МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
ДОНЕЦКОЙ НАРОДНОЙ РЕСПУБЛИКИ
ГОУ ВПО «ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ»
ФИЗИКО-ТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

В. Н. КОТЕНКО, Ю. В. КОТЕНКО

ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ НИЗКОГО УРОВНЯ

КУРС ЛЕКЦИЙ

Донецк
ГОУ ВПО «ДонНУ»
2016

УДК 004.431
ББК З973.2-018.1я73-2
К 731

Авторы:
В. Н. Котенко, старший преподаватель кафедры;
Ю. В. Котенко, ассистент кафедры

Ответственный за выпуск:
Т. В. Шарий, канд. техн. наук, доц.

Утверждено на заседании ученого совета
физико-технического факультета ГОУ ВПО «ДонНУ»
Протокол № 1 от 27.09.2016

К 731 Котенко В. Н.

Программирование на языках низкого уровня: курс лекций для студентов укрупненной группы направлений подготовки 09.00.00 «Информатика и вычислительная техника» направления подготовки 09.03.01 «Информатика и вычислительная техника» квалификационного уровня «Бакалавр» / В. Н. Котенко. – Донецк: ГОУ ВПО «ДонНУ», 2016. – 103 с.

Курс лекций содержит подробное описание каждой темы и примеры, разъясняющие теоретический материал.

Предназначен для студентов всех направлений подготовки, изучающих программирование на языках низкого уровня.

УДК 004.431
ББК З973.2-018.1я73-2

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данный курс лекций предназначен для формирования знаний студента о фундаментальных понятиях, общих принципах организации и функционирования программ на машинно-ориентированных языках, методах и средствах проектирования и построения программ на языках программирования низкого уровня и предполагает наличие у студентов знаний об организации и функционировании ЭВМ, а также современных языках программирования. В курсе лекций использованы таблицы, схемы, графики, а также примеры программных кодов, которые разъясняют и закрепляют теоретический материал.

В издании излагаются основные понятия и принципы программирования на языках низкого уровня. Рассматриваются такие темы, как исполняемые компоненты Windows, системные библиотеки и компиляторы ассемблера, структура программы, компиляция и компоновка программ, процедуры, консольные и оконные приложения, элементы управления окна, ресурсы приложений, работа с файлами в системе Windows, отладка программ.

Курс лекций построен таким образом: в начале каждой темы приводятся ключевые определения, перечисляются основные функции соответствующей темы программирования на языках программирования низкого уровня, а затем детально рассматривается механизм их реализации.

В конце курса лекций приводится список рекомендуемой литературы для изучения дисциплины.

Тема 1 Исполняемые компоненты Windows. Системные библиотеки. Компиляторы ассемблера

1.1 Преимущества знания низкоуровневых языков

В настоящее время создано множество языков программирования, удобных для решения любых задач. Большинство из них являются языками высокого уровня.

Ассемблер же, самый древний язык программирования, – это язык программирования низкого уровня. До него программирование осуществлялось в машинных кодах. Спустя годы ассемблер вернулся в десятку самых популярных языков программирования.

Преимущества, которые дает знание ассемблера:

1. Глубокое понимание работы компьютера и операционной системы

Если программист пишет программу на языке высокого уровня, знание ассемблера помогает понять, как она будет выполняться, как хранятся переменные, как вызываются функции. А это позволяет избежать очень многих ошибок. Есть люди, которые знают программирование только на уровне языка, т. е. знают, что надо написать, чтобы получить результат. А сам внутренний механизм работы для них остается непонятным. Человек, владеющий ассемблером, будет лучше программировать и на других языках.

2. Максимальная гибкость при работе с аппаратными ресурсами

Используя ассемблер, можно делать с аппаратурой компьютера все что угодно. Языки же высокого уровня ограничены компилятором и используемыми библиотеками. Такие современные языки, как Java и C#, вообще не позволяют работать с аппаратными ресурсами и операционной системой напрямую.

3. Оптимизация программ по скорости выполнения

Современные компиляторы неплохо оптимизируют код, поэтому писать на ассемблере все подряд не имеет смысла. Однако если пишется,

например, программа для шифрования или архивации больших файлов, то применение ассемблера позволит в несколько раз увеличить скорость ее выполнения. Причем достаточно реализовать на ассемблере небольшой критически важный участок программы, который производит вычисления или сложные преобразования, а интерфейс может быть написан на языке высокого уровня.

4. Оптимизация программ по размеру кода

Программа на ассемблере значительно меньше по размерам аналогичной программы на другом языке программирования. Для современных персональных компьютеров и серверов с терабайтными дисками и гигабайтами памяти это не играет большой роли. Но для микроконтроллеров, где всего несколько килобайт памяти, маленький размер программы очень важен. Чем меньше программа, тем меньше памяти требуется и тем проще и дешевле будет используемая микросхема.

5. Дизассемблирование и отладка

Знание ассемблера позволяет открыть дизассемблером любую программу, заглянуть внутрь и изучить механизм ее работы, даже не имея исходников. Ассемблер также незаменим при отладке программ. Иногда бывают ошибки и в самих компиляторах и, казалось бы, корректно написанный код выполняется вовсе не так, как предполагалось. Чтобы обнаружить такую ошибку надо посмотреть, во что скомпилировался код, а разобраться в этом без ассемблера невозможно.

1.2 Основные сведения об операционной системе Windows

Windows – это операционные системы защищенного режима (32-битные и 64-битные версии). Система безопасности этих операционных систем построена на разделении кода пользователя и системного кода. Код пользователя работает в режиме «user mode» и на него наложено множество ограничений. Системный код работает в режиме «kernel mode» и почти ничем не ограничен.

Основанная единица выполнения в Windows – это *поток* (по терминологии защищенного режима – задача). Потоки объединяются в процессы. В общем случае одна программа – это один процесс. В процессе может быть сколько угодно потоков. Каждый процесс обособлен от всех остальных. Это достигается за счет того, что у каждого процесса своя собственная виртуальная память. Тем не менее, если программе надо получить доступ в памяти других процессов, она может осуществить это через специальные системные сервисы.

Диспетчеризация потоков осуществляется на основе приоритетов. Приоритеты у потоков являются динамическими, т. е. могут меняться в зависимости от того, что делает поток. В некоторый момент времени процессором всегда выполняется поток с наибольшим приоритетом.

1.2.1 Память в Windows

Каждый процесс имеет свое собственное виртуальное адресное пространство. Адресное пространство любого процесса разбито на две равные части: *память процесса* и *память системы*. Младшие 2 Гбайт памяти являются памятью процесса, старшие 2 Гбайт – памятью системы. Память системы одна для всех процессов, она недоступна из режима пользователя «user mode» даже для чтения. Любое обращение к ней приводит к ошибке доступа и завершению приложения.

В некоторых случаях под память процесса выделяется 3 Гбайт памяти, а под память системы – 1 Гбайт. Так делается тогда, когда используются приложения, требовательные к памяти, которым 2 Гбайт памяти недостаточно. Для 64-битных систем этот метод потерял актуальность.

Адреса в диапазоне 0h – FFFFh никому не доступны. Эта память нужна для выявления нулевых указателей. Любой указатель, значение которого меньше 100000h, считается нулевым. Таким образом, каждому

процессу в Win32 в общем случае доступно 2 Гбайт (за вычетом 64 Кбайт) виртуальной памяти.

1.2.2 Исполняемые компоненты Windows

В операционных системах (ОС) Windows имеется несколько типов исполняемых файлов. Все они имеют формат *PE (Portable Executable)*. Наиболее часто используемые исполняемые компоненты в Windows: EXE (приложение), DLL (динамическая библиотека), SYS (драйвер).

EXE-файлы – это самый распространенный тип исполняемых файлов в Windows. В них находятся программы.

DLL-файлы – это динамически загружаемые библиотеки. В них хранятся функции и процедуры, которые могут использовать другие исполняемые компоненты.

SYS-файлы – это файлы драйверов режима ядра. В них находится код нулевого кольца операционной системы.

Файлы формата *PE* состоят из *заголовка* и *секций*. Секция в *PE*-файле – это его основная составляющая единица. В заголовке содержатся основные характеристики файла и таблица секций.

Рассмотрим базовые характеристики исполняемых файлов. Самые главные характеристики файла – *точка входа* и *база образа*, т. е. указание, по какому адресу должен быть загружен данный модуль.

При создании исполняемого файла компоновщик должен вставить в код программы вместо меток конкретные адреса и подразумевает, что этот код будет загружен по некоторому базовому адресу. Загрузчик Windows должен знать, по какому адресу надо загрузить данный исполняемый файл, и именно для этого используется поле базы образа в заголовке исполняемого файла. Очень часто при загрузке файлов DLL и SYS адрес, указанный в поле базы образа, является уже занятым или просто недоступным. Если при загрузке исполняемого файла адрес, указанный в поле базы образа, уже занят, то он грузит файл по другому адресу и при

этом загрузчику надо подправить в коде программы все обращения к данным. Для этого загрузчику будут нужны *релокейшены*. *Релокейшены* содержат информацию о командах, в которых есть обращения к памяти для поправки адресов. Точкой входа содержит адрес, с которого начнется выполнение исполняемого файла.

Каждый файл может иметь таблицу импорта и экспорта. С помощью *таблицы импорта* исполняемый файл может импортировать функции, которые находятся в других модулях, загруженных в текущее адресное пространство, и использовать эти функции как свои. Для того чтобы другие модули могли использовать функции из данного модуля, адреса этих функций должны быть прописаны в таблице экспорта.

Как было сказано выше, в заголовке *PE*-файла содержится таблица секций, она описывает каждую секцию в *PE*-файле: начало данных в секции, размер данных, адрес, куда должна быть спроецирована данная секция и ее характеристики. Наиболее часто в секциях находятся данные, код, импорты, экспорты и релокейшены.

Итак, файлы с расширением .EXE являются обычными программами. В 99,99 % случаев *EXE-файл* представляет процесс, в память которого он загружен. Файлы с расширением .DLL являются библиотеками, где содержатся функции, которые могут использовать другие программы или другие DLL. Файлы с расширением .SYS являются драйверами режима ядра. Код, содержащийся в них, выполняется на нулевом уровне привилегий, в режиме ядра. В файлах DLL и SYS точка входа указывает на инициализирующую функцию.

1.2.3 Системные библиотеки и подсистемы

В коде Win32, выполняемом в режиме пользователя, запрещены любые прямые обращения к устройствам и портам ввода-вывода. Это значит, что любые обращения к портам ввода-вывода, вызов прерываний и выполнение привилегированных инструкций приведут к ошибке и

завершению программы. Обращение к памяти, на которую спроецированы регистры устройств, обращение к другим важным областям памяти (например, 0B8000h) ничего не даст. Без обращения к внешним устройствам и портам ввода-вывода польза от программ, работающих в третьем кольце, нулевая. Для того чтобы они могли обратиться к внешним устройствам и наладить взаимодействие с «окружающим миром», операционная система предоставляет программам *API-функции (Application Program Interface)*.

Программирование в Windows основывается на использовании интерфейса прикладного программирования *API*. Он предоставляет программисту набор готовых классов, функций, структур и констант. Их количество составляет около двух тысяч. *API-функции* обеспечивают взаимодействие приложения с внешними устройствами и ресурсами операционной системы.

Все *API-функции* содержатся в системных DLL-библиотеках. Самые главные из DLL-библиотек:

1) *kernel32.dll* – взаимодействие с системой;

2) *user32.dll* – пользовательский интерфейс;

3) *gdi32.dll* – графика.

Библиотека *kernel32.lib* предназначена для работы с объектами ядра операционной системы, ее функции позволяют управлять памятью и другими системными ресурсами. Библиотека *user32.lib* отвечает за окна и интерфейс пользователя, в ней сосредоточены функции для управления окнами, обработки сообщений, работы с меню, таймерами и т.п. Библиотека *gdi32.dll* обеспечивает графический интерфейс операционной системы. В состав библиотеки входят функции управления выводом на экран монитора, управления выводом принтера, функции для работы со шрифтами и т. п.

Функции библиотеки *kernel32.dll* в основном являются оболочками вокруг функций из *ntdll.dll*.

Функции из библиотеки *ntdll.dll* являются переходниками к функциям ядра Windows. Эти функции принимают параметры, подготавливают их к вызову команды «`sysenter`». *Библиотека ntdll.dll – важнейший компонент пользовательской подсистемы Windows и является основополагающей для всех подсистем*, так как именно через нее пользовательский код может взаимодействовать с кодом ядра. Эта библиотека загружается в память любого процесса одной из первых и всегда по одному и тоже адресу. *Библиотека kernel32.dll является основополагающей для подсистемы Win32*. Она загружается во все процессы Win32 одной из первых (после *ntdll*) и всегда по одному и тому же адресу.

1.2.4 Модель вызова функций в Win32

В системах Win32 при вызове всех системных функций используется модель вызова *stdcall*.

Согласно этой модели параметры функций передаются через стек в обратном порядке. При этом за очистку стека от параметров ответственна вызываемая функция. Например, если у функции есть три параметра, то вызов по соглашению *stdcall* будет выглядеть так:

```
Push param3
Push param2
Push param1
Call FunctionAddr
```

Результат выполнения функции будет содержаться в регистре EAX.

При использовании *API-функций* следует помнить, что они сохраняют значение не всех регистров общего назначения. *Соглашение stdcall* предусматривает сохранение содержимое регистров EBX, ESI, EDI и EBP. При написании функций обратного вызова также надо обязательно сохранять содержимое этих регистров, поскольку код системных функций не ожидает их изменения.

1.2.5 Выполнение программ в Win32

При загрузке исполняемого файла загрузчик Windows выполняет такие действия:

1) создает для файла виртуальное адресное пространство размером 4 Гбайт, причём нижние 2 Гбайт из них доступны приложению;

2) загружает системные библиотеки *ntdll.dll*, *kernel32.dll* и библиотеки, указанные в таблице импорта файла;

3) создает первичный поток процесса, который начинает свое выполнение с точки входа программы.

Во время выполнения процесса, вернее, его потоков, ему запрещены какие-либо обращения к портам ввода-вывода и вызов каких-либо прерываний, запрещена работа с привилегированными регистрами и выполнение привилегированных команд. Чтобы программы могли работать с внешними устройствами, Windows предоставляет им *API-функции*, позволяющие им работать с внешними устройствами, взаимодействовать с системой, а также друг с другом. *API-функции* находятся в системных библиотеках. Каждая функция, которая работает с ресурсом, охраняемым системой (файлы, процессы, устройства и т. д.), вызывает соответствующую функцию ядра системы.

Резюмируем все вышесказанное. Операционная система помещает программу в некоторое изолированное адресное пространство, разрешая ей взаимодействовать с «внешним миром» посредством функций (системных сервисов), которые сама же и предоставляет. Операционная система избавляет программиста, который пользуется ассемблером, от множества забот, которые вообще-то не должны его касаться. Например, от работы с системными регистрами и структурами, взаимодействия с внешними устройствами, реализации работы с файловой системой и т. д.

В связи с этим программирование на ассемблере под Win32 намного легче. Например, при работе с файлами программисту не нужно заботиться о том, какая же модель жесткого диска установлена на компьютере:

достаточно просто вызывать функции, которые предоставляет операционная система, а она уже сама разберется со всеми проблемами.

1.2.6 Типы структур программ Windows

Возможны *три типа структур программ* для Windows:

1) диалоговая (основное окно – диалоговое);

2) консольная;

3) классическая (GUI, graphical user interface).

Диалоговые приложения для Windows имеют минимальный интерфейс связи с пользователем и передают информацию посредством диалоговых окон (например, окна сообщения `MessageBox`).

Консольные приложения представляет собой программу, работающую в текстовом режиме. Работа консольного приложения напоминает работу MS-DOS. Но это лишь внешнее впечатление. Консольное приложение обеспечивается специальными функциями Windows. Диалог с пользователем ведется посредством консоли. Примером консольного приложения является Far.

Оконные приложения строится на базе набора *функций API*, составляющих графический интерфейс пользователя GUI. Главным элементом такого приложения является окно. Окно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы, по сути, также являются окнами, но обладающими особыми свойствами. События, происходящие с этими элементами (и самим окном), приводят к приходу сообщений в процедуру окна. Диалог с пользователем ведется посредством графического интерфейса.

1.3 Компиляторы ассемблера

Ассемблер – это машинно-ориентированный язык программирования, позволяющий использовать все ресурсы ЭВМ в процессе написания

программы, а также компилятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке. Основной его недостаток – ассемблер для каждого микропроцессора индивидуален. Под каждую архитектуру процессора и под каждую операционную систему (или их семейство) есть свой ассемблер. Существуют также так называемые *«кросс-ассемблеры»*, позволяющие на машинах с одной архитектурой (или в среде одной операционной системы) ассемблировать программы для другой целевой архитектуры (или другой операционной системы) и получать исполняемый код в формате, пригодном к исполнению на целевой архитектуре или в среде целевой операционной системы.

На языке ассемблера пишут:

1) программы, требующие максимальной скорости выполнения: ядра операционных систем реального времени, основные компоненты компьютерных игр и критичные по времени участки программ;

2) программы, взаимодействующие с внешними устройствами: драйверы, программы, работающие напрямую с портами, звуковыми и видеокартами;

3) программы, использующие полностью возможности процессора: ядра многозадачных операционных систем, серверы;

4) программы, полностью использующие возможности операционной системы: вирусы, антивирусы, защита от несанкционированного доступа, программы обхода защиты от несанкционированного доступа.

Достоинства языка ассемблера: максимальная оптимизация программ как по скорости выполнения, так и по размеру; максимальная адаптация под соответствующий процессор.

Недостатки языка ассемблера: трудоемкость написания программы больше, чем на языке высокого уровня; трудоемкость чтения;

непереносимость на другие платформы, кроме совместимых; ассемблер малопригоден для совместных проектов.

1.3.1 Ассемблеры для операционной системы DOS

Наиболее известными ассемблерами для операционной системы DOS являлись пакеты *Borland Turbo Assembler (TASM)* и *Microsoft Macro Assembler (MASM)*. Также в свое время был популярен простой ассемблер A86. Изначально они поддерживали лишь 16-битные команды (до появления процессора Intel 80386). Более поздние версии *TASM* и *MASM* поддерживают и 32-битные команды, а также все команды, введенные в более современных процессорах, и системы команд, специфические для конкретной архитектуры (такие, как MMX, SSE, 3DNow! и т. д.).

Особенностью *TASM* всегда была быстрота трансляции и обширный набор макросредств. Кроме поддержки стандартного режима, совместимого с ассемблером *MASM*, данный ассемблер поддерживал также расширенный режим «IDEAL». Несмотря на то что уже много лет *Turbo Assembler* не поддерживается корпорацией Borland, он до сих пор популярен в кругах программистов.

1.3.2 Ассемблеры для операционной системы Windows

Вместе с операционной системой Windows появилось расширение *TASM*, именуемое *TASM32*, позволившее создавать программы для выполнения в среде Windows. Последняя известная версия *TASM* – пятая, включая различные к ней дополнения. Но официально развитие программы полностью остановлено.

Ассемблер MASM предназначен для x86-микропроцессоров и ведет свою историю с 1981 г. Первоначально был создан корпорацией Microsoft для программирования в операционной системе MS-DOS. Долгое время он конкурировал по популярности с ассемблером *TASM*, пока последний ушел в небытие. В дальнейшем *MASM* был адаптирован для программирования в операционной системе Windows. Последние версии включены в наборы

DDK (Driver Development Kit). В настоящее время корпорация Microsoft распространяет пакет *MASM* в составе пакета Visual Studio.NET (выпущена 64-битовая версия *MASM* и 64-битовый компоновщик *LINK.EXE*). Для более удобного создания программ на ассемблере для Windows, появился пакет, названный *MASM32*, поддерживаемый независимыми группами разработчиков (сетевая поддержка: http://www.masm32.com/). Пакет содержит большое количество примеров, библиотек, документации и утилит. Де-факто в настоящее время *MASM* среди других ассемблеров является законодателем мод в области программирования в операционной системе Windows.

1.3.3 Ассемблеры для операционной системы Linux

Несколько иначе обстоит ситуация с ассемблерами для другой известной операционной системы Linux. В ее состав входит компилятор GCC (GNU Compiler Collection), включающий в себя ассемблер *GAS (GNU Assembler)*. Поскольку *GAS* был разработан для поддержки компиляторов Unix, он использует стандартный синтаксис AT&T, который несколько отличается от большинства ассемблеров, основанных на синтаксисе Intel. Ассемблер *GAS* включен в стандартный комплект операционных систем линеек Unix и Linux и может быть запущен посредством утилиты GCC.

1.3.4 Ассемблер для операционной системы ReactOS

Ассемблер RosAsm – 32-битовый Win32-x86ассемблер, выпущенный согласно лицензии GNU GPL.

Проект RosAsm поддерживает операционную систему ReactOS. RosAsm производится вместе с интегрированной средой программирования, в которую входят ассемблер, компоновщик, отладчик, редактор ресурсов, дизассемблер.

