC_WRITE,    0,    NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
            mov [_hFile], eax
            ;Записываем текст в файл
            invoke WriteFile, eax, [_lpBuffer], [_SizeTextBytes], _r, NULL
            ;Закрываем файл
            invoke CloseHandle, [_hFile]
            mov [_hFile], NULL
            ;Освобождаем память
            invoke HeapFree, [_hHeap], 0, [_lpBuffer]
            mov [_lpBuffer], NULL

    _EndDialog:
            invoke EndDialog, [hwnddlg], 0

    ExitProc:
            xor eax, eax
            ret
    endp

    section '.data' data readable
    _Filter du 'Text files', 00, '*.txt', 00, 00
    _DefExt du 'txt', 00

    section '.rdata' data readable writeable
    _hHeap dd NULL
    _hEdit dd NULL
    _lpBuffer dd NULL
    _SizeTextSymbols dd 00
    _SizeTextBytes dd 00
    _hFile dd NULL
    _r dd 00
    _OpenFile OPENFILENAME
    _FileName rf 512

    section '.idata' data import readable writeable
    library kernel, 'kernel32.dll',\
            user  , 'user32.dll',\
            comdlg, 'comdlg32.dll'
```

```
    import kernel,\
           GetModuleHandle, 'GetModuleHandleW',\
           GetProcessHeap , 'GetProcessHeap',\
           HeapAlloc      , 'HeapAlloc',\
           HeapFree       , 'HeapFree',\
           CreateFile     , 'CreateFileW',\
           WriteFile      , 'WriteFile',\
           CloseHandle    , 'CloseHandle',\
           ExitProcess    , 'ExitProcess'

    import user,\
           DialogBoxParam , 'DialogBoxParamW',\
           GetDlgItem     , 'GetDlgItem',\
           EndDialog      , 'EndDialog' ,\
           GetWindowTextLength, 'GetWindowTextLengthW',\
           GetWindowText  , 'GetWindowTextW'

    import comdlg,\
           GetSaveFileName, 'GetSaveFileNameW'

    section '.rsrc' resource data readable
    directory RT_DIALOG, dialogs

    resource dialogs,\
           ID_MainWindow, LANG_NEUTRAL, MainWindow

    dialog    MainWindow,    'GetSaveFileName',    0,    0,    300,    300,
WS_CAPTION+WS_SYSMENU+DS_CENTER
           dialogitem    'EDIT',    '',    ID_EDIT,    0,    0,    300,    300,
WS_VISIBLE+ES_MULTILINE+ES_WANTRETURN
    enddialog
```

Переменные окружения

Операционная система Windows имеет ряд настраиваемых переменных, именуемых переменными окружения. Эти переменные определяют парамеры текущего пользователя системы, а также некоторые параметры всей системы в целом.

Почему я рассказываю об этих переменных в главе, посвященной работе с файлами? Потому что основная часть этих переменных задает пути к наиболее известным директориям. Именно эти переменные и будут нас интересовать в первую очередь. В таблице ниже представлены некторые переменные окружения и их описания.

Переменная окружения	Описание
ALLUSERSPROFILE	Содержит размещение профиля «All Users». Пример[1]: «C:\Documents and Settings\All Users»
APPDATA	Содержит путь к каталогу, используемому по умолчанию для размещения данных приложений текущего пользователя. Пример: «C:\Documents and

[1] Этот и последующие примеры приводятся для моего домашнего компьютера с ОС Microsoft Windows XP Professional SP3.

	Settings\Сергей\Application Data»
COMMONPROGRAMFILES	Расположение каталога «Common Files». Пример: «C:\Program Files\Common Files»
COMPUTERNAME	Имя компьютера. Пример: «COMPNEW»
COMSPEC	Путь до исполняемого файла командной строки Windows. Пример: «C:\WINDOWS\system32\cmd.exe»
HOMEDRIVE	Буква диска, на котором расположен основной каталог пользователя. Пример: «C:»
HOMEPATH	Основной каталог текущего пользователя без буквы диска. Пример: «\Documents and Settings\Сергей»
LOGONSERVER	Сетевое имя компьютера. Пример: «\\COMPNEW»
NUMBER_OF_PROCESSORS	Количество процессоров (ядер) в системе
OS	Название операционной системы. Пример: «Windows_NT»
PATH	Путь, а точнее пути разделенные точкой с запятой поиска исполняемых файлов. Пример (привожу лишь часть всей строки): «C:\Program Files\Borland\Delphi7\Bin;C:\Program Files\Borland\Delphi7\Projects\Bpl\;C:\WINDOWS\system32;C:\WINDOWS\System32\Wbem;»
PATHEXT	Список расширений файлов, котрые рассматриваются операционной системой как исполняемые. Расширения разделяются символом точка запятая. Пример: «.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH»
PROCESSOR_ARCHITECTURE	Архитектура процессора. Пример: «x86».
PROCESSOR_IDENTIFIER	Описание процессора. Пример: «x86 Family 6 Model 23 Stepping 6, GenuineIntel»
PROGRAMFILES	Путь к каталогу «Program Files». Пример: «C:\Program Files»
SYSTEMDRIVE	Буква диска, на который установлена операционная система. Пример: «C:»
SYSTEMROOT	Путь к каталоу Windows. Пример: «C:\WIDOWS»
TEMP или TMP	Путь к каталогу для хранения временных файлов. Пример: «C:\DOCUME~\F942~1\LOCALS~1\Temp»
USERNAME	Имя текущего пользователя. Пример: «Сергей»
USERPROFILE	Путь к профилю текущего пользователя. Пример: «C:\Documents and Settings\Сергей»
WINDIR	Аналогично SYSTEMROOT

Для того, чтобы узнать значение какой-либо переменной окружения используется функция GetEnvironmentVariable (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
DWORD WINAPI GetEnvironmentVariable(
  __in_opt   LPCTSTR lpName,  //Адрес строки с именем переменной,
                              //значение которой мы хотим поучить (см.
                              //таблицу выше)
  __out_opt  LPTSTR lpBuffer, //Арес буфера, в котрый будет записано
                              //значение переменной
  __in       DWORD nSize      //Размер буфера lpBuffer в символах включая
                              //нулевой символ
);
```

В случае успеха данная функция вернет количество символов, записанных в целевой буфер. Если размера целевого буфера недостаточно для размещения в нем требуемого значения, функция вернет требуемый размер буфера. Если же функции не удается выполниться по какой-либо другой причине тогда она возвращает нуль.

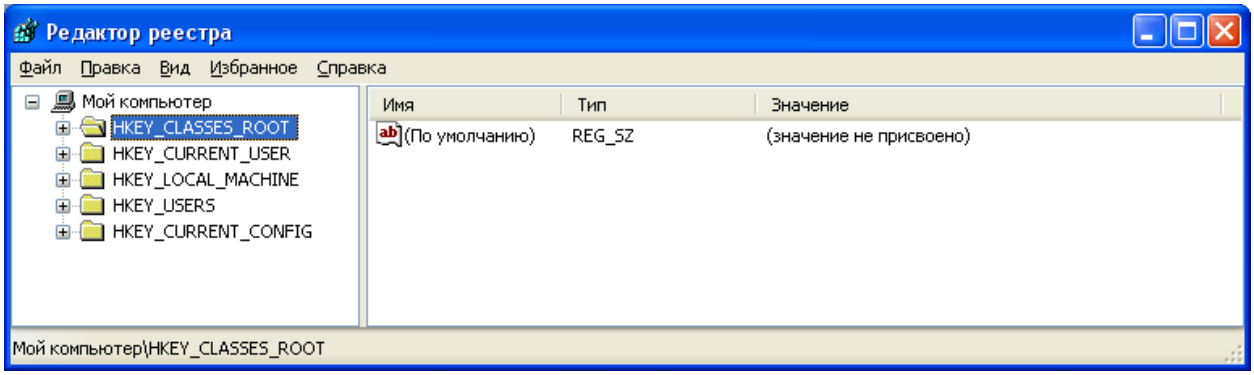
На этом теория можно сказать окончена. Теперь перейдем к практике. В качестве примера напишем небольшую программу, котая будет создавать пустой тектовый файл в каталоге «Мои документы» текущего пользователя. Вот полный исходный код этого приложения:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
;Подклбючаем этот файл так как будем работать с кирилицей
include 'D:\FASM1\INCLUDE\ENCODING\WIN1251.inc'

section '.code' code readable executable
start:
        ;Получаем дескриптор системной кучи процесса
        invoke GetProcessHeap
        mov [_hHeap], eax
        ;Выделяем память
        invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, 512
        mov [_lpBuffer], eax
        ;Получаем путь к профилю текущего пользователя
        invoke GetEnvironmentVariable, _NameVariable, eax, 233
        ;К полученому пути присоединяем «недостающую часть»
        invoke lstrcat, [_lpBuffer], _FileName
        ;Создаем файл
        invoke     CreateFile,     [_lpBuffer],     GENERIC_READ,0,NULL,
CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL, NULL
        ;Закрываем созданный файл
        invoke CloseHandle, eax
        ;Освобождаем память
        invoke HeapFree, [_hHeap], 0, [_lpBuffer]
        mov [_lpBuffer], NULL
        invoke ExitProcess, 0

section '.data' data readable
_NameVariable du 'USERPROFILE', 0
_FileName du '\Мои документы\File.txt', 0

section '.rdata' data readable writeable
```

```
_hHeap dd NULL
_lpBuffer dd NULL

section '.idata' data import readable writeable
library kernel, 'kernel32.dll'

import kernel,\
       GetProcessHeap, 'GetProcessHeap',\
       HeapAlloc     , 'HeapAlloc',\
       HeapFree      , 'HeapFree',\
       GetEnvironmentVariable, 'GetEnvironmentVariableW',\
       lstrcat       , 'lstrcatW',\
       CreateFile    , 'CreateFileW',\
       CloseHandle   , 'CloseHandle',\
       ExitProcess   , 'ExitProcess'
```

Глава 17. Реестр
Общие сведения

Реестр представляет собой сложную базу данных, в которой хранятся настройки и служебная информация операционной системы и программ, установленных на компьютере. Редактор реестра (regedit) представляет всю эту информацию в виде древовидной структуры.



На рисунке выше вы можете видеть шесть корневых разделов реестра. Структура реестра похожа на файловую систему, в которой есть директории (разделы реестра) и файлы (ключи реестра), содержащие в себе какую-то информацию. Единственное отличие ключа от раздела состоит в характере хранимых в нем данных: раздел хранит в себе подразделы и ключи, а ключ хранит в себе значение какого-то типа.

Ниже мы кратко опишем назначение корневых разделов реестра (описание всего реестра займет не одну сотню страниц убористым шрифтом)[1].

HKEY_LOCAL_MACHINE – в нем содержится вся информация, относящаяся к данному компьютеру, такая как: информация о драйверах, об установлоенном программном обеспечении, конфигурации программного обеспечения (часто наименование сокращают до HKLM).

HKEY_CLASSES_ROOT - является зеркалом раздела HKEY_LOCAL_MACHINE\Software\Classes. Содержит информацию о зарегистрированных типах файлов и объектах COM и ActiveX. Сокращенно обозначают HKCR.

HKEY_CURRENT_CONFIG – зеркало раздела «HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current». Содержит информацию о текущей конфигурации компьютера (а точнее о его периферийном оборудовании).

HKEY_USERS – Содержит информацию обо всех загруженных профилях пользователей компьютера. Сокращенно обозначают HKU.

HKEY_CURRENT_USER – копия информации из раздела HKU но, только для текущего пользователя. Сокращенно: HKCU.

---

[1] Если кому то все же интересено, то устройству реестра различных операционных систем семейства Windows (между ними существуют небольшие различия) посвящена не одна книга

Для значений ключей реестра предусмотрено несколько предопределенных типов. Они описаны в таблице ниже:

Тип	Описание
REG_BINARY	Необработанные двоичные данные, то есть простая последовательность байт
REG_DWORD	32-разрядное беззнаковое целое
REG_EXPAND_SZ	Строка переменной длины
REG_MULTI_SZ	Многострочный текст
REG_SZ	Строка фиксированной длины
REG_NONE	Данные, не имеющие определенного типа

Открытие и закрытие раздела

Для открытия того или иного раздела реестра используется функция RegOpenKeyEx (A/W) из библиотеки Advapi32.dll. Вот ее прототип:

```
LONG WINAPI RegOpenKeyEx(
  __in        HKEY hKey,         //Дескриптор уже открытого ключа, раздела
  __in_opt    LPCTSTR lpSubKey, //Адрес строки с именем открываемого
                                 //подраздела раздела hKey
  __reserved  DWORD ulOptions,   //Зарезервировано. Должно быть равно нулю
  __in        REGSAM samDesired,//Маска прав доступа к открываемому
                                 //разделу
  __out       PHKEY phkResult    //Адрес, по которому следует записать
                               //дескриптор открываемого раздела
);
```

Параметр hKey может задвать как дескриптор ранее открытого раздела, так и один из предопределенных дескрипторов: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE и HKEY_USERS. Полагаю объяснять каким разделам реестра они соотвествуют не надо.

Параметр samDesired содержит комбинацию флагов доступа к открываемому разделу. В таблице ниже представлены некоторые флаги доступа.

Флаг доступа	Описание
KEY_ALL_ACCESS	Полный доступ
KEY_CREATE_SUB_KEY	Возможность создавать подразделы
KEY_ENUMERATE_SUB_KEYS	Возможность перечислять дочерние ключи и подразделы
KEY_EXECUTE	То же что KEY_READ
KEY_NOTIFY	Получать уведомление об изменении подразделов или ключей в открываемом разделе
KEY_QUERY_VALUE	Возможность получать значений ключей в открываемом разделе

KEY_READ	Комбинация флагов KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS и KEY_NOTIFY
KEY_SET_VALUE	Возможность создавать, удалять и изменять значения ключа
KEY_WRITE	Комбинация флагов KEY_SET_VALUE и KEY_CREATE_SUB_KEY

В случае успеха функция RegOpenKeyEx возвращает значение ERROR_SUCCESS. Если же произошла какая-то ошибка, то возвращается ненулевой код ошибки.

Для открытия раздела HKEY_CURRENT_USER компания Microsoft рекомендует использовать функцию RegOpenCurrentUser из той юже библиотеки. Вот ее прототип:

