Министерство образования и науки Российской Федерации
Тольяттинский государственный университет
Институт энергетики и электротехники
Кафедра «Промышленная электроника»

Е.С. Глибин, А.В. Прядилов

ПРОГРАММИРОВАНИЕ ЭЛЕКТРОННЫХ УСТРОЙСТВ

Электронное учебное пособие

ISBN 978-5-8259-0834-2

УДК 004.42(075.8)
ББК 32.973.26-018я73

Рецензенты:
канд. техн. наук, руководитель отдела закупок оборудования и услуг ООО «Джейко Раша» *Д.А. Яковлев*;
канд. техн. наук, доцент Тольяттинского государственного университета *А.А. Шевцов*.

Глибин, Е.С. Программирование электронных устройств : электронное учеб. пособие / Е.С. Глибин, А.В. Прядилов. – Тольятти : Изд-во ТГУ, 2014. : 1 оптический диск

В учебном пособии рассматривается программирование электронных схем на базе микропроцессоров: устройств вывода (дисплеи и принтеры), ввода информации (клавиатуры, мыши), сетевых коммуникаций, многоядерных и многопроцессорных систем, USB-интерфейса. Приведен как теоретический материал, так и примеры программ с подробным разбором их работы.

Предназначено для студентов, обучающихся по направлению подготовки бакалавров 210100.62 «Электроника и наноэлектроника», при изучении дисциплины «Программирование электронных устройств».

Текстовое электронное издание

Рекомендовано к изданию научно-методическим советом Тольяттинского государственного университета.

Минимальные системные требования: IBM PC-совместимый компьютер: Windows XP/Vista/7/8; 500 МГц или эквивалент; 128 Мб ОЗУ; SVGA; Adobe Reader.

Редактор *Е.Ю. Жданова*
Технический редактор *З.М. Малявина*
Корректор *Т.Д. Савенкова*
Компьютерная верстка: *Л.В. Сызганцева*
Художественное оформление, компьютерное проектирование: *Г.В. Карасева*

Дата подписания к использованию 13.11.2014.
Объем издания 1,3 Мб.
Комплектация издания: компакт-диск, первичная упаковка.
Заказ № 1-60-13.

Издательство Тольяттинского государственного университета
445667, г. Тольятти, ул. Белорусская, 14
тел.: 8(8482) 53-91-47, www.tltsu.ru

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данного пособия – помочь в изучении дисциплины «**Программирование электронных устройств**» студентам направления подготовки бакалавров 210100.62 «Электроника и наноэлектроника». В пособии приведены начальные сведения о программировании периферийных устройств персонального компьютера, сетевых интерфейсов, многопроцессорных систем, USB-устройств на уровне взаимодействия программы с операционной системой.

Учебное пособие состоит:

1) из введения;
2) рекомендаций по изучению дисциплины;
3) основного теоретического материала;
4) заключения.

В ***рекомендациях по изучению дисциплины*** указаны цели и задачи дисциплины, даны рекомендации по изучению тем, библиографический список, а также контрольные вопросы.

В основном разделе рассматриваются теоретические вопросы.

В каждой теме приведен пример практической реализации программы в операционной среде *Windows*.

Для успешного изучения материала необходимо владеть языком программирования *Cu*.

РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ДИСЦИПЛИНЫ

Дисциплина «Программирование электронных устройств» посвящена практическому изучению программирования электронных схем на базе микропроцессоров. В курсе изучается программирование устройств вывода информации (дисплеи и принтеры), ввода информации (клавиатуры, мыши), сетевых коммуникаций на уровне операционной системы.

Целями изучения дисциплины «Программирование электронных устройств» являются формирование знаний о современном состоянии микропроцессорной техники и получение практических навыков разработки программ, обеспечивающих функционирование электронных схем на базе микропроцессоров.

При изучении дисциплины решаются следующие **задачи:**

1) формирование практических навыков разработки алгоритмов, написания и отладки программ;
2) ознакомление с технологиями проектирования программных средств, построения пользовательского интерфейса, отображения графической информации и параллельного программирования;
3) изучение плат ввода-вывода данных.

Учебный курс «Программирование электронных устройств» базируется на знаниях и навыках, приобретенных при изучении дисциплины «Информационные технологии». Ключевым требованием при изучении курса является владение основами программирования на языке *Си*, что подразумевает хорошее понимание следующего необходимого минимума:

1) лексических основ языка;
2) различных переменных базовых типов;

3) операторов условий и циклов;
4) переменных составных типов – одномерных и многомерных массивов, структур;
5) указателей в языке *Си*;
6) определений, описаний и вызовов функций.

Эти вопросы в данном пособии не рассматриваются.

Каждая глава основного раздела состоит из двух логически связанных частей: в начале главы приводится теоретический материал, затем рассматривается практическое написание программы. Теоретический материал включает описание необходимых понятий и терминов, механизмов и подходов, которые используются при написании программ (например, краткий принцип работы сети Интернет с точки зрения программирования с использованием понятий IP-адрес, порт, сетевой протокол в главе, посвященной сетевому программированию). Практический материал включает рабочий пример и подробное описание его работы на уровне отдельных команд, их параметров в рамках языка *Си*. В конце главы приводится список контрольных вопросов для проверки понимания изложенного материала.