1.3.5 Переносимые ассемблеры

Также существует открытый проект ассемблера, версии которого доступны под различные операционные системы и который позволяет получать объектные файлы для этих систем. Называется этот ассемблер *NASM (Netwide Assembler)*. *Ассемблер NASM* – свободно распространяемый ассемблер (лицензия GNU LGPL). Ассемблер может быть использован для написания 16-, 32-, 64-битовых программ для различных платформ. *NASM* был создан Саймоном Тэтхемом совместно с Юлианом Холлом и в настоящее время развивается маленькой командой разработчиков. Первоначально он был выпущен согласно его собственной лицензии, но позже эта лицензия, после многочисленных проблем, была изменена на GNU LGPL. Ассемблер поддерживает множество форматов для различных операционных систем, что позволяет вести разработку для одной операционной системы, используя другую операционную систему. Сетевая поддержка *NASM* расположена на сайтах http://www.opennet.ru/docs/RUS/nasm и http://www.nasm.us.

Ассемблер YASM – это переписанная с нуля версия *NASM* под лицензией BSD (с некоторыми исключениями). В настоящее время его развивает Питер Джонсон и Майкл Ерман. Будучи динамично развивающимся проектом, *YASM* предлагает прямую поддержку пользователей, которые ищут новые особенности. Кроме Intel-синтаксиса, применяемого, например, в *NASM*, *YASM* также поддерживает и AT&T-синтаксис, распространенный в Unix-системах. Ассемблер *YASM* построен модульно, что позволяет легко добавлять новые формы синтаксиса, препроцессоры и т. п. Сетевая поддержка расположена на сайте http://www.tortall.net/.

FASM (Flat Assembler) – быстро развивающийся и завоевывающий популярность мультиплатформенный ассемблер, используют настоящие ценители языка. Автор *FASM* Томаш Грыштар. Этот многопроходной ассемблер распространяется с исходным кодом. Есть версии для операционных систем KolibriOS (написана на самом *FASM*), Linux и

Windows. Использует Intel-синтаксис. *FASM* также поддерживает инструкции AMD64.

Ассемблер написан на себе самом, поэтому является одним из самых быстрых. *FASM* обладает такими достоинствами: сильный макроязык, поддержка разнообразных форматов выходных файлов и несколько измененный в лучшую сторону, но немного непривычный для старой школы синтаксис. В большинстве случаев FASM позволяет обходиться без компоновщика. Но с ним можно использовать и обычные компоновщики (например, *LINK.EXE*), поскольку ассемблер поддерживает основные форматы объектных модулей. Сетевая поддержка ассемблера расположена на сайтах http://flatassembler.net и http://fasmassembler.narod.ru.

1.3.6 AVR ассемблер

Микроконтроллеры прочно вошли в нашу жизнь. На микроконтроллерах можно собрать индикаторы, вольтметры, приборы для дома (устройства защиты, коммутации, термометры), металлоискатели, игрушки, роботы и т. п. В настоящее время широкое распространение получили микроконтроллеры AVR фирмы Atmel. Объем памяти микроконтроллеров не превышает нескольких Кбайт. Поэтому размер программ для них должен быть минимизирован.

AVR ассемблер – родной язык программирования микроконтроллеров AVR. Преимущества программирования на ассемблере:

1) полный и непосредственный контроль над процессором;

2) программы занимают минимум объема памяти всех видов;

3) максимум быстродействия.

На данный момент существуют два компилятора производства Atmel. Вторая версия – попытка исправить не очень удачную первую.

Тема 2 Структура программы. Компиляция и компоновка

2.1 Использование визуальной интегрированной среды разработки приложений для создания исполняемого файла

Программа, написанная на ассемблере *MASM*, может состоять из нескольких частей – *модулей*. В каждом модуле могут быть определены один или несколько сегментов данных, стека и кода.

Программа должна включать один главный модуль, с которого начинается ее выполнение. Модуль может содержать программные сегменты, сегменты данных и стека, объявленные при помощи соответствующих директив. Перед объявлением сегментов нужно указать модель памяти при помощи директивы `.model`.

Модель памяти *flat*, которую мы будем использовать, предполагает несегментированную конфигурацию программы. Данные и код размещены в одном сегменте. Для разработки программы для модели *flat* перед директивой `.model flat` следует разместить одну из директив: `.386, .486, .586` или `.686`. Желательно указывать тот тип процессора, который используется в машине. Операционная система автоматически инициализирует сегментные регистры при загрузке программы. Адресация данных и кода является ближней (near).

Рассмотрим пример простейшей программы на языке *MASM*:

```
; Простейшая программа First_Program.asm
.686p
.model flat, stdcall
option casemap : none
.data
.code
start:
ret
end start
```

В данной программе всего одна команда микропроцессора – «`ret`». В общем случае эта команда используется для выхода из процедуры.

Остальная часть программы относится к работе транслятора.

`.686P` – разрешены команды защищенного режима Pentium Pro. Директива выбирает поддерживаемый набор команд ассемблера, указывая модель процессора. Буква P, указанная в конце директивы, сообщает транслятору о работе процессора в защищенном режиме.

`.model flat, stdcall` – *flat* (плоская модель памяти), *stdcall* (соглашение, используемое для вызова функций WinAPI). Аргументы функций передаются через стек справа налево. Очистку стека производит вызываемая функция.

`option casemap : none` – включить чувствительность к прописным или строчным символам в именах меток и процедур.

`.data` – блок программы, содержащий данные. В MS-DOS это называлось сегментом. В Windows (для прикладной программы) сегменты отсутствуют, точнее есть один большой плоский сегмент. Такие блоки здесь называются *секциями*. Им можно задать различные свойства. Например, запретить запись в нее, или сделать секцию доступной для других программ.

Данная программа не использует стек, поэтому секция `.stack` отсутствует.

`.code` – блок программы, содержащей код.

`start` – метка.

`end start` – конец программы и сообщение компилятору, что начинать выполнение программы надо с метки `start`. Каждая программа должна содержать директиву `end`, отмечающую конец исходного кода. Все строки, которые следуют за директивой `end`, игнорируются. Метка, указанная после директивы `end`, сообщает транслятору имя главного модуля, с которого начинается выполнение программы. Если программа содержит один модуль, метку после директивы `end` можно не указывать.

Чтобы представленный выше текст превратить в исполняемый файл, воспользуемся IDE (Integrated Development Environment) Visual Studio 2015. В среде Visual Studio 2015 для создания скомпилированной

программы first_program.exe необходимо выполнить шаги:

1) создать пустой проект с именем, например, First_Program:

Пройти по цепочке «Файл» – «Создать» – «Проект» – «Шаблоны» – «Visual C++» – «Пустой проект». В поле ввода «Имя» указать First_Program, а в поле ввода «Расположение» – полное имя папки, куда будут помещены файлы проекта. Нажать кнопку «ОК»;

2) добавить в проект пустой файл с именем First_Program и расширением `.asm`:

Пройти по цепочке «Проект» – «Добавить новый элемент» – «Файл C++ (.cpp)». В поле ввода «Имя» ввести First_Program.asm. Нажать кнопку «Добавить»;

3) в окно ввода кода программы ввести код приведенной выше простейшей программы. Сохранить файл;

4) в окне «Обозреватель решений» кликнуть правой кнопкой мыши по имени проекта First_Program (выделено жирным шрифтом). Далее пройти по цепочке «Зависимости сборки» – «Настройки сборки» и в доступных файлах настройки сборки отметить галочкой masm(.targets,.props). Нажать кнопку «ОК»;

5) в окне «Обозреватель решений» кликнуть правой кнопкой мыши по имени проекта First_Program.asm. Выбрать пункт меню «Свойства». На странице свойств файла First_Program.asm в левой части диалогового окна выбрать «Свойства конфигурации», «Общие». В правой части диалогового окна кликнуть пункт «Тип элемента» и в выпадающем списке компиляторов выбрать «Microsoft Macro Assembler», т.е. указать что файл будет компилироваться компилятором *MASM*;

6) теперь надо указать параметры компоновки. В окне «Обозреватель решений» снова кликнуть правой кнопкой мыши по имени проекта First_Program (выделено жирным шрифтом). На странице свойств проекта First_Program пройти по цепочке «Свойства конфигурации» –

«Компоновщик» – «Система». В правой части диалогового окна выбрать пункт «Подсистема» и в выпадающем списке указать Windows(/SUBSYSTEM:WINDOWS). Нажмите кнопку «ОК»;

7) для компиляции и компоновки пройти по цепочке «Сборка» – «Собрать решение» (или комбинация клавиш Ctrl–Shift–B). В результате создастся исполняемый файл «First_Program.exe», который можно запускать на выполнение;

8) для запуска программы выбрать «Отладка» – «Запуск без отладки» (или комбинация клавиш Ctrl–F5).

Примечание – Так как программа ничего не делает, то выполнение исполняемого файла не даст никакого визуального эффекта.

При использовании IDE Visual Studio 2015 для создания исполняемого файла неявно запускаются транслятор (компилятор) *ML.EXE* и компоновщик (линковщик) *LINK.EXE*, а их параметры задаются визуально.

ML.EXE – это транслятор языка макроассемблера. Главная задача программы – перевести все команды ассемблера из текстового исходного кода в байты машинных команд. *ML.EXE* не создает готовую программу под конкретную операционную систему и ее формат. Он делает промежуточный объектный файл (с расширением obj). *ML.EXE* – компилирует и компонует (если указан ключ /LINK) один или более исходных файлов на языке ассемблера. Параметры командной строки являются зависимыми от регистра. *ML.EXE* – программа с интерфейсом командной строки.

Вид строки:

```
ML [ключи] список_файлов [/link <ключи_линковщика>],
```

где

ключи – параметры компиляции (ключи чувствительны к регистру);

список_файлов – имена исходных файлов, которые будут транслироваться;

/link – параметр, с которым указываются ключи_линковщика;

ключи_линковщика – параметры командной строки, которые будут переданы линковщику.

Главная характеристика трансляции – это тип выходного объектного файла. *ML.EXE* умеет создавать такие типы obj-файлов:

1) MS COFF (Common Object File Format) – объектный формат, используемый для компиляции программ под Windows;

2) Intel OMF – объектный формат, который в основном используется для создания exe-модулей под DOS.

Список и описание параметров компилятора можно посмотреть на странице https://msdn.microsoft.com/ru-ru/library/s0ksfwcf.aspx.

LINK.EXE – связывает объектные файлы и библиотеки в формате COFF для создания исполняемого файла (EXE) или библиотеки динамической компоновки (DLL).

Главная функция *LINK.EXE* – компоновать объектные файлы в исполняемые модули определенного формата.

Так как линковщик обрабатывает только объектные файлы, он независим от языка, на котором был написан исходный код. Линковщик Visual Studio 2015 умеет компоновать с наращиванием. Если исходник большого приложения был слегка изменен, такой метод значительно ускоряет процесс перекомпоновки.

LINK.EXE – это программа с интерфейсом командной строки.

Вид строки:

```
LINK [ключи] [входные_obj-файлы] [@файл_опций],
```

где

ключи – список возможных опций;

входные_obj-файлы – имена объектных файлов через пробел или имя одного файла;

файл_опций – имя текстового файла (с любым расширением), в котором лежат указания линковщику. Формат указаний соответствует ключам.

Можно использовать несколько таких файлов опций (@имя1, @имя2, @имяX).

Список и описание параметров компоновщика можно посмотреть на странице https://msdn.microsoft.com/ru-ru/library/y0zzbyt4.aspx.

Рассмотрим, какие ключи компилятора были заданы неявно при создании исполняемого файла в IDE Visual Studio 2015.

В окне «Обозреватель решений» снова кликните правой кнопкой мыши по имени проекта First_Program (выделено жирным шрифтом). На странице свойств проекта First_Program пройдите по цепочке «Свойства конфигурации» – «Microsoft Macro Assembler» – «Командная строка». В правой части диалогового окна будут видны все параметры компилятора *ML.EXE* вида

```
ml.exe /c /nologo /Zi /Fo"Debug\%(FileName).obj" /W3
/errorReport:prompt /Ta
```

Значения ключей

/c – компиляция без компоновки, т.е. создание obj-файла.

/nologo – не показывать заголовочный текст компилятора.

/Zi – создает полную отладочную информацию. Используется на этапе отладки программы. Транслятор включает в объектный файл полную отладочную информацию, которую линковщик (только с ключом /DEBUG) формирует в отладочный в pdb-файл (program database) и связывает с исполняемым модулем. С таким файлом в отладчике можно будет видеть исходный код и всю информацию типа имен функций, переменных, меток и т. д.

/Fo<файл> – задает альтернативное имя для объектного файла. Если необходимо, чтобы объектный файл имел имя, отличное от имени исходного файла, то необходимо указать это имя в поле <файл>.

/W<число> – устанавливает уровень предупреждений (перечень событий, трактуемых как предупреждения). Число может быть 0, 1, 2 или 3.

/errorReport – если трансляция завершается ошибкой, можно

воспользоваться /ERRORREPORT, чтобы отправить в корпорацию MicroSoft сведения о ней.

/Ta<file> – для компилирования файлов, расширение которых не .asm.

Рассмотрим, какие ключи компоновщика были заданы неявно при создании исполняемого файла в IDE Visual Studio 2015.

В окне «Обозреватель решений» нужно кликнуть правой кнопкой мыши по имени проекта First_Program (выделено жирным шрифтом). На странице свойств проекта First_Program пройдите по цепочке «Свойства конфигурации» – «Компоновщик» – «Командная строка». В правой части диалогового окна появятся параметры компоновщика *LINK.EXE*:

```
/OUT:"C:\Users\v.kotenko\Desktop\First_Program\First_Program\
Debug\First_Program.exe" /MANIFEST /NXCOMPAT /PDB:"C:\Users\
v.kotenko\Desktop\First_Program\First_Program\Debug\First_Prog
ram.pdb" /DYNAMICBASE "kernel32.lib" "user32.lib" "gdi32.lib"
"winspool.lib"  "comdlg32.lib"  "advapi32.lib"  "shell32.lib"
"ole32.lib" "oleaut32.lib" "uuid.lib" "odbc32.lib" "odbccp32.lib
" /DEBUG:FASTLINK /MACHINE:X86 /INCREMENTAL /PGD:"C:\Users\
v.kotenko\Desktop\First_Program\First_Program\Debug\First_Prog
ram.pgd" /SUBSYSTEM:WINDOWS /MANIFESTUAC:"level='asInvoker'
uiAccess='false'" /ManifestFile:"Debug\First_Program.exe.inter
mediate.manifest" /ERRORREPORT:PROMPT /NOLOGO /TLBID:1
```

Значения ключей

/OUT – задает имя выходного файла.

/MANIFEST – создает параллельный файл манифеста и при необходимости включает его в двоичный.

/NXCOMPAT – помечает исполняемый файл как проверенный на совместимость с признаком предотвращения выполнения данных как кода.

/PDB – определяет имя файла базы данных программы (Program Database – PDB), содержащего информацию для отладки.

/DYNAMICBASE – определяет, следует ли создавать исполняемый

образ, базовый адрес которого может быть случайным образом изменен во время загрузки с помощью технологии ASLR (Address Space Layout Randomization). Включен по умолчанию.

/DEBUG – создает отладочную информацию для EXE- и DLL-файлов. Отладочная информация помещается в pdb-файл.

/MASHINE – указывает целевую платформу.

/INCREMENTAL – управляет инкрементной компоновкой. Включен по умолчанию.

/PGD<файл> – задает файл базы данных PGD для профильных оптимизаций. Профильная оптимизация позволяет повысить качество выходного файла .

/SUBSYSTEM – указывает операционной системе, как запускать исполнимый файл.

/MANIFESTUAC – указывает, следует ли внедрять в манифест программы сведения о контроле учетных записей.

/MANIFESTFILE – изменяет имя файла манифеста, заданное по умолчанию.

/ERRORREPORT – позволяет предоставлять сведения о внутренних ошибках компоновщика напрямую в Microsoft.

/NOLOGO – отключает загрузочный баннер.

/TLBID – указывает идентификатор ресурса библиотеки типов, создаваемой компоновщиком

2.2 Компиляция и компоновка программы с использованием командной строки и командного файла

На самом деле указывать такое количество параметров, которое использовалось при создании исполняемого файла в IDE Visual Studio 2015, нет необходимости, так как многие ключи установлены по умолчанию. Для создания исполняемого файла необходимо запустить из командной строки операционной системы сначала компилятор *ML.EXE*, а затем компоновщик *LINK.EXE* с указанием для них параметров

трансляции и компоновки.

Для получения из исходного файла First_Program.asm объектного файла First_Program.obj достаточно командной строки

```
ML.EXE /c First_Program.asm
```

Параметр /c обозначает компиляцию без компоновки.

Для получения из объектного файла First_Program.obj исполняемого файла First_Program.exe достаточно командной строки

```
LINK /subsystem:Windows First_Program.obj
```

Параметр `/subsystem:Windows` указывает операционной системе, как запускать исполняемый файл.

Чтобы не набирать каждый раз вышеприведенные строки и выполнять рутинные операции по удалению промежуточных файлов, а также для систематизации рабочих файлов в своих каталогах, создадим мини-среду для преобразования исходного ассемблерного файла в исполняемый файл.

Создадим каталог с именем LLPL (Low-level programming languages). В нем будем размещать наши рабочие файлы.

В папке LLPL создадим подкаталог MY_SDK. В каталог скопируем:

1) компилятор *ML.EXE*. Полный путь к файлу имеет вид C:\Program Files\Microsoft Visual Studio 14.0\VC\bin\ml.exe;

2) компоновщик *LINK.EXE*. Полный путь к файлу имеет вид C:\Program Files\Microsoft Visual Studio 14.0\VC\bin\link.exe;

3) необходимый для работы компоновщика файл mspdb140.dll. Путь: C:\Program Files\Microsoft Visual Studio 14.0\Common7\IDE\mspdb140.dll.

Если прописать путь к файлу в системной переменной PATH и перезапустить компьютер, то необходимость копировать файл отпадет;

4) объектные библиотеки *kernel32.lib*, *user32.lib*, *gdi32.lib*. Пути:
C:\Program Files\Windows Kits\10\Lib\10.0.10586.0\um\x86\kernel32.lib
C:\Program Files\Windows Kits\10\Lib\10.0.10586.0\um\x86\user32.lib
C:\Program Files\Windows Kits\10\Lib\10.0.10586.0\um\x86\gdi32.lib

При необходимости позднее можно подключить библиотеку *comdlg32.lib*, которая содержит функции для работы со стандартными диалоговыми окнами (FileOpen, FileSave и т. п.) и другие.

Содержимое папки MY_SDK показано на рис. 2.1.

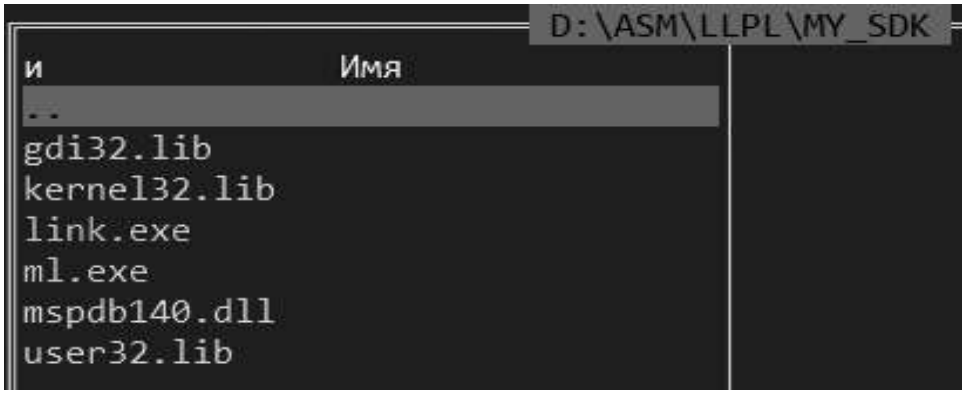

Рис. 2.1 – Файлы необходимые для компиляции и компоновки

Последним шагом является создание командного файла. Создаем любым текстовым редактором в папке LLPL файл ASM.BAT и заносим в него строки:

```
@echo off
my_sdk\ml.exe /c %1.asm
my_sdk\link.exe /DEFAULTLIB:my_sdk\kernel32.lib
/DEFAULTLIB:my_sdk\user32.lib /DEFAULTLIB:my_sdk\gdi32.lib
/SUBSYSTEM:WINDOWS  %1.obj
if exist %1.obj del %1.obj
dir %1.*
pause
```

Как видно из файла сначала производится компиляция без компоновки компилятором *ML.EXE*. А затем осуществляется компоновка компоновщиком *LINK.EXE*, при которой использованием ключа /DEFAULTLIB добавляются три библиотеки.

Ключ /DEFAULTLIB:library добавляет одну библиотеку в список библиотек, в которых LINK выполняет поиск при разрешении ссылок. Поиск в библиотеке, заданной с помощью параметра /DEFAULTLIB, начинается после выполнения поиска в библиотеках, заданных в командной строке, и до выполнения поиска в библиотеках по умолчанию, перечисленных в obj-файлах.

Готовая и настроенная мини-среда показана на рис. 2.2.