```
LONG WINAPI RegOpenCurrentUser(
  __in   REGSAM samDesired,     //Маска прав доступа к разделу
  __out  PHKEY phkResult        //Адрес, по которому будет записан
                                //дескриптор открытого раздела
);
```

В случае успеха данная функция возвращает значение ERROR_SUCCESS, в противном случае она вовзращает ненулевой код ошибки.

Поскольку мы получаем дескриптор раздела, то по окончании работы с ним мы должны закрыть раздел. Делается это с помощью функции RegCloseKey из той же библиотеки. Вот ее прототип:

```
LONG WINAPI RegCloseKey(
  __in  HKEY hKey        //Дескриптор закрываемого раздела
);
```

Если данная функция завершается успешно, то она возвращает значение ERROR_SUCCESS. Если же в процессе работы произойдет какая-то ошибка, то функция вернет ненулевое значение кода ошибки.

Создание и удаление раздела

Для создания раздела используется функция RegCreateKeyEx (A/W) из библиотеки Advapi32.dll. Вот ее прототип:

```
LONG WINAPI RegCreateKeyEx(
  __in        HKEY hKey,         //Дескриптор раздела, в котором
                                 //создается подраздел
  __in        LPCTSTR lpSubKey,  //Адрес строки с именем создаваемого
                                 //раздела
  __reserved  DWORD Reserved,    //Зарезервировано, должно быть равно нулю
  __in_opt    LPTSTR lpClass,    //игнорируется системой
  __in        DWORD dwOptions,   //Опции
  __in        REGSAM samDesired, //Маска прав доступа к создаваемому
                                  //разделу
  __in_opt    LPSECURITY_ATTRIBUTES lpSecurityAttributes, //Атрибуты
                                                          //защиты
  __out       PHKEY phkResult,    //Адрес, по которому будет записан
                                  //дескриптор созданного раздела
  __out_opt   LPDWORD lpdwDisposition //Адрес, по которому будет записан
                                      //результат выполнения функции
);
```

Теперь по порядку.

Парамеры hKey, lpSubKey и samDesired здесь аналогичны соответствующим параметрам в функции RegOpenKeyEx (A/W), уже рассмотренной нами. Повторяться я не буду.

Параметр dwOptions принимает одно из значений, описанных в таблице ниже:

Значение	Описание
REG_OPTION_NON_VOLATILE	При перезагрузке системы раздел сохраняется, то есть информация о нем хранится в файле а не в памяти
REG_OPTION_VOLATILE	Информация о разделе сохраняется в памяти и удаляется при перезагрузке системы
REG_OPTION_BACKUP_RESTORE	Функция игнорирует значение параметра samDesired и пытается открыть ключ с требуемым доступом для резервного сохранения или восстановления ключа.

По адресу, указанному в параметре lpdwDisposition функцией заносится одно из двух значений:

REG_CREATED_NEW_KEY – был создан новый раздел;

REG_OPENED_EXISTING_KEY – раздел с таким именем уже существует, и он был открыт. В этом случае функция RegCreateKeyEx(A/W) ведет себя аналогично функции RegOpenKeyEx(A/W), рассмотренной нами ранее.

Если функция завершается успехом, то она возвращает значение ERROR_SUCCESS, в противном случае она возвращает отличный от нуля код ошибки.

Для удаления раздела (при этом будут удалены все ключи и подразделы удаляемого раздела) используется функция RegDeleteKey (A/W) из той же библиотеки. Вот ее прототип:

```
LONG WINAPI RegDeleteKey(
  __in  HKEY hKey,        //Дескриптор раздела более высокого по
                          //отношению к удаляемому разделу
  __in  LPCTSTR lpSubKey  //Адрес строки с именем удаляемого раздела
);
```

Если функция завершается успехом, то возвращается ERROR_SUCCESS, в противном случае возвращается ненулевой код ошибки.

Чтение и изменение значения ключа

Теперь перейдем к ключам. Как я уже говорил выше, каждый ключ реестра хранит какое-то значение определенного типа. Для того чтобы прочитать это значение используется функция RegQueryValueEx (A/W) из библиотеки Advapi32.dll. Вот ее прототип:

```
LONG WINAPI RegQueryValueEx(
  __in         HKEY hKey,            //Дескриптор открытого раздела с
                                     //нужным нам ключом
  __in_opt     LPCTSTR lpValueName,  //Адрес строки с наименованием
                                     //нужного нам ключа
  __reserved   LPDWORD lpReserved,   //Зарезервировано должно быть NULL
  __out_opt    LPDWORD lpType,       //Адрес, по которому бует записан код
                                     //типа значения ключа
  __out_opt    LPBYTE lpData,        //Адрес, по которому будут записаны
                                     //читаемые данные ключа
  __inout_opt  LPDWORD lpcbData      //Адрес, по которому хранится размер
                                     //буфера lpData в байтах
);
```

Если функция завершается успешно, она возвращает значение ERROR_SUCCESS, а по адресу lpcbData записывается количество байт, записанных в буфер lpData. Если же функция завершается с ошибкой, то она возвращает код ошибки.

Для записи данных ключа реестра (при этом, если ключа с таким наименованием не существует, то он автоматически будет создан) используется функция RegSetValueEx (A/W) из той же библиотеки. Вот как она выглядит:

```
LONG WINAPI RegSetValueEx(
  __in        HKEY hKey,           //Дескриптор открытого раздела,
                                   //значение ключа которого меняется
  __in_opt    LPCTSTR lpValueName, //Адрес строки с именем ключа
  __reserved  DWORD Reserved,      //Зарезервировано, должен быть ноль
  __in        DWORD dwType,        //код типа устанавливаемого значения
  __in_opt    const BYTE *lpData,  //Адрес буфера с записываемыми данными
  __in        DWORD cbData         //Размер записываемых данных в байтах
);
```

Типы значений ключей реестра мы уже обсуждали в разделе «Общие сведения» данной главы.

Данная функция в случае успеха возвращает значение ERROR_SUCCESS. В случае же ошибки она возвращает код ошибки.

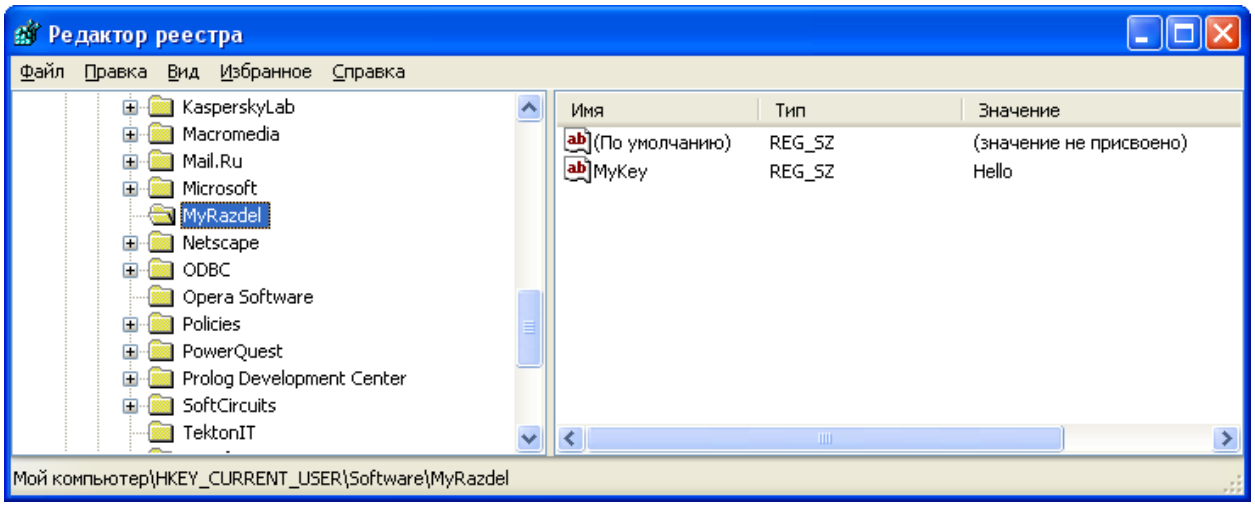
Пример программы

Будем считать, что с теорией разобрались. Теперь пора перейти к практике. В качестве примера напишем небольшую программу, демонстрирующую работу с реестром. В этой программе в реестре будет создан раздел, а в нем ключ с определенным значением. После чего этот раздел будет удален (не стоит превращать реестр в откровенную помойку). Для того чтобы пользователь смог убедиться в том, что программа работает, после каждого действия будет выводиться сообщение об этом. До тех пор пока пользователь не закроет окно с сообщением, следующее действие выполняться не будет. Смотрим код:

```
    format PE GUI 4.0
    entry start
    include 'F:\FASM1\INCLUDE\win32a.inc'
    include 'F:\FASM1\INCLUDE\ENCODING\win1251.inc'

    section '.code' code readable executable
    start:
            ;Создаем раздел реестра
            invoke  RegCreateKeyEx,HKEY_CURRENT_USER,  _RazdelName,  0,  NULL,
REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL,phkRazdel, NULL
            invoke MessageBox, HWND_DESKTOP,_RazdelCreated, _Probel, MB_OK
            ;В созданном разделе создаем ключ с определенным значением
            invoke RegSetValueEx, [phkRazdel], _KeyName, 0, REG_SZ,_Value, 12
            invoke MessageBox, HWND_DESKTOP, _ValueSet, _Probel, MB_OK
            ;Закрываем ранее созданный раздел
            invoke RegCloseKey, [phkRazdel]
            mov [phkRazdel], NULL
            ;Удаляем раздел вместе с ключом
            invoke RegDeleteKey,HKEY_CURRENT_USER, _RazdelName
            invoke MessageBox, HWND_DESKTOP, _RazdelDeleted,_Probel, MB_OK
            invoke ExitProcess, 0

    section '.data' data readable
    _RazdelName du 'Software\MyRazdel' , 0
    _KeyName du 'MyKey', 0
    _Value du 'Hello', 0

    _RazdelCreated du 'Раздел реестра успешно создан', 0
    _ValueSet du 'Ключу присвоено значение',0
    _RazdelDeleted du 'Раздел успешно удален',0

    _Probel du ' ', 0

    section '.rdata' data readable writeable
    phkRazdel dd NULL

    section '.idata' data import readable writeable
    library  kernel, 'kernel32.dll',\
             advapi, 'advapi32.dll',\
```

```
          user   , 'user32.dll'

import kernel,\
       ExitProcess, 'ExitProcess'

import advapi,\
       RegCreateKeyEx, 'RegCreateKeyExW',\
       RegCloseKey   , 'RegCloseKey',\
       RegSetValueEx , 'RegSetValueExW',\
       RegDeleteKey  , 'RegDeleteKeyW'

import user,\
       MessageBox    , 'MessageBoxW'
```

Промежуточный результат работы программы (при не закрытом окне с сообщением «Ключу присвоено значение») представлен на рисунке ниже:



Обращаю ваше внимание на то, что для того, чтобы внесенные в реестр изменения, отобразились в редакторе реестра, может понадобиться его перезапуск.

Глава 18. Файлы инициализации
Структура файлов инициализации

Файлы инициализации[1], как и реестр, предоставляют программам воможность хранить в них какие-то свои настроечные данные, и некоторые программы этим пользуются. По сути это текстовые файлы, которые могут быть отредактированы в стандартном блокноте, имеют определенную структуру и расширение ini.

Логически каждый файл инициализации разбит на несколько разделов. Каждый раздел включает в себя один или несколько параметров различного типа. Ниже, в качестве примера, приводится фрагмент содержимого файла инициализации компилятора Flatasm:

```
[Compiler]
Memory=16384
Priority=0
[Options]
SecureSelection=0
AutoBrackets=0
AutoIndent=1
SmartTabs=1
OptimalFill=1
ReviveDeadKeys=0
ConsoleCaret=1
```

Здесь: «Compiler» и «Options» (в квадратных скобках) – имена разделов.

«Memory» и «Priority» - имена параметров раздела «Compiler», а числа 16384 и 0 являются значениями этих параметров.

«SecureSelection», «AutoBrackets», «AutoIndent», … - имена параметров раздела «Options».

Параметры могут быть всего двух типов: целочисленным и строковым.

Несмотря на то, что программам рекомендуется использовать для хранения своих настроечных данных реестр, Windows API включает в себя несколько функций для работы с файлами инициализации. Некоторые из них мы и рассмотрим[2].

---

[1] Они же файлы настроек и ini-файлы.

[2] Более подробно о работе с файлами инициализации смотрите в книге Р. Саймона «Microsoft Windows API: справочник системного программиста»

Чтение параметров

Для чтения значения целочилемленного значения параметра может быть использована функция GetPrivateProfileInt(A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
UINT WINAPI GetPrivateProfileInt(
  __in  LPCTSTR lpAppName,  //Адрес строки с именем раздела, из которого
                            //считывается значение параметра
  __in  LPCTSTR lpKeyName,  //Адрес строки с именем читаемого параметра
  __in  INT nDefault,       //Значение, которое будет возвращено данной
                            //функцией если параметра с таким именем в
                            //файле инициализации нет
  __in  LPCTSTR lpFileName  //Адрес строки с именем файла инициализации,
                            //из которого читается параметр
);
```

Данная функция возвращает значение читаемого параметра, или значение nDefault в случае ошибки.

Обращаю ваше внимание на то, что при работе с файлами инициализации их не нужно самостоятельно открывать и закрывать. Все эти действия операционная система выполняет сама по мере надобности.

Для чтения строкового значения может быть использована функция GetPrivateProfileString(A/W) из той же библиотеки. Вот ее прототип:

```
DWORD WINAPI GetPrivateProfileString(
  __in   LPCTSTR lpAppName,        //Адрес строки с именем раздела
  __in   LPCTSTR lpKeyName,        //Адрес строки с именем параметра
  __in   LPCTSTR lpDefault,        //Адрес строки по умочлчанию
  __out  LPTSTR lpReturnedString, //Адрес буыера, в который будет
                                   //записана читаемая строка
  __in   DWORD nSize,              //Размер буфера под строку в символах
  __in   LPCTSTR lpFileName        //Адрес строки с именем файла
);
```

Данная функция возвращает количество символов, скопированных в буфер lpReturnedString, без учета нулевого символа. Если читаемая строка не помещается полностью в буфер lpReturnedString, то она обрезается.