Рекомендуемая литература

В качестве дополнительного учебного материала по языку программирования *Си* могут быть использованы следующие источники:

1. Баженова, И.Ю. Введение в программирование : учеб.пособие / И.Ю. Баженова, В.А. Сухомлин. – М. : Интернет-ун-т информ. технологий : БИНОМ. Лаб. знаний, 2007. – 326 с.
2. Макконнелл, С. Совершенный код = CODE COMPLETE : Мастер-класс / С. Макконнелл. – М. : Рус. ред. ; СПб. : Питер, 2008, 2008. – 867 с.
3. Павловская, Т.А. C/C++. Программирование на языке высокого уровня : учеб. для вузов / Т.А. Павловская. – СПб. : Питер, 2007. – 460 с.
4. Павловская, Т.А. C/C++. Программирование на языке высокого уровня : учеб. для вузов / Т.А. Павловская. – СПб. : Питер, 2006. – 460 с.

При изучении тем настоящего учебного пособия в качестве дополнительной литературы могут быть использованы:

1. Мартынов, Н.Н. Программирование для *Windows* на C/C++ : фундаментальный учебник-самоучитель: в 2 т. / Н.Н. Мартынов. – М. : Бином, 2006. – Т. 2. – 480 с.
2. Сван, Т. Программирование для *Windows* в Borland C++ / Т. Сван ; пер. с англ. В. Тимофеева. – М. : Бином, 1995. – 479 с.
3. Фаронов, В.В. Практика *Windows*-программирование / В.В. Фаронов. – М. : Информпечать, 1996. – 247 с.
4. Хонекамп, Д. Введение в профессиональное программирование под *Windows* : пер. с нем. / Д. Хонекамп, П. Вилькен. – М. : Эком, 1996. – 654 с.
5. Хьюз, К. Параллельное и распределенное программирование с использованием C++ = Parallel and Distributed Programming Using C++ / К. Хьюз, Т. Хьюз ; пер. с англ. и ред. Н.М. Ручко. – М. : СПб. : Киев : Вильямс, 2004. – 667 с.
6. Черносвитов, А. Visual C++7 : учебный курс / А. Черносвитов. – СПб. : Питер, 2002. – 528 с.

Контрольные вопросы

1. Понятие событийно-управляемой программы.
2. Минимальное приложение для ОС Windows.
3. Программирование клавиатуры.
4. Программирование манипулятора типа «мышь».
5. Устройства ввода пользовательской информации.
6. Методы графических построений в электронных устройствах.
7. Вывод информации с помощью принтера.
8. Организация пользовательского интерфейса.
9. Определение пользовательского интерфейса, классификация интерфейсов.
10. Виды графического интерфейса пользователя. Текстовый интерфейс. Организация интерфейса в ОС Windows.
11. Технологии скоростного обмена данными по сети. Понятие сетевой модели.
12. Сетевая модель DOD. Сетевой протокол, транспортный протокол, IP-адрес, DNS-сервер.

13. Архитектура «клиент – сервер». Написание серверных приложений на *Cи*.
14. Windows Sockets. Функции Беркли. Написание клиентских приложений на *Cи*.
15. Программирование сетевых коммуникаций.
16. Параллельные системы.
17. Параллельные программы на основе передачи сообщений.
18. Многопоточное программирование.
19. Распределение работы между параллельными потоками в OpenMP.
20. Распределение работы между параллельными потоками в POSIX Threads.
21. USB-интерфейс.

1. УПРАВЛЯЕМЫЕ СОБЫТИЯМИ ПРОГРАММЫ

Основной чертой всех Windows-приложений является то, что они поддерживают оконный интерфейс, используя при этом множество стандартных элементов управления (кнопки, переключатели, линейки, окна редактирования, списки и т. д.). Эти элементы поддерживаются с помощью динамических библиотек (DLL), которые являются частью операционной системы (ОС). Именно поэтому элементы доступны любым приложениям, и ваше первое приложение имеет почти такой же облик, как и любое другое. Принципиально важным отличием Windows-приложений от приложений DOS является то, что все они – программы, управляемые событиями (event-driven applications). Приложения DOS – программы с фиксированной последовательностью выполнения. Разработчик программы задает последовательность выполнения операторов, и система строго ее соблюдает. В случае программ, управляемых событиями, разработчик не может заранее предсказать последовательность вызовов функций и даже выполнения операторов своего приложения, так как эта последовательность определяется на этапе выполнения кода.

Программы, управляемые событиями, обладают большей гибкостью в смысле выбора пользователем порядка выполнения операций. Характерно то, что последовательность действий часто определяется операционной системой и зависит от потока сообщений о событиях в системе. Большую часть времени приложение, управляемое событиями, находится в состоянии ожидания событий, точнее сообщений о них. Сообщения могут поступать от различных источников, но все они попадают в одну очередь системных сообщений. Только некоторые из них система передаст в очередь сообщений вашего приложения.

В случае многопотокового приложения сообщение приходит активному потоку (thread) приложения. Приложение постоянно выполняет цикл ожидания сообщений. Как только придет адресованное ему сообщение, управление будет передано его оконной процедуре.

Наступление события обозначается поступлением сообщения. Все сообщения *Windows* имеют стандартные имена, многие из которых начинаются с префикса WM_ (*Windows Message*). Например, WM_PAINT именует сообщение о том, что необходимо перерисовать содержимое окна того приложения, которое получило это сообщение. Идентификатор сообщения WM_PAINT – это символьная константа, обозначающая некое число. Другой пример: при создании окна система посылает сообщение WM_CREATE. Вы можете ввести в оконную процедуру реакцию на это сообщение для того, чтобы произвести какие-то однократные действия.

Программист может создать и определить какие-то свои собственные сообщения, действующие в пределах зарегистрированного оконного класса. В этом случае каждое новое сообщение должно иметь идентификатор, превышающий зарезервированное системой значение WM_USER (0×400). Допустим, вы хотите создать сообщение о том, что пользователь нажал определенную клавишу в тот момент, когда клавиатурный фокус находится в особом окне редактирования с уже зарегистрированным классом. В этом случае новое сообщение можно идентифицировать так:#define WM_MYEDIT_PRESSED WM_USER + 1.