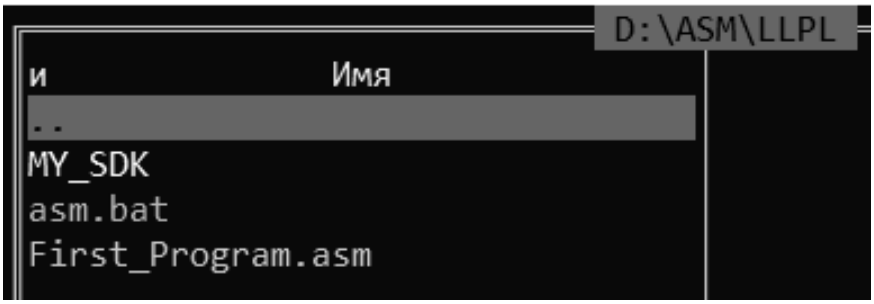

Рис.2.2 – Сформированная мини-среда создания исполняемого файла

Создаем в папке LLPL любым текстовым редактором файл First_Program.asm или копируем его из того места, где он был создан с использованием IDE Visual Studio 2015.

В командной строке вводим: ASM.BAT First_Program (без расширения!) и нажимаем клавишу Enter. В папке LLPL появляется файл First_Program.exe.

Примечание – для работы с мини-средой рекомендуется использовать файловый менеджер Far. Программа имеет встроенный редактор для набора теста исходного файла на языке ассемблера и отображаемую в нижней части экрана командную строку, в которую можно вводить имя командного файла с параметрами.

2.3 Структура программы

Программа на языке ассемблера имеет такую структуру:

```
.686p
.model flat, stdcall
option casemap: none
.data
     <инициализированные данные>
.data?
     <неинициализированные данные>
.const
     <константы>
.code
<метка>
     <код>
end <метка>
```

Директива `.686p` указывает компилятору, что необходимо использовать набор операций процессора определенного поколения, директива `.model` указывает используемую модель памяти `flat`, соглашения о вызовах `stdcall` определяют порядок передачи параметров и порядок очистки стека, директива `option casemap: none` заставляет компилятор ассемблера различать большие и маленькие буквы в метках и именах процедур.

Директивы `.data`, `.data?`, `.const` и `.code` определяют секции программы. Начало одной секции отмечает конец предыдущей. Есть две группы секций: данных и кода. Секция `.data` содержит инициализированные данные программы. Секция `.data?` содержит неинициализированные данные программы. Неинициализированные данные не занимают места в исполняемом файле. Необходимо всего лишь сообщить компилятору, сколько места понадобится, когда программа загрузится в память.

Секция `.const` содержит объявления констант, используемых программой. Константы не могут быть изменены. Попытка изменить константу вызывает аварийное завершение программы.

Использовать все три секции не обязательно.

Есть только одна секция для кода – `.code`. Предложения `<метка>` и `end <метка>` устанавливают границы кода. Обе метки должны быть идентичны. Код должен располагаться между ними.

Любая программа под Windows должна как минимум корректно завершится. Для этого необходимо вызвать функцию Win32 API `ExitProcess` из библиотеки *kernel32.lib*. Так как мы не подключали файлы с расширением `.inc`, содержащие прототипы функций, то необходимо явно указать прототип: имя функции, количество ее параметров и их размер: `ExitProcess PROTO STDCALL :DWORD`.

```
.686p
.model flat, stdcall
```

```
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.code
program:
      push 0
      call ExitProcess
end program
```

Это пример минимальной программы на языке ассемблера, которая делает только одно – корректно завершается. Команда «`push`» кладет в стек параметр для процедуры *ExitProcess*, который определяет код завершения. Значение 0 – код нормального завершения программы. Команда «`call`» вызывает процедуру *ExitProcess*.

Добавим в программу вывод простого диалогового окна с одной кнопкой «ОК». Для этого зададим прототип и воспользуемся вызовом функции Win32 API *MessageBoxA* из библиотеки *user32.lib*.

Параметры функции *MessageBoxA*:

1) дескриптор окна владельца, которое создает окно сообщения. Если этот параметр равен нулю, то окно сообщения не имеет окна владельца;

2) указатель на символьную строку с нулем в конце, которая содержит сообщение, показываемое в окне;

3) указатель на символьную строку с нулем в конце, которая содержит заголовок диалогового окна (окна сообщения);

4) устанавливает содержание и режим работы диалогового окна. Этим параметром может быть комбинация флажков. Если этот параметр равен нулю, то окно содержит одну командную кнопку «ОК».

```
.686p
.model flat, stdcall
option casemap: none
ExitProcess PROTO STDCALL :DWORD
MessageBoxA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD
; Секция данных
.data
      TextMsg  db 'Это первая программа для Win32',0
      TitleMsg db 'Язык ассемблера Masm32!',0
; секция кода
.code
program:
      ; вывод диалогового окна
```

```
        push 0
        push offset TitleMsg
        push offset TextMsg
        push 0
        call MessageBoxA
        ; завершение программы
        push 0
        call ExitProcess
end program
```

В секции `.data` определены две строки – *TextMsg* и *TitleMsg*, являющиеся параметрами функции *MessageBoxA*.

2.4 Макродирективa Invoke

Основное преимущество *MASM* – это макродирективa `invoke`. Она позволяет вызывать *API-функции* с проверкой количества и типа параметров. Это почти тот же вызов `call`, но эта макродирективa проверяет количество параметров и их типы. Пример вызова функции:

```
invoke <функция>, <параметр1>, <параметр2>, <параметр3>.
```

Чтобы использовать `invoke` для вызова функции, необходимо определить ее прототип:

```
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD
```

Эта директива объявляет процедуру, названную `testproc`, у которой три параметра размером DWORD.

Теперь, если написать

```
invoke testproc, 1, 2, 3, 4,
```

MASM выдаст ошибку потому, что процедура `testproc` имеет три параметра, а не четыре. *MASM* также осуществляет контроль соответствия типов, т.е. проверяет, имеют ли параметры правильный тип (размер).

В `invoke` можно использовать ADDR вместо OFFSET. Тогда после сборки кода адрес сформируется в правильной форме.

Нижеприведенный код создает процедуру, названную `testproc`, с тремя параметрами. Прототип используется макросом `invoke`. Все параметры можно использовать в коде процедуры, они автоматически извлекутся из стека.

```
; прототип процедуры
testproc PROTO STDCALL :DWORD, :DWORD, :DWORD
.code
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
....
ret
testproc endp
```

В процедурах можно использовать локальные переменные:

```
testproc proc param1:DWORD, param2:DWORD, param3:DWORD
LOCAL var1: DWORD;  ; первая локальная переменная
LOCAL var2: BYTE    ; вторая локальная переменная
mov edx, param2     ; edx = param2
mov eax, param3     ; eax = param3
mov ecx, param1     ; ecx = param1
mov var2, cl        ; var2 = cl
mov var1, eax       ; var1 = eax
ret
testproc endp
```

Нельзя использовать локальные переменные вне процедуры. Они сохранены в стеке и удаляются при возврате из процедуры.

Теперь можно переписать программу вывода диалогового окна с кнопкой «ОК», используя макрос `invoke`.

```
.686p
.model flat, stdcall
option casemap: none
ExitProcess PROTO STDCALL :DWORD
MessageBoxA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD
.data
     TextMsg  db 'Это первая программа для Win32',0
     TitleMsg db 'Язык ассемблера Masm32!',0
.code
program:
     ; вывод диалогового окна
     invoke MessageBoxA,0,ADDR TextMsg, ADDR TitleMsg,0
     ; завершение работы программы
     invoke ExitProcess,0
end program
```

В *MASM* широко используются директивы сравнения и повторения: `.IF`, `.REPEAT`, `.WHILE`:

Директива `.IF` выполняет `фрагмент_программы_1`, если логическое выражение истинно, и `фрагмент_программы_2`, если оно ложно:

```
.IF логическое_выражение
     ; фрагмент_программы_1
.ELSE
     ; фрагмент_программы_2
.ENDIF
```

Директива `.REPEAT` повторяет выполнение `фрагмент_программы`, пока условие не истинно:

```
.REPEAT
     ; фрагмент_программы
.UNTIL условие
```

Директива `.WHILE` – инверсия директивы `.REPEAT`. Она повторяет выполнение `фрагмент_программы`, пока условие истинно:

```
.WHILE условие
; фрагмент_программы
.ENDW
```

Можно использовать директиву `.BREAK`, чтобы прервать цикл:

```
.WHILE edx==1
     inc eax
     .IF eax==7
          .BREAK
     .ENDIF
.ENDW
```

Если eax станет равным семи, то цикл `.WHILE` будет прерван.

Директива `.CONTINUE` осуществляет переход на код, проверяющий условие цикла в конструкциях `.REPEAT` и `.WHILE`.

2.5 Форматированный вывод

Для вывода в окно результатов работы программы, исходных данных и т. п. необходимо сформировать выходную строку, в которую поместить данные, преобразовав их в соответствии с необходимым форматом. Это позволяет осуществить функция *wsprintfA*, которая записывает форматированные данные в указанный буфер. Все аргументы преобразуются и копируются в выходной буфер согласно с соответствующей спецификацией формата в строке формата. Функция добавляет завершающий нулевой символ к записываемым символам, но в возвращаемое функцией значение количества записанных в буфер

символов он не включается. Функция находится в библиотеке *user32.lib*. Прототип функции имеет вид `wsprintfA PROTO C : VARARG`

Рассмотрим пример деления значений двух ячеек mem1 и mem2 и вывода результата в диалоговое окно с использованием `wsprintfA`:

```
.686p
.model flat, stdcall
option casemap: none
ExitProcess PROTO STDCALL :DWORD
MessageBoxA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD
wsprintfA PROTO C :VARARG
.data
     TitleMsg db 'Деление значений двух ячеек!',0
     ; указываем буфер для форматированного вывода
     buffer db 128 dup(0)
     ; указываем строку формата со спецификациями форматов
     format db 'Частное %d / %d = %d. Остаток = %d.', 0
     ; задаем исходные данные
     mem1 dd 275
     mem2 dd 30
.code
program:
     ; загружаем в регистр eax значение ячейки mem1
     mov eax, mem1
     ; расширяем значение в eax на edx:eax
     cdq
     ; загружаем в регистр ebx значение ячейки mem2
     mov ebx, mem2
     ; делим содержимое edx:eax на содержимое ebx
     idiv ebx
     ; формируем строку вывода по заданному формату
     invoke wsprintfA, addrbuffer, addr format, mem1,mem2,eax,edx
     ; выводим результат в диалоговое окно
     invoke MessageBoxA, 0, ADDR buffer, ADDR TitleMsg,0
     invoke ExitProcess,0
end program
```

Результат работы программы показан на рис. 2.1.

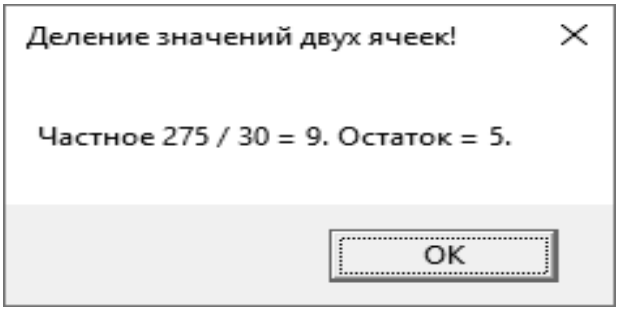

Рис. 2.1 – Результат работы программы деления содержимого двух ячеек памяти

Тема 3 Процедуры

3.1 Команды работы со стеком

Работа со стеком связана с процедурами, так как *стек* используется для передачи параметров и для хранения локальных данных процедур.

Для того чтобы положить данные в стек, используется команда «`push`». Формат команды: `push <операнд>`.

Операнд может быть регистром, ячейкой памяти или непосредственным операндом. Размер операнда должен быть 2 или 4 байта. Операнд кладется на вершину стека, а значение регистра ESP уменьшается на размер операнда (рис. 3.1).

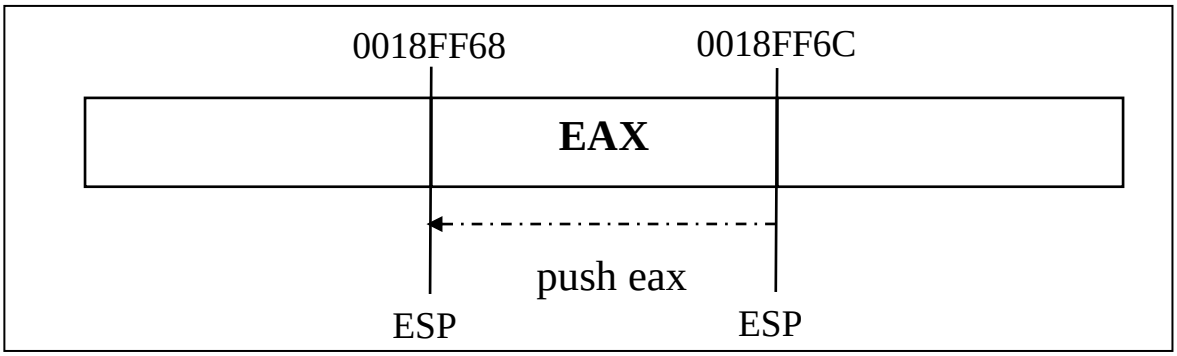

Рис. 3.1 – Выполнение команды «push»

Для того чтобы взять данные из стека, используется команда «`pop`». Формат команды: `pop <операнд>`

Из вершины стека берутся 2 или 4 байта и помещаются в указанный регистр или ячейку памяти. Значение регистра ESP увеличивается на размер операнда (рис. 3.2).

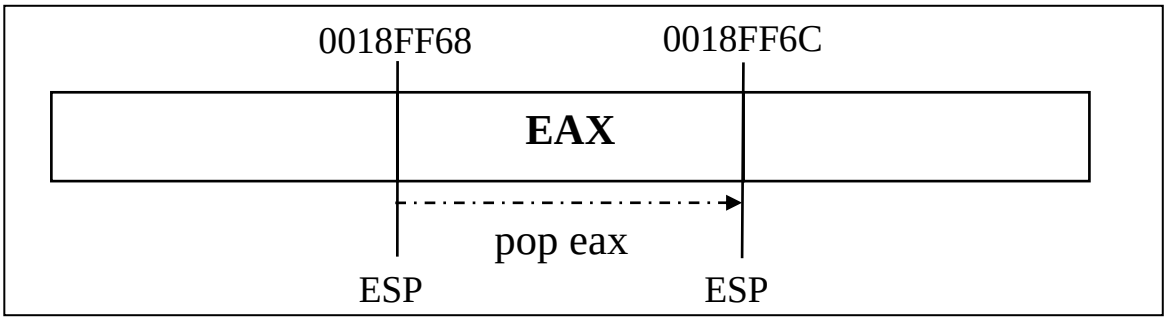

Рис. 3.2 – Выполнение команды «pop»

Существуют еще команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое всех регистров общего назначения.

Команда «`pusha`» сохраняет в стеке содержимое регистров AX, CX, DX, BX, SP, BP, SI, DI. Команда «`pushad`» сохраняет в стеке содержимое регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Для регистра (E)SP сохраняется значение, которое было до того, как положили регистры в стек. После этого значение регистра (E)SP изменяется как обычно.

Команды «`popa`» и «`popad`» противоположны предыдущим – они восстанавливают из стека значения регистров (E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, (E)AX. Содержимое регистра (E)SP не восстанавливается из стека, а изменяется как обычно.

Команда «`pushf`» сохраняет в стеке младшие 16 бит регистра флагов. Команда «`pushfd`» сохраняет в стеке все 32 бита регистра флагов.

Команда «`popf`» восстанавливает из стека младшие 16 бит регистра флагов. Команда «`popfd`» восстанавливает все 32 бита регистра флагов.

3.2 Синтаксис процедуры. Вызов и возврат из процедуры

Описание процедуры на ассемблере выглядит таким образом:

```
<имя процедуры> PROC
     <тело процедуры>
<имя процедуры> ENDP
```

Несмотря на то что после имени процедуры не ставится двоеточие, оно является меткой, обозначающей первую команду процедуры.

В языке ассемблера имена и метки, описанные в процедуре, не локализуются внутри нее, поэтому они должны быть уникальны.

Обычно процедуры размещают либо в конце секции кода после вызова `ExitProcess`, либо в начале секции кода после директивы `.code`.

Вызов процедуры – это, по сути, передача управления на первую команду процедуры. Для этого управления можно использовать команду безусловного перехода на метку, являющуюся именем процедуры. Можно

даже не использовать директивы *proc* и *endp*, а написать обычную метку с двоеточием после вызова функции *ExitProcess*.

Обращаться к процедуре можно из разных мест основной программы, а потому и возврат из процедуры должен осуществляться в разные места. Сама процедура не знает, куда надо вернуть управление, зато это знает основная программа. Поэтому при обращении к процедуре основная программа должна сообщить ей адрес возврата, т. е. адрес той команды, на которую процедура должна сделать переход по окончании своей работы. Поскольку при разных обращениях к процедуре будут указываться разные адреса возврата, то и возврат управления будет осуществляться в разные места программы. Адрес возврата принято передавать через стек, как показано в нижеприведенном примере:

```
.686
.model flat, stdcall
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.code
program:
     push L
     jmp  Procedure
L:   nop
     push 0
     call ExitProcess
Procedure:
     pop eax
     jmp eax
end program
```

Однако так обычно не делают. Система команд ассемблера включает специальные команды для вызова процедуры и возврата из нее:

```
call <имя процедуры> ; вызов процедуры
ret                  ; возврат из процедуры
```

Команда «`call`» записывает адрес следующей за ней команды в стек и осуществляет переход на первую команду указанной процедуры. Команда «`ret`» считывает из вершины стека адрес и выполняет переход по нему.

```
.686
.model flat, stdcall
option casemap: none
```

```
ExitProcess PROTO STDCALL :DWORD
.code
program:
     call Procedure
     push 0
     call  ExitProcess

Procedure proc
     ret
Procedure endp

end program
```

3.3 Передача параметров в процедуру

Существуют несколько способов передачи параметров в процедуру.

1. Параметры можно передавать через регистры

Если процедура получает небольшое число параметров, идеальным местом для их передачи оказываются регистры. Существуют соглашения о вызовах, предполагающие передачу параметров через регистры ECX и EDX. Этот метод самый быстрый, но он удобен только для процедур с небольшим количеством параметров.

2. Параметры можно передавать в глобальных переменных

Параметры процедуры можно записать в глобальные переменные, к которым затем будет обращаться процедура. Однако этот метод является неэффективным, и его использование может привести к тому, что рекурсия и повторная входимость станут невозможными.

3. Параметры можно передавать в блоке параметров

Блок параметров – это участок памяти, содержащий параметры и располагающийся в сегменте данных. Процедура получает адрес начала этого блока при помощи любого метода передачи параметров (в регистре, в переменной, в стеке, в коде или даже в другом блоке параметров).

4. Параметры можно передавать через стек

Передача параметров через стек – самый распространенный способ. Именно его используют такие языки высокого уровня, как C++ и Pascal. Параметры помещаются в стек непосредственно перед вызовом процедуры.

Возникает два вопроса: кто должен удалять параметры из стека (процедура или вызывающая ее программа) и в каком порядке помещать параметры в стек? Оба варианта имеют свои «за» и «против». Если стек освобождает процедура, то код программы получается меньшим, а если за освобождение стека от параметров отвечает вызывающая программа, то становится возможным вызвать несколько функций с одними и теми же параметрами последовательными командами «`call`». Первый способ, более строгий, используется при реализации процедур в языке Pascal, а второй, дающий больше возможностей для оптимизации, – в языке C++.

Основное соглашение о вызовах языка Pascal предполагает, что параметры кладутся в стек в прямом порядке. Соглашения о вызовах языка C++, в том числе одно из основных соглашений о вызовах ОС Windows `stdcall`, предполагают, что параметры помещаются в стек в обратном порядке. Это делает возможной реализацию функций с переменным числом параметров (как, например, `printf`). При этом первый параметр определяет число остальных параметров.

```
push <параметр N>
. . . . . . . . .
push <параметр 1>
call Procedure
```

В приведенном выше участке кода в стек кладется несколько параметров и затем вызывается процедура. Следует помнить, что команда «`call`» также кладет в стек адрес возврата. Таким образом, перед выполнением первой команды процедуры стек будет выглядеть так, как показано на рис. 3.3.

Рис. 3.3 – Стек перед выполнением первой команды процедуры

Адрес возврата оказывается в стеке поверх параметров. Однако поскольку в рамках своего участка стека процедура может обращаться без ограничений к любой ячейки памяти, нет необходимости перекладывать куда-то адрес возврата, а потом возвращать его обратно в стек. Для обращения к первому параметру используют адрес [ESP + 4] (прибавляем 4, так как на архитектуре Win32 адрес имеет размер 32 бита), для обращения ко второму параметру – адрес [ESP + 8] и т. д.