Запись параметров

Для записи параметров используется функция WritePrivateProfileString(A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI WritePrivateProfileString(
  __in  LPCTSTR lpAppName, //Адрес строки с именем раздела
  __in  LPCTSTR lpKeyName, //Адрес строки с именем парметра
  __in  LPCTSTR lpString,  //Адрес записываемой строки
  __in  LPCTSTR lpFileName //Адрес строки с именем файла
);
```

Если данная функция завершается успешно, то она возвращает ненулевое значение, и ноль в случае ошибки.

Отдельной функции для записи целочисленных значений нет. Поэтому в своих программах вам придется самостоятельно приводить их к строковому виду.

Пример программы

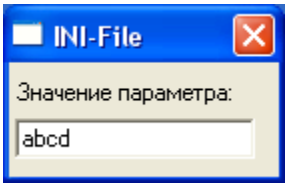
В качестве примера напишем небольшую программу. Пусть она при своем открытии читает строковый параметр из файла инициализации «file.ini» расположенного в том же каталоге, что и программа, и отображает его в поле ввода. При закрытии программа будет записывать строку из поля ввода в этот же файл в качестве нового значения параметра. Ниже приводится полный исходный текст такой программы:

```
format PE GUI 4.0
entry start
include 'D:\FASM\INCLUDE\WIN32A.INC'
include 'D:\FASM\INCLUDE\ENCODING\WIN1251.INC'

ID_DIALOG = 1
ID_STATIC = 2
ID_EDIT   = 3

section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
        invoke DialogBoxParam, eax, ID_DIALOG, HWND_DESKTOP, DialogProc,
0
        invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
    cmp [msg], WM_INITDIALOG
    je InitDialog
    cmp [msg], WM_CLOSE
    je FreeDialog
    jmp ExitProc
InitDialog:
    ;Получаем дескриптор системной кучи процесса, он нужен для работы с
    ;динамической памятью
    invoke GetProcessHeap
    mov [hHeap], eax
    ;Выделяем память под один символ
    invoke HeapAlloc, eax, HEAP_ZERO_MEMORY, 2
    mov [lpbuffer], eax
    ;Определяем количество символов в строке с текущей директорией
    ;процесса
    invoke GetCurrentDirectory, 1, eax
    ;Пересчитываем размер памяти, необходимой для хранения полного имени
    ;файла
    push eax
    shl eax, 1
    ;Добавляем размер имени файла
    add eax, 20
    ;Увеливичиваем размер буфера до необходимого размера
    invoke HeapReAlloc, [hHeap], HEAP_ZERO_MEMORY, [lpbuffer], eax
    mov [lpbuffer], eax
```

```
        pop ebx
        ;Получаем наименование текущей директории процесса
        invoke GetCurrentDirectory, ebx, eax
        ;Присоединяем к ней имя файла инициализации, прлучим его полное
        ;наименование
        invoke lstrcat, [lpbuffer], filename
        ;Читаем параметр из файла инициализации
        invoke  GetPrivateProfileString,  rasdelname,  parametername,  error,
value, 20, [lpbuffer]
        ;Выводим прочитанную строку в поле ввода
        invoke SetDlgItemText, [hwnddlg], ID_EDIT, value
        jmp ExitProc
    FreeDialog:
        ;Читаем из поля ввода измененнное значение параметра
        invoke GetDlgItemText, [hwnddlg], ID_EDIT, value, 20
        ;Записываем новое значение параметра в файл инициализации
        invoke  WritePrivateProfileString,  rasdelname,  parametername,  value,
[lpbuffer]
        ;Освобождаем память, которую мы выделяли из кучи
        invoke HeapFree, [hHeap], 0, [lpbuffer]
        mov [lpbuffer], NULL
        invoke EndDialog, [hwnddlg], 0
    ExitProc:
        xor eax, eax
        ret
    endp

    section '.data' data readable
    rasdelname du 'Razdel', 00
    parametername du 'Parameter', 00
    filename du '\file.ini', 00
    error du 'Error', 0

    section '.rdata' data readable writeable
    value rw 20    ;Буфер для хранения значения параметра
    lpbuffer dd ?  ;Адрес буфера с полным наименованием файла инициализации
    hHeap dd ?     ;Дескриптор системной кучи процесса

    section '.idata' import data readable writeable

    library kernel, 'kernel32.dll',\
            user  , 'user32.dll'

    import kernel,\
           ExitProcess, 'ExitProcess',\
           GetModuleHandle, 'GetModuleHandleW',\
           GetCurrentDirectory, 'GetCurrentDirectoryW',\
           GetPrivateProfileString, 'GetPrivateProfileStringW' ,\
           GetProcessHeap, 'GetProcessHeap' ,\
           HeapAlloc, 'HeapAlloc',\
           HeapFree, 'HeapFree',\
           HeapReAlloc, 'HeapReAlloc',\
           lstrcat, 'lstrcatW',\
           WritePrivateProfileString, 'WritePrivateProfileStringW'

    import user,\
           DialogBoxParam, 'DialogBoxParamW',\
           EndDialog, 'EndDialog',\
           GetDlgItemText, 'GetDlgItemTextW',\
           SetDlgItemText, 'SetDlgItemTextW'

    section '.rsrc' resource data readable
    directory RT_DIALOG, dialogs
```

```
    resource dialogs,\
             ID_DIALOG, LANG_NEUTRAL, maindialog

    dialog      maindialog,      'INI-File',      0,      0,      90,      35,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+DS_CENTER
        dialogitem 'Static', 'Значение параметра:', ID_STATIC, 2, 5, 80, 12,
WS_VISIBLE
        dialogitem 'Edit', '', ID_EDIT, 2, 17, 80, 12, WS_VISIBLE+WS_BORDER
    enddialog
```

На рисунке ниже представлено главное окно этого приложения:



Глава 19. Клавиатура и мышь

Мышь

Когда пользователь выполняет мышью какое-то действие: будь то простой щелчок кнопкой мыши или ее простое перемещение, приложению, которому принадлежит фокус ввода, отправляется соответствующее сообщение, которое оно может обработать. В таблице ниже представлены основные события, связанные с мышью.

Сообщение	Описание
WM_LBUTTONDOWN	Пользователь нажал на левую кнопку мыши
WM_LBUTTONDBLCLK	Двойной щелчок левой кнопкой мыши
WM_LBUTTONUP	Пользователь отпустил левую кнопку мыши[1]
WM_MBUTTONDBLCLK	Двойной щелчок центральной кнопкой мыши
WM_MBUTTONDOWN	Нажата центральная кнопка мыши
WM_MBUTTONUP	Отпущена центральная кнопка мыши
WM_MOUSEACTIVATE	Окно приложения вновь стало активным, так как по нему щелкнули мышью
WM_MOUSELEAVE	Курсор мыши вышел за пределы окна приложения
WM_MOUSEMOVE	Перемещение курсора мыши
WM_RBUTTONDBLCLK	Двойной щелчок правой кнопкой мыши
WM_RBUTTONDOWN	Нажата правая кнопка мыши
WM_RBUTTONUP	Отпущена правая кнопка мыши

Во всех этих сообщениях (кроме WM_MOUSEACTVATE и WM_MOUSELEAVE) параметр wParam задает код клавишы, которая была в нажатом состоянии во время выполнения того или иного действия мышью. В таблице ниже представлены основные коды клавиш:

Код клавишы	Описание
MK_CONTROL	Клавиша Ctrl
MK_LBUTTON	Левая кнопка мыши
MK_MBUTTON	Центральная кнопка мыши
MK_RBUTTON	Правая кнопка мыши
MK_SHIFT	Клавиша Shift

Параметр lParam задает координаты курсора мыши в момент того или иного дествия. Старшее слово задает координату по вертикали, а младшее по горизонтали. При этом начало координат располагается в левом верхнем углу клиентской области окна[2].

[1] Одинарный щелчок левой кнопкой мыши на самом деле представляет последовательность двух сообщений: WM_LBUTTONDOWN (нажали на кнопку) и WM_LBUTTONUP (отпустили кнопку)

[2] Если говорить упрощенно, то клиентская область окна представляет собой часть окна без его заголовка

В качестве примера напишем небольшую программу. В ней будет создано модальное диалоговое окно со статическим элементом управления. Причем этот элемент управления будет перемещаться по щелчку мыши на место этого щелчка. Ниже привожу полный исходный код этого приложения:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
ID_DIALOG = 1
ID_STATIC = 2
section '.code' code readable executable
start:
       invoke GetModuleHandle, 0
       invoke DialogBoxParam, eax, ID_DIALOG, HWND_DESKTOP, DialogProc, 0
       invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
     cmp [msg], WM_INITDIALOG
     je InitDialog
     cmp [msg], WM_LBUTTONDOWN
     je LButtonDown
     cmp [msg], WM_CLOSE
     je FreeDialog
     jmp ExitProc
InitDialog:
     invoke GetDlgItem, [hwnddlg], ID_STATIC
     mov [_hwndStatic], eax
     jmp ExitProc
LButtonDown:
     mov eax, [lparam]
     xor ebx, ebx
     mov bx, ax
     shr eax, 16
     invoke SetWindowPos, [_hwndStatic],0,ebx,eax,0,0, SWP_NOSIZE
     jmp ExitProc
FreeDialog:
     invoke EndDialog, [hwnddlg], 0
ExitProc:
     xor eax, eax
     ret
endp
section '.rdata' data readable writeable
_hwndStatic dd NULL

section '.idata' import data readable writeable
library kernel, 'kernel32.dll',\
        user  , 'user32.dll'

import kernel,\
       GetModuleHandle, 'GetModuleHandleA',\
       ExitProcess, 'ExitProcess'

import user,\
       DialogBoxParam, 'DialogBoxParamA',\
       EndDialog, 'EndDialog',\
       GetDlgItem, 'GetDlgItem',\
       SetWindowPos, 'SetWindowPos'

section '.rsrc' resource data readable
directory RT_DIALOG, dialogs

resource dialogs,\
```

```
            ID_DIALOG, LANG_NEUTRAL, form1

    dialog      form1,      'Mouse',      100,      100,      100,      100,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+DS_CENTER
          dialogitem 'STATIC', 'Hello', ID_STATIC, 50, 50, 30, 12, WS_VISIBLE
    enddialog
```

В этом приложении для перемещения элемента управления мы использовалди функцию SetWindowPos из библиотеки user32.dll. Подробно эту функцию мы уже рассматривали в главе «взаимодействие с окнами других приложений».

Клавиатура

При работе пользователя с клавиатурой активному приложению отправляются специальные сообщения, по которым приложение узнает: какая именно кнопка была нажата. Это позволяет приложению обрабатывать нажатия тех или иных клавиш. Основные события клавиатуры представлены в таблице ниже:

Сообщение	Описание
WM_CHAR	Представляет собой сообщение WM_KEYDOWN обработанное функцией TranslateMessage
WM_DEADCHAR	Представляет собой сообщение WM_KEYUP обработанное функцией TranslateMessage
WM_KEYDOWN	Нажата несистемная клавиша[1]
WM_KEYUP	Отпущена несистемная клавиша
WM_SYSDEADCHAR	Представляет собой сообщение WM_SYSKEYDOWN обработанное функцией TranslateMessage
WM_SYSKEYDOWN	Нажата системная клавиша
WM_SYSKEYUP	Отпущена системная клавиша
WM_UNICHAR	Эквиваленто WM_CHAR, но использует Unicode формат. Данное сообщение предназначено чтобы отправлять или пересылать символы Unicode окнам ANSI.

Во всех этих сообщениях параметр wParam задает нажатую или отпущенную клавишу. А вот каким именно образом он ее задает, зависит от сообщения, а точнее, от того вызывается функция TranslateMessage или нет. Например, параметр wParam сообщения WM_CHAR содержит в себе непосредственный код вводимого с клавиатуры символа, а тот же параметр сообщения WM_KEYDOWN содержит в себе виртуальный код нажатой

[1] Системными клавишами называются клавиши, которые нажимаются совместно с клавишей Alt. Например: Alt+Tab, Alt+F4, Alt+Shift и другие. Все остальные клавишы (нажатые без клавишы Alt) называются несистемными клавишами

клавишы. Функция TranslateMessage как раз и осуществляет преобразование виртуальных кодов клавиш в коды соотвествующих им символов.