Каждое новое сообщение должно увеличивать значение идентификатора по сравнению с WM_MYEDIT_PRESSED. Максимально допустимым значением для идентификаторов такого типа является число 0×7FFF. Если вы хотите создать сообщение, действующее в пределах всего приложения и не конфликтующее с системными сообщениями, то вместо константы WM_USER следует использовать другую константу WM_APP (0×8000). В этом случае можно наращивать идентификатор вплоть до 0×BFFF.

Рассмотренная модель выработки и прохождения сообщений поможет понять структуру, принятую для всех *Windows*-приложений. Простейшее из них должно состоять как минимум из двух функций:

1) функции *WinMain*, с которой начинается выполнение программы и которая «закручивает» цикл ожидания сообщений;

2) оконной процедуры, которую вызывает система, направляя ей соответствующие сообщения.

Каждое приложение в системе, основанной на сообщениях, должно уметь получать и обрабатывать сообщения из своей очереди. Основу такого приложения в системе Windows составляет функция WinMain, которая содержит стандартную последовательность действий. Однако обрабатывается большинство сообщений окном – объектом операционной системы Windows.

С точки зрения пользователя, окно – это прямоугольная область экрана, соответствующая какому-то приложению или его части. Приложение может управлять несколькими окнами, среди которых обычно выделяют одно главное – окно-рамку. В операционной системе окно в большинстве случаев рассматривается как конечный пункт, куда направляются сообщения. С точки зрения программиста, окно – это объект, атрибуты которого (тип, размер, положение на экране, вид курсора, меню, значок, заголовок) должны быть сначала сформированы, а затем зарегистрированы системой. Манипуляция окном осуществляется посредством специальной оконной функции, которая имеет вполне определенную, устоявшуюся структуру.

Функция WinMain выполняется первой в любом приложении. Ее имя зарезервировано операционной системой. Она в этом смысле является аналогом функции main, с которой начинается выполнение С-программы для DOS-платформы. Имя оконной процедуры произвольно и выбирается разработчиком. Система Windows регистрирует это имя, связывая его с приложением. Главной целью функции WinMain является регистрация оконного класса, создание окна и запуск цикла ожидания сообщений.

Рассмотрим более подробно структуру традиционного Windows-приложения.

```
// Стандартный включаемый файл Windows
#include <windows.h>

// Прототип функции обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// Функция вызывается автоматически, когда программа
запускается
```

```
    int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, PSTR szCmdLine, int iCmdShow)
    {
       HWND         hWnd;
       MSG          msg;
       WNDCLASSEX   wndclass;

       // Настройка класса окна
       wndclass.cbSize        = sizeof(WNDCLASSEX);
       wndclass.style         = CS_HREDRAW | CS_VREDRAW;
       wndclass.lpfnWndProc   = WndProc;
       wndclass.cbClsExtra    = 0;
       wndclass.cbWndExtra    = 0;
       wndclass.hInstance     = hInstance;
       wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
       wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
       wndclass.hbrBackground =
(HBRUSH)GetStockObject(DKGRAY_BRUSH);
       wndclass.lpszMenuName  = NULL;
       wndclass.lpszClassName = "Window Class"; // Имя класса
       wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

       // Регистрация класса окна
       if(RegisterClassEx(&wndclass) == 0)
       {
           // Сбой программы, выход
           return 0;
       }

       // Создание окна
       hWnd = CreateWindowEx(
          WS_EX_OVERLAPPEDWINDOW,
          «Window Class»,             // Имя класса
          «Приложение Windows»,     // Текст заголовка
          WS_OVERLAPPEDWINDOW,
          0,
          0,
          320,
          200,
          NULL,
          NULL,
          hInstance,
          NULL);

       // Отображение окна

       ShowWindow(hWnd, iCmdShow);

       // Обработка сообщений, пока программа не будет прервана
       while(GetMessage(&msg, NULL, 0, 0))
       {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
       }
```

```
        return (int)msg.wParam;
    }

    // Функция обратного вызова для обработки сообщений
    LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg, WPARAM
wParam, LPARAM lParam)
    {
        HDC         hDC;
        PAINTSTRUCT ps;

        switch(iMsg)
        {
            // Вызывается, когда пользователь отпускает левую
кнопку мыши
            case WM_LBUTTONUP:
            MessageBox(hWnd, TEXT(«Вы кликнули!»), TEXT(«событие»),
MB_OK);
                return(0);
            // Вызывается, когда окно обновляется
            case WM_PAINT:
                hDC = BeginPaint(hWnd, &ps);
                Ellipse(hDC, 50, 20, 200, 100);
                EndPaint(hWnd, &ps);
                return(0);
            // Вызывается, когда пользователь закрывает окно
            case WM_DESTROY:
                PostQuitMessage(0);
                return(0);
            default:
                return DefWindowProc(hWnd, iMsg, wParam, lParam);
        }
    }
```

В двух первых строках кода указываются включаемые заголовочные файлы и приводятся прототипы функций. В программировании для Windows необходим только один заголовочный файл с именем windows.h.

Для нашего примера требуется единственный прототип функции для обработчика сообщений. Любая программа для Windows должна содержать функцию, которая будет вызываться для обработки сообщений и должна соответствовать следующему прототипу:

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

При вызове функции WinMain система передает ей параметры:

hInstance – описатель экземпляра приложения. Это адрес прило-

жения, загруженного в память. В Windows NT/2000 этот адрес для всех приложений имеет одно и то же значение 0×00400000 (4 Мб);

hPrevInstance – описатель предыдущего экземпляра приложения. Этот параметр устарел и теперь не используется в приложениях Win32;

szCmdLine – указатель на командную строку. Мы не будем использовать этот параметр;

iCmdShow – состояние окна при начальной демонстрации.