После завершения работы процедуры необходимо освободить стек. Если используется соглашение о вызовах `stdcall` (или любое другое, предполагающее, что стек освобождается процедурой), то в команде «`ret`» следует указать суммарный размер в байтах всех параметров процедуры. Тогда команда «`ret`» после извлечения адреса возврата прибавит к регистру ESP указанное значение, освободив таким образом стек.

Передача параметров и возврат из процедуры с использованием соглашения о вызовах `stdcall`:

```
.686
.model flat, stdcall
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.data
     x dd 0
     y dd 4
.code
program:
     push y ; в стек первый параметр размером 4 байта
     push x ; в стек второй параметр размером 4 байта
     call Procedure
     push 0
     call ExitProcess
Procedure proc
     ret 8 ; указываем, что надо освободить 8 байт стека
Procedure endp
end program
```

Если же используется соглашение о вызовах `cdecl` (или любое другое, предполагающее, что стек освобождается вызывающей

программой), то после команды «call» следует поместить команду, которая прибавит к регистру ESP нужное значение.

Передача параметров и возврат из процедуры с использованием соглашения о вызовах cdecl:

```
.686
.model flat, c
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.data
     x dd 0
     y dd 4
.code
program:
     push y ; в стек первый параметр размером 4 байта
     push x ; в стек второй параметр размером 4 байта
     call Procedure
     add esp, 8 ; освобождаем 8 байт стека
     push 0
     call ExitProcess
Procedure proc
     ret  ; используем команду возврата без параметров
Procedure endp
end program
```

5. Параметры можно передавать в потоке кода

В этом необычном методе передаваемые процедуре данные размещаются прямо в коде программы, сразу после команды «call». Чтобы прочитать параметр, процедура должна использовать его адрес, который автоматически передается в стеке как адрес возврата из процедуры. Процедура должна будет изменить адрес возврата на первый байт после конца переданных параметров перед выполнением команды «ret».

```
.686
.model flat, stdcall
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.code
program:
     call Procedure; кладет в стек адрес следующей команды
     db 'string',0 ; в этом случае – адрес начала строки
     push 0
     call ExitProcess
Procedure proc
```

```
     pop esi ; извлекаем из стека адрес начала строки
     xor ax,ax ; обнуляем счетчик символов строки
L1:  mov bl, [esi] ; раносим в BL байт, по адресу ESI
     inc esi ; увеличиваем значение в регистре ESI на 1
     inc eax ; увеличиваем значение счетчика на 1
     cmp bl, 0 ; Сравниваем прочитанный символ с нулем
     jne L1 ; если не 0, переходим к началу цикла
     push esi ; в стек адрес байта, следующего за строкой
     ret ; возврат из процедуры
Procedure endp
end program
```

3.4 Передача результата процедуры

Для передачи результата процедуры обычно используется регистр EAX. Этот способ используется не только в программах на языке ассемблера, но и в программах на языке C++. Объекты, имеющие размер не более 8 байт, могут передаваться через регистровую пару EDX : EAX. Вещественные числа передаются через вершину стека вещественных регистров. Если эти способы не подходят, то следует передать в качестве параметра адрес ячейки памяти, куда будет записан результат.

Передача параметров через стек, возврат результата через EAX:

```
.686
.model flat, c
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.data
     a dd 76
     b dd -8
     d dd ?
.code
program:
     push b              ; кладем параметры в стек
     push a
     call Procedure
     add esp, 8          ; освобождаем 8 байт стека
     mov d, eax          ; d = a - b
     push 0
     call ExitProcess
Procedure proc
     mov eax, [esp + 4]  ; заносим в EAX первый параметр
     mov edx, [esp + 8]  ; заносим в EDX второй параметр
     sub eax, edx        ; в EAX - разность параметров
     ret
Procedure endp
end program
```

Передача параметров через стек, возврат результата по адресу:

```
.686
.model flat, c
option casemap: none
ExitProcess PROTO STDCALL :DWORD
.data
     a dd 76
     b dd -8
     d dd ?
.code
program:
     push offset d; в стек адрес переменной для результата
     push b
     push a
     call Procedure
     add esp, 12         ; освобождаем 12 байт стека
     push 0
     call ExitProcess
Procedure proc
     mov eax, [esp + 4] ; заносим в EAX первый параметр
     mov edx, [esp + 8] ; заносим в EDX второй параметр
     sub eax, edx ; в EAX получилась разность параметров
     mov edx, [esp + 12] ; заносим в EDX адрес результата
     mov [edx], eax ; записываем результат по адресу в EDX
     ret
Procedure endp
end program
```

3.5 Сохранение регистров в процедуре

Регистров очень мало, и нет возможности указать, что основная программа использует, например, регистры EAX, ECX, EBP, ESP, а процедура – регистры EBX, EDX, ESI, EDI. Кроме того, существуют правила, которые изменить нельзя: в регистре ESP хранится адрес вершины стека, а команды умножения и деления всегда используют регистры EAX и EDX.

Чтобы основная программа могла продолжить вычисления, процедура должна при выходе восстановить те значения регистров, которые были до начала выполнения процедуры. Для этого процедуре придется предварительно сохранить значения регистров. Все вышесказанное относится также к случаю, когда одна процедура вызывает другую процедуру.

Особенно внимательно следует относиться к регистрам ESI, EDI, EBP и EBX. Windows использует их для своих целей и не ожидает, что вы измените их значение.

Если вы пишете отдельные процедуры, которые затем будут использоваться в другой программе, то сохранение и восстановление регистров становится жизненно необходимой операцией.

Сохранить значения регистров можно в стеке: по одному с помощью команды «`push`», или все сразу с помощью команды «`pushad`». В первом случае в конце процедуры нужно будет восстановить значения сохраненных регистров с помощью команды «`pop`» в обратном порядке. Во втором случае для восстановления значений регистров используется команда «`popad`».

При сохранении регистров указатель стека изменится на некоторое значение, зависящее от количества сохраненных регистров. Это нужно учитывать при вычислении адресов параметров процедуры, передаваемых через стек.

Процедура получает из стека два параметра по 4 байта:

```
Procedure proc
     push esi ; сохраняем используемые регистры
     push edi
     mov esi, [esp + 12] ; извлекаем из стека первый параметр
     mov edi, [esp + 16] ; извлекаем из стека второй параметр
     . . . . . . . .
     pop edi ; извлекаем сохранённые регистры из стека
     pop esi ; в обратном порядке
     ret
Procedure endp
```

Процедура получает из стека два параметра по 4 байта:

```
Procedure proc
     pushad ; сохраняем все регистры
     mov eax, [esp + 4 + 32] ; извлекаем первый параметр
     mov ebx, [esp + 8 + 32] ; извлекаем второй параметр
     . . . . . . . . .
     popad ; извлекаем сохраненные регистры из стека
     ret
Procedure endp
```

3.6 Локальные данные процедур

Процедуры часто нуждаются в локальных данных, которые размещаются в стеке. Для того чтобы отвести место под локальные переменные в процедуре на языке ассемблера, достаточно просто вычесть из регистра ESP размер требуемой памяти. После этого все вызываемые процедуры будут «знать», что место в стеке занято, и размещать свои данные в незанятой части стека.

При вызове других процедур, а также в ходе выполнения текущей процедуры в стек могут быть положены другие данные. При этом значение регистра ESP изменится. Поэтому ESP не является надежной точкой отсчета для адресов локальных переменных. Для того чтобы получить такую точку отсчета, значение регистра ESP переписывают в регистр EBP, предварительно сохранив значение регистра EBP в стеке. В этом случае регистр EBP отмечает часть стека, занятую на момент начала работы процедуры (регистр EBP – указатель базы кадра стека). При таком подходе первый параметр процедуры всегда находится по адресу [EBP + 8]. Адреса локальных переменных отсчитываются от регистра EBP с отрицательным смещением. По окончании работы процедуры значение регистра ESP восстанавливается по регистру EBP, а значение регистра EBP – из стека.

```
Procedure proc
     var_104 = byte ptr -104h ; последняя локальная переменная
     var_4   = dword ptr -4 ; первая локальная переменная
     param_1 = dword ptr  8
     param_2 = dword ptr  0Ch
     push ebp ; сохраняем EBP в стеке
     mov  ebp, esp ; переписываем ESP в EBP
     sub  esp, 104h ; уменьшаем ESP на 104h
     mov  edx, [ebp + param1] ; заносим в edx первый параметр
     mov  eax, [ebp + param2] ; заносим в eax второй параметр
     push ebx ; сохраняем в стеке значения трех регистров
     push esi
     push edi
     . . . . . . . .
     pop  edi ; восстанавливаем из стека значения регистров
     pop  esi
     pop  ebx
     mov  esp, ebp ; переписываем EBP в ESP
     pop  ebp ; извлекаем из стека значение в EBP
     ; теперь ESP указывает на адрес возврата
     ret
```

```
Procedure endp
```

На рис. 3.4 показаны локальные данные и параметры процедуры.

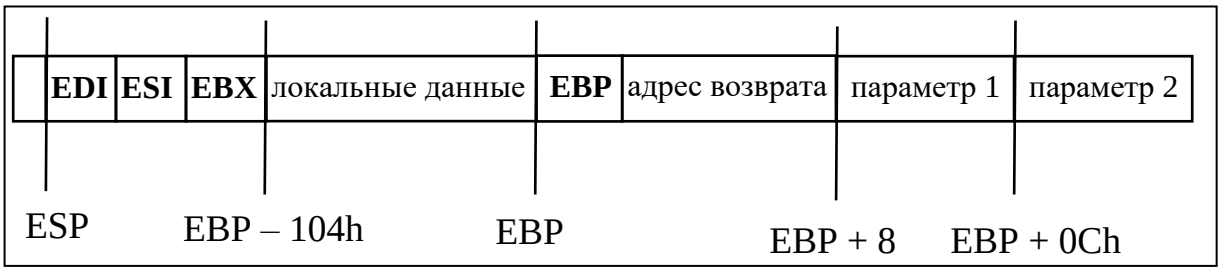

Рис. 3.4 – Локальные данные и параметры процедуры

Такой способ позволяет также отводить различное количество места под локальные данные, и при необходимости не заботится о парности команд «push» и «pop».

3.7 Рекурсивные процедуры

Рекурсия требует много места для хранения локальных данных на каждом своем шаге. Рекурсивные процедуры обычно выполняются не очень быстро, поэтому Ассемблеру рекурсия не свойственна, но при желании на нем можно написать такие процедуры. В процедуре на ассемблере, как и на других языках, должна быть терминальная ветвь, в которой нет рекурсивного вызова, и рабочая ветвь.

При реализации рекурсивных процедур становится особенно важным использование стека для передачи параметров и адреса возврата, что позволяет хранить данные, относящиеся к разным уровням рекурсивных вызовов, в разных областях памяти.

Рассмотрим программу с рекурсивной процедурой вычисления факториала целого беззнакового числа. Процедура получает параметр через стек и возвращает результат через регистр EAX.

```
.686p
.model flat, stdcall
option casemap :none
ExitProcess PROTO STDCALL :DWORD
MessageBoxA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD
wsprintfA PROTO C :DWORD,:VARARG
```

```
.data
     TitleMsg db 'Лабораторная работа:Вычисление факториала',0
     formatA     db 'Факториал %d = %d', 0
     otvet       db 128 dup(0)
.const
     num equ 5

.code
start:
     push num
     call FactorialProc
     invoke wsprintfA, addr otvet, addr formatA, num, eax
     invoke MessageBoxA, 0, addr otvet, addr TitleMsg, 0
     invoke ExitProcess,0

   FactorialProc  proc
     mov eax, [esp + 4] ; заносим в EAX параметр процедуры
     test  eax, eax ; проверяем значение в регистре EAX
     jz final; если EAX = 0, то обходим рекурсивную ветвь
     dec eax ; уменьшаем значение в регистре EAX на 1
     push eax ; кладем в стек параметр для следующего
              ; рекурсивного вызова
     call FactorialProc ; вызываем процедуру
     add esp, 4 ; очищаем стек, так как процедура использует RET
                ; без параметров
     mul dword ptr [esp + 4] ; умножаем EAX, хранящий
                             ; результат предыдущего вызова, на
                             ; параметр текущего вызова процедуры
     ret ; возврат из процедуры (без параметров)
     final: inc eax ; если EAX был равен 0, записываем в EAX 1
     ret ; возврат из процедуры (без параметров)
   FactorialProc  endp
end start
```

Тема 4 Консольные приложения

4.1 Понятие консольного приложения

Консольные приложения представляют собой систему средств взаимодействия пользователя с компьютером, основанную на использовании буквенно-цифрового режима дисплея. Консольные приложения очень компактны не только в откомпилированном виде, но и в текстовом варианте, и имеют такие же возможности обращаться к ресурсам Windows посредством *API-функций*, как и обычные графические приложения.

Для работы с консольными приложениями Windows используются соответствующие функции Windows API. В программе на ассемблере записываются прототипы вызываемых функций, а сами функции расположены в подключаемом файле библиотеки (*.lib).

4.2 Минимальное консольное приложение

Пример простейшего консольного приложения:

```
.686p
.model flat, stdcall

; прототипы внешних функций
SetConsoleTitleA PROTO :DWORD
GetStdHandle PROTO :DWORD
WriteConsoleA PROTO :DWORD,:DWORD,:DWORD,:DWORD,:DWORD
CharToOemA PROTO :DWORD, :DWORD
Sleep PROTO :DWORD
ExitProcess PROTO :DWORD
.const
STD_OUTPUT_HANDLE equ -11; константа Win API

.data

sConsoleTitle db 'First Console Application',0
consoleOutHandle dd ? ; дескриптор консоли вывода
bytesWritten dd ? ; количество байт
message db "Привет всем!",13,10,0 ;
h EQU $ - message ; длина текстовой строки (константа)

.code
start:
; выводим заголовок консоли
invoke SetConsoleTitleA, offset sConsoleTitle
;Получаем дескриптор консоли вывода и сохраняем его
```

```
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov consoleOutHandle, eax
; перекодируем сообщение в формат OEM
INVOKE CharToOemA, offset message, offset message
mov eax, h
; выводим строку в консоль: дескриптор консоли, указатель
; на выводимую строку, длина строки, число выведенных байт
INVOKE WriteConsoleA, consoleOutHandle, offset message,\
eax, offset bytesWritten, 0
INVOKE Sleep, 2000
INVOKE ExitProcess,0
END start
```

Для перекодировки русского текста, введенного в кодировке CP1251 в кодировку OEM, читаемую в консоли, используется функция `CharToOem`:

```
BOOL CharToOem(
LPCTSTR lpszSrc,
LPSTR lpszDst)
```

Параметр `lpszSrc` – указатель на строку-источник.

Параметр `lpszDst` – указатель на строку-приемник.

Возвращаемое значение – единица, в случае успешной перекодировки.

4.3 Функции работы с консолью

4.3.1 Создание и освобождение консоли

Консольные приложения могут создать свою консоль. В этом случае весь ввод-вывод будет производиться в эту консоль. Если приложение консоль не создает, то здесь может возникнуть двоякая ситуация: либо наследуется консоль, в которой программа была запущена (например, консоль Far), либо Windows создает для приложения свою консоль.

Для создания своей консоли используется функция `AllocConsole`:

```
BOOL WINAPI AllocConsole(void);
```

По завершении программы все выделенные консоли автоматически освобождаются. Это можно сделать и принудительно, используя функцию `FreeConsole`:

```
BOOL WINAPI FreeConsole(void);
```

4.3.2 Получение дескриптора устройства и установка заголовка окна

Для получения дескриптора стандартного устройства используется функция `GetStdHandle`:

```
HANDLE WINAPI GetStdHandle(
_In_ DWORD nStdHandle);
```

Параметр nStdHandle может принимать значения:

```
STD_INPUT_HANDLE =  - 10 устройство ввода;
STD_OUTPUT_HANDLE = - 11 устройство вывода;
STD_ERROR_HANDLE = - 12 ошибка.
```

Для установки заголовка окна консоли используется функция `SetConsoleTitle`:

```
BOOL WINAPI SetConsoleTitle( _In_ LPCTSTR lpConsoleTitle);
```

Параметр `lpConsoleTitle` – указатель на строку имени консоли, заканчивающуюся нулевым символом.

4.3.3 Вывод в консоль и чтение из буфера консоли

Для вывода текстовой информации в консоль используется функция `WriteConsole`:

```
BOOL WINAPI WriteConsole(
_In_ HANDLE hConsoleOutput,
_In_ const VOID* lpBuffer,
_In_ DWORD nNumberOfCharsToWrite,
_Out_ LPDWORD lpNumberOfCharsWritten,
_Reserved_ LPVOID lpReserved);
```

ее параметры:

1) `hConsoleOutput` – дескриптор буфера вывода консоли, который может быть получен при помощи функции GetStdHandle;
2) `lpBuffer` – указатель на буфер, в котором находится выводимый текст;
3) `nNumberOfCharsToWrite` – количество выводимых символов;
4) `lpNumberOfCharsWritten` – указывает на переменную типа двойное слово, куда будет помещено количество действительно выведенных символов;

5) `lpReserved` – резервный параметр, должен быть равен нулю.

Буфер, который содержит выводимый текст, не обязательно должен заканчиваться нулем, поскольку для данной функции указывается количество выводимых символов.

Для чтения строки из буфера консоли используется функция `ReadConsole`:

```
BOOL WINAPI ReadConsole(
_In_ HANDLE hConsoleInput,
_Out_ LPVOID lpBuffer,
_In_ DWORD nNumberOfCharsToRead,
_Out_ LPDWORD lpNumberOfCharsRead,
_In_Opt_ LPVOID pInputControl);
```

ее параметры:

1) `hConsoleInput` – дескриптор буфера ввода консоли, полученного функцией `GetStdHandle`;
2) `lpBuffer` – указатель на буфер, куда помещается вводимый текст;
3) `nNumberOfCharsToRead` – длина буфера ввода;
4) `lpNumberOfCharsRead` – указатель на переменную, в которую будет помещено количество действительно введенных символов;
5) `pInputControl` – резервный параметр, должен быть равен нулю.

4.3.4 Определение размеров окна консоли, установка позиции курсора и атрибутов символов

Для определения размеров окна консоли используется функция `SetConsoleScreenBufferSize`:

```
BOOL WINAPI SetConsoleScreenBufferSize(
_In_ HANDLE hConsoleOutput,
_In_ COORD dwSize);
```

с параметрами:

1) `hConsoleOutput` – дескриптор буфера вывода консоли;
2) `dwSize` – структура, задающая размер окна консоли:

```
COORD STRUC
     X DW ?
     Y DW ?
```

```
COORD ENDS
```

Для установки позиции курсора в окне консоли используется функция `SetConsoleCursorPosition`:

```
BOOL WINAPI SetConsoleCursorPosition(
_In_ HANDLE hConsoleOutput,
_In_ COORD dwCursorPosition);
```

содержащая такие параметры:

1) `hConsoleOutput` – дескриптор буфера выходной консоли;
2) `dwCursorPosition` – структура координат COORD, определяющая позицию курсора.

Для установки атрибутов символов, выводимых в буфер окна консоли, используется функция `SetConsoleTextAttribute`:

```
BOOL WINAPI SetConsoleTextAttribute(
_In_ HANDLE hConsoleOutput,
_In_ WORD wAttributes);
```

с параметрами:

1) `hConsoleOutput` – дескриптор буфера вывода консоли;
2) `wAttributes` – атрибуты символов, получаемые путем комбинации констант:

```
FOREGROUND_BLUE equ 1h          ;синий
FOREGROUND_GREEN equ 2h         ;зеленый
FOREGROUND_RED equ 4h           ;красный
FOREGROUND_INTENSITY equ 8h     ;интенсивный
BACKGROUND_BLUE equ 10h         ;синий фон букв
BACKGROUND_GREEN equ 20h        ;зеленый фон букв
BACKGROUND_RED equ 40h          ;красный фон букв
BACKGROUND_INTENSITY equ 80h    ;интенсивный фон букв
```

Для установки атрибутов символов для определенного числа символьных ячеек, начиная с заданных координат в буфере экрана, используется функция `FillConsoleOutputAttribute`:

```
BOOL WINAPI FillConsoleOutputAttribute(
_In_ HANDLE hConsoleOutput,
_In_ WORD wAttribute,
_In_ DWORD nLength,
_In_ COORD dwWriteCoord,
_Out_ LPDWORD lpNumberOfAttrsWritten);
```

с такими параметрами:

1) `hConsoleOutput` – дескриптор буфера вывода консоли;
2) `wAttribute` – атрибут цвета символа и его фона в консоли;
3) `nLength` – количество ячеек символов, которые устанавливаются в заданный цвет;
4) `dwWriteCoord` – координаты первой закрашиваемой ячейки;
5) `lpNumberOfAttrsWritten` – указатель на идентификатор, в который записывается количество реально закрашенных ячеек.