Коды виртуальных клавиш обеспечивают независимость от аппаратуры компьютера, а точнее от используемой клавиатуры. Сами эти коды стандартизованы и постоянны. Некоторые из них представлены в таблице ниже

Виртуальный код клавишы	**Клавиша**
от 30h до 39h	Клавиши от 0 до 9 вдоль верхней части клавиатуры
от 41h до 5Ah	Клавиши от A до Z
VK_ADD	+ на цифровой части клавиатуре
VK_BACK	Backspace
VK_CAPITAL	Caps Lock
VK_CONTROL	Ctrl
VK_DECIMAL	Точка на цифровой части клавиатуры
VK_DELETE	Delete
VK_DIVIDE	/ на цифровой части клавиатуры
VK_END	End
от VK_F1 до VK_F12	Клавиши от F1 до F12
VK_HOME	Home
VK_INSERT	Insert
VK_LWIN	Левая клавиша Win
VK_MULTIPLY	* на цифровой части клавиатуры
VK_NEXT	Page Down
VK_NUMLOCK	Num Lock
от VK_NUMPAD0 до VK_NUMPAD9	Клавиши от 0 до 9 на цифровой части клавиатуры
VK_PRIOR	Page Up
VK_RIGHT	Стрелка вправо
VK_RWIN	Правая клавиша Win
VK_SCROLL	Scroll Lock
VK_SHIFT	Shift
VK_SUBTRACT	- на цифровой части клавиатуры

В качестве примера напишем небольшую программу, которая будет отлавливать нажатие трех клавиш (Ctrl, Shift и F3) и выводить на экран какая клавиша была нажата. Вот полный исходный код:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
ID_DIALOG = 1
section '.code' code readable executable
start:
       invoke GetModuleHandle, 0
       invoke DialogBoxParam, eax, ID_DIALOG, HWND_DESKTOP, DialogProc, 0
       invoke ExitProcess, 0

proc DialogProc hwnddlg, msg, wparam, lparam
     cmp [msg], WM_CLOSE
     je FreeDialog
     cmp [msg], WM_KEYDOWN
     je wmkeydown
     jmp ExitProc
wmkeydown:
     cmp [wparam], VK_CONTROL
     je vkcntrl
     cmp [wparam], VK_SHIFT
     je vkshft
     cmp [wparam], VK_F3
     je vkf3
     jmp ExitProc
vkcntrl:
     mov eax, _cntrl
     jmp shwmsg
vkshft:
     mov eax, _shft
     jmp shwmsg
vkf3:
     mov eax, _f3
shwmsg:
     invoke MessageBox, HWND_DESKTOP, eax, _prbl, MB_OK
     jmp ExitProc
FreeDialog:
     invoke EndDialog, [hwnddlg], 0
ExitProc:
     xor eax, eax
     ret
endp

section '.data' data readable
_prbl db ' ', 0
_cntrl db 'Ctrl', 0
_shft db 'Shift', 0
_f3 db 'F3', 0

section '.idata' import data readable writeable
library kernel, 'kernel32.dll',\
        user  , 'user32.dll'

import kernel,\
       GetModuleHandle, 'GetModuleHandleA',\
       ExitProcess, 'ExitProcess'

import user,\
       DialogBoxParam, 'DialogBoxParamA',\
       EndDialog, 'EndDialog',\
       MessageBox, 'MessageBoxA'

section '.rsrc' resource data readable
directory RT_DIALOG, dialogs
```

```
    resource dialogs,\
            ID_DIALOG, LANG_NEUTRAL, form1

    dialog      form1,      'Keyboard',      100,      100,      100,      100,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU+DS_CENTER
    enddialog
```

Глава 20. Разработка динамических библиотек

Динамические библиотеки очень похожи на своих «собратьев» (обычных исполняемых файлов). Для создания библиотеки в параметре format нужно указать:

```
Format PE GUI 4.0 DLL
```

Так, Flatasm поймет, что вы разрабатываете динамическую библиотеку, а не простой исполняемый файл.

Таблица экспорта

Основное назначение динамических библиотек (файлов dll) – это предоставление набора функций, которые программист может использовать в своих программах. Для описания функций, предоставляемых данной библиотекой служит специальная структура данных – таблица экспорта. Обращаю ваше внимание на то, что в этой таблице могут быть представлены не все функции (процедуры) этой библиотеки. Такие функции называются неэкспортируемыми. Они, как правило, используются для внутренних целей библиотеки.

В общем виде таблица экспорта задается следующим образом:

```
;Задаем секцию с таблицей экспорта
section '.edata' export data readable
  ;имя нашей библиотеки после ключевого слова export
  export 'MYDLL.DLL',\
         ;Перечень экспортируемых функций с их строковыми представлениями
         ShowMessage,'ShowMessage'
```

В данном примере экспортируется всего одна функция «ShowMessage»

Как видно из этого примера таблица экспорта в чем-то похожа на таблицу импорта, которую мы уже рассматривали.

Статическая и динамическая загрузка

Перед тем как программа сможет обратиться к той или иной функции библиотеки, сама библиотека должна быть загружена в память процесса. Загрузку библиотеки можно осуществить двумя путями: статически и динамически.

При статической загрузке нужная библиотека просто указывается в таблице импорта основного модуля программы. При этом библиотека загружается в память автоматически при старте программы. Именно этот способ мы и использовали раньше. Недостаток этого метода стоит в том, что библиотека всегда присутствует в памяти процесса, даже если процесс обращается к ней всего один раз в начале или конце своей работы (или вообще не обращается в результате действий пользователя).

При динамической загрузке требуемая библиотека загружается в память процесса только тогда, когда она действительно нужна, и может быть выгружена из памяти сразу после окончания работы с ней.

Загрузка библиотеки осуществляется функцией LoadLibrary (A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
HMODULE WINAPI LoadLibrary(
  __in  LPCTSTR lpFileName  //адрес строки с именем библиотеки
);
```

В случае успешной загрузки библиотеки данная функция возвращает ее дескриптор. Если же произошла какая-то ошибка, то она вернет значение NULL.

Для выгрузки библиотеки из памяти используется функция FreeLibrary из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI FreeLibrary(
  __in  HMODULE hModule  //Дескриптор выгружаемой библиотеки
);
```

В случае успеха данная функция возвращает ненулевое значение и нуль в случае ошибки.

Хорошо, с загрузкой и выгрузкой библиотек будем считать, что разобрались. Теперь резонно возникает вопрос: как вызвать ту или иную функцию, расположенную в загруженной библиотеке. Для того чтобы определить адрес нужной функции используется функция GetProcAddress из библиотеки kernel32.dll. Вот ее прототип:

```
FARPROC WINAPI GetProcAddress(
  __in  HMODULE hModule,    //Дескриптор библиотеки
  __in  LPCSTR lpProcName   //Адрес строки с именем нужной функции
);
```

В случае успеха данная функция вернет адрес искомой функции, а в случае ошибки значение NULL. Обращаю ваше внимание на то, что данная функция работает исключительно с ANSI строками.

Функция dllmain

Строго говоря, динамические библиотеки не имеют точки входа. Однако при использовании библиотеки часто требуется проинициализировать некоторые переменные библиотеки или выполнить другие действия перед тем как будет вызвана какя-либо функция из этой библиотеки. Для этих целей используется callback функция dllmain, которая вызывается операционной системой при выполнении каких-либо действий с динамической библиотекой. Вот ее прототип:

```
BOOL WINAPI DllMain(
  __in  HINSTANCE hinstDLL, //дескриптор библиотек
  __in  DWORD fdwReason,    //Причина вызова функции
  __in  LPVOID lpvReserved  //используется для внутренних целей ОС
);
```

Параметр fdwReason может принимать одно из значений представленных в таблице ниже:

Причина вызова	Описание
DLL_PROCESS_ATTACH	Библиотека была загружена в адресное пространство процесса
DLL_PROCESS_DETACH	Библиотека будет выгружена из адресного пространства процесса
DLL_THREAD_ATTACH	Процесс создал новый поток, в котором используется эта библиотека
DLL_THREAD_DETACH	Завершение работы потока

Данная функция должна вернуть TRUE, если она выполнилась без ошибок и FALSE в противном случае. Если система вызывает dllmain при загрузке библиотеки в память, а она возвращает FALSE, то система тут же вызывает dllmain еще раз, но, уже указывая в качестве причины вызова выгрузку из адресного пространства процесса, и выгружает эту библиотеку. Функция LoadLibrary (если библиотека загружалась с ее помощью) при этом возвращает значение NULL.

Адрес этой функции, если она есть (вы можете создавать библиотеки и без нее) и указывается в динамических библиотеках в качестве точки входа.

Таблица перемещаемых элементов

В PE заголовке любого PE-файла (exe, dll) есть специальное значение ImageBase, которое задает желательный базовый адрес загрузки этого файла, то есть виртуальный адрес, по которому должен быть размещен PE заголовок этого файла. Когда, при попытке загрузить в память какой-либо PE-файл, неожиданно выяснится, что его желательный базовый адрес загрузки уже занят каким-либо другим ранее загруженным файлом, то загрузчик прибегает к перемещению загружаемого файла. Что это значит? Это значит, что данный файл загружается в память так, что его фактический базовый адрес загрузки отличается от желательного. Соответственно, адреса всех процедур и переменных в этих двух случаях (когда файл загружен по желательному адресу, и когда загрузить его по этому адресу не удалось) будут различными. Резонно возникает вопрос: как это скажется на работоспособности данного приложения.

Давайте посмотрим на примере. Допустим, в нашем модуле есть такая команда:

```
mov eax, _OurString
……..
_OurString du ‘Our string’, 0
```

В скомпилированном виде он может выглядеть так:

```
mov eax, 00402004h
```

А теперь, вопрос: что произойдет, если данный модуль будет подвергнут перемещению. Правильно: наша строка _OurString окажется размещенной по совершенно другому адресу. Это приведет к тому, что представленный выше код будет работать неправильно. Как же быть? Полностью застраховаться от перемещения мы не можем.

Тут нам на помощь приходит так называемая таблица перемещаемых элементов. Что это такое? Если говорить упрощенно, то это специальная служебная таблица, в которой описаны участки программы, чувствительные к перемещению файла (например, такие, как в нашем примере). В общем виде процесс загрузки того или иного модуля загрузчиком может быть описан так:

1) Пытаемся разместить загружаемый модуль по его желательному базовому адресу загрузки.

2) Если это удается, то считаем, что загрузка завершена, иначе переходим к шагу 3.

3) Перемещаем модуль и проверяем наличие таблицы перемещаемых элементов

4) Если таблицы перемещаемых элементов нет, то оставляем все так, как есть.

5) Если же таблица имеется, то в соотвествии с ней корректируем команды, чувтствительные к перемещению таким образом, чтобы они сохранили свое назначение в условиях перемещения.

Как видите, таблица перемещаемых элементов вполне способна решить возникшую проблему. Но как ее создать? Для этого нам достаточно задать соотвествующую секцию под эту таблицу (все остальное за нас сделает Flatasm), примерно так:

```
section '.reloc' fixups data discardable
```

Данная таблица может быть создана как у динамической библиотеки, так и у обычного исполняемого файла. Но, так как последние практически никогда не подвергаются перемещению, в отличие от библиотек, у них, как правило, таблицу перемещаемых элементов не задают.

Пример

Теперь пришло время перейти от теории к практике. В качестве примере напишем небольшую динамическую библиотеку, экспортирующую всего одну функцтю Show. Вот полный исходный код этой библиотеки:

```
format PE DLL
entry dllmain
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
;функция dllmain
proc dllmain hInstDLL, fdwReason, lpvReserved
     cmp [fdwReason], DLL_PROCESS_ATTACH
     je dllProcessAttach
     cmp[fdwReason], DLL_PROCESS_DETACH
     je dllProcessDetach
     mov eax, TRUE
     ret
dllProcessAttach:
     mov ebx, _Loaded
     jmp dllshow
dllProcessDetach:
     mov ebx, _UnLoaded
dllshow:
     invoke MessageBox, HWND_DESKTOP, ebx, _Probel, MB_OK
     mov eax, TRUE
     ret
endp
;сама функция, экспортировать которую мы и будем
proc ShowMessage, lpStr
     invoke MessageBox, HWND_DESKTOP, [lpStr], _Probel, MB_OK
     ret
endp
section '.data' data readable
_Probel du ' ', 0
_Loaded du 'Library was loaded', 0
_UnLoaded du 'Library was unloaded', 0
;таблица экспорта
section '.edata' export data readable
export 'MYDLL.DLL',\
       ShowMessage, 'ShowMessage'

section '.idata' import data readable writeable
library user, 'user32.dll'
import user,\
        MessageBox, 'MessageBoxW'
;таблица перемещаемых элементов
section '.reloc' fixups data discardable
```

Теперь напишем небольшое приложение, которое будет работать с этой библиотекой. Вот его полный исходный код:

```
format PE GUI 4.0
entry start
include 'D:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
start:
       invoke LoadLibrary, _LibraryName
       invoke GetProcAddress, eax, _FunctionName
       stdcall eax, _Hello
       invoke ExitProcess, 0

section '.data' data readable
_Hello du 'Hello', 0
_LibraryName du 'mydll.dll', 0
_FunctionName db 'ShowMessage', 0

section '.idata' import data readable writeable
library kernel, 'kernel32'

import kernel,\
       LoadLibrary, 'LoadLibraryW',\
       GetProcAddress, 'GetProcAddress',\
       ExitProcess, 'ExitProcess'
```

Как легко видно из приведенного кода мы используем динамическую загрузку библиотеки.

Глава 21. Разработка консольных приложений
Объявление консольного приложения

До этого времени мы рассматривали исключительно графические приложения Windows теперь пришло время поговорить и о консольных приложениях. Строго говоря разница между графическими и консольными приложениями является чисто условной: предполагается, что консольные приложения работают исключительно с консолью и не создают никаких окон, а графические, наоборот, создают одно или несколько окон и работают исключительно с ними, обходя консоль стороной. Однако, заставить вас соблюдать это условие никто не может. Поэтому консольные приложения могут работать с окнами, а графические с консолью.

При описании формата выходного файла вы может явно указать, что создаете консольное приложение. Графические приложения мы всегда объявляли так:

```
format PE GUI 4.0
```

а консольные объявляются похожим образом:

```
format PE Console
```

При таком объявлении вы явно указываете, что создаете консольное приложение. Что нам это дает? При таком подходе операционная система сама создаст консоль. Это несколько упрощает разработку (не нужно самому создавать консоль), но может вызвать и трудности: как быть если, например, наша программа не сразу начинает работать с консолью, а перед этим выполняет какие-то действия. Может быть вашему приложению вообще не нужна консоль. Но операционная система не спросит вас и создаст консоль даже тогда, когда вы в ней (консоли) явно не нуждаетесь.

Единственный способ избежать этого – объявить свое приложение как графическое. Нет, я ни в коем случае не призываю вас отказываться от этого формата исполняемых файлов. Вы сами решаете файл какого типа вы хотите получить. Автоматическое создание консоли – удобный механизм, который вы можете использовать в своих задачах. Но для лучшего понимания данной темы далее мы будем создавать консоль вручную.

Ручное создание консоли

Для создания консоли используется функция AllocConsole из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI AllocConsole(void);
```

Как видите у данной функции нет параметров. В случае успеха она возвращает ненулевое значение, а в случае ошибки – нуль.

Погодите, а как же дескриптор? – спросите вы. А нет его. Дело в том, что у приложения не может быть более одной консоли: или одна, или ни одной. Третьего как говорится не дано. Поэтому и дескриптор консоли не нужен: операционная система и так знает, какая консоль закреплена за данным приложением.