Ранее в Win16 второй параметр использовался в целях экономии ресурсов, но в Win32 – это NULL, так как каждый экземпляр приложения теперь выполняется в своем собственном виртуальном адресном пространстве процесса емкостью 4 Гб. Все экземпляры процесса загружаются, начиная с одного и того же адреса в этом пространстве.

Имя функции может отличаться, но параметры должны оставаться неизменными. Это вызвано тем, что Windows автоматически вызывает данную функцию и не сможет работать правильно, если вы измените параметры.

Первый элемент, упоминаемый в функции WinMain(), – это объект, используемый для создания окна. Объект является структурой типа WNDCLASSEX и очень важен для создания окна программы.

Ниже приведен прототип структуры WNDCLASSEX:

```
typedef struct _WNDCLASSEX {
  UINT    cbSize;
  UINT    style;
  WNDPROC lpfnWndProc;
  int     cbClsExtra;
  int     cbWndExtra;
  HANDLE  hInstance;
  HICON   hIcon;
  HCURSOR hCursor;
  HBRUSH  hbrBackground;
  LPCTSTR lpszMenuName;
  LPCTSTR lpszClassName;
  HICON   hIconSm;
} WNDCLASSEX;
```

Первая переменная типа UINT называется cbSize. Она используется для указания размера структуры данных. Обычно для инициализации этого элемента структуры используется выражение sizeof(WNDCLASSEX). Инициализацию этого элемента данных можно увидеть в приведенном выше листинге.

Второй элемент структуры также имеет тип UINT и называется style. Как указывает имя, элемент style используется для задания стиля создаваемого окна. Удобство данного элемента данных в том, что можно указывать комбинации одних стилей с другими, используя несколько флагов, объединенных поразрядной операцией ИЛИ (|). Некоторые стили приведены в табл. 1.

Таблица 1

Некоторые стили окон

Значение	Действие
CS_DBLCLKS	Простой стиль. Когда он указан, Windows будет посылать окну сообщение о двойном щелчке каждый раз, когда пользователь выполняет двойной щелчок кнопкой мыши в пределах области окна. Это может показаться странным, но многие приложения самостоятельно запоминают время каждого щелчка кнопки мыши, чтобы определить, был ли выполнен двойной щелчок
CS_HREDRAW	Этот стиль заставляет перерисовывать все окно в случае изменения его ширины
CS_NOCLOSE	Запрещает выполнение закрытия окна через системное меню
CS_VREDRAW	Заставляет перерисовывать все содержимое окна в случае изменения высоты окна

Третий элемент структуры имеет тип WNDPROC, является указателем на функцию и называется lpfnWndProc. Он должен указывать на функцию обработки сообщений Windows, которую окно использует, чтобы принимать сообщения. Это очень важно, и функция, на которую ссылаются здесь, должна полностью соответствовать прототипу, приведенному в коде.

Четвертый элемент структуры относится к типу int и называется cbClsExtra. Это целое число задает количество байт, которые будут вы-

делены сразу за структурой данных класса окна. Практически всегда это значение устанавливается равным нулю.

Пятый элемент также относится к типу int и называется cbWndExtra. Это целое число задает количество байтов, которые будут выделены сразу за экземпляром окна. Работа с ним аналогична обращению с предыдущим элементом структуры.

Шестой элемент структуры имеет тип HANDLE и называется hInstance. Дескриптор, который задается здесь, является дескриптором экземпляра, к которому относится оконная процедура класса. В большинстве случаев можно задать значение дескриптора hInstance, получаемого функцией WinMain() в одном из параметров.

В приложениях Win32 используется немало новых типов данных. Многие из них имеют префикс H, который является сокращением слова Handle – дескриптор, описатель. Описатели разных типов (HWND, HPEN, HBITMAP и т. д.) являются посредниками, которые помогают найти нужную структуру данных в виртуальном мире Windows. Объекты или ресурсы Windows, такие как окна, файлы, потоки, перья, кисти, области, представлены в системе структурами языка C, и адреса этих структур могут изменяться. В случае нехватки реальной памяти Windows выгружает из памяти ненужные в данный момент времени объекты и загружает на их место объекты, требуемые приложением. В системной области оперативной памяти Windows поддерживает таблицу, в которой хранятся физические адреса объектов. Для поиска объекта и управления им сначала следует получить у системы его дескриптор (место в таблице, индекс). Важно иметь в виду, что физический адрес объекта – понятие для Windows, а не для программиста. Описатель типа HANDLE можно уподобить номеру мобильного телефона, с помощью которого отыскивается объект, перемещающийся в виртуальном мире Windows.

Седьмой параметр называется hIcon и имеет тип HICON.

Тип HICON – это не что иное, как тип HANDLE. Данный дескриптор указывает на класс значка, используемого окном. Класс значка в действительности является ресурсом значка (иконкой). Функция LoadIcon() загружает ресурс значка из исполняемой программы. Хотя ресурсы компилируются внутрь исполняемого файла

программы (с расширением exe) для Windows, все равно необходимо загружать их, поэтому и требуется вызов данной функции. Вот как выглядит ее прототип:

```
HICON LoadIcon(
HINSTANCE hInstance,
LPCTSTR lpIconName
);
```

У функции всего два параметра: HINSTANCE и LPCTSTR. Первый параметр с именем hInstance содержит дескриптор экземпляра модуля, чей исполняемый файл содержит значок, который предполагается использовать. Если параметру присвоить значение NULL, то поиск ресурса, в данном случае иконки, будет осуществляться среди стандартных. Второй параметр является указателем на строку, содержащую имя загружаемого значка. Взглянув на разбираемый пример, можно увидеть, что для инициализации данного параметра используется константа IDI_APPLICATION. Ее значение соответствует значку, используемому по умолчанию для приложений, который можно видеть во многих программах для Windows. Некоторые другие иконки перечислены в табл. 2.