4.3.5 Получение информации о клавиатуре и мыши

Для получения информации о клавиатуре и мыши в консольном режиме используется функция `ReadConsoleInput`:

```
BOOL WINAPI ReadConsoleInput(
_In_ HANDLE hConsoleInput,
_Out_ PINPUT_RECORD lpBuffer,
_In  DWORD nLength,
_Out_ LPDWORD lpNumberOfEventsRead);
```

Имеющая параметры:

1) `hConsoleInput` – дескриптор входного буфера консоли;
2) `lpBuffer` – указатель на структуру (или массив структур) `INPUT_RECORD`, в которой содержится информация о событиях, происшедших с консолью;
3) `nLength` – количество получаемых информационных записей (структур);
4) `lpNumberOfEventsRead` – указатель на двойное слово, содержащее количество реально полученных записей.

Структура `INPUT_RECORD` имеет вид

```
INPUT_RECORD STRUCT
  EventType              WORD ?
  two_byte_alignment     WORD ?
  UNION
    KeyEvent                 KEY_EVENT_RECORD            <>
    MouseEvent               MOUSE_EVENT_RECORD          <>
    WindowBufferSizeEvent    WINDOW_BUFFER_SIZE_RECORD   <>
```

```
    MenuEvent                MENU_EVENT_RECORD          <>
    FocusEvent               FOCUS_EVENT_RECORD         <>
  ENDS
INPUT_RECORD ENDS
```

Структура `INPUT_RECORD` используется для получения события, происшедшего в консоли.

Всего системой зарезервировано пять типов событий:

```
KEY_EVENT equ 1h ; клавиатурное событие
MOUSE_EVENT equ 2h  ; событие с мышью
WINDOW_BUFFER_SIZE_EVENT equ 4h ; изменился размер окна
MENU_EVENT equ 8h ; зарезервировано
FOCUS_EVENT equ 10h ; зарезервировано
```

Остальные байты структуры зависят от происшедшего события.

Событие `KEY_EVENT` описывается структурой

```
KEY_EVENT_RECORD STRUCT
bKeyDown DD ?   ; при нажатии клавиши значение поля больше нуля
wRepeatCount DW ?    ; количество повторов при удержании клавиши
wVirtualKeyCode DW ?     ; виртуальный код клавиши
wVirtualScanCode DW ?    ; скан-код клавиши
UNION
   ; для функции (ReadConsoleInputW)
   UnicodeChar DW ?      ; код символа в формате Unicode
   ; для функции (ReadConsoleInputA)
   AsciiChar DB ?        ; код символа в формате ASCII
ENDS;
dwControlKeyState DD ?   ; состояния управляющих клавиш
; может являться суммой таких констант:
; RIGHT_ALT_PRESSED equ 1h
; LEFT_ALT_PRESSED equ 2h
; RIGHT_CTRL_PRESSED equ 4h
; LEFT_CTRL_PRESSED equ 8h
; SHIFT_PRESSED equ 10h
; NUMLOCK_ON equ 20h
; SCROLLLOCK_ON equ 40h
; CAPSLOCK_ON equ 80h
; ENHANCED_KEY equ 100h
KEY_EVENT_RECORD ENDS
```

Событие `MOUSE_EVENT` описывается структурой

```
MOUSE_EVENT_RECORD STRUCT
dwMousePosition COORD <>      ; координаты курсора мыши
dwButtonState DD ?            ; состояние кнопок мыши
; первый бит - левая кнопка
; второй бит - правая кнопка
; третий бит - средняя кнопка
; бит установлен - кнопка нажата
```

```
dwControlKeyState DD ?   ; состояние управляющих клавиш
dwEventFlags DD ?        ; может содержать значения:
; MOUSE_MOVED equ 1h     ; было движение мыши
; DOUBLE_CLICK equ 2h    ; был двойной щелчок
MOUSE_EVENT_RECORD ENDS
```

Событие `WINDOW_BUFFER_SIZE_EVENT` описывается структурой

```
WINDOW_BUFFER_SIZE_RECORD STRUCT
dwSize COORD <>          ; новый размер консольного окна
WINDOW_BUFFER_SIZE_RECORD ENDS
```

События `MENU_EVENT_RECORD` и `FOCUS_EVENT_RECORD` зарезервированы и описываются такими структурами:

```
MENU_EVENT_RECORD STRUCT
dwCommandId DD ?
MENU_EVENT_RECORD ENDS

FOCUS_EVENT_RECORD STRUCT
bSetFocus DD ?
FOCUS_EVENT_RECORD ENDS
```

Структура, объединяющая все типы событий – это структура `INPUT_RECORD`.

4.4 Пример консольного приложения

Рассмотрим пример консольного приложения подсчета количества символов в строке. Программа состоит из двух файлов. Файл *string_length.inc* содержит основные конструкции и константы Windows, необходимые при написании программы, а также основные функции. Файл *string_length.asm* содержит текст программы:

```
; файл string_length.inc
; прототипы внешних функций
FreeConsole  PROTO
AllocConsole PROTO
GetStdHandle PROTO :DWORD
SetConsoleTitleA PROTO :DWORD
WriteConsoleA PROTO :DWORD,:DWORD,:DWORD,:DWORD,:DWORD
ReadConsoleA  PROTO :DWORD,:DWORD,:DWORD,:DWORD,:DWORD
ReadConsoleInputA PROTO :DWORD,:DWORD,:DWORD,:DWORD
SetConsoleScreenBufferSize  PROTO :DWORD, :DWORD
SetConsoleCursorPosition    PROTO :DWORD, :DWORD
SetConsoleTextAttribute  PROTO :DWORD, :DWORD
FillConsoleOutputAttribute PROTO :DWORD, :DWORD, :DWORD,\
```

```
                                    :DWORD, :DWORD
CharToOemA PROTO :DWORD, :DWORD
ExitProcess PROTO :DWORD

; прототипы своих функций
ReadSymbol  PROTO :DWORD, :DWORD, :DWORD
IntToStr PROTO :DWORD, :DWORD
LENSTR PROTO :DWORD
PrintStr PROTO :DWORD, :DWORD

; константы дескрипторов буфера
STD_INPUT_HANDLE equ -10
STD_OUTPUT_HANDLE equ -11
STD_ERROR_HANDLE equ -12

; структура координат
COORD STRUC
     X DW ?
     Y DW ?
COORD ENDS

; цвет окна консоли
FOREGROUND_BLUE equ 1h         ; синий цвет букв
FOREGROUND_GREEN equ 2h        ; зеленый цвет букв
FOREGROUND_RED equ 4h          ; красный цвет букв
FOREGROUND_INTENSITY equ 8h    ; повышенная интенсивность
BACKGROUND_BLUE equ 10h        ; синий цвет фона
BACKGROUND_GREEN equ 20h       ; зеленый цвет фона
BACKGROUND_RED equ 40h         ; красный цвет фона
BACKGROUND_INTENSITY equ 80h   ; повышенная интенсивность

; тип события
KEY_EVENT equ 1h                ; клавиатурное событие
MOUSE_EVENT equ 2h              ; событие с мышью
WINDOW_BUFFER_SIZE_EVENT equ 4h ; изменился размер окна
MENU_EVENT equ 8h               ; зарезервировано
FOCUS_EVENT equ 10h             ; зарезервировано

; константы - состояния клавиатуры
RIGHT_ALT_PRESSED equ 1h
LEFT_ALT_PRESSED equ 2h
RIGHT_CTRL_PRESSED equ 4h
LEFT_CTRL_PRESSED equ 8h
SHIFT_PRESSED equ 10h
NUMLOCK_ON equ 20h
SCROLLLOCK_ON equ 40h
CAPSLOCK_ON equ 80h
ENHANCED_KEY equ 100h

; события мыши
MOUSE_MOVED equ 1h; было движение мыши
DOUBLE_CLICK equ 2h; был двойной щелчок

; описание событий структуры INPUT_RECORD
; событие мыши
MOUSE_EVENT_RECORD STRUCT
   dwMousePosition COORD <>
   dwButtonState DWORD ?
```

```
   dwControlKeyState DWORD ?
   dwEventFlags DWORD ?
MOUSE_EVENT_RECORD ENDS

; событие клавиатуры
KEY_EVENT_RECORD STRUCT
   bKeyDown DD ?
   wRepeatCount DW ?
   wVirtualKeyCode DW ?
   wVirtualScanCode  DW ?
   UNION
        UnicodeChar DW ?
        AsciiChar DB ?
   ENDS
   dwControlKeyState DD ?
KEY_EVENT_RECORD ENDS

; изменение размера окна консоли
WINDOW_BUFFER_SIZE_RECORD STRUCT
   dwSize COORD <>
WINDOW_BUFFER_SIZE_RECORD ENDS

MENU_EVENT_RECORD STRUCT
   dwCommandId DWORD ?
MENU_EVENT_RECORD ENDS

FOCUS_EVENT_RECORD STRUCT
   bSetFocus DWORD ?
FOCUS_EVENT_RECORD ENDS

; Структура INPUT_RECORD
INPUT_RECORD STRUCT
   EventType DW ?
   DW ?
   UNION
       KeyEvent KEY_EVENT_RECORD <>
       MouseEvent MOUSE_EVENT_RECORD <>
       WindowBufferSizeEvent WINDOW_BUFFER_SIZE_RECORD <>
       MenuEvent MENU_EVENT_RECORD <>
       FocusEvent FOCUS_EVENT_RECORD <>
   ENDS
INPUT_RECORD ENDS

; Секция данных содержит временные переменные
.data
@CO DD ?
@numBytes DD ?
KeyEvent INPUT_RECORD <>
@SYMBOL DB ?

.code

;-------------------------------------------------------------

; Функция считывания символа в консоли
; consoleInHandle - дескриптор буфера консоли ввода
; consoleOutHandle - дескриптор буфера консоли вывода
; Display - управление отображением символа:
```

```
; 0 - символ отображается
; 1 - символ не отображается
; функция возвращает считанный символ в регистре al

ReadSymbol proc consoleInHandle:DWORD, consoleOutHandle:DWORD,\
                                        Display:DWORD
@L1:
INVOKE ReadConsoleInputA, consoleInHandle, \
                          offset KeyEvent, 1, offset @CO
CMP KeyEvent.EventType, KEY_EVENT
JNE @L1
; сохранение введенного символа
MOV AL, KeyEvent.KeyEvent.AsciiChar
MOV @SYMBOL, AL
CMP Display,0
JNE @L2
; вывод символа
INVOKE WriteConsoleA, consoleOutHandle, OFFSET @SYMBOL,\
                      1, OFFSET @numBytes, 0
; Считывание события клавиатуры отпускания клавиши
@L2:
INVOKE ReadConsoleInputA, consoleInHandle,\
                          offset KeyEvent, 1,offset @CO
CMP KeyEvent.EventType, KEY_EVENT
JNE @L2
mov eax,0
mov al, @SYMBOL
ret
ReadSymbol endp

;-------------------------------------------------------------

; Функция представления целого числа в текстовой форме
; Number - целое число
; Str1 - указатель на строку, в которую будет помещено
; представление числа
; функция возвращает длину строки символов в регистре EAX

IntToStr proc Number:DWORD, Str1:DWORD
MOV EAX, Number
MOV EDI, Str1
MOV ECX, 0
CMP EAX,0
JGE @I1
; число отрицательное,
; заносим знак «минус» в Str1, получаем дополнительный код
MOV DL, '-'
MOV [EDI], DL
INC EDI
NOT EAX
INC EAX
@I1:
; ноль или положительное число
; делим на 10, занося остатки в стек
MOV EBX, 10
```

```
MOV EDX, 0
IDIV EBX
PUSH EDX
INC ECX
CMP EAX,0
JG @I1
@I2:
; достаем из стека цифры, добавляем 30h и заносим в Str1
POP EDX
ADD DL, 30h
MOV [EDI],DL
INC EDI
LOOP @I2
; заносим 0 в конец строки Str1
MOV DL,0
MOV [EDI], DL
; вычисляем длину строки Str1
INC EDI
MOV EAX, EDI
SUB EAX, Str1
ret
IntToStr endp
;-------------------------------------------------------------
; Определение длины строки
; Str1 - указатель на строку
; функция возвращает длину строки символов в регистре EAX
LENSTR PROC Str1:DWORD
CLD ; обработка строки слева направо
MOV EDI, Str1  ; указатель на первый символ строки
MOV EBX, EDI   ; сохраняем этот указатель в регистре EBX
MOV ECX, 100   ; ограничить длину строки
MOV EAX, 0     ; символ, который ищем в строке
REPNE SCASB    ; найти символ 0
SUB EDI, EBX   ; длина строки, включая 0
MOV EAX, EDI
DEC EAX        ; вычитаем единицу из длины строки
RET
LENSTR ENDP
;-------------------------------------------------------------
; Вывод строки в окно консоли
; StrPtr - указатель на выводимую строку, оканчивающуюся 0
; consoleOutHandle - дескриптор буфера консоли вывода
PrintStr proc StrPtr:DWORD, Handle:DWORD
INVOKE CharToOemA, StrPtr, StrPtr
INVOKE LENSTR, StrPtr; определение длины строки
INVOKE WriteConsoleA, Handle, StrPtr, eax, OFFSET @numBytes, 0
ret
PrintStr endp

; файл string_length.asm
```

```
.686p
.MODEL FLAT, stdcall
include string_length.inc

.DATA
consoleOutHandle DD ?    ; дескриптор выходного буфера
consoleInHandle DD ?     ; дескриптор входного буфера
COUNT DD 0               ; счетчик количество символов
numBytes DD ?
TITL DB "Подсчет количества символов в строке",0
IN_STR DB "Введите строку: ",0
IN_SYM DB "Введите символ: ",0
BUF DB 200 dup (?) ; для ввода строки
Len DD ? ; длина введенной строки
Yes DB 13,10,"Количество символов: ",0
No DB 13,10,"Символ не найден.",0
SymCount DB 10 dup(?)    ; строка с количеством символов
CRD COORD <?>            ; структура координат

.CODE

START:
; образовать консоль, вначале освободить уже существующую
INVOKE FreeConsole
INVOKE AllocConsole
; получить дескрипторы консолей ввода и вывода
INVOKE GetStdHandle, STD_INPUT_HANDLE
MOV consoleInHandle, EAX
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
MOV consoleOutHandle, EAX
; задать заголовок окна консоли
INVOKE CharToOemA, OFFSET TITL, OFFSET TITL
INVOKE SetConsoleTitleA, OFFSET TITL
; задать размер окна консоли
MOV CRD.X, 80
MOV CRD.Y, 25
MOV EAX, CRD
INVOKE SetConsoleScreenBufferSize, consoleOutHandle, EAX
; задать цвет окна консоли
INVOKE FillConsoleOutputAttribute, consoleOutHandle,\
                BACKGROUND_BLUE + BACKGROUND_GREEN,\
                2000, 0, offset numBytes
; установить позицию курсора
MOV CRD.X,0
MOV CRD.Y,2
MOV EAX, CRD
INVOKE SetConsoleCursorPosition, consoleOutHandle, EAX
; задать цветовые атрибуты выводимого текста
INVOKE SetConsoleTextAttribute, consoleOutHandle,\
     FOREGROUND_BLUE + BACKGROUND_BLUE + BACKGROUND_GREEN
; вывод сообщения «Введите строку:»
INVOKE PrintStr, offset IN_STR, consoleOutHandle
; Ввод строки
INVOKE ReadConsoleA, consoleInHandle, OFFSET BUF, 200,\
                                  OFFSET numBytes, 0
```

```
; сохранить длину введенной строки
MOV EAX, numBytes
MOV Len, EAX
; вывод сообщения «Введите символ:»
INVOKE PrintStr, offset IN_SYM, consoleOutHandle
; считывание символа
INVOKE ReadSymbol, consoleInHandle, consoleOutHandle, 0
; поиск символа в строке
; символ содержится в регистре AL
CLD ; поиск символа с начала строки DF = 0
LEA EDI, BUF;
MOV ECX, Len; количество повторений равно длине строки
FIND:
Repne SCASB
JNE FAILED; просмотр строки завершен
INC COUNT
JMP FIND
; символ не найден
FAILED:
CMP COUNT, 0
JNE FOUND ; количество символов не 0 – идем на FOUND
INVOKE PrintStr, offset No, consoleOutHandle
JMP EXIT
; символ найден
FOUND:
INVOKE PrintStr, offset Yes, consoleOutHandle
INVOKE IntToStr, COUNT, offset SymCount
INVOKE PrintStr, offset SymCount, consoleOutHandle
EXIT:
INVOKE ReadSymbol, consoleInHandle, consoleOutHandle, 1
INVOKE ExitProcess,0
END START
```

На рис. 4.1 показана работа консольного приложения для подсчета количества символов в строке.

Рис. 4.1 – Подсчет количества символов в строке

Тема 5 Оконные приложения

5.1 Сообщения и их структура

Главным элементом оконного приложения в среде Windows является *окно*, которое может содержать *элементы управления*: кнопки, списки, окна редактирования, полосы прокрутки и т. п. *Все они также являются окнами, обладающими особыми свойствами*. События, происходящие с элементами управления и с самим окном, приводят к формированию *сообщений* – специально оформленных групп данных.

Приложение не знает порядка появления сообщений, поэтому оно должно быть построено таким образом, чтобы обеспечить корректную и предсказуемую работу при поступлении сообщений любого типа – *системных* или *пользовательских*.

Отличительным признаком сообщения является его код, который для системных сообщений лежит в диапазоне от 1 до 0x3FF. Так как с кодами работать в программе неудобно, то каждому коду сопоставляется своя символическая константа, по имени которой можно определить источник сообщения. Например, при перемещении мыши возникает

сообщение WM_MOUSEMOVE (код 0x200), при нажатии на левую кнопку мыши – сообщение WM_LBUTTONDOWN (код 0x201). При перерисовке окно получает сообщение WM_PAINT. Эти события относятся к классу аппаратных, поскольку в их обработке участвуют драйверы внешних устройств. Например, при нажатии клавиши драйвер клавиатуры формирует пакет данных и пересылает его в форме сообщения в системную очередь сообщений Windows. Рассмотрим дальнейшую судьбу сообщения.

Для приложений время для выполнения распределяется между потоками приложения. *Как минимум приложение создает один поток*. *Для каждого потока создается своя очередь сообщений, которая не имеет фиксированного размера*. Сообщения из системной очереди распределяются между очередями сообщений потоков, откуда затем извлекаются приложением с помощью функции `GetMessage()`.

В какую очередь потока направляется сообщение из системной очереди? Предположим, что на экране много окон приложений и возникло событие, инициированное мышью. В этом случае сообщение от мыши будет адресованы потоку-владельцу окна, над которым находится курсор. Значит, сообщения от мыши ставятся в очередь потока, активизировавшего текущее окно.

Источниками сообщений, помимо устройств, могут быть прикладные программы или операционные системы. Если сообщение создается в прикладной программе и посылается в Windows, чтобы операционная система выполнила требуемые действия, то оно имеет код, превышающий 0x3FF. Программист может предусмотреть собственные сообщения и направлять их в различные окна приложения для оповещения о различных ситуациях в вычислительной системе.

Сообщения передаются в приложение с помощью специальной структуры `MSG`, включающей шесть полей и описанной в файле *Windows.inc*.

Поля структуры `MSG`:

1) дескриптор окна, которому адресовано сообщение;

2) код данного сообщения;

3) дополнительная информация `wParam`;

4) дополнительная информация `lParam`;

5) время отправления сообщения;

6) позиция курсора мыши на момент отправления сообщения.

Все объекты (окна, файлы, процессы, потоки, события и т. д.) и системные ресурсы в Windows описываются с помощью *дескрипторов*. Различают дескрипторы экземпляров приложений (Handle of Instance, HInstance), окон (HWND), пиктограмм (HIcon), шрифтов (HFont), перьев (HPen) и т. д. Определения этих дескрипторов доступны при подключении файла *Windows.inc*. Так как сообщение посылается определенному окну, то его дескриптор указывается в структуре сообщения. Поля `wParam` и `lParam` содержат дополнительные данные, необходимые для обработки сообщения. Их содержимое различается для сообщений каждого типа. Например, для сообщения WM_MOUSEMOVE поле `wParam` содержит информацию о состоянии клавиш мыши (нажаты или отпущены), а также о состоянии клавиш Ctrl и Shift, а поле `lParam` содержит позицию курсора относительно начала клиентской области окна.

Приложение может обрабатывать не все сообщения, часть из них обрабатываются самой операционной системой либо до попадания сообщения в очередь потока, либо после извлечения сообщения из очереди и «осознания», что приложение сообщение данного типа не интересует. Рассмотрим первый из этих случаев. Например, если щелкнуть левой кнопкой мыши на пункте меню, то вместо сообщения WM_LBUTTONDOWN формируется сообщение WM_COMMAND, параметры которого содержат идентификатор пункта меню, над которым был курсор мыши в момент щелчка. Это избавляет приложение от

необходимости анализа положения курсора мыши при оценке, попадает ли курсор в прямоугольную область, соответствующую пункту меню.

При программировании под Windows необходимо знать правила наименования различных объектов. Для конструирования имен объектов используется система венгерской записи, согласно которой перед именем объекта ставятся символы, по которым можно определить тип переменной. Эти символы называются *префиксом*. Так, префикс «`sz`» (String Zero) означает символьную строку, заканчивающуюся двоичным нулем, «`c`» (Char) – символ, «`dw`» (Double Word) – 32-битную переменную, «`lpsz`» (Long Pointer of String Zero) – указатель на символьную строку, заканчивающуюся двоичным нулем, «`h`» (Handle) – дескриптор (описатель) объекта, содержащий информацию об объекте.