Для уничтожения консоли используется функция FreeConsole из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI FreeConsole(void);
```

Здесь все похоже: входных параметров нет, в случае успеха возвращается ненулевое значение, а в случае ошибки – нуль.

Напоминаю, что вызывать данные функции нужно лишь в том случае, если вы создаете графическое приложение, которое будет работать с консолью. Если же вы изначально создаете консольное приложение, то создавать консоль еще раз вам не нужно.

Заголовок окна консоли

Для чтения окна заголовка окна консоли используется функция GetConsoleTitle(A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
DWORD WINAPI GetConsoleTitle(
  __out  LPTSTR lpConsoleTitle, //Адрес буфера куда следует записать
                                //читаемый заголовок
  __in   DWORD nSize            //Размер буфера в символах
);
```

В случае успеха данная функция возвращает длину заголовка окна в символах, а в случае ошибки она возвращает ноль.

Для установки нового заголовка окна консоли используется функция SetConsoleTitle (A/W) из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI SetConsoleTitle(
  __in  LPCTSTR lpConsoleTitle //Адрес строки с новым заголовком
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки нуль.

Буфер экрана

Ввод и вывод на консоль осуществляется посредством буфера экрана. Что это такое? По сути это служебная структура данных, используемая операционной системой для организации ввода и вывода на консоль. Более подробно про строение буфера обмена и консоли в целом вы можете прочесть у Александра Побегайло в его книге «Системное программирование в Windows».

Создать буфер экрана можно с помощью функции CreateConsoleScreenBuffer из библиотеки kernel32.dll. Вот ее прототип:

```
HANDLE WINAPI CreateConsoleScreenBuffer(
  __in        DWORD dwDesiredAccess,    //Режимы доступа
  __in        DWORD dwShareMode,        //Режимы разделения доступа
  __in_opt    const SECURITY_ATTRIBUTES *lpSecurityAttributes, //атрибуты
                                                               //защиты
  __in        DWORD dwFlags,            //Флаги
  __reserved  LPVOID lpScreenBufferData //Зарезервировано
);
```

Теперь подробнее:

dwDesiredAccess – определяет способ доступа к создаваемому буферу экрану. Этот параметр может принимать одно из двух значений или их сочетание:

GENERIC_READ – буфер будет доступен на чтение;

GENERIC_WRITE – буфер будет доступен на запись.

dwShareMode – определяет, может ли буфер экрана использоваться несколькими процессами одновременно. Если этот параметр равен нулю, то данным буфером может пользоваться только тот процесс, который его создал. Также он может принимать любую комбинацию следующих значений:

FILE_SHARE_READ – другим процессам разрешается чтение;

FILE_SHARE_WRITE – другим процессам разрешается запись;

Вроде бы все просто и понятно. Однако, не совсем. Дело в том, что для корректного отображения в консоли символов, вводимых пользователем с клавиатуры должно быть установлено значение FILE_SHARE_WRITE (одно или совместно со значением FILE_SHARE_READ). С чем это связано, я точно сказать не могу, могу лишь предполагать, что это как-то связано с внутренней архитектурой консоли.

lpSecurityAttributes – задает атрибуты защиты. Но так как в данной книге атрибуты зашиты мы не рассматриваем, то будем ставить сюда значение NULL (использовать атрибуты по умолчанию).

dwFlags. Здесь должно быть установлено значение CONSOLE_TEXTMODE_BUFFER. Из заголовочного файла WinCon.h мы

узнаем, что численное значение этой константы единица. Нам это пригодится так как заголовочные файлы FASMa не описывают эту константу.

И последний параметр lpScreenBufferData должен быть равен NULL. Он оставлен для будущих целей (не спрашивайте каких именно – не знаю).

В случае успеха данная функция вернет дескриптор созданного буфера экрана, а в случае ошибки значение INVALID_HANDLE_VALUE.

Для уничтожения буфера экрана используется уже известная нам функция CloseHandle из библиотеки kernel32.dll. На всякий случай напомню ее прототип:

```
BOOL WINAPI CloseHandle(
  __in  HANDLE hObject //Дескриптор уничтожаемого объекта
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки – ноль.

Однако создать буфер экрана мало, нужно еще связать его с консолью. Делается это функцией SetConsoleActiveScreenBuffer из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI SetConsoleActiveScreenBuffer(
  __in  HANDLE hConsoleOutput  //Дескриптор буфера экрана
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки – ноль.

Стандартные потоки ввода-вывода

Они до глубины души знакомы любому программисту пишущему на Си: stdin, stdout и stderr[1]. Но мы ведь пишем на ассемблере. Ну и что. Мы тоже можем их использовать. Для их получения используется функция GetStdHandle из библиотеки kernel32.dll. Вот ее прототип:

```
HANDLE WINAPI GetStdHandle(
  __in  DWORD nStdHandle     //тип потока
);
```

Параметр nStdHandle может принимать одно из трех значений: STD_INPUT_HANDLE, STD_OUTPUT_HANDLE, STD_ERROR_HANDLE. Назначение каждого из них, думаю, ясно из их названия.

В случае успеха данная функция вовзращает дескриптор соответствующего потока ввода-вывода. А в случае ошибки она возвращает значение INVALID_HANDLE_VALUE. Если с приложением не связано

[1] Stdin – стандартный поток ввода (обычно, клавиатура), stdout – стандартный поток вывода (обычно, консоль), stderr – стандартный поток вывода сообщений об ошибках (обычно, консоль). Более подробно смотри в учебниках по языку Си.

никаких стандартных потоков ввода-вывода (то есть у него нет консоли), то данная функция вернет значение NULL.

Закрывать или уничтожать полученные дескрипторы не надо. Они будут автоматически уничтожены при уничтожении консоли.

Ввод и вывод на консоль

Начнем с вывода. Вывод на консоль осуществляется функцией WriteConsole(A/W) из библиотеки kernel32.dll. Вот ее прототип:

```
BOOL WINAPI WriteConsole(
  __in        HANDLE hConsoleOutput,  //Дескриптор буфера экрана
  __in        const VOID *lpBuffer,   //Адрес выводимой на консоль строки
  __in        DWORD nNumberOfCharsToWrite,//Количество выводимых символов
  __out       LPDWORD lpNumberOfCharsWritten, //Адрес, по которому будет
                     //записано количество фактически выведенных символов
  __reserved  LPVOID lpReserved //Зарезервировано, должно быть NULL
);
```

В случае успеха функция возвращает ненулевое значение, а в случае ошибки ноль.

Ввод с консоли осуществляется функцией ReadConsole (A/W) из той же библиотеки. Вот ее прототип:

```
BOOL WINAPI ReadConsole(
  __in      HANDLE hConsoleInput, //дескриптор станартного потока ввода
  __out     LPVOID lpBuffer,      //Адрес по которому будут записываться
                                  //читаемые с консоли данные
  __in      DWORD nNumberOfCharsToRead, //Количество символов которые
                                        //следует прочесть
  __out     LPDWORD lpNumberOfCharsRead, //Адрес по которому будет
                       //записано количество реально прочитанных символов
  __in_opt  LPVOID pInputControl //Адрес специальной структуры
             //описывающей признак окончания ввода (будем оставлять NULL)
);
```

В случае успеха данная функция вовзащает ненулевое значение, а в случае ошибки – нуль.

Обращаю ваше внимание на то, что если функция WriteConsole(A/W) в качестве параметра ожидает дескриптор буфера экрана, то функция ReadConsole(A/W) ожидает дескриптор стандартного потока ввода stdin. Также важным различием является то, что если в функции WriteConsole параметр lpNumberOfCharsWritten может быть равен NULL, то в функции ReadConsole это является ошибкой применительно к параметру lpNumberOfCharsRead.

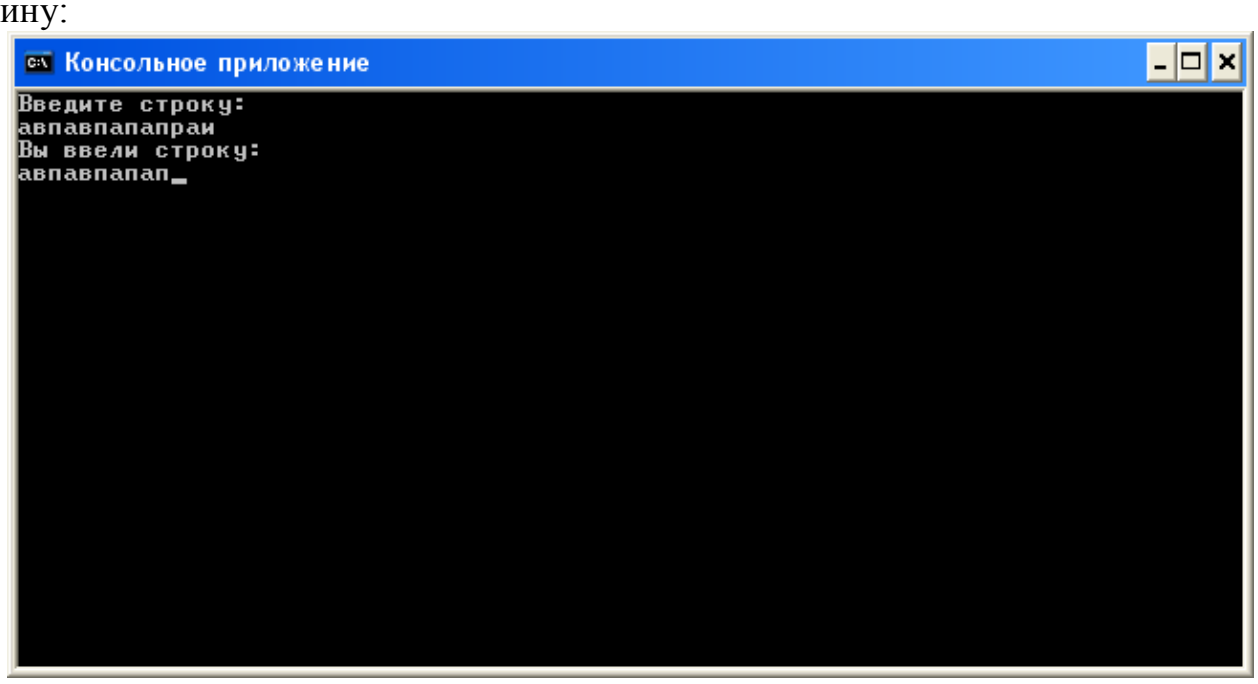
Пример программы

На этом, рассмотрение теории прекратим и перейдем наконец к практике. Ниже приводится исходный текст программы, в которой создается консоль, пользователю предлагается ввести строку после чего, введенная им строка повторно выводится на консоль. Ну и наконец после двухсекундной задержки консоль уничтожается.

```
format PE GUI 4.0
entry start

include 'D:\FASM\INCLUDE\win32a.inc'
include 'D:\FASM\INCLUDE\ENCODING\win1251.inc'

CONSOLE_TEXTMODE_BUFFER = 1

section '.code' code readable executable
start:
      ;Создаем консоль
      invoke AllocConsole
      ;Убеждаемся в том, что консоль успешно создана
      test eax, eax
      jz ErrorCreateConsole
      ;Устанавливаем заголовок окна консоли
      invoke SetConsoleTitle,_Title
      ;Создаем буфер экрана
      invoke CreateConsoleScreenBuffer, GENERIC_READ+GENERIC_WRITE,
                  FILE_SHARE_WRITE , NULL, CONSOLE_TEXTMODE_BUFFER,NULL
      ;Убеждаемся в том, что буфер экрана успешно создан
      cmp eax, INVALID_HANDLE_VALUE
      je ErrorCreateScreenBuffer
      ;Сохраняем дескриптор буфера экрана
      mov [_hscreenbuffer], eax
      ;Связываем буфер экрана с консолью
      invoke SetConsoleActiveScreenBuffer, eax
      ;Выводим в консоль предложение ввести строку
      invoke WriteConsole, [_hscreenbuffer],_String1,17, NULL, NULL
      ;Получаем дескриптор стандартного потока ввода
      invoke GetStdHandle, STD_INPUT_HANDLE
      ;Читаем строку вводимую пользователем
      invoke ReadConsole, eax,_inputstring,10,_countreadsymbols,NULL
      ;Выводим на консоль сообщение о том, что хотим повторить
      ;прочитанную ранее строку
      invoke WriteConsole, [_hscreenbuffer],_String2,19,NULL,NULL
      ;Выводим строку введенную пользователем
      invoke WriteConsole, [_hscreenbuffer],_inputstring,10,NULL,NULL
      ;Делаем временную задержку в 2 секунды
      invoke Sleep, 2000
      ;Уничтожаем буфер экрана
      invoke CloseHandle, [_hscreenbuffer]
      ;Уничтожаем консоль
      invoke FreeConsole
      jmp Exit
ErrorCreateConsole:
      invoke   MessageBox,HWND_DESKTOP,   _ErrorCreateConsole,   _Error,
MB_ICONERROR
      jmp Exit
ErrorCreateScreenBuffer:
      invoke MessageBox, HWND_DESKTOP, _ErrorCreateScreenBuffer, _Error,
MB_ICONERROR
```

```
Exit:
      invoke ExitProcess, 0

section '.data' data readable
_ErrorCreateConsole du 'Ошибка при попытке создания консоли', 0
_ErrorCreateScreenBuffer du 'Ошибка при попытке создания буфера экрана',
0
_Error du 'Ошибка', 0
_Title du 'Консольное приложение',0
_String1 du 'Введите строку:',0, 10, 13
_String2 du 'Вы ввели строку:',0, 10, 13

section '.rdata' data readable writeable
_hscreenbuffer dd 00
_countreadsymbols dd 00
_inputstring rd 22

section '.idata' data import readable writeable
library kernel, 'kernel32.dll',\
        user, 'user32.dll'
import kernel,\
       AllocConsole, 'AllocConsole',\
       FreeConsole , 'FreeConsole',\
       GetStdHandle, 'GetStdHandle',\
       ExitProcess , 'ExitProcess' ,\
       SetConsoleTitle, 'SetConsoleTitleW',\
       CreateConsoleScreenBuffer, 'CreateConsoleScreenBuffer',\
       SetConsoleActiveScreenBuffer, 'SetConsoleActiveScreenBuffer',\
       WriteConsole, 'WriteConsoleW',\
       ReadConsole , 'ReadConsoleW',\
       CloseHandle , 'CloseHandle',\
       Sleep       , 'Sleep'
import user,\
       MessageBox  , 'MessageBoxW'
```

Если все правильно сделано, то вы увидите приблизительно следующую картину:



Глава 22. Обработка ошибок SEH

Структурная обработка исключений (Structured Exception Handling, SEH) представляет собой механизм, позволяющий приложениям обрабатывать различные исключительные ситуации (такие как: деление на ноль, ошибка при обращении к памяти и др.). В основе этого механизма лежит односвязный список структур _EXCEPTION_REGISTRATION:

```
_EXCEPTION_REGISTRATION struc
      prev        dd    ?     ; адрес следующего элемента списка
      handler     dd    ?     ; Адрес процедуры-обработчика
_EXCEPTION_REGISTRATION ends
```

Эти структуры также называют SEH-фреймами. Адрес первой структуры списка хранится по адресу: fs:[00h]. Поле prev последнего элемента списка равно -1. Когда происходит какое-либо исключение, операционная система переходит в начало списка и по списку вызывает обработчики исключений до тех пор, пока либо не будет найден обработчик, который сможет обработать эту ошибку, либо не будет достигнут конец списка обработчиков.

Теперь поговорим о процедуре обработчике. По сути, это обычная процедура, которая ответственна за обработку тех или иных исключительных ситуаций. Для уведомления операционной системы о том обработано исключение или нет, данная процедура должна вернуть одно из значений представленных ниже:

EXCEPTION_EXECUTE_HANDLER (1)[1] – исключение не обработано. В этом случае ОС продолжает поиск обработчиков. При этом если ни один обработчик так и не сможет обработать это исключение приложение будет прибито.

EXCEPTION_CONTINUE_SEARCH (0) – исключение успешно обработано. В этом случае операционная система возвращает управление на ту инструкцию, при выполнении которой произошла ошибка. При этом, если причина ошибки не была устранена, то программа уходит в бесконечный цикл: исключение-обработчик-исключение-…

EXCEPTION_CONTINUE_EXECUTION (-1) – исключение не обработано. В этом случае ОС вызывает этот обработчик еще раз. При этом программа может уйти в бесконечный цикл.

С этим я думаю все понятно. Гораздо сложнее разобраться с входными параметрами функции-обработчика. Вот прототип этой функции:

[1] Здесь я привожу численные значения этих констант, так как inc-файлы не содержат их описания.