Таблица 2

Некоторые стандартные иконки

Значение	Описание
IDI_APPLICATION	Значок, используемый по умолчанию для приложений. Он применяется и в рассматриваемом примере. В большинстве случаев его можно использовать, если только не требуется нестандартный значок для приложения
IDI_ASTERISK	Значок в виде небольшого овала с буквой «i» внутри
IDI_ERROR	Красный круг с крестом внутри
IDI_EXCLAMATION	Желтый треугольник с восклицательным знаком внутри
IDI_QUESTION	Значок с вопросительным знаком
IDI_WINLOGO	Небольшой логотип Windows

Восьмой элемент структуры данных WNDCLASSEX очень похож

на седьмой, за исключением того, что он задает используемый окном курсор. Его тип HCURSOR, а имя – hCursor. Тип HCURSOR – это еще один замаскированный дескриптор. Обычно здесь указывается значение дескриптора класса курсора, который будет использован в программе для ее собственного курсора. Функция LoadCursor() похожа на функцию LoadIcon() за исключением того, что она загружает ресурсы курсора, а не ресурсы значка. Вот ее прототип:

```
HCURSOR LoadCursor(
HINSTANCE hInstance,
LPCTSTR lpCursorName
);
```

Первый параметр называется hInstance и содержит дескриптор экземпляра модуля, чей исполняемый файл содержит курсор, который вы собираетесь использовать. Второй параметр – это указатель на строку, содержащую имя загружаемого курсора. Как можно видеть, в рассматриваемом примере я параметр имеет значение IDC_ARROW. Оно соответствует стандартному курсору Windows в виде стрелки. Некоторые другие курсоры перечислены в табл. 3.

Таблица3

Стандартные курсоры

Значение	Описание
IDC_APPSTRING	Курсор в форме стандартной стрелки с присоединенными к ней песочными часами. Обычно данный курсор устанавливается, когда программа занята
IDC_ARROW	Стандартный курсор Windows
IDC_CROSS	Создает курсор, выглядящий как перекрестье прицела
IDC_HELP	Этот курсор выглядит как стандартная стрелка с присоединенным к ней вопросительным знаком. Его хорошо использовать, когда пользователю предоставляется возможность задать вопрос
IDC_IBEAM	Курсор в форме буквы «I». Обычно используется в режиме ввода и редактирования текста
IDC_NO	Курсор в виде перечеркнутого круга. Его можно использовать, когда пользователь наводит курсор на область, которая не реагирует на щелчки кнопок мыши

Значение	Описание
IDC_SIZEALL	Курсор с перекрещенными стрелками. Применяется, когда пользователь изменяет размер окна или графического элемента
IDC_SIZENESW	Еще один курсор для изменения размера. В отличие от предыдущего курсора, у которого стрелки направлены во все четыре стороны, здесь стрелки направлены только на северо-восток и юго-запад
IDC_SIZENS	То же, что и предыдущий курсор, но стрелки направлены на север и юг
IDC_SIZENWSE	То же, что и предыдущие два курсора, но стрелки направлены на северо-запад и юго-восток
IDC_SIZEWE	Еще один курсор со стрелками. В данном случае они направлены на запад и восток
IDC_UPARROW	Курсор в виде стрелки, направленной вверх
IDC_WAIT	Курсор в виде песочных часов

Девятый элемент структуры hbrBackground имеет тип HBRUSH и определяет цвет фона окна. Можно задать цвет, используя константы вида:

```
wndclass.hbrBackground =(HBRUSH)COLOR_GRAYTEXT;
```

И другие константы, начинающиеся с **COLOR_**.

Или, как показано в примере, использовать дескриптор стандартной кисти, полученный с помощью функции GetStockObject(). Функция GetStockObject() часто используется для того, чтобы получить дескриптор одной из встроенных кистей, шрифтов, палитр или перьев. Дело в том, что в Windows есть несколько предопределенных типов, которые могут быть использованы. Прототип функции следующий:

```
HGDIOBJ GetStockObject(int fnObject);
```

Ее единственный параметр представляет собой целое число, идентифицирующее предопределенный объект операционной системы. В примере используется встроенный объект DKGRAY_BRUSH. Он окрашивает фон окна в темно-серый цвет.

Десятый элемент структуры данных WNDCLASSEX – это завершающаяся нулевым символом строка с именем lpszMenuName. Она со-

держит имя ресурса меню, используемого окном. NULL указывает на отсутствие меню в программе.

Одиннадцатый элемент структуры данных также является строкой, которая должна завершаться нулевым символом. Его имя – lpszClassName. Как сказано в имени, эта строка используется для задания имени класса окна. Имя класса является уникальным идентификатором типа класса. Поэтому очень важно, чтобы заданное здесь имя не использовалось для других классов окон программы.

Двенадцатый, и последний, элемент структуры WNDCLASSEX – это переменная с именем hIconSm. Она аналогична элементу данных hIcon, за исключением того, что здесь задается используемый программой маленький значок.

Класс окна должен быть зарегистрирован. Ниже приведен фрагмент кода, выполняющий это действие.

```
If(RegisterClassEx(&wndclass) == 0)
{
  // Сбой программы, выход
  return 0;
}
```

Вызов функции RegisterClassEx() необходим, чтобы потом можно было создать окно. Эта функция регистрирует класс в системе Windows. Если класс не зарегистрирован, то его невозможно использовать для создания окна.