Сообщения от Windows имеют префикс «`WM_`», префикс «`BM_`» соответствует сообщениям от кнопок, префикс «`EM_`» – сообщениям от текстовых полей редактирования (EditBox), «`LB_`» – сообщениям от списков.

5.2 Оконные сообщения и функции работы с окнами

Окно – это не только область на экране для вывода, но еще и адресат событий и сообщений в среде Windows.

Окно идентифицируется по дескриптору, который является переменной типа HWND и однозначно определяет каждое окно в системе. Windows организует свои окна в иерархическую структуру. Каждое окно имеет родителя. Корнем дерева всех окон является окно рабочего стола, создаваемого Windows при загрузке.

Для всех окон верхнего уровня (для главных окон приложений и других перекрывающихся и всплывающих окон приложений) родительским окном является рабочий стол.

Родитель дочернего окна – окно верхнего уровня или другое дочернее окно, более высокого уровня по иерархии.

Между окнами верхнего уровня (перекрывающиеся и всплывающие окна) существует еще одна иерархическая связь. Владельцем окна верхнего уровня может быть другое окно того же уровня. Окно, имеющее владельца, всегда отображается поверх него и исчезает при его минимизации. Типичным случаем владения одного окна верхнего уровня другим является приложение, отображающее диалоговое окно. Диалоговое окно не является дочерним (оно всплывающее), но его владельцем остается окно приложения.

Рассмотрим наиболее часто обрабатываемые сообщения:

1. WM_CREATE – посылается окну перед тем, как оно станет видимым, при получении сообщения приложение может инициализировать нужные данные;

2. WM_DESTROY – посылается окну, которое уже удалено с экрана и должно быть разрушено;

3. WM_CLOSE – указывает, что окно должно быть закрыто. Приложение может при его обработке, например, вывести диалоговое окно подтверждения закрытия окна;

4. WM_QUIT – сообщение, требующее завершить приложение;

5. WM_ACTIVATE – указывает, что окно верхнего уровня будет активизировано или деактивизировано. Сообщение сначала посылается окну, которое должно быть деактивизировано, а потом окну, которое должно быть активизировано;

6. WM_SHOWWINDOW – указывает, что окно должно быть скрыто или отображено;

7. WM_ENABLE – посылается окну, когда оно становится доступным или недоступным. Недоступное окно не может принимать вводимые данные от мыши или клавиатуры;

8. WM_MOVE – указывает, что расположение окна изменилось;

9. WM_SIZE – указывает, что размер окна был изменен;

10. WM_SETFOCUS – указывает получение окном фокуса клавиатуры;

11. WM_KILLFOCUS – указывает, что окно должно потерять фокус клавиатуры;

Рассмотрим функции, позволяющие приложению исследовать иерархию окон, находить, перемещать, изменять режим отображения, изменять вид окна:

1. *AnimateWindow* – дает возможность производить специальные эффекты при показе или сокрытии. Существуют четыре типа мультипликации: ролик, слайд, свертывание или развертывание и плавное перетекание;

2. *CloseWindow* – закрывает (но не разрушает) указанное окно;

3. *FindWindow* – используется для поиска окна верхнего уровня по имени его класса или по его заголовку;

4. *FlashWindow* – предназначена для создания окна с мигающим заголовком, используется для привлечения внимания к нему;

5. *FlashWindowEx* – усовершенствованный вариант *FlashWindow*;

6. *GetClientRect* – возвращает координаты клиентской области;

7. *GetParent* – возвращает дескриптор родительского окна;

8. *GetDesktopWindow* – возвращает дескриптор окна рабочего стола;

9. *GetTitleBarInfo* – возвращает информацию о строке заголовка;

10. *GetWindow* – предоставляет наиболее гибкий способ работы с иерархией окон. В зависимости от значения второго параметра эту функцию можно использовать для получения идентификатора родительского окна, владельца, окон одного уровня или дочерних окон;

11. *GetWindowPlacement* – возвращает данные о расположении;

12. *GetWindowTextLength* – возвращает длину (количество символов) текста строки заголовка, если имеется область заголовка. Если окно – элемент управления, функция возвращает длину текста внутри элемента управления;

13. *IsChild* – проверяет, является ли окно дочерним или порожденным для указанного родительского окна;

14. *IsWindow* – определяет, существует ли окно с заданным дескриптором;

15. *IsWindowVisible* – возвращает информацию о состоянии заданного окна;

16. *MoveWindow* – изменяет расположение и размеры. Для окна верхнего уровня расположение вычисляется относительно левого верхнего угла экрана. Для дочернего окна расположение вычисляется относительно левого верхнего угла клиентской области родительского окна;

17. *OpenIcon* – восстанавливает свернутое окно;

18. *SetWindowPlacement* – устанавливает в состояние показа и восстанавливает, свертывает и развертывает;

19. *SetWindowText* – копирует текст строки из буфера в заголовок окна. Если окно – элемент управления, текст из буфера копируется в элемент управления;

20. *ShowWindow* – устанавливает состояние показа;

21. *WindowFromPoint* – отыскивает дескриптор окна, которое содержит заданную точку.

5.3 Минимальное оконное приложение

Итак, оконные приложения строятся по принципам *событийно-управляемого программирования* – стиля программирования, при котором поведение компонента системы определяется набором возможных внешних событий и ответных реакций компонента на них. Такими компонентами в Windows являются окна.

С каждым окном в Windows связана определенная функция обработки событий – *оконная функция*. События для окон называются *сообщениями*. Сообщение относится к тому или иному типу, идентифицируемому определенным кодом (32-битным целым числом), и сопровождается парой 32-битных параметров (`wParam` и `lParam`), интерпретация которых зависит от типа сообщения.

Задача любого оконного приложения – создать главное окно и сообщить Windows функцию обработки событий для этого окна. Все основное для приложения будет происходить именно в функции обработки событий главного окна.

В Windows программа пассивна. После запуска она ждет, когда ей уделит внимание операционная система, которая делает это посылкой сообщений. Сообщения могут быть разного типа, они функционируют в системе достаточно хаотично, и приложение не знает, какого типа сообщение придет следующим. Логика построения Windows-приложения должна обеспечивать корректную и предсказуемую работу при поступлении сообщений любого типа.

Классическое оконное приложение, как правило, состоит по крайней мере из двух функций:

1) *стартовой*, создающей главное окно WinMain;

2) *обработки сообщений окна* (оконная функция WndProc).

Выполнение любого оконного приложения начинается с главной функции. Она содержит код, осуществляющий инициализацию приложения в операционной системе, с помощью которого система узнает о новом приложении и его свойствах. Для этого в WinMain описывается и регистрируется класс окна приложения, а затем создается и отображается на экране окно приложения зарегистрированного класса. Видимым для пользователя результатом работы главной функции является появление на экране нового графического объекта – окна. Последним действием кода главной функции является создание цикла обработки сообщений. После его создания приложение начинает взаимодействовать с вычислительной системой через сообщения.

Обработка же поступающих приложению сообщений осуществляется с помощью *специальной функции*, называемой *оконной*. *Оконная функция* уникальна тем, что может быть вызвана только из операционной системы, а не из приложения, которое ее содержит.

Рассмотрим пример создания оконного приложения Windows с подробными комментариями:

```
.686p
.model flat, stdcall
option casemap:none
```

1. Задаем прототипы используемых приложением API-функций

Получение дескриптора экземпляра приложения:

```
GetModuleHandleA PROTO STDCALL :DWORD
```

Загрузка стандартного курсора или курсора из файла ресурсов и получение дескриптора курсора:

```
LoadCursorA PROTO STDCALL :DWORD,:DWORD
```

Загрузка стандартной иконки или иконки из файла ресурсов и получение дескриптора иконки:

```
LoadIconA PROTO STDCALL :DWORD,:DWORD
```

Регистрация класса окна:

```
RegisterClassExA PROTO STDCALL :DWORD
```

Создание окна и получение дескриптора окна:

```
CreateWindowExA PROTO STDCALL :DWORD,:DWORD, :DWORD, :DWORD,
:DWORD, :DWORD, :DWORD, :DWORD, :DWORD, :DWORD, :DWORD, :DWORD
```

Установка состояния показа окна:

```
ShowWindow PROTO STDCALL :DWORD,:DWORD
```

Обновление содержимого окна посылкой сообщения WM_PAINT непосредственно в оконную процедуру:

```
UpdateWindow PROTO STDCALL :DWORD
```

Извлечение сообщения из очереди:

```
GetMessageA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD
```

Перевод сообщений формата виртуальных клавиш в символьные сообщения:

```
TranslateMessage  PROTO STDCALL :DWORD
```

Пересылка сообщения оконной процедуре:

```
DispatchMessageA  PROTO STDCALL :DWORD
```

Вызов заданной по умолчанию оконной процедуры для обеспечения обработки по умолчанию любых сообщений окна, которые приложение не

обрабатывает. Эта функция гарантирует, что каждое сообщение будет обработано. Она должна быть вызвана с теми же параметрами, какие и у оконной процедуры.

```
DefWindowProcA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD
```

Указание системе, что поток сделал запрос на прекращение работы. Обычно используется в ответ на сообщение `WM_DESTROY`:

```
PostQuitMessage PROTO STDCALL :DWORD
```

Получение указателя на командную строку, содержащую имя текущей программы и ее аргументы:

```
GetCommandLineA PROTO STDCALL
```

Создание логической кисти указанного цвета:

```
CreateSolidBrush PROTO STDCALL :DWORD
```

Завершение процесса и всех его потоков:

```
ExitProcess PROTO STDCALL :DWORD
```

Условное название точки входа приложения:

```
WinMain PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD.
```

2. Описываем структуры приложения

Структура, содержащая информацию о координатах указателя мыши в момент помещения сообщения в очередь `POINT`:

```
POINT STRUCT
     ; координата X указателя мыши в момент отправки сообщения
     x DWORD ?
     ; координата Y указателя мыши в момент отправки сообщения
     y DWORD ?
POINT ENDS
```

Структура с информацией о сообщении `MSG`:

```
MSG STRUCT
     ; дескриптор окна, которому предназначено сообщение
     hwnd DWORD ?
     ; идентификатор сообщения
     message DWORD ?
     ; параметр сообщения wParam
     wParam DWORD ?
     ; параметр сообщения lParam
     lParam DWORD ?
     ; время помещения сообщения в очередь
     time DWORD ?
     ; координаты указателя мыши в момент отправки сообщения
```

```
     pt POINT <>
MSG ENDS
```

Структура, содержащая атрибуты класса окна `WNDCLASSEXA`:

```
WNDCLASSEXA STRUCT
     ; размер структуры в байтах (30h)
     cbSize DWORD ?
     ; флаги, указывающие стили класса
     style DWORD ?
     ; указатель на оконную процедуру
     lpfnWndProc DWORD ?
     ; количество дополнительных байтов класса, которые
     ; размещаются вслед за структурой класса окна
     cbClsExtra DWORD ?
     ; количество дополнительных байтов окна, которые
     ; размещаются вслед за внутренней структурой окна
     cbWndExtraDWORD ?
     ; дескриптор экземпляра приложения, который содержит
     ; оконную процедуру для класса
     hInstance DWORD ?
     ; дескриптор значка класса, дескриптор ресурса значка
     hIcon DWORD ?
     ; дескриптор курсора класса, дескриптор ресурса курсора
     hCursor DWORD ?
     ; дескриптор кисти фона класса, дескриптор физической
     ; кисти, которая используется при создании фона окна
     hbrBackground DWORD ?
     ; указатель на символьную строку с именем ресурса меню
     класса(если поле равно 0, у класса окна нет меню)
     lpszMenuNameDWORD ?
     ; указатель на символьную строку с именем класса
     lpszClassName DWORD ?
     ; дескриптор мелкого значка, связанного с данным окном
     hIconSm DWORD ?
WNDCLASSEXA ENDS
```

3. Определяем данные в секции данных

```
.data
     ; символьная строка, которая определяет имя класса окна
     ClassName db "SimpleWinClass",0
     ; символьная строка, которая определяет имя окна
     AppName   db "Наше первое окно",0
     ; дескриптор экземпляра приложения
     hInstance dd 00000000h
     ; указатель на командную строку текущего процесса
     CommandLine    dd 00000000h
```

4. Определяем константы

```
.const
     ; идентификатор сообщения, генерируемого при закрытии окна
     WM_DESTROY equ 2h
```

```
; идентификатор сообщения, генерируемого при нажатии
; несистемной клавиши
WM_KEYDOWN equ 100h
; виртуальный код клавиши ESC
VK_ESCAPE equ 1Bh
; идентификатор стандартной иконки
IDI_APPLICATIONequ 32512
; идентификатор стандартного курсора-стрелки
IDC_ARROW equ 32512
; идентификатор нормального режима показа окна
SW_SHOWNORMAL equ 1
; идентификатор перерисовки окна при изменении ширины окна
CS_HREDRAW equ 2h
; идентификатор перерисовки окна при изменении высоты окна
CS_VREDRAW equ 1h
; идентификатор цвета кисти фона класса окна
COLOR_BTNFACE equ 15
; идентификатор заданной по умолчанию позиции или размера
CW_USEDEFAULT equ 80000000h
; идентификатор перекрытия окна другими окнами
WS_OVERLAPPED equ 0h
; идентификатор окна с заголовком
WS_CAPTION equ 0C00000h
; идентификатор окна с системным меню в заголовке
WS_SYSMENU equ 80000h
; идентификатор окна с рамкой
WS_THICKFRAME equ 40000h
; идентификатор окна с кнопкой свертывания окна
WS_MINIMIZEBOX equ 20000h
; идентификатор окна с кнопкой развертывания окна
WS_MAXIMIZEBOX equ 10000h
; идентификатор перекрывающегося окна с заголовком,
; системным меню, рамкой, кнопками свертывания и
; развертывания
WS_OVERLAPPEDWINDOW equ WS_OVERLAPPED OR WS_CAPTION OR
WS_SYSMENU OR WS_THICKFRAME OR WS_MINIMIZEBOX OR
WS_MAXIMIZEBOX
; идентификатор показа окна способом, заданным по умолчанию
SW_SHOWDEFAULT equ 10
```

5. Определяем команды и функции секции кода

```
.code
start:
; получаем дескриптор экземпляра приложения hInstance
     invoke GetModuleHandleA, 0
     mov hInstance,eax
; получаем указатель на командную строку текущего процесса
     invoke GetCommandLineA
     mov CommandLine,eax
; вызываем главную функцию
     invoke WinMain, hInstance, 0 , CommandLine, SW_SHOWDEFAULT
; завершаем процесс
```

```
     invoke ExitProcess,eax

COMMENT *
Главная функция приложения WinMain. Заполняет поля структуры,
содержащей атрибуты класса окна, регистрацию класса окна,
создание, отображение и обновление содержимого окна,
организует цикл обработки сообщений.
*
     WinMain proc hInst:DWORD ,hPrevInst:DWORD, CmdLine:DWORD,
                                                CmdShow:DWORD
     ; локальная переменная типа структуры класса окна
     LOCAL wc:WNDCLASSEXA
     ; локальная переменная типа структуры сообщения
     LOCAL msg:MSG
     ; локальная переменная – дескриптор окна
     LOCAL hwnd:DWORD
; заполнение полей структуры с атрибутами класса окна
     ; размер структуры
     mov wc.cbSize,SIZEOF WNDCLASSEXA
     ; перерисовка окна при изменении его ширины и высоты
     mov wc.style, CS_HREDRAW or CS_VREDRAW
     ; указатель на функцию обработки сообщений окна
     mov wc.lpfnWndProc, OFFSET WndProc
     ; число дополнительных байт за структурой класса окна
     mov wc.cbClsExtra,0
     ; число дополнительных байт за экземпляром окна
     mov wc.cbWndExtra,0
     ; дескриптор экземпляра приложения
     push hInst
     pop wc.hInstance
     ; фон окна – синий
     invoke CreateSolidBrush,00FF0000h
     mov wc.hbrBackground,eax
     ; оконное меню отсутствует
     mov wc.lpszMenuName,0
     ; имя класса окна
     mov wc.lpszClassName,OFFSET ClassName
     ; стандартный значок и мелкий значок приложения
     invoke LoadIconA,0,IDI_APPLICATION
     mov wc.hIcon,eax
     mov wc.hIconSm,eax
     ; стандартный курсор-стрелка
     invoke LoadCursorA,0,IDC_ARROW
     mov wc.hCursor,eax

; регистрация класса окна
     invoke RegisterClassExA, addr wc

COMMENT *
Создание окна. Параметры: дополнительный стиль, имя класса,
заголовок, стиль, позиция x, позиция y, ширина, высота,
дескриптор окна-родителя, дескриптор меню, дескриптор
экземпляра приложения, указатель на данные
```

```
*
     invoke CreateWindowExA,0,ADDR ClassName, ADDR AppName,\
     WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,\
     CW_USEDEFAULT,CW_USEDEFAULT,0,0, hInst,0
     mov hwnd,eax; описатель созданного окна

; отображение окна
     invoke ShowWindow, hwnd,SW_SHOWNORMAL

; обновление содержимого окна
     invoke UpdateWindow, hwnd

; цикл обработки сообщений
     .WHILE 1
          ; получить сообщение
          invoke GetMessageA, ADDR msg,0,0,0
          .BREAK .IF (!eax)
          ; преобразовать аппаратное сообщение в символьное
          invoke TranslateMessage, ADDR msg
          ; передать сообщение оконной функции для обработки
          invoke DispatchMessageA, ADDR msg
     .ENDW
     mov eax,msg.wParam
     ret
     WinMain endp

COMMENT *
Оконная функция. Вызывается, когда в структуру msg попадает
следующее сообщение, выбранное из входной очереди. Анализирует
код сообщения и обрабатывает его.
*
     WndProc proc hWnd:DWORD, wMsg:DWORD, wParam:DWORD,
                                          lParam:DWORD
          ; сообщение о необходимости уничтожения окна
          .IF wMsg==WM_DESTROY
               ; помещение в очередь сообщение WM_QUIT
               invoke PostQuitMessage,0
          ; если нажата клавиша
          .ELSEIF wMsg==WM_KEYDOWN
               ; и это код клавиши ESC
               .IF wParam==VK_ESCAPE
                    ; помещение в очередь сообщение WM_QUIT
                    invoke PostQuitMessage, 0
               .ENDIF
          .ELSE
               ; вызов функции обработки сообщений по умолчанию
               invoke DefWindowProcA,hWnd,wMsg,wParam,lParam
               ret
          .ENDIF
          xor eax,eax
          ret
     WndProc endp
end start
```

Тема 6 Элементы управления окна

Главным элементом программы в среде Windows является *окно*. Оно может содержать элементы управления: кнопки, списки, окна редактирования и др. Эти элементы также являются окнами, но обладающими особым свойством: события, происходящие с этими элементами (и самим окном), приводят к приходу сообщений в процедуру окна. Список основных элементов управления окна приведен в таблице 6.1

Таблица 6.1 – Основные элементы управления окна

Системный класс	**Предназначение**
BUTTON	Кнопка
COMBOBOX	Комбинированное окно (окно со списком и поля выбора).
EDIT	Окно редактирования текста
LISTBOX	Окно со списком
SCROLLBAR	Полоса прокрутки
STATIC	Статический элемент (текст)

Создание элементов управления окна осуществляется функцией *CreateWindow* или *CreateWindowEx*. Функция возвращает дескриптор элемента управления окна, который может быть впоследствии использован для анализа элемента управления, с которым связано обрабатываемое событие. Дескриптор кнопки, например, передается в оконную функцию в качестве параметра `lParam`.

Таблицу стилей элементов управления окна можно устанавливать в параметре `dwStyle`, как и для создания родительского окна. Обязательно указывается, что создаваемое окно является дочерним – `WS_CHILD`.

6.1 Кнопка

Кнопка – маленькое прямоугольное дочернее окно, по которому пользователь может щелкать мышью, чтобы включить или выключить его.

Кнопки управления могут использоваться самостоятельно или в группах. Они могут быть подписаны или появляться без текста. Кнопки управления обычно изменяют свой вид, когда пользователь щелкает по ним мышью.

При нажатии кнопки операционная система генерирует сообщение `WM_COMMAND` с параметром `lParam`, который соответствует дескриптору кнопки.