```
EXCEPTION_DISPOSITION
 __cdecl _except_handler(
     struct _EXCEPTION_RECORD *ExceptionRecord, //адрес структуры
                        //ExceptionRecord, описывающей параметры исключения
     void * EstablisherFrame,        //адрес текущего SEH-фрейма
     struct _CONTEXT *ContextRecord, //адрес структуры CONTEXT, в которой
                         //хранятся значения регистров на момент исключения
     void * DispatcherContext //адрес служебной структуры диспетчера
                              //исключений (именно он занимается поиском
                              //и вызовом обработчиков)
     );
```

Поскольку структура DispatcherContext используется системой для своих внутренних целей, то рассматривать ее мы не будем.

Ниже приводится прототип структуры ExceptionRecord:

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;    //Код ошибки
  DWORD                    ExceptionFlags;   //Флаги исключения
  struct _EXCEPTION_RECORD *ExceptionRecord;//адрес структуры
                                             //ExceptionRecord
  PVOID                    ExceptionAddress;//Адрес, по которому
                                             //произошло исключение
  DWORD                    NumberParameters;//количество параметров
                                             //исключения
  ULONG_PTR
          ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
                                           //Массив параметров исключения
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

Поле ExceptionFlags строго говоря представляет собой всего один флаг EXCEPTION_NONCONTINUABLE, который обычно взводится не ОС, а обработчиком. При взведенном флаге обработчик не может вернуть значение EXCEPTION_CONTINUE_EXECUTION. Таким образом, данный флаг говорит о том, что данное исключение никак не может быть обработано, то есть приложение, в котором возникло такое исключение уже никак нельзя спасти. Если же обработчик нарушит «договоренность» и вернет системе EXCEPTION_CONTINUE_EXECUTION, то при попытке выполнить инструкцию, даже если причина ошибки устранена, произойдет ошибка с кодом STATUS_NONCONTINUABLE_EXCEPTION. Данное исключение также будет передаваться с взведенным флагом EXCEPTION_NONCONTINUABLE.

Для того чтобы понять, зачем в структуре ExceptionRecord указатель на другой экземпляр структуры ExceptionRecord представим такую ситуацию: предположим, что во время выполнения программы произошло исключение 1. Операционная система начала раскрутку цепочки SEH-фреймов и вызвала какой-то обработчик. Во время работы этого обработчика происходит какая-то ошибка 2. Что произойдет в этом случае? В этом случае ОС вновь начнет раскрутку SEH-фреймов в поиске обработчика. При вызове какого-то обработчика ему будет передаваться адрес структуры ExceptionRecord, описывающей ошибку 2, возникшей при попытке обработки ошибки 1. А вот поле ExceptionRecord этой структуры будет указывать на структуру, описывающую исключение 1. В этом случае говорят, что исключение 2

вложено в исключение 1. Причем степень такой вложенности ничем не ограничена.

Параметр NumberParameters задает количество элементов в массиве ExceptionInformation, в котором хранятся дополнительные параметры исключения. Обычно этот массив не содержит в себе какой-либо информации, однако он может быть использован приложением для передачи в процедуру-обработчик каких-либо данных. Правда для этого программе придется воспользоваться функцией RaiseException из библиотеки kernel32.dll. Данная функция используется для генерирования исключения. Вот ее прототип:

```
void WINAPI RaiseException(
  __in  DWORD dwExceptionCode,         //Код исключения
  __in  DWORD dwExceptionFlags,        //Флаги исключения
  __in  DWORD nNumberOfArguments,      //Количество параметров исключения
  __in  const ULONG_PTR *lpArguments   //Массив параметров исключения
);
```

Аргументы этой функции соответствуют полям структуры ExceptionRecord, поэтому пояснять я их не буду. Данная функция не имеет возвращаемых значений.

Будем считать, что со структурой ExceptionRecord разобрались, теперь у нас на очереди структура Context (ее иногда называют контекстом потока). Данная структура содержит в себе значения регистров процесса на момент исключения. Поэтому большинство ее полей не нуждаются в комментариях. Понятно, что она различается на разных архитектурах. Вот прототип этой структуры для процессоров x86:

```
typedef struct _CONTEXT
 {
     DWORD ContextFlags;
     DWORD   Dr0;
     DWORD   Dr1;
     DWORD   Dr2;
     DWORD   Dr3;
     DWORD   Dr6;
     DWORD   Dr7;
     FLOATING_SAVE_AREA FloatSave;
     DWORD   SegGs;
     DWORD   SegFs;
     DWORD   SegEs;
     DWORD   SegDs;
     DWORD   Edi;
     DWORD   Esi;
     DWORD   Ebx;
     DWORD   Edx;
     DWORD   Ecx;
     DWORD   Eax;
     DWORD   Ebp;
     DWORD   Eip;
     DWORD   SegCs;
     DWORD   EFlags;
     DWORD   Esp;
     DWORD   SegSs;
```

```
    BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION[1]];
} CONTEXT;
```

Здесь нужно пояснить всего три поля ContextFlags, FloatSave и ExtendedRegisters. Первое поле представляет собой сочетание флагов, задающих различный состав структуры CONTEXT[2].Наиболее интересные флаги представлены в таблице ниже:

Флаг	Описание
CONTEXT_DEBUG_REGISTERS	Включить в состав структуры отладочные регистры (Dr0-Dr7)
CONTEXT_FLOATING_POINT	Включить поле FloatSave
CONTEXT_SEGMENTS	Включить сегментные регистры (gs, fs, es, ds)
CONTEXT_INTEGER	Включить регистры общего назначения (edi, esi, ebx, edx, ecx, eax)
CONTEXT_CONTROL	Включить регистры управления (ebp, eip, cs, регистр флагов, esp, ss)
CONTEXT_EXTENDED_REGISTERS	Включить массив ExtendedRegisters
CONTEXT_FULL	CONTEXT_CONTROL+ CONTEXT_INTEGER+ CONTEXT_SEGMENTS
CONTEXT_ALL	Все вышеперечисленные флаги

Поле FloatSave представляет собой структуру _FLOATING_SAVE_AREA размером 70h байт. Информации по поводу назначения этой структуры мне так и не удалось найти. Даже msdn не описывает поля этой структуры, а просто отсылает к заголовочному файлу winnt.h, в котором содержится описание этой структуры. Скорее всего, она используется системой для своих внутренних целей. Как бы то ни было, мы не будем использовать эту структуру при обработке ошибок.

Массив ExtendedRegisters хранит в себе значения дополнительных регистров процессора, например, таких, как регистры SSE и MMX расширения.

На этом общая теория по структурной обработке исключений окончена. Вы еще не забыли, с чего мы начинали. Теперь, для того чтобы у вас в голове все встало на свои места, перейдем к практике.

[1] MAXIMUM_SUPPORTED_EXTENSION = 512

[2] Данная возможность изменения состава структуры CONTEXT может быть использована при определении контекста другого потока, с помощью функции GetThreadContext. Мы ее рассматривать не будем. Данная функция предельно проста.

Пример программы

Для закрепления выше изложенного материала, напишем небольшую программу. В ней будет искусственно допущена ошибка, которая будет обработана соответствующим обработчиком. Исходный текст этой программы вместе с комментариями представлен ниже:

```
format PE GUI 4.0
entry start
include 'C:\FASM1\INCLUDE\win32a.inc'

EXCEPTION_CONTINUE_SEARCH = 0

section '.code' code readable executable
start:
      ;Создаем свой SEH фрейм и добавляем его в список обработчиков
      push excepthandler
      push dword [fs:00]
      mov [fs:00], esp
      ;вызываем ошибку деления на ноль
      xor edx, edx
      xor ebx, ebx
      div ebx
      ;Удаляем свой SEH фрейм. Приводим список обработчиков к первозданному виду
      mov eax, [esp]
      mov [fs:00], eax
      add esp, 8
      ;Сообщаем пользователю,что мы продолжили работу после ошибки
      invoke MessageBox, HWND_DESKTOP, _ContinueWork, _Probel, MB_OK
      ;завершаем работу программы
      invoke ExitProcess, 0
;Обработчик исключений
proc excepthandler ExceptionRecord, EstablisherFrame, Context, DispatcherContext
      ;Сообщаем пользователю, что мы отловили исключение
      invoke MessageBox, HWND_DESKTOP, _CaughtException, _Probel, MB_ICONERROR
      ;Заносим в регистр ebx адрес структуры CONTEXT
      mov ebx, [Context]
      ;Заносим в регистр eax значение регистра ebx на момент исключения
      mov eax, [ebx+0A4h]
      ;Увеличиваем его на единицу
      inc eax
      ;Записываем новое значение регистра ebx
      mov [ebx+0A4h], eax
      ;Выходим из процедуры, сообщая, что мы обработали исключение
      mov eax, EXCEPTION_CONTINUE_SEARCH
      ret
endp
section '.data' data readable
_CaughtException du 'I caught exception', 0
_ContinueWork du 'I continue work', 0
_Probel du ' ', 0

section '.idata' import data  readable writeable
library kernel , 'kernel32.dll',\
        user   , 'user32.dll'

import kernel,\
       ExitProcess, 'ExitProcess'
```

```
import user,\
       MessageBox, 'MessageBoxW'
```

Здесь 0A4h – это смещение поля Ebx от начала структуры CONTEXT.

Функция SetUnhandledExceptionFilter

Ответьте мне на вопрос: что произойдет если во время поиска обработчика исключения, ОС так и не найдет обработчика, который смог бы обработать возникшее исключение? Данное приложение будет прибито, но не сразу. Перед этим операционная система вызовет так называемый обработчик необработанных исключений, давая приложению последний шанс восстановить свою работоспособность. Данный обработчик представляет собой фунцкию, имеющую следующий прототип:

```
LONG WINAPI UnhandledExceptionFilter(
  __in  struct _EXCEPTION_POINTERS *ExceptionInfo
);
```

Данная функция принимает всего один параметр ExceptionInfo представляющий собой адрес структуры EXCEPTION_POINTERS, прототип этой структуры представлен ниже:

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord; //Адрес структуры ExceptionRecord
  PCONTEXT          ContextRecord;   //Адрес структуры CONTEXT
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Обе эти структуры (ExceptionRecord и CONTEXT) мы уже рассматривали, повторяться я не буду.

Данная функция, так же как и обычный обработчик структурных исключений должен вернуть одно из предопределенных значений EXCEPTION_EXECUTE_HANDLER, EXCEPTION_CONTINUE_SEARCH или EXCEPTION_CONTINUE_EXECUTE, говорящих системе как поступить дальше: вернуть управление приложению или прибить его. Но вот интерпретация этих значений уже иная:

EXCEPTION_EXECUTE_HANDLER (1) – Исключение не обработано. При этом система еще раз проходится по списку обработчиков, вызвая каждый из них. Если исключение так и остается необработанным приложение прихлопывается без вывода соответствующего окна об ошибке.

EXCEPTION_CONTINUE_SEARCH (0) – исключение не обработано. При этом система прихлопывает приложение с выводом соответствующего окна.

EXCXEPTION_CONTINUE_EXECUTION (-1) – исключение обработано. В этом случае операционная система возвращает управление на ту инструкцию, при выполнении которой произошла ошибка. При этом, если

причина ошибки не была устранена, то программа уходит в бесконечный цикл: исключение-обработчик-исключение-…

Читатель, наверное, скажет: все это конечно хорошо, но как я могу использовать это в своей программе. Очень просто. Дело в том, что мы можем устанавливать свой обработчик в качестве обработчика необработанных исключений. Делается это с помощью функции SetUnhandledExceptionFilter из библиотеки kernel32.dll. Вот ее прототип:

```
LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter(
  __in  LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
                                                  //Адрес функции-обработчика
);
```

Данная функция возвращает в случае успеха адрес замещенного обработчика необработанных исключений, или NULL в случае ошибки.

В качестве примера давайте перепишем рассмотренную выше программу так, чтобы в ней использовался фильтр необработанных исключений. Исходный код такой программы представлен ниже:

```
format PE GUI 4.0
entry start
include 'C:\FASM1\INCLUDE\win32a.inc'