Прототип функции следующий:

```
ATOM RegisterClassEx(
CONST WNDCLASSEX *lpwcx
);
```

Первый и единственный необходимый для нее параметр является указателем на структуру данных WNDCLASSEX. Функция возвращает значение типа ATOM, которое можно сравнить с NULL. Если возвращаемое функцией RegisterClassEx() значение не равно нулю, ее выполнение завершилось успешно.

Итак, класс окна зарегистрирован, и программа переходит к действительному созданию окна.

Для создания окна используется функция CreateWindowEx(). Ее можно применять для создания дочерних, всплывающих или перекрывающихся окон. При создании окна указываются используемый класс, имя приложения и некоторые другие параметры. Прототип функции выглядит следующим образом:

```
HWND CreateWindowEx(
  DWORD dwExStyle,
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName,
  DWORD dwStyle,
  int x,
  int y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  LPVOID lpParam
);
```

Первый параметр имеет тип DWORD и называется dwExStyle. Он похож на определяющий стиль элемент структуры WNDCLASSEX, но задает дополнительные стили окна.

WS_EX_OVERLAPPEDWINDOW является достаточно распространенным стилем, благодаря чему скомпилированная и запущенная программа выглядит, как большинство приложений Windows.

LPCTSTR lpClassName – имя класса для создаваемого окна (это имя использовалось при регистрации класса).

LPCTSTR lpWindowName – имя окна.

DWORD dwStyle – стиль окна.

Int x – позиция по горизонтали верхнего левого угла окна.

Int y – позиция по вертикали.

Int nWidth – ширина окна.

Int nHeight – высота окна.

HWND hWndParent – используется для создания «дочернего окна» («child window»). Сюда передается дескриптор «родительского окна» («parent window»).

HMENU hMenu – дескриптор меню (если hMenu равно нулю, используется меню класса, указанного в lpClassName).

HINSTANCE hInstance – экземпляр приложения.

LPVOID lpParam – указатель на пользовательский параметр окна. Этот указатель со всеми остальными параметрами функции CreateWindow будет занесен в структуру CREATESTRUCT. В сообщениях WM_CREATE или WM_NCCREATE параметр lParam будет содержать указатель на эту структуру.

Функция CreateWindow возвращает уникальный дескриптор окна HWND. Если функция вернула ноль, значит, во время создания окна произошла ошибка.

Создание окна не приводит к его отображению. Чтобы окно было действительно выведено на экран, необходимо вызвать функцию ShowWindow(). Ее прототип:

```
BOOL ShowWindow(
  HWND hWnd,
  int nCmdShow
);
```

Первый параметр задает дескриптор отображаемого окна. Это действительно просто, поскольку дескриптор уже подготовлен функцией, создавшей окно. Второй параметр – это целое число, определяющее, как будет отображаться окно. Можно использовать константы вида SW_MINIMIZE (запуск в свернутом состоянии) или отобразить, как рекомендуется операционной системой, передав в функцию параметр, полученный функцией WinMain от Windows.

Чтобы приложения Windows знали, когда их окна закрываются, перемещаются или изменяют размеры, они должны принимать сообщения Windows. В общем случае приложения для Windows должны всегда иметь цикл сообщений. Чтобы проверить наличие ожидающих обработки сообщений, вызывается функция GetMessage():

```
BOOL GetMessage(
  LPMSG lpMsg,
  HWND hWnd,
  UINT wMsgFilterMin,
  UINT wMsgFilterMax
);
```

В первом параметре функция ожидает указатель на объект MSG. Структура данных MSG содержит всю информацию о любом обнаруженном сообщении.

Второй параметр указывает, для какого окна проводится проверка наличия сообщений. Он необходим потому, что программа может управлять несколькими окнами. В рассматриваемом примере есть только одно окно, поэтому просто передается дескриптор окна, созданного функцией CreateWindow(). Можно указать NULL, что будет означать обработку сообщений всех окон программы.

Третий параметр задает нижнюю границу кодов получаемых сообщений. Нужно получать все сообщения, поэтому данному параметру присваиваетсязначение 0.

Четвертый параметр позволяет задать верхнюю границу кодов получаемых сообщений. Поскольку необходимо получать все сообщения, значение этого параметра также равно 0.

Перед тем как отправить сообщение в очередь сообщений, его необходимо транслировать в символьные данные. Это делает функция TranslateMessage(). Для нее требуется единственный параметр – указатель на транслируемое сообщение.

После того как сообщение транслировано в символьные данные, его нужно поместить в очередь сообщений с помощью функции DispatchMessage().

Подобно функции TranslateMessage(), функция DispatchMessage() требует единственного параметра. Цель этой функции – отправить прошедшее трансляцию сообщение в очередь сообщений программы. После вызова этой функции сообщение попадает в функцию обработки сообщений. Последняя строка кода функции WinMain() возвращает значение wParam последнего сообщения Windows, извлеченного функцией получения сообщений.

Как видно, главная функция программы завершается циклом while, который будет обрабатываться до тех пор, пока GetMessage() не вернет ложное значение. Это происходит при приеме сообщения WM_QUIT, генерация которого осуществляется функцией PostQuitMessage().

Стартовая заготовка иллюстрирует стандартную последовательность действий при создании Windows-приложения на базе API-функций. Обратите внимание на то, что функция WndProc нигде явно не вызывается, хотя именно она выполняет всю полезную работу.