Для обработки нажатия кнопки:

1. В секцию `.data` добавим строки

```
ButtonClassName db "button",0 ; имя класса кнопки
ButtonText db "My First Button",0 ; надпись на кнопке
```

2. В секцию `.data`? добавим строку (дескриптор кнопки)

```
hwndButton HWND ?
```

3. В секцию `.const` добавим строку с идентификатором кнопки

```
ButtonID equ 1
```

строку со стилем кнопки – создание обычной кнопки, которая посылает сообщение WM_COMMAND родительскому окну, когда пользователь нажимает кнопку

```
BS_DEFPUSHBUTTON equ 1h
```

и строку с кодом уведомления:

```
BN_CLICKED equ 0.
```

4. В оконной функции обработаем сообщение WM_CREATE, создав окно кнопки и получив его дескриптор:

```
. . . . . . . . . . . . . . .
.ELSEIF wMsg==WM_CREATE
    invoke CreateWindowExA,0, ADDR ButtonClassName,\
    ADDR ButtonText,\
    WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
    75, 70, 140, 25, hWnd, ButtonID, hInstance,0
    mov hwndButton, eax
. . . . . . . . . . . . . . .
```

Затем обработаем сообщение WM_COMMAND:

```
. . . . . . . . . . . . . . .
.ELSEIF wMsg == WM_COMMAND
```

```
mov eax, wParam
.IF lParam != 0 ; выбран элемент управления
    .IF ax == ButtonID ; дескриптор кнопки
    shr eax,16
    .IF ax==BN_CLICKED ; нажата кнопка
    invoke MessageBoxA,0,ADDR AppName,ADDR ClassName,MB_OK
    .ENDIF
    .ENDIF
.ENDIF
. . . . . . . . . . . . . .
```

6.2 Поле редактирования

Поле редактирования – прямоугольное дочернее окно, внутри которого пользователь может напечатать с клавиатуры текст. Пользователь выбирает элемент управления и дает ему фокус клавиатуры, щелкая по нему мышью или перемещая в него курсор путем нажатия клавиши табуляции. Пользователь может вводить текст, когда окно редактирования текста отображает мигающий курсор.

Для считывания информации из поля редактирования используется функция `GetWindowText`. Параметрами функции являются дескриптор поля редактирования, указатель на текстовую строку и максимальное количество символов. Возвращаемое значение – длина считанной текстовой строки.

Для установки текста в поле редактирования используется функция `SetWindowText`. Параметрами функции являются дескриптор поля редактирования и указатель на текстовую строку. В случае успешного завершения функция возвращает ненулевое значение.

Для вывода текста в поле редактирования при нажатии кнопки:

1. В секцию `.data` добавим строки

```
EditClassName db "edit",0 ; имя класса поля редактирования
EditText db "My First Edit",0 ; выводимый в поле текст
```

2. В секцию `.data`? добавим строку (дескриптор поля)

```
hwndEdit HWND ?
```

3. В секцию `.const` добавим строку с идентификатором поля

```
EditID equ 2
```

и строки со стилем поля – выравнивание текста по левому краю и автоскроллинг текста в поле редактирования:

```
ES_LEFT equ 0h
ES_AUTOHSCROLL equ 80h
```

4. В оконной функции обработаем сообщение WM_CREATE, создав окно поля редактирования и получив его дескриптор:

```
. . . . . . . . . . . . . .
.ELSEIF wMsg==WM_CREATE
     invoke CreateWindowExA, WS_EX_CLIENTEDGE,\
     ADDR EditClassName, 0 , WS_CHILD or WS_VISIBLE or \
     WS_BORDER or ES_LEFT or ES_AUTOHSCROLL,\
     50, 50, 70, 20, hWnd, EditID, hInstance, 0
     mov hwndEdit, eax
. . . . . . . . . . . . . .
```

Затем обработаем сообщение WM_COMMAND:

```
. . . . . . . . . . . . . .
.ELSEIF wMsg == WM_COMMAND
mov eax, wParam
.IF lParam != 0 ; выбран элемент управления
    .IF ax == ButtonID ; дескриптор кнопки
    shr eax,16
    .IF ax==BN_CLICKED ; нажата кнопка
    ; выводим текст в поле редактирования
    invoke SetWindowTextA, hwndEdit, ADDR EditText
    .ENDIF
    .ENDIF
.ENDIF
. . . . . . . . . . . . . .
```

6.3 Статический текст

Статический текст – текстовое поле, окно или прямоугольник, используемый для надписей, не подлежащих редактированию.

Для установки статического текста используется все та же функция SetWindowText.

Для вывода текста в статическое поле при нажатии кнопки:

1. В секцию `.data` добавим строки

```
StaticClassName db "edit",0 ; имя класса статического текста
```

```
StaticText db "My First Static",0 ; выводимый текст
```

2. В секцию `.data`? добавим строку (дескриптор статического текста)

```
hwndStatic HWND ?
```

3. В секцию `.const` добавим строку с идентификатором текста

```
StaticID equ 3
```

и строку со стилем статического текста – выравнивание по центру –

```
SS_CENTER equ 1h.
```

4. В оконной функции обработаем сообщение WM_CREATE, создав окно статического текста и получив его дескриптор:

```
. . . . . . . . . . . . . . .
.ELSEIF wMsg==WM_CREATE
     invoke CreateWindowExA, WS_EX_CLIENTEDGE,\
     ADDR StaticClassName, 0, WS_CHILD or WS_VISIBLE \
     or SS_CENTER, 50, 180, 170, 20, hWnd, StaticID,\
     hInstance, 0
     mov hwndStatic, eax
. . . . . . . . . . . . . . .
```

Затем обработаем сообщение WM_COMMAND:

```
. . . . . . . . . . . . . . .
.ELSEIF wMsg == WM_COMMAND
mov eax, wParam
.IF lParam != 0 ; выбран элемент управления
    .IF ax == ButtonID ; дескриптор кнопки
    shr eax,16
    .IF ax==BN_CLICKED ; нажата кнопка
    ; выводим статический текст
    invoke SetWindowTextA, hwndStatic, ADDR StaticText
    .ENDIF
    .ENDIF
.ENDIF
. . . . . . . . . . . . . . .
```

6.4 Пример использования элементов управления

Напишем программу нахождения суммы двух чисел. Программа должна создать два поля редактирования для ввода чисел, кнопку «Рассчитать», при нажатии на которую выводится статический текст, соответствующий сумме, и кнопку «Очистить», при нажатии на которую

удаляются введенные числа и рассчитанная сумма. Пример работы программы показан на рис. 5.1.

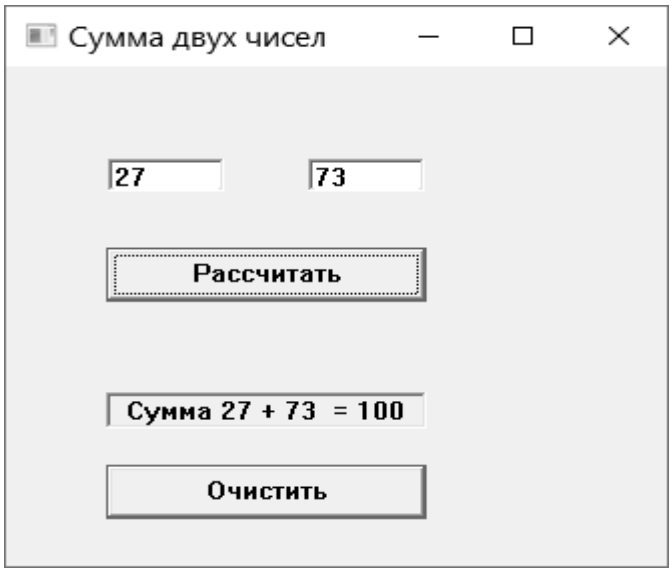

Рис. 5.1 – Пример работы программы сложения двух чисел

В секцию `.data` программы добавим имена системных классов поля редактирования, кнопки и статического текста:

```
EditClassName db "edit",0
ButtonClassName db "button",0
StaticClassName db "static",0
```

В эту же секцию добавим строки-надписи для поля редактирования и кнопок, строку-результат и строку формата для вывода результата:

```
StaticText db "Тут будет сумма",0
Button1Text db "Рассчитать",0
Button2Text db "Очистить",0
buffer db 128 dup(0)
format db 'Сумма %d + %d  = %d', 0
```

В секцию `.data?` добавим дескрипторы двух полей редактирования, двух кнопок и статического текста:

```
hwndEdit1 HWND ?
hwndEdit2 HWND ?
hwndButton1 HWND ?
hwndButton2 HWND ?
hwndStatic HWND ?
```

В секцию `.const` добавим идентификаторы двух полей редактирования, двух кнопок и статического текста:

```
Edit1ID equ 1
Edit2ID equ 2
Button1ID equ 3
Button2ID equ 4
StaticID equ 5
```

В оконной функции обработаем сообщение WM_CREATE и создадим два поля редактирования, две кнопки и статический текст и сохраним их дескрипторы:

```
.........................................................
.ELSEIF uMsg==WM_CREATE

; создание первого поля редактирования
invoke CreateWindowExA,WS_EX_CLIENTEDGE, ADDR EditClassName,0,\
WS_CHILD or WS_VISIBLE or WS_BORDER or ES_LEFT or \
ES_AUTOHSCROLL, 50, 50, 70, 20, hWnd, Edit1ID, hInstance, 0
mov hwndEdit1, eax
invoke SetFocus, hwndEdit1

; создание второго поля редактирования
invoke CreateWindowExA,WS_EX_CLIENTEDGE, ADDR EditClassName,0,\
WS_CHILD or WS_VISIBLE or WS_BORDER or ES_LEFT or \
ES_AUTOHSCROLL, 150, 50, 70, 20, hWnd,Edit2ID, hInstance, 0
mov hwndEdit2, eax

; создание кнопки для вычисления суммы
invoke CreateWindowExA,0, ADDR ButtonClassName,ADDR Button1Text,\
WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
50, 100, 170, 30, hWnd, Button1ID, hInstance, 0
mov hwndButton1, eax

; создание кнопки очистки
invoke CreateWindowExA,0, ADDR ButtonClassName,ADDR Button2Text,\
WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
50, 220, 170, 30, hWnd, Button2ID,hInstance, 0
mov hwndButton2, eax

; создание статического текста
invoke CreateWindowExA,WS_EX_CLIENTEDGE, ADDR StaticClassName,\
ADDR StaticText, WS_CHILD or WS_VISIBLE or SS_CENTER,\
50, 180, 170, 20, hWnd, StaticID, hInstance, 0
mov hwndStatic, eax
```

Обработаем сообщение WM_COMMAND. Оно может исходить как от элементов управления, так и от меню. Чтобы провести различие между ними, используется параметр lParam. Если он равен нулю, то текущее сообщение WM_COMMAND было послано меню, иначе – сообщение послано элементом управления.

Нельзя использовать параметр wParam, чтобы отличить меню и элемент управления, так как ID меню и ID элемента управления могут быть идентичными и код уведомления может быть равен нулю.

В таблице 6.2 показано содержание параметров wParam и lParam.

Таблица 6.1 – Содержание параметров wParam и lParam

Сообщение	Старшее слово wParam	Младшее слово wParam	lParam
От элемента управления	Код уведомления	ID элемента управления	Дескриптор элемента
От меню	0	ID меню	0

```
.........................................................
.ELSEIF uMsg==WM_COMMAND
mov eax,wParam
; если сообщение послано элементом управления, а не меню
.IF lParam != 0
     ; и нажата кнопка «Рассчитать»
     .IF ax == Button1ID
     shr eax,16
          ; и код уведомления – одинарный клик мышкой по кнопке
          .IF ax == BN_CLICKED
          ; получаем в буфер содержимое первого поля
          invoke GetWindowTextA, hwndEdit1, ADDR buffer,10
          ; преобразуем в целое число и сохраняем в стеке
          invoke StrToIntA, ADDR buffer
          push eax
          ; получаем в буфер содержимое второго поля
          invoke GetWindowTextA, hwndEdit2,ADDR buffer,10
          ; преобразуем в целое и сохраняем в регистре ecx
          invoke StrToIntA, ADDR buffer
          mov ecx, eax
          ; восстанавливаем из стека первое слагаемое
          pop ebx
          ; складываем второе и первое слагаемое
          add eax, ebx
          ; формируем строку вывода по заданному формату
          invoke wsprintfA,addr buffer,addr format,ebx,ecx,eax
          ; выводим результат в статическое поле
          invoke SetWindowTextA, hwndStatic, ADDR buffer
          .ENDIF
     .ENDIF
     ; если нажата кнопка «Очистить»
     .IF ax == Button2ID
     shr eax,16
          ; и код уведомления – одинарный клик мышкой по кнопке
          .IF ax == BN_CLICKED
          ; очищаем поля редактирования и статического текста
```

```
            invoke SetWindowTextA, hwndEdit1, 0
            invoke SetWindowTextA,hwndEdit2, 0
            invoke SetWindowTextA,hwndStatic, 0
            ; устанавливаем фокус на первое поле редактирования
            invoke SetFocus, hwndEdit1
            .ENDIF
        .ENDIF
.ENDIF
```

В вышеприведенном коде использована функция перевода строки в число `StrToIntA`, которая преобразует строку в целое число и помещает его в регистр `eax`. Функция находится в библиотеке *shlwapi.lib*. Путь к ней:

C:\Program Files\Windows Kits\10\Lib\10.0.10586.0\um\x86\shlwapi.lib

Скопируйте файл *shlwapi.lib* в свою папку MY_SDK.

В командном файле ASM.BAT замените строку

```
my_sdk\link.exe /DEFAULTLIB:my_sdk\kernel32.lib
/DEFAULTLIB:my_sdk\user32.lib /DEFAULTLIB:my_sdk\gdi32.lib
/SUBSYSTEM:WINDOWS  %1.obj
```

на строку

```
my_sdk\link.exe /DEFAULTLIB:my_sdk\kernel32.lib
/DEFAULTLIB:my_sdk\user32.lib /DEFAULTLIB:my_sdk\gdi32.lib
/DEFAULTLIB:my_sdk\shlwapi.lib /SUBSYSTEM:WINDOWS  %1.obj
```

Добавьте к прототипам функций программы прототип функции

`StrToIntA`:

```
StrToIntA PROTO STDCALL :DWORD
```

Тема 7 Ресурсы приложений

7.1 Понятие ресурса

Составной частью проекта в Windows является *файл определения ресурсов*. У самой Windows есть некоторые предопределенные данные (предопределенные курсоры, иконки и кисти). Точно так же почти в каждой программе для Windows есть некоторые данные, которые определяются еще до начала ее работы, особым образом добавляются в выполняемый файл и используются во время ее функционирования. Яркими примерами таких данных являются *иконки* и *курсоры мыши*.

Кроме них, к числу ресурсов относятся используемые в программе изображения; строки символов; меню; ускорители клавиатуры; диалоговые окна; шрифты; ресурсы, определяемые пользователем.

Итак, ресурсы определяются до начала работы программы и добавляются в исполняемый файл, но при его загрузке в память сами ресурсы в память не загружаются. Только в случае, если тот или иной ресурс требуется для работы, программа сама загружает его в память.

Возможность использования того или иного атрибута в качестве ресурса не означает, что программист не может создавать их в программе. Например, в старой известной программе Program Manager при перетаскивании иконки с место на место курсор меняет свою форму и принимает форму, подобную перетаскиваемой иконке. Естественно, что в этом случае курсоры определяются программой.

7.2 Стандартные и нестандартные ресурсы

Все ресурсы, заранее определенные в Win32 API, называются стандартными. Для работы с ними существуют специальные функции. Эта стандартность ограничивает возможности программиста. Для того чтобы можно было преодолеть эти ограничения, был создан *особый тип ресурсов – определяемые пользователем ресурсы*. Используя именно этот тип, можно предоставить в распоряжение программы практически любые

данные. В таком случае платой за универсальность является усложнение программы, так как забота о манипулировании данными из ресурсов лежит уже не на системе, а на программе, использующей их. Программа может только получить указатель на данные ресурсов, загруженные в память средствами Windows. Дальнейшая работа с ресурсами осуществляется исключительно самой программой.

7.3 Подключение ресурсов к исполняемому файлу

Ресурсы создаются отдельно от файлов программы и добавляются в исполняемый файл при ее линковке. Подавляющее большинство ресурсов содержится в *файлах ресурсов*, имеющих расширение `.RC`. Имя файла ресурсов обычно совпадает с именем исполняемого файла программы. Так, если имя программы MYPROG.EXE, то имя файла ресурсов – MYPROG.RC.

Некоторые типы ресурсов (меню, например) можно описать на специальном языке и воспользоваться при этом обычным текстовым редактором. Другие ресурсы (иконки, курсоры, изображения) тоже описываются в текстовом виде, но частью их описания является последовательность шестнадцатеричных цифр, описывающих изображения. Можно, конечно, попробовать написать эту последовательность и в текстовом редакторе, но, наверное, в этом случае сложность создания ресурса приблизится к сложности написания программы, а возможно, и превысит ее. Обычно для создания ресурсов пользуются специальными средствами – *редакторами ресурсов*. Они позволяют создавать ресурсы, визуально контролировать правильность этого процесса, после чего сохранять их в файлах ресурсов.

Часто используют «смешанный» способ редактирования ресурсов. Например, при визуальном редактировании диалоговых окон достаточно трудно точно установить его элементы именно так, как хочется. Необходимо установить все элементы *приблизительно* на те места, где они

должны находиться, после чего сохранить ресурсы в виде файла с расширением `.RC`. Затем отредактировать RC-файл как обычный текстовый файл, точно указывая при этом все размеры и позиции.

При создании RC-файлов программист может столкнуться с одной тонкостью. Некоторые ресурсы, такие, как иконки, курсоры, диалоговые окна, изображения (bitmap'ы) могут быть сохранены в отдельных файлах с расширениями `.ICO`, `.CUR`, `.DLG`, `.BMP` соответственно. В этом случае в RC-файлах делаются ссылки на упомянутые файлы.

После создания файла ресурсов его нужно откомпилировать специальным компилятором ресурсов RC.EXE.

После компиляции файла ресурсов компилятором ресурсов создается файл, имеющий расширение `.RES`. Этот RES-файл используется линкером для добавления ресурсов в исполняемый файл. При необходимости RES-файлы могут создаваться и редакторами ресурсов. В каком формате создавать ресурсы и как присоединять их к исполняемому файлу, зависит от потребностей программиста.

Шаги для включения ресурсов в исполняемый файл:

1. Создание RC-файла, при необходимости включающего ссылки на файлы с расширением .ICO, .CUR, .BMP, .DLG, .MNU и т.д. Используется редактор ресурсов (может быть использован текстовый и графический).

2. Редактирование RC-файла в текстовом виде. Используется текстовый редактор.

3. Компиляция RC-файла, получение RES-файла. Используется компилятор ресурсов.

4. Добавление ресурсов, содержащихся в RES-файле, в исполнимый файл. Используется линкер.

7.4 Создание собственной иконки приложения

Для использования собственной иконки нужно:

1. Скачать с интернет-сайта понравившийся файл-картинку иконки с расширением `.ICO`, например, ICON.ICO. Если расширение файла `.PNG`, то можно воспользоваться on-line конвертером файлов изображений в файлы `.ICO` на сайте http://image.online-convert.com/ru/convert-to-ico.

2. Используя текстовый редактор, создать файл ресурсов, например, ICON.RC и записать в него строку

```
ICON_MAIN ICON ICON.ICO
```

где `ICON_MAIN` – уникальное имя, которое используется как идентификатор ресурса в исходной программе, а `ICON.ICO` – имя файла, содержащего ресурс. Если файл располагается не в текущей директории, то кроме его имени должен быть указан и полный путь к нему. В этом случае параметр заключается в двойные кавычки.

3. В секцию `.DATA` исходного кода программы добавить строку

```
IconName db 'ICON_MAIN', 0
```

В секции `.CODE` исходного кода программы заменить строку, формирующую стандартную иконку приложения

```
invoke LoadIconA, 0, IDI_APPLICATION
```

на строку, создающую собственную иконку

```
invoke LoadIconA, hInst, OFFSET IconName.
```

4. Создать командный файл ICO.BAT. Он позволяет осуществить компиляцию исходного файла ICON.ASM компилятором ML.EXE (на выходе файл ICON.OBJ), компиляцию файла ресурсов ICON.RC компилятором ресурсов RC.EXE (на выходе файл ICON.RES) и компоновку программы компоновщиком LINK.EXE (параметрами компоновки являются файлы ICON.OBJ и ICON.RES). Код файла ICO.BAT:

```
     @echo off
     my_sdk\ml.exe /c %1.asm
     my_sdk\rc.exe %1.rc
     my_sdk\link.exe  /DEFAULTLIB:my_sdk\kernel32.lib
/DEFAULTLIB:my_sdk\user32.lib /DEFAULTLIB:my_sdk\gdi32.lib
/SUBSYSTEM:WINDOWS  %1.obj %1.res
     if exist %1.obj del %1.obj
```

```
if exist %1.res del %1.res
dir %1.*
pause
```

В командной строке ввести: ICO.BAT ICON (без расширения!) и нажать клавишу Enter. При отсутствии ошибок будет сгенерирован файл ICON.EXE.

7.5 Подключение меню к окну

Меню располагаются сразу под заголовком окна и позволяют пользователю выбрать из предоставляемых программой. Существуют также всплывающие меню, которые могут появляться в любой точке экрана. Обычно их содержание зависит от того, на каком окне щелкнули клавишей мыши.