EXCEPTION_CONTINUE_EXECUTION = -1

section '.code' code readable executable
start:
      ;Устанавливаем свой обработчик необработанных исключений
      invoke SetUnhandledExceptionFilter, TopLevelExceptionFilter
      ;вызываем ошибку деления на ноль
      xor edx, edx
      xor ebx, ebx
      div ebx
      ;Сообщаем пользователю что мы продолжили работу после ошибки
      invoke MessageBox, HWND_DESKTOP, _ContinueWork, _Probel, MB_OK
      ;завершаем работу программы
      invoke ExitProcess, 0

;Обработчик необработанных исключений
proc TopLevelExceptionFilter ExceptionPointers
     ;Сообщаем пользователю, что мы отловили исключение
     invoke   MessageBox,   HWND_DESKTOP,   _CaughtException,   _Probel,
MB_ICONERROR
     ;Заносим в регистр ebx адрес структуры EXCEPTION_POINTERS
     mov ebx, [ExceptionPointers]
     ;Заносим в регистр ebx адрес структуры CONTEXT
     mov ebx, [ebx+4]
     ;Заносим в регистр eax содержимое регистра ebx в момент исключения
     mov eax, [ebx+0A4h]
     ;Увеличиваем его на единицу
     inc eax
     ;Записываем новое значение регистра ebx
     mov [ebx+0A4h], eax
     mov eax, EXCEPTION_CONTINUE_EXECUTION
     ret
endp
```

```
section '.data' data readable
_CaughtException du 'I caught exception', 0
_ContinueWork du 'I continue work', 0
_Probel du ' ', 0

section '.idata' import data  readable writeable
library kernel , 'kernel32.dll',\
        user   , 'user32.dll'

import kernel,\
       SetUnhandledExceptionFilter, 'SetUnhandledExceptionFilter',\
       ExitProcess, 'ExitProcess'

import user,\
       MessageBox, 'MessageBoxW'
```

VEH

Векторная обработка исключений (Vectored Exception Handling, VEH) – это механизм похожий на SEH, но реализованный независимо от него. VEH появился в ОС Windows XP. По сути, это тот же самый же SEH, но немного измененный с целью упрощения работы программиста. Для добавления обработчика в цепочку VEH-обработчиков используется функция AddVectoredExceptionHandler из библиотеки kernel32.dll. Вот ее прототип:

```
PVOID WINAPI AddVectoredExceptionHandler(
  __in  ULONG FirstHandler,  //Порядок обработчика в цепочке обработчиков
  __in  PVECTORED_EXCEPTION_HANDLER VectoredHandler //адрес функции-
                                                    //обработчика
);
```

Если параметр FirstHandler отличен от нуля, тогда обработчиков добавляется в начало списка. В противном случае он будет добавлен в конец списка обработчиков.

В случае успеха данная функция возвращает дескриптор добавленного обработчика. В случае же ошибки она возвращает NULL.

Для удаления обработчика из списка обработчиков используется функция RemoveVectoredExceptionHandler из той же библиотеки вот ее прототип:

```
ULONG WINAPI RemoveVectoredExceptionHandler(
  __in  PVOID Handler //Дескриптор удаляемого обработчика
);
```

В случае успеха данная функция возвращает ненулевое значение, и ноль в случае ошибки.

Сам обработчик исключений имеет следующий вид:

```
LONG CALLBACK VectoredHandler(
  __in  PEXCEPTION_POINTERS ExceptionInfo //Адрес структуры
                                          //EXCEPTION_POINTERS
);
```

В случае если исключение успешно обработано она должна вернуть значение EXCEPTION_CONTINUE_EXECUTION (-1). При этом ОС вернет управление на инструкцию, вызвавшую исключение. Если же исключение не обработано и следует продолжить поиск обработчика, то данная функция должна вернуть значение EXCEPTION_CONTINUE_SEARCH (0).

При использовании механизма VEH следует помнить, что на системах младше Windows XP он не реализован.

Ниже представлен исходный код программы, рассматриваемой выше, написанной с использованием VEH.

```
format PE GUI 4.0
entry start
include 'I:\FASM1\INCLUDE\win32a.inc'

EXCEPTION_CONTINUE_EXECUTION = -1

section '.code' code readable executable
start:
      ;Добавляем свой обработчик исключений
      invoke AddVectoredExceptionHandler,1, VectoredHandler
      mov edi, eax
      ;Вызываем ошибку деления на ноль
      xor edx, edx
      xor ebx, ebx
      div ebx
      ;Удаляем обработчик исключений
      invoke RemoveVectoredExceptionHandler, edi
      ;Сообщаем пользователю, что мы продолжили работу после ошибки
      invoke MessageBox, HWND_DESKTOP, _ContinueWork, _Probel, MB_OK
      ;завершаем работу программы
      invoke ExitProcess, 0
;Обработчик исключений
proc  VectoredHandler ExceptionInfo
      push ebx
      ;Сообщаем пользователю, что мы отловили исключение
      invoke    MessageBox,    HWND_DESKTOP,    _CaughtException,    _Probel,
MB_ICONERROR
      ;Заносим в регистр ebx адрес структуры EXCEPTION_POINTERS
      mov ebx, [ExceptionInfo]
      ;Заносим в регистр ebx адрес структуры CONTEXT
      mov ebx, [ebx+4]
      ;Заносим в регистр eax значение регистра ebx на момент исключения
      mov eax, [ebx+0A4h]
      ;Увеличиваем его на единицу
      inc eax
      ;Записываем новое значение регистра ebx
      mov [ebx+0A4h], eax
      ;Выходим из процедуры, сообщая, что мы обработали исключение
      mov eax, EXCEPTION_CONTINUE_EXECUTION
      pop ebx
      ret
endp
section '.data' data readable
_CaughtException du 'I caught exception', 0
_ContinueWork du 'I continue work', 0
_Probel du ' ', 0

section '.idata' import data  readable writeable
library kernel , 'kernel32.dll',\
        user   , 'user32.dll'
```

```
import kernel,\
       AddVectoredExceptionHandler, 'AddVectoredExceptionHandler',\
       RemoveVectoredExceptionHandler, 'RemoveVectoredExceptionHandler',\
       ExitProcess, 'ExitProcess'

import user,\
       MessageBox, 'MessageBoxW'
```

Функции GetLastError и FormatMessage

Описывая ту или иную функцию API, мы говорили, что в случае какой-либо ошибки она возвращает определенное значение (NULL, 0, -1 или еще что-то в этом же духе). Логично возникает вопрос: а можно ли как-нибудь узнать подробности ошибки, что конкретно произошло? Оказывается можно.

Для этого используется функция GetLastError из библиотеки kernel32.dll. Вот ее прототип:

```
DWORD WINAPI GetLastError(void);
```

Данная функция не имеет входных параметров, а возвращает она код последней ошибки.

Код конечно хорошо. Но если вы сообщите пользователю: «программе не удалось сделать то-то по причине: 0005». Боюсь он вас просто не поймет. Для того чтобы преобразовать код ошибки в символьную строку с описанием этой ошибки используется функция FormatMessage (A/W) из той же библиотеки. Вот ее прототип:

```
DWORD WINAPI FormatMessage(
  __in      DWORD dwFlags,    //Набор флагов
  __in_opt  LPCVOID lpSource, //Зависит от флагов
  __in      DWORD dwMessageId,//Код, который нужно преобразовать в сообщение
  __in      DWORD dwLanguageId,//Идентификатор языка
  __out     LPTSTR lpBuffer,  //Адрес буфера, куда следует записать сообщение
  __in      DWORD nSize,      //Размер буфера в символах
  __in_opt  va_list *Arguments//Массив значений вставляемых в сообщение
);
```

Параметр dwFlags представляет собой сочетание флагов, основные из которых перечислены в таблице ниже:

Флаг	Описание
FORMAT_MESSAGE_ ALLOCATE_BUFFER	Если указан данный флаг, то функция сама выделит объем памяти необходимый для размещения сформированного строкового сообщения. При этом по адресу, переданному в lpBuffer, будет записан адрес выделенного функцией буфера. Параметр nSize игнорируется. Выделенный функцией буфер должен быть освобожден функцией LocalFree.
FORMAT_MESSAGE_ ARGUMENT_ARRAY	Параметр Arguments представляет собой адрес массива значений параметров, подставляемых в форматную строку
FORMAT_MESSAGE_ FROM_HMODULE	Параметр lpSource представляет собой дескриптор модуля, в котором хранится таблица соответствия между кодом сообщения и его строковым представлением. Именно по этой таблице и будет искаться расшифровка сообщения.
FORMAT_MESSAGE_ FROM_STRING	Параметр lpSource содержит адрес форматной строки, по которой формируется сообщение[1].
FORMAT_MESSAGE_ FROM_SYSTE M	Функция будет искать строковое представление ошибки в системной таблице соответствий. Если одновременно с этим флагом установлен флаг FORMAT_MESSAGE_FROM_HMODULE, то функция сначала будет искать описание в модуле, дескриптор которого хранится в параметре lpSource. И, если в нем не найдет описания тогда будет искать в системной таблице.

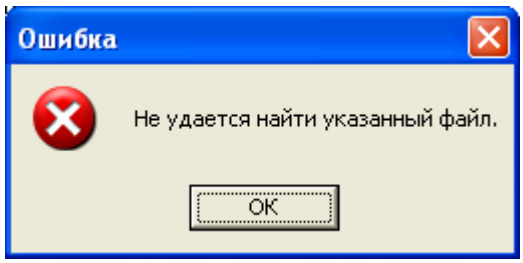
Ниже приводится исходный текст программы, демонстрирующей работу с этой функцией. В этой программе предпринимается попытка открыть несуществующий файл, и выводится сообщение об ошибке, сформированное системой.

```
format PE GUI 4.0
entry start
include 'I:\FASM1\INCLUDE\win32a.inc'
section '.code' code readable executable
start:
        invoke  CreateFile,  _FileName,  GENERIC_READ,  FILE_SHARE_READ,  NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
        ;Убеждаемся в том, что произошла ошибка
        cmp eax, INVALID_HANDLE_VALUE
        jnz FileExist
        ;Такой файл не существует
        invoke GetLastError
        ;Сформируем сообщение об ошибке
        invoke      FormatMessage,      FORMAT_MESSAGE_ALLOCATE_BUFFER      +
FORMAT_MESSAGE_FROM_SYSTEM, NULL, eax, LANG_NEUTRAL, _lpBuffer, 0, NULL
```

[1] Функцию FormatMessage можно использовать не только для получения строкового описания ошибки по ее коду, но и как аналог С-шной функции vsnprintf, осуществляющей форматирование строки по шаблону.

```
        ;Покажем пользователю сообщение об ошибке
        invoke MessageBox, HWND_DESKTOP, [_lpBuffer], NULL, MB_ICONERROR
        ;Освобождаем память и выходим из программы
        invoke LocalFree, [_lpBuffer]
        mov [_lpBuffer], NULL
        jmp ExitProgram
FileExist:
        ;Такой файл существует, закрываем его
        invoke CloseHandle, eax
ExitProgram:
        invoke ExitProcess, 0

section '.data' data readable writeable
_FileName du 'C:\fgdshgfysadgchzjxbc.dat',0
_lpBuffer dd ?

section '.idata' data import readable writeable
library kernel, 'kernel32.dll' ,\
        user, 'user32.dll'
import kernel,\
       CreateFile, 'CreateFileW',\
       CloseHandle, 'CloseHandle',\
       GetLastError, 'GetLastError',\
       FormatMessage, 'FormatMessageW',\
       LocalFree, 'LocalFree',\
       ExitProcess, 'ExitProcess'
import user,\
       MessageBox, 'MessageBoxW'
```

Если все сделано правильно, при запуске этой программы вы увидите приблизительно следующее окно:



Глава 23. Ресурсы

Типы ресурсов

Ресурсы представляют собой своеобразный «склад», на котором программа может хранить какие-то свои вспомогательные данные. Сами ресурсы представляют собой древовидную структуру. На первом уровне задается тип ресурса, а самом нижнем (зависит от типа ресурса) описывается то, что он собой представляет.

Для описания ресурсов, как правило, выделяется отдельная секция «.rsrc» (см. раздел «Секции»). После ключевого слова «directory» перечисляются типы ресурсов, используемых в программе. Пример приведен в листинге:

```
directory RT_DIALOG, dialogs,\
          RT_CURSOR, cursor
```

В данном примере указывается, что в программе будут использоваться ресурсы двух типов (RT_DIALOG и RT_CURSOR), а также назначаются для них псевдонимы (dialogs и cursor соответственно) для дальнейшей детализации ресурсов. В нижеприведенной таблице перечислены основные типы ресурсов.

Тип ресурса	Описание
RT_CURSOR	Курсоры
RT_BITMAP	Изображения формата bmp
RT_ICON	Иконка
RT_MENU	Меню
RT_DIALOG	Диалоговое окно
RT_STRING	Таблица строк
RT_FONTDIR	Набор шрифтов
RT_FONT	Шрифт
RT_GROUP_CURSOR	Коллекция курсоров
RT_GROUP_ICON	Коллекция иконок
RT_VERSION	Версия
RT_ANICURSOR	Анимированный курсор
RT_ANIICON	Анимированная иконка

Идентификатор языка

При описании некоторых ресурсов необходимо указать язык интерфейса. Это делается на втором уровне ресурсов.

Идентификатор языка состоит из двух частей: основного идентификатора языка (отражает используемый язык) и идентификатора «подязыка» (отражает страну, регион).

Поскольку указание идентификатора страны (региона) необязательно, то мы опишим лишь некоторые идентификаторы языков. Они представлены в таблице:

Строковое представление	**Описание**
LANG_NEUTRAL	Язык по умолчанию (в текущей системе)
LANG_ENGLISH	Английский
LANG_FRENCH	Французский
LANG_GERMAN	Немецкий
LANG_GREEK	Греческий
LANG_ITALIAN	Итальянский
LANG_RUSSIAN	Русский
LANG_SPANISH	Испанский

Нужно заметить, что они могут сочетаться друг с другом.

Приведем пример задания двух шаблонов диалоговых окон с разными языками:

```
resource dialogs,\
         1, LANG_ENGLISH,              form1,\
         2, LANG_RUSSIAN+LANG_GREEK, form2
```

Здесь:

“resource” – зарезервированное слово;

1 и 2 — идентификаторы описываемых ресурсов;

form1 и form2 — псевдонимы форм.

Подробно описание в ресурсах шаблона диалогового окна и меню было описано в соответствующих главах книги. Здесь же мы приведем примеры задания некоторых других ресурсов таких, как картинка, строка и иконка.

Изображение

Изображение имеет тип ресурса RT_BITMAP. Ниже приводится пример задания ресурса-картинки:

```
;Определяем тип ресурса картинка и даем псевдоним bitmaps
directory RT_BITMAP, bitmaps
;Задаем язык ресурса и его идентификатор(ID_BITMAP)
resource bitmaps,\
        ID_BITMAP,LANG_NEUTRAL, main_bitmap
;Задаем сам ресурс
bitmap main_bitmap, 'picture.bmp'
```

В данном примере мы задаем ресурс-картинку с идентификатором ID_BITMAP. Первоначально картинка находится в файле picture.bmp. После компилирования программы она будет помещена в сам исполняемый файл.

Для того чтобы загрузить картинку из модуля используется функция LoadImage (A/W) из библиотеки user32.dll. Вот ее прототип:

```
HANDLE WINAPI LoadImage(
  __in_opt  HINSTANCE hinst,  //Дескриптор модуля из которого загружается
                              //ресурс
  __in      LPCTSTR lpszName, //Идентификатор или адрес строки с именем
                              //загружаемого ресурса
  __in      UINT uType,       //Тип загружаемого ресурса
  __in      int cxDesired,    //Ширина загружаемого объекта в пикселях
  __in      int cyDesired,    //Высот загружаемого объекта в пикселях
  __in      UINT fuLoad       //Набор флагов загрузки
);
```

Параметр uType может принимать одно из трех допустимых значений. Он указывает ресурс какого типа мы хотим загрузить. Функция LoadImage (A/W) может быть использована и для их загрузки. В таблице ниже представлены допустимые значения параметра uType:

Допустимое значение	Описание
IMAGE_BITMAP	Картинка
IMAGE_CURSOR	Курсор
IMAGE_ICON	Иконка

Параметр fuLoad представляет собой комбинацию флагов, задающих параметры загрузки ресурса. В таблице ниже представлены основные флаги, которые могут быть использованы в этом параметре:

Флаг	Описание
LR_DEFAULTCOLOR	Отображать изображение в тех цветах, в которых оно и есть
LR_DEFAULTSIZE	Использовать стандартные размеры загружаемых объектов. При этом параметры cxDesired и cyDesired игнорируются
LR_LOADFROMFILE	Загрузить изображение из файла, имя которого указано в строке lpszName
LR_MONOCHROME	Загрузить изображение в черно-белом виде
LR_SHARED	Влияет на порядок уничтожения иконок и курсоров

Данная функция возвращает дескриптор загруженного изображения или NULL в случае ошибки.

Для удаления (освобождения) ресурса–картинки используется функция DeleteObject из библиотеки gdi32.dll. Вот ее прототип:

```
BOOL DeleteObject(
  __in  HGDIOBJ hObject //Дескриптор удаляемого изображения
);
```

В случае успеха данная функция возвращает значение отличное от нуля, и ноль в случае ошибки.

Пример работы с ресурсом изображения мы приводили в главе «элементы управления» когда говорили о статическом элементе управления.

Иконка

Для описания ресурса иконки нужно определить как минимум два ресурса с типами RT_GROUP_ICON (группа иконок) и RT_ICON (иконка). Группа иконок включает в себя одну или несколько иконок. Ниже приводится пример задания иконки в ресурсах:

```
;Определяем типы ресурсов
directory RT_GROUP_ICON, Icons,\
          RT_ICON, my_icon
;Задаем язык группы иконок
resource Icons,\
         ID_ICONS, LANG_NEUTRAL, myicons
;Задаем язык иконки
resource my_icon,\
         ID_ICON, LANG_NEUTRAL, myicon
;Задаем саму иконку
icon myicons, myicon, 'IconFile.ico'
```

Здесь: ID_ICONS и ID_ICON – идентификаторы группы иконок и иконки соответственно;

myicons и myicon – псевдонимы группы иконок и иконки соответственно;

IconFile.ico – файл, в котором содержится иконка. При компиляции приложения она будет записана в сам исполняемый файл.

Для загрузки иконки из ресурсов может быть использована как функция LoadImage (A/W) рассмотренная ранее (с параметром uType равным IMAGE_ICON) так и функция LoadIcon (A/W) из той же библиотеки. Прототип последней представлен ниже:

```
HICON WINAPI LoadIcon(
  __in_opt  HINSTANCE hInstance, //Дескриптор модуля, из которого
                                 //загружается иконка
  __in      LPCTSTR lpIconName   //Идентификатор группы иконок[1]
);
```

В случае успеха данная функция вернет дескриптор загруженной иконки, а в случае ошибки значение NULL.

Уничтожать загруженную иконку следует с помощью функции DestroyIcon из библиотеки user32.dll. Вот ее прототип:

```
BOOL WINAPI DestroyIcon(
  __in  HICON hIcon   //Дескриптор уничтожаемой иконки
);
```

В случае успеха данная функция возвращает ненулевое значение, а в случае ошибки значение нуль.

Важным моментом является то, что данную функцию нельзя использовать для иконок загруженных функцией LoadImage (A/W) с флагом LR_SHARED. Такие иконки будут уничтожены при выгрузке из памяти модуля, из которого они были загружены.

Ниже приводится исходный код пограммы демонстрирующий работу с иконкой. В ней создается окно со своей иконкой, хранящейся в ресурсах приложения.

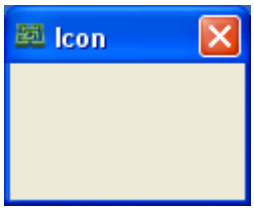
```
format PE GUI 4.0
entry start
include 'G:\FASM\INCLUDE\win32a.inc'
ID_ICONS = 1
ID_ICON  = 2
section '.code' code readable executable
start:
        invoke GetModuleHandle, 0
        mov [wc.lpfnWndProc], wndproc
        mov [wc.hInstance], eax
        mov [wc.lpszClassName], _ClassName
        ;Загружаем иконку из ресурсов
        invoke LoadIcon, [wc.hInstance], ID_ICONS
        mov [wc.hIcon], eax
        mov [wc.hbrBackground], COLOR_BTNSHADOW
        invoke RegisterClass, wc
```

[1] Да, именно группы иконок, а не иконки. В функцию LoadImage также следует передавать именно идентификатор группы иконок.

```
            invoke     CreateWindowEx,     0,     _ClassName,     _WindowName,
WS_VISIBLE+WS_CAPTION+WS_SYSMENU,   0,0,   100,   100,   HWND_DESKTOP,   NULL,
[wc.hInstance], NULL
            ;Цикл обработки сообщений
    StartLoop:
            invoke GetMessage, msg, NULL, 0, 0
            cmp eax, 1
            jb ExitProgramm
            jne StartLoop

            invoke TranslateMessage, msg
            invoke DispatchMessage, msg
            jmp StartLoop
    ExitProgramm:
            ;Уничтожаем ранее загруженную иконку
            invoke DestroyIcon, [wc.hIcon]
            invoke ExitProcess, 0
            ;Оконная процедура
            proc wndproc, hwnd, uMsg, wParam, lParam
                 cmp [uMsg], WM_CLOSE
                 jz msgCloseWindow
                 cmp [uMsg], WM_DESTROY
                 jz msgDestroyWindow
                 jmp ExitWndProc
            msgCloseWindow:
                 invoke DestroyWindow, [hwnd]
                 jmp ExitWndProc
            msgDestroyWindow:
                 invoke PostQuitMessage, 0
            ExitWndProc:
                 invoke DefWindowProc, [hwnd], [uMsg], [wParam], [lParam]
                 ret
            endp

    section '.data' data readable
    _ClassName db 'MyWindowClass', 0
    _WindowName db 'Icon', 0

    section '.rdata' data readable writeable
    wc WNDCLASS
    msg MSG

    section '.idata' import data readable writeable
    library kernel, 'kernel32.dll',\
            user  , 'user32.dll'  ,\
            gdi   , 'gdi32.dll'

    import kernel,\
           ExitProcess    , 'ExitProcess',\
           GetModuleHandle, 'GetModuleHandleA'


    import user,\
           RegisterClass  , 'RegisterClassA',\
           CreateWindowEx , 'CreateWindowExA',\
           DestroyWindow  , 'DestroyWindow',\
           PostQuitMessage, 'PostQuitMessage',\
           DefWindowProc  , 'DefWindowProcA',\
           LoadIcon       , 'LoadIconW',\
           DestroyIcon    , 'DestroyIcon',\
           GetMessage     , 'GetMessageA',\
           TranslateMessage,'TranslateMessage',\
           DispatchMessage, 'DispatchMessageA'
```

```
section '.rsrc' resource data readable
;типы ресурсов
directory RT_GROUP_ICON, Icons,\
          RT_ICON, my_icon
;язык группы иконок
resource Icons,\
         ID_ICONS, LANG_NEUTRAL, myicons
;язык иконки
resource my_icon,\
         ID_ICON, LANG_NEUTRAL, myicon
;описываем саму иконку из группы иконок
icon myicons, myicon, 'IconFile.ico'
```

Если все сделано правильно, то при запуске этой программы вы должны увидеть приблизительно следующее окно с нашей иконкой в левом верхнем углу:



Строка

В ресурсах можно описывать и Unicode строки. В этом случае тип ресурса равен RT_STRING. Пример описания простой строки «String» представлен ниже:

```
;Тип ресурса
directory RT_STRING, mystring
;Язык
resource mystring,\
         ID_STRING, LANG_NEUTRAL, my_string
;Сама строка
resdata my_string
       dw 0000,0006
       du 'String', 0
endres
```

Макрос resdata…endres используется для задания фактического содержимого описываемого ресурса. Из примера видно, что строковый ресурс представляет собой два слова, за которыми идет сама Unicode строка с нулем на конце. Теперь о словах.

Первое слово задает индекс символа в строке, с которого она начинается (нумерация с нуля).

Второе слово задает количество символов в строке, включая завершающий нуль.

Эти два слова позволяют выводить не всю строку, а только ее часть.

Для того чтобы прочитать строку из ресурсов используется функция LoadString (A/W) из библиотеки user32.dll. Вот ее прототип:

```
int WINAPI LoadString(
  __in_opt  HINSTANCE hInstance,//Дескриптор модуля, из которого читается
                                //строка
  __in      UINT uID,           //Идентификатор ресурса строки
  __out     LPTSTR lpBuffer,    //Адрес буфера, в который следует
                                //записать читаемую строку
  __in      int nBufferMax      //Максимальное количество символов,
                                //записываемых в буфер
);
```

В случае успеха данная функция возвращает количество символов записанных в буфер lpBuffer, а в случае ошибки ноль.

Ниже приводится исходный текст программы, демонстрирующий работу со строкой, заданной в ресурсах:

```
format PE GUI 4.0
entry start
include 'G:\FASM\INCLUDE\win32a.inc'
ID_STRING=1       ;Идентификатор ресурса строки
LengthString =15h ;Длина строки с нулевым символом
section '.code' code readable executable
start:
      invoke GetModuleHandle, 0
      ;Загружаем строку из ресурсов
      invoke LoadString, eax, ID_STRING, _lpstr, LengthString
      invoke MessageBox, HWND_DESKTOP, _lpstr, _lpstr, MB_OK
      invoke ExitProcess, 0

section '.rdata' data readable writeable
_lpstr rb  LengthString*2 ;буфер под загружаемую строку


section '.idata' import data readable writeable
library kernel, 'kernel32.dll',\
        user  , 'user32.dll'
import kernel,\
       GetModuleHandle, 'GetModuleHandleA',\
       ExitProcess, 'ExitProcess'
import user,\
       LoadString, 'LoadStringW',\
       MessageBox, 'MessageBoxW'
;Ресурсы
section '.rsrc' resource data readable
directory RT_STRING, mystring
resource mystring,\
         ID_STRING, LANG_NEUTRAL, my_string

resdata my_string
       dw 0000, LengthString
       du 'String from resources', 0
endres
```

Ресурсы, загруженные из res файла

Если вы не хотите вручную описывать ресурсы в тексте программы, вы можете загружать их из заранее подготовленного res файла. Сам такой файл можно подготовить в любой программе по редактированию ресурсов (лично я использовал программу Restorator 2007). Для указания того, что секция ресурсов загружается из внешнего файла нужно объявить эту секцию приблизительно следующим образом:

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

Здесь ‘MyDialogRes.res’ – это заранее подготовленный res файл, из которого Flatasm загрузит секцию ресурсов.

Для того чтобы продемонстрировать как это работает, напишем небольшую программу. В этой программе из шаблона, хранящегося в ресурсах, будет создаваться диалоговое окно[1]. Исходный текст программы представлен ниже:

```
format PE GUI 4.0
entry start
include 'D:\FASM\INCLUDE\win32a.inc'
section '.code' code readable executable

start:
        invoke GetModuleHandle, 0
        ;Создаем модальное диалоговое окно
        invoke DialogBoxParam, eax, _resname, HWND_DESKTOP, DialogProc, 0
        invoke ExitProcess, 0
;Диалоговая процедура
proc DialogProc hwnddlg, msg, wparam, lparam
     xor eax, eax
     cmp [msg], WM_CLOSE
     je FreeDialog
     ret
FreeDialog:
     invoke EndDialog, [hwnddlg], 0
     xor eax,eax
     ret
endp

section '.data' data readable
;Строковое наименование нужного нам ресурса
_resname db 'RES1', 0

section '.idata' import data readable writeable
 library kernel, 'KERNEL32.DLL',\
         user  , 'USER32.DLL'

 import kernel,\
        GetModuleHandle, 'GetModuleHandleA',\
        ExitProcess    , 'ExitProcess'

 import user,\
```

[1] Подробнее об этом читай в главе, посвященной диалоговым окнам и описанию их шаблонов

```
        DialogBoxParam, 'DialogBoxParamA',\
        EndDialog     , 'EndDialog'
;Шаблон диалогового окна
section '.rsrc' resource from 'MyDialogRes.res' data readable
```

Также в этом примере демонстрируется идентификация ресурсов не по идентификатору (как мы привыкли это делать), а по наименованию (RES1).