Теперь рассмотрим, как устроена оконная процедура WndProc. Ее имя уже дважды появлялось в тексте программы. Сначала был объявлен ее прототип, затем оно было присвоено одному из полей структуры типа WNDCLASSEX. Поле имеет тип указателя на функцию с особым прототипом оконной функции. Здесь полезно вспомнить, что имя функции трактуется компилятором C++ как ее адрес.

Оконная процедура должна «просеивать» все посылаемые ей сообщения и обрабатывать те из них, которые были выбраны программистом для обеспечения желаемой функциональности. Типичной структурой оконной процедуры является switch-блок, каждая ветвь которого содержит обработку одного сообщения. В первом примере оконная процедура реагирует только на три сообщения:

1) WM_LBUTTONUP – о щелчке при отпускании пользователем левой кнопки мыши;
2) WM_PAINT – о необходимости перерисовать клиентскую область окна;
3) WM_DESTROY – о необходимости закрыть окно. Сообщение WM_DESTROY (окно уничтожено) посылается системой уже после того, как окно исчезло с экрана. Мы реагируем на него вызовом функции PostQuitMessage, которая указывает системе, что поток приложения требует своего завершения, путем посылки сообщения WM_QUIT. Его параметром является код завершения, который мы указываем при вызове PostQuitMessage.

Рассмотренная структура приложения Win32 позволяет сделать вывод, что в подавляющем числе случаев развитие приложения сосредоточено внутри оконной процедуры, а не в функции WinMain. Развитие приложения заключается в том, что в число обрабатываемых сооб-

щений (messages) включаются новые. Для этого программист должен вставлять новые case-ветви в оператор switch (msg).

Если оконная процедура не обрабатывает какое-либо сообщение, то управление передается в ветвь default. В этой ветви мы вызываем функцию DefWindowProc, которая носит название оконной процедуры по умолчанию. Эта функция гарантирует, что все сообщения будут обработаны, то есть удалены из очереди. Возвращаемое значение зависит от посланного сообщения.

Результат компиляции и запуска примера показан на рис. 1.

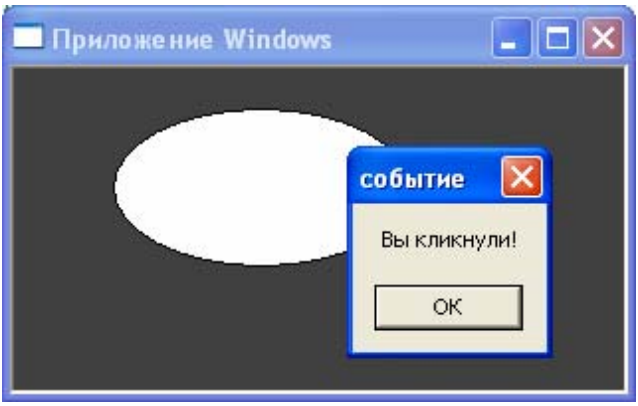

Рис. 1. Результат выполнения примера

Контрольные вопросы

1. Чем отличается последовательность выполнения команд традиционной DOS-программы и современного Windows-приложения?
2. Что такое событие и сообщение в рамках программирования в ОС Windows? Какие бывают сообщения?
3. Какова простейшая структура событийно-управляемой программы на *Си*?
4. Что такое оконная процедура, каково ее назначение?

5. Почему такие параметры, как адрес оконной процедуры или иконка курсора, задаются при регистрации класса окна, а название окна – в аргументах функции для его создания?
6. Что такое цикл сообщений?
7. Требуется ли приложению обрабатывать все возможные виды сообщений, существующих в ОС?
8. Можно ли просто вывести что-то на экран в произвольном месте кода программы? Обоснуйте свой ответ.
9. Как происходит завершение программы? Каково назначение функции PostQuitMessage()?
10. Оконная процедура нигде не вызывается явно из главной функции программы, хотя все основные команды сосредоточены именно в ней. Когда и чем она вызывается?

2. ВИЗУАЛИЗАЦИЯ ЦИФРОВЫХ ДАННЫХ

2.1. Вывод графической информации на дисплей

Хотя некоторым приложениям Windows для организации полного интерфейса пользователя нужны только окна, диалоги и управляющие элементы, многим приложениям (например, программам-осциллографам или генераторам функций) требуется наличие графических средств. Эта простейшая графика может быть в форме линий, кривых, фигур, текста и т. д.

Для предоставления приложениям графических функциональных возможностей Windows имеет стандартный набор функций, называемый интерфейсом графических устройств GDI. Функции GDI предоставляют возможности рисования, которые не зависят от используемого устройства вывода. Например, одни и те же функции можно использовать для организации вывода на дисплей или принтер. Аппаратная независимость реализуется через использование драйверов устройств, которые переводят функции GDI в команды, воспринимаемые используемым устройством вывода. Это означает, что не нужно беспокоиться о том, как конкретное устройство (например, определенная марка принтеров) работает с графическим образом. Подобная абстракция аппаратной части является очень распространенной. Ценой универсальности являются достаточно ограниченные возможности по работе с графикой и невысокое быстродействие графических команд. Насыщенные визуальными данными программы следует разрабатывать с использованием специализированных программных средств: OpenGL, GLES, Direct3D и т. п.

В отличие от традиционных графических программ DOS программы Windows никогда не выводят элементы изображения непосредственно на экран или принтер, а записывают их в логической последовательности, называемой контекстом устройства. Контекст устройства – это виртуальная поверхность с присущими ей атрибутами, такими как перо, кисть, шрифт, цвет фона, цвет текста и текущая позиция. Для приложения, независимо от того, какое это на самом деле устройство, все контексты устройства выглядят аналогично.

Как правило, приложения выполняют свою работу по рисованию содержимого окна во время обработки сообщения Windows WM_PAINT, хотя часто требуется рисовать и во время обработки других сообщений. В любом случае приложению нужно придерживаться следующей последовательности действий:

1) получение или создание контекста отображения;
2) установка необходимых атрибутов в контексте отображения;
3) выполнение операций рисования;
4) освобождение либо удаление контекста отображения.

Последнее действие (освобождение либо удаление контекста отображения) должно быть обязательно выполнено, так как контекст – очень ресурсоёмкая структура данных. Если создать множество контекстов, а затем не удалить (или не освободить) их, система станет работать нестабильно из-за нехватки ресурсов, единственным выходом станет ее перезагрузка.

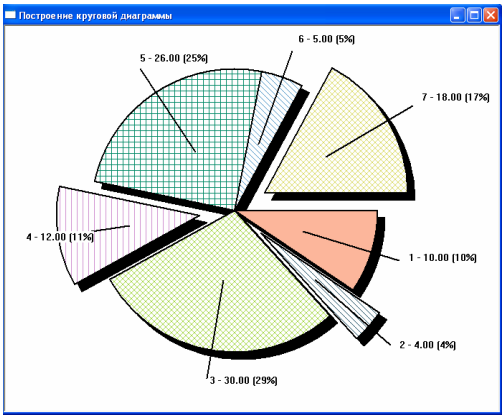

Рис. 2. Тестовая программа, выполняющая визуализацию данных

Рассмотрим особенности работы устройств вывода информации на примере программы построения круговой диаграммы, окно которой показано на рис. 2.

Программа основана на примере простейшей программы Windows.

Пример текста на языке C++ рабочей программы (для удобства новый код выделен жирным шрифтом):

```
#define  _USE_MATH_DEFINES
#include <math.h>

#include <stdio.h>
#include <time.h>

// Стандартный включаемый файл Windows
#include <windows.h>

const float g_data[] = { 10.0f, 4.0f, 30.0f, 12.0f, 26.0f, 5.0f,
18.0f };

struct Brush
{
     int            fnStyle;
     COLORREF  clrref;
};

unsigned int   g_iNumOfPies;
Brush*              g_pBrushes;

// Прототип функции обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void Diagram(HDC hDC,
               int x,
               int y,
               int r,
               bool fShadow,
               const float* pData,
               const Brush* pBrushes,
               unsigned int iNumOfPies);

void DiagramPie(HDC hDC,
                    int x,
                    int y,
                    int r,
                    float s,
                    float e,
                    int fnStyle,
                    COLORREF clrref,
                    const char* szLabel);
```

```
void ShadyDiagram(HDC hDC,
                  int x,
                  int y,
                  int r,
                  int iShadowDepth,
                  const float* pData,
                  const Brush* pBrushes,
                  unsigned int iNumOfPies);

// Функция вызывается автоматически, когда программа запускается

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance
PSTR szCmdLine, int iCmdShow)
{
     HWND         hWnd;
     MSG          msg;
     WNDCLASSEX   wndclass;

     // Настройка класса окна
     wndclass.cbSize         = sizeof(WNDCLASSEX);
     wndclass.style          = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc    = WndProc;
     wndclass.cbClsExtra     = 0;
     wndclass.cbWndExtra     = 0;
     wndclass.hInstance      = hInstance;
     wndclass.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor        = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =
(HBRUSH)GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = "Window Class"; // Имя класса
     wndclass.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);

     // Регистрация класса окна
     if(RegisterClassEx(&wndclass) == 0)
     {
          // Сбой программы, выход
          return 0;
     }

     // Создание окна
     hWnd = CreateWindowEx(
          WS_EX_OVERLAPPEDWINDOW,
          «Window Class»,                              // Имя
класса
          «Построение круговой диаграммы»,   // Текст заголовка
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          700,
          580,
          NULL,
          NULL,
```

```
            hInstance,
            NULL);

      // Отображение окна
      ShowWindow(hWnd, iCmdShow);

      // Обработка сообщений, пока программа не будет прервана
      while(GetMessage(&msg, NULL, 0, 0))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);

      }

      return (int)msg.wParam;
}

// Функция обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg, WPARAM wParam,
LPARAM lParam)
{
      HDC                  hDC;
      PAINTSTRUCT          ps;
      unsigned int   I;

      switch(iMsg)
      {
      case WM_CREATE:
            g_iNumOfPies   =    sizeof(g_data) / sizeof(g_data[0]);

            g_pBrushes           =    new Brush[g_iNumOfPies];

            srand((unsigned int)time(NULL));

            for (I = 0u; I < g_iNumOfPies; ++i)
            {
                  g_pBrushes[i].fnStyle    =    rand() % 7;
                  g_pBrushes[i].clrref     =    RGB(rand() % 0xFF,
rand() % 0xFF, rand() % 0xFF);
            }

            break;
      // Вызывается, когда окно обновляется
      case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);
            ShadyDiagram(hDC, 320, 260, 200, 10, g_data,
g_pBrushes, g_iNumOfPies);
            EndPaint(hWnd, &ps);
            break;
      // Вызывается, когда пользователь закрывает окно
      case WM_DESTROY:
            if (g_pBrushes)
                  delete[] g_pBrushes;
```

```
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, iMsg, wParam, lParam);
        }

        return 0;
}

void Diagram(HDC hDC, int x, int y, int r, bool fShadow, const
float* pData, const Brush* pBrushes, unsigned int iNumOfPies)

{
        HPEN            hOldPen, hPen;
        char            szLabel[BUFSIZ];
        int                 dx, dy;
        float               e, s;
        float               k;
        unsigned int    I;

        for (I = 0u, s = 0