Главное меню программы – древовидная структура. Корень дерева – это непосредственно главное меню. Оно представляет только структуру в памяти, не отображается на экране, не содержит ни одного элемента, но хранит указатель на список структур, описывающих подключаемые к нему элементы и *всплывающие Popup-Menu*. В свою очередь, *Popup-Menu* должно указать на список структур очередного, более низкого уровня и т.д. Конечные элементы меню никаких указателей на списки не имеют, но хранят идентификатор действия (идентификатор элемента меню), которое должна произвести программа при выборе данного элемента меню. Эта многоуровневая древовидная структура описывается в файле ресурсов. Описание меню имеет вид

```
MenuName MENU [параметры] ; это главное меню
{
  Описание всех Popup-Menu и элементов меню второго уровня
}
```

В данном случае MenuName – это имя создаваемого меню. Слово MENU обозначает начало его определения.

В Win32 API для описания меню существуют два ключевых слова:

1. `POPUP` – специфицирует всплывающее меню;

2. `MENUITEM` – описывает обычный элемент меню.

Всплывающее меню описывается таким образом:

```
POPUP "Имя" [,параметры] ; описание POPUP-меню
{
  Описание всех Popup-Menu и элементов очередного уровня
}
```

У конечного элемента меню в его описании есть еще одна характеристика – идентификатор действия:

```
MENUITEM "Имя", MenuID [,параметры]
```

В обоих случаях «Имя» – это тот текст, который будет выведен на экран при отображении меню (при описании главного меню выводимого на экран текста нет). В том случае, когда вместо имени окна записано слово SEPARATOP (без кавычек), на месте элемента меню появляется горизонтальная линия. Обычно эти горизонтальные линии используются для разделения элементов подменю на логические группы.

Если в имени меню встречается символ «&», то следующий за амперсандом символ на экране будет подчеркнут одинарной чертой. Этот элемент меню можно будет вызывать с клавиатуры посредством одновременного нажатия клавиши Alt и подчеркнутого символа.

MenuID – идентификатор действия. Он может быть передан функции окна, содержащего меню. Значение идентификатора определяется пользователем. Функция окна в зависимости от полученного MenuID производит определенные действия.

Параметры же описывают способ появления элемента на экране. Возможные значения параметров:

1. CHECKED – рядом с именем элемента может отображаться значек, говорящий о том, что соответствующий флаг установлен;
2. ENABLED – элемент меню доступен;
3. DISABLED – элемент меню недоступен, но отображается как обычный;
4. GRAYED – элемент меню недоступен и отображается серым цветом;

5. MENUBREAK – горизонтальные меню размещают следующие элементы в новой строке, а вертикальные – в новом столбце;

6. MENUBARBREAK – то же, что и предыдущее, но в случае вертикального меню столбцы разделяются вертикальной линией.

Создадим описание небольшого меню. Горизонтальное меню (menubar) позволит выбирать подменю «File», «Examples» и конечный элемент «Help». Подменю «File» будет содержать элементы «Open» и «Exit», разделенные горизонтальной линией, а подменю «Examples» – несколько конечных элементов.

Ниже приведен текст скрипта для этого меню:

```
#define IDM_OPEN 101
#define IDM_EXIT 102
#define IDM_EXAMPLE_11 103
#define IDM_EXAMPLE_12 104
#define IDM_EXAMPLE_21 105
#define IDM_EXAMPLE_22 106
#define IDM_HELP 111
MyMenu MENU
{
 POPUP "&File"
 {
       MENUITEM "&Open", IDM_OPEN
       MENUITEM SEPARATOR
       MENUITEM "Exit", IDM_EXIT
 }
 POPUP "&Examples"
 {
       POPUP "Example 1"
       {
             MENUITEM "1&1", IDM_EXAMPLE_11
             MENUITEM "1&2", IDM_EXAMPLE_12
       }
       POPUP "Example 2"
       {
             MENUITEM "2&1", IDM_EXAMPLE_21
             MENUITEM "2&2", IDM_EXAMPLE_22
       }
}
 MENUITEM "&Help", IDM_HELP
}
```

Идентификаторы действия есть только у MENUITEM'ов. Popup-Menu идентификаторов не содержит.

Сохраним описание меню в файл menu.rc.

В основную программу в секцию `.const` добавим строки:

```
IDM_OPEN equ 101
IDM_EXIT equ 102
IDM_EXAMPLE_11 equ 103
IDM_EXAMPLE_12 equ 104
IDM_EXAMPLE_21 equ 105
IDM_EXAMPLE_22 equ 106
IDM_HELP equ 111
```

В секцию `.data` добавим строку

```
MenuName db 'MyMenu',0
```

В поле структуры класса окна `lpszMenuName` занесем указатель на строку с именем меню

```
mov wc.lpszMenuName,OFFSET MenuName
```

В оконной функции обработаем сообщение WM_COMMAND. Если выбран пункт меню `IDM_HELP`, выведем диалоговое окно с информацией о программе:

```
. . . . . . . . . . . . . . .
.ELSEIF wMsg == WM_COMMAND
     mov eax,wParam
     .IF lParam == 0         ; выбран элемент меню
         .IF ax == IDM_HELP  ; выбран пункт HELP
         invoke MessageBoxA,0,ADDR AppName,ADDR ClassName,MB_OK
        .ENDIF
     .ENDIF
. . . . . . . . . . . . . . .
```

Тема 8 Работа с файлами в системе Windows

8.1 Создание, открытие и закрытие файла

Создание и открытие файла производится функцией `CreateFile`:

```
HANDLE WINAPI CreateFile(
_In_ LPCTSTR lpFileName,
_In_ DWORD dwDesiredAccess,
_In_ DWORD dwShareMode,
_In_ opt LPSECURITY_ATTRIBUTES lpSecurityAttributes,
_In_ DWORD dwCreationDisposition,
_In_ DWORD dwFlagsAndAttributes,
_In_ opt HANDLE hTemplateFile);
```

с параметрами:

1) `lpFileName` – указатель на ASCII-строку с именем (путем) открываемого или создаваемого файла;
2) `dwDesiredAccess` – тип доступа к файлу:

   `GENERIC_READ = 80000000b - доступ для чтения;`
   `GENERIC_WRITE = 40000000b - доступ для записи;`
   `GENERIC_READ + GENERIC_WRITE = 0C0000000h - доступ для чтения-записи;`
3) `dwShareMode` – режим разделения файлов между разными процессами. Может принимать значения:

   `0 - монополизация доступа к файлу;`
   `FILE_SHARE_READ = 00000001h – другие процессы могут открыть файл, но только для чтения, запись в файл монополизирована процессом, открывшим файл;`
   `FILE_SHARE_WRITE = 00000002h – другие процессы могут открыть файл, но только для записи, чтение в файл монополизировано процессом, открывшим файл;`
   `FILE_SHARE_READ + FILE_SHARE_WRITE = 00000003h – другие процессы могут открывать файл для чтения-записи;`
4) `lpSecurityAttributes` – указатель на структуру `SecurityAttributes`, определяющую защиту связанного с файлом объекта ядра; при отсутствии защиты заносится NULL;
5) `dwCreationDisposition` – действия для случаев, когда файл существует или не существует, параметр может принимать значения:

   `CREATE_NEW = 1 – создать новый файл, если файл не существует; если файл существует, то функция завершается формированием ошибки;`

CREATE_ALWAYS = 2 — создать новый файл, если файл не существует; если он существует, то заместить новым;

OPEN_EXISTING = 3 — открыть файл, если он существует; если файл не существует, то формируется ошибка;

OPEN_ALWAYS = 4 — открыть файл при его существовании и создать его, если файла нет;

TRUNCATE_EXISTING = 5 — открыть файл с усечением его до нулевой длины; если файл не существует, то формируется ошибка;

6) dwFlagsAndAttributes — флаги и атрибуты; этот параметр используется для задания характеристик создаваемого файла:

FILE_ATTRIBUTE_READ0NLY = 00000001h – файл только для чтения;

FILE_ATTRIBUTE_HIDDEN = 00000002h – скрытый файл;

FILE_ATTRIBUTE_SYSTEM = 00000004h – системный файл;

FILE_ATTRIBUTE_DIRECTORY = 00000010h – каталог;

FILE_ATTRIBUTE_ARCHIVE = 00000020h – архивный файл;

FILE_ATTRIBUTE_N0RMAL = 00000080h – обычный файл для чтения-записи (этот атрибут нельзя комбинировать с другими);

FILE_ATTRIBUTE_TEMPORARY = 00000100h – временный файл;

FILE_FLAG_WRITE_THR0UGH = 80000000h – не использовать промежуточное кэширование при записи на диск, а все изменения записывать прямо на диск;

FILE_FLAG_NO_BUFFERING = 20000000h – не использовать средства буферизации операционной системы;

FILE_FLAG_RANDOM_ACCESS = 10000000h – случайный доступ к файлу, может использоваться системой для оптимизации кэширования файла;

FILE_FLAG_SEQUENTIAL_SCAN = 08000000h – последовательный доступ к файлу;

FILE_FLAG_DELETE_ON_CLOSE=04000000h – удалить файл после его закрытия;

FILE_FLAG_OVERLAPPED = 40000000h – асинхронный доступ к файлу (синхронность означает то, что программа, вызвавшая функцию для доступа к файлу, приостанавливается до тех пор, пока не закончит работу функция ввода-вывода);

7) hTemplateFile — при создании нового файла значением данного параметра является дескриптор другого существующего и предварительно открытого файла, а новый файл создается с теми же значениями атрибутов и флагов, что и у файла, дескриптор которого указан в параметре.

При удачном завершении функция CreateFile возвращает в регистре EAX дескриптор нового файла. В случае неудачи функция возвращает в регистре EAX значение NULL.

Закрытие файла производится функцией CloseHandle:

```
BOOL WINAPI CloseHandle(
_In_ HANDLE hObject);
```

Параметр `hObject` – дескриптор, полученный при открытии файла функцией `CreateFile`.

При удачном завершении функция `CloseHandle` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

8.2 Удаление файла

Удаление файла производится функцией `DeleteFile`:

```
BOOL WINAPI DeleteFile(
_In_ LPCTSTR lpFileName);
```

Параметр `lpFileName` – указатель на ASCIIZ-строку с именем (путем) удаляемого файла. Перед удалением файл необходимо закрыть, хотя в некоторых версиях Windows это не является обязательным.

При удачном завершении функция `DeleteFile` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

8.3 Установка текущей файловой позиции

Доступ к содержимому файла может быть произвольным (прямым) и последовательным. Обычно, функции ввода-вывода работают с *файловым указателем*. Но необходимо иметь в виду, что файловый указатель связан только с описателем файла. Его значение равно текущему номеру позиции в файле, с которой будет производиться чтение-запись данных при очередном вызове функции ввода-вывода. В первый момент после открытия значение указателя равно нулю, т. е. он указывает на начало файла. Функции, производящие чтение-запись, меняют значение файлового указателя на количество прочитанных или записанных байт.

При необходимости, а при организации прямого доступа к файлу без этого не обойтись, значение файлового указателя можно изменять с помощью функции `SetFilePointer`:

```
DWORD WINAPI SetFilePointer(
_In_ HANDLE hFile,
_In_ LONG lDistanceToMove,
_In_opt_ PLONG lpDistanceToMoveHigh,
_In_ DWORD dwMoveMethod);
```

ее параметры:

1. `hFile` – дескриптор файла, в котором производится позиционирование указателя позиции (получен функцией `CreateFile`);

2. `lDistanceToMove` – расстояние в байтах, на которое необходимо переместить указатель; это число может трактоваться как знаковое и беззнаковое: его отрицательное значение соответствует случаю, когда указатель требуется перемещать к началу файла;

3. `lpDistanceToMoveHigh` – используется при необходимости создания 64-битового знакового значения указателя позиции как значение старших 32 битов указателя;

4. `dwMoveMethod` – определяет трактовку параметров `lDistanceToMove` и `lpDistanceToMoveHigh`:

```
FILE_BEGIN = 0 — указатель позиции - это значение без знака, заданное содержимым полей 2 и 3;
FILE_CURRENT = 1 — текущее значение указателя позиции складывается со знаковым значением, заданным в полях 2 и 3;
FILE_END = 2 — значение, определяемое содержимым полей 2 и 3, должно представлять собой отрицательное значение, и смещение в файле вычисляется как сумма, в которой первое слагаемое определяется полями 2 и 3, а второе является размером файла.
```

8.4 Получение размера файла

Получение размера файла осуществляет функция `GetFileSize` :

```
DWORD WINAPI GetFileSize(
_In_ HANDLE hFile,
_Out_opt_ LPDWORD lpFileSizeHigh);
```

Параметр `hFile` – дескриптор файла, размер которого будет получен.

Параметр `lpFileSizeHigh` – указатель на старшее двойное слово, если размер памяти занимает больше 32 бит; если не требуется, то данный параметр равен нулю.

Функция возвращает младшее двойное слово размера файла в регистре EAX. В случае ошибки функция возвращает константу `INVALID_FILE_SIZE = 0xFFFFFFFF`.

8.5 Чтение данных из файла

Чтение данных из файла осуществляется функцией `ReadFile`:

```
BOOL WINAPI ReadFile(
_In_ HANDLE hFile,
_Out_ LPVOID lpBuffer,
_In_ DWORD nNumberOfBytesToRead,
_Out_opt_ LPDWORD lpNumberOfBytesRead,
_InOut_opt_ LPOVERLAPPED lpOverlapped);
```

включающей параметры:

1) `hFile` – дескриптор файла или устройства ввода, с которым производится операция чтения;
2) `lpBuffer` – указатель на буфер, в который заносятся данные при чтении из файла или устройства;
3) `nNumberOfBytesToWrite` – максимальное количество байт для чтения;
4) `lpNumberOfBytesWritten` – указатель на переменную, которая получает число прочитанных байт при использовании синхронного ввода. Функция `ReadFile` устанавливает это значение в ноль, прежде чем что-либо делать или проверять ошибки. Если это асинхронная операция параметр необходимо установить в NULL;
5) `lpOverlapped` – указатель на структуру, используемую в процессе асинхронного ввода-вывода (для синхронного режима – `NULL`).

При удачном завершении функция `ReadFile` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

8.6 Запись данных в файл

Запись в файл осуществляется функцией `WriteFile`:

```
BOOL WINAPI WriteFile(
```

```
_In_ HANDLE hFile,
_In_ LPCVOID lpBuffer,
_In_ DWORD nNumberOfBytesToWrite,
_Out_opt_ LPDWORD lpNumberOfBytesWritten,
_InOut_opt_ LPOVERLAPPED lpOverlapped);
```

содержащей параметры:

1) `hFile` – дескриптор файла или устройства вывода, с которым производится операция записи;

2) `lpBuffer` – указатель на буфер, содержащий данные, которые будут записаны в файл или на устройство вывода;

3) `nNumberOfBytesToWrite` – число байт, которые будут записаны в файл или на устройство вывода;

4) `lpNumberOfBytesWritten` – указатель на переменную, которая получает число записанных байт при использовании синхронной передачи. Функция `WriteFile` устанавливает это значение в ноль, прежде чем что-либо делать или проверять ошибки. Если это асинхронная операция параметр необходимо установить в NULL;

5) `lpOverlapped` – указатель на структуру, используемую в процессе асинхронного ввода-вывода (для синхронного режима `NULL`).

При удачном завершении функция `WriteFile` возвращает ненулевое значение в регистре `EAX`. В случае неудачи функция возвращает в регистре `EAX` значение `NULL`.

8.7 Пример работы с файлами

Рассмотрим пример программы, которая считывает строки из файла и дублирует их в тот же файл после последней строки, дополнительно выводя эти же строки на консоль.

```
.686p
.model flat, stdcall
option casemap:none

; прототипы используемых функций
GetStdHandle PROTO STDCALL :DWORD
WriteConsoleA PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD,:DWORD
CreateFileA PROTO STDCALL \
:DWORD,:DWORD,:DWORD,:DWORD,:DWORD,:DWORD,:DWORD
```

```
ReadFile PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD,:DWORD
WriteFile PROTO STDCALL :DWORD,:DWORD,:DWORD,:DWORD,:DWORD
CloseHandle PROTO STDCALL :DWORD
CharToOemA PROTO STDCALL :DWORD,:DWORD
ExitProcess PROTO STDCALL :DWORD

; описания констант Windows
.const
STD_INPUT_HANDLE equ  -10
STD_OUTPUT_HANDLE equ -11
STD_ERROR_HANDLE equ -12
FILE_SHARE_READ  equ 1h
FILE_SHARE_WRITE equ 2h
FILE_ATTRIBUTE_READONLY equ 1h
FILE_ATTRIBUTE_HIDDEN equ 2h
FILE_ATTRIBUTE_SYSTEM equ 4h
FILE_ATTRIBUTE_DIRECTORY equ 10h
FILE_ATTRIBUTE_ARCHIVE equ 20h
FILE_ATTRIBUTE_NORMAL equ 80h
FILE_ATTRIBUTE_TEMPORARY equ 100h
FILE_ATTRIBUTE_COMPRESSED equ 80
FILE_BEGIN equ 0
FILE_CURRENT equ 1
FILE_END equ 2
CREATE_NEW equ 1
CREATE_ALWAYS equ 2
OPEN_EXISTING equ 3
OPEN_ALWAYS equ 4
TRUNCATE_EXISTING equ 5
GENERIC_READ equ 80000000h
GENERIC_WRITE equ 40000000h
GENERIC_EXECUTE equ 20000000h
GENERIC_ALL equ 10000000h

.data
file db "file.txt",0      ; имя файла
hFile dd 0                ; дескриптор файла
TitleText db 'Работа с файлами',0  ; заголовок окна
dOut dd 0                 ; дескриптор вывода консоли
NumWri dd 0               ; действительное количество символов
buf db 1024 dup(?)        ; буфер ввода-вывода
bufoem db 1024 dup(?)     ; буфер ввода-вывода для консоли
n db 13,10                ; перевод строки

.code
start:

; получаем dOut - дескриптор вывода консоли
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov dOut, eax

; открываем файл: доступ на чтение-запись; монопольный доступ
; защита файла не требуется; если файла нет, то он создается
; получаем hFile - дескриптор файла
INVOKE CreateFileA, offset file, GENERIC_READ or \
GENERIC_WRITE, 0, 0, OPEN_ALWAYS, 0, 0
```

```
mov hFile, eax ;дескриптор файла

; читаем строки из файла в буфер
INVOKE ReadFile, hFile, offset buf, 1024, offset NumWri, 0
; преобразуем в формат oem для вывода в консоль
INVOKE CharToOemA, offset buf, offset bufoem

; Выводим в консоль
INVOKE WriteConsoleA, dOut, offset bufoem, NumWri, \
offset NumWri, 0

; сохраняем в стек количество считаных байт
push NumWri

; переводим строку в файле (добавляем пустую строку)
INVOKE WriteFile, hFile, offset n, 2, offset NumWri, 0

; восстанавливаем из стека количество считанных байт
pop NumWri

; дублируем в файл строки
INVOKE WriteFile, hFile, offset buf, NumWri, offset NumWri, 0

; закрываем файл
INVOKE CloseHandle, hFile
INVOKE ExitProcess, 0
end start
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Аблязов Р. З. Программирование на ассемблере на платформе x86-64 / Р. З. Аблязов. – М.: ДМК Пресс, 2011. – 304 с.

2. Галисеев Г. В. Ассемблер для Win32. Самоучитель / Г. В. Галисеев. – М.: Вильямс, 2007. – 368 с.

3. Калашников О. А. Ассемблер – это просто / О. А. Калашников. – СПб.: БХВ-Петербург, 2011. – 336 с.

4. Пирогов В. Ю. Ассемблер для Windows. / В. Ю. Пирогов. – 4-е изд. – СПб.: БХВ-Петербург, 2007. – 896 с.

5. Юров В. И. Ассемблер для вузов. / В. И. Юров. – 2-е изд. – СПб.: Питер, 2010. – 637 с.

СПИСОК ЭЛЕКТРОННЫХ РЕСУРСОВ

1. Сравнение ассемблерных трансляторов [Электрон. ресурс] – Режим доступа: http://likameta.narod.ru/passembler/asm1.html. – Загл. с экрана.

2. Assembler для Windows [Электрон. ресурс] – Режим доступа: http://i-assembler.ru/index.html#14. – Загл. с экрана.

3. Программирование на языке ассемблера [Электрон. ресурс] – Режим доступа: http://natalia.appmat.ru/c&c++/assembler.html. – Загл. с экрана.

4. Функции Win API для работы с окнами [Электрон. ресурс] – Режим доступа: http://www.realcoding.net/article/view/404. – Загл. с экрана.

5. Структура оконного приложения [Электрон. ресурс] – Режим доступа: http://prog-cpp.ru/winmain/. – Загл. с экрана.

6. Элементы управления окна [Электрон. ресурс] – Режим доступа: http://prog-cpp.ru/winelements/ – Загл. с экрана.

7. The MASM32 SDK [Электрон. ресурс] – Режим доступа: http://www.masm32.com/. – Загл. с экрана.

Учебное издание

Котенко Владислав Николаевич,

Котенко Юлия Владиславовна

Программирование на языках низкого уровня: курс лекций

Редактор М. В. Совпель

План издания 2016 г., поз. № 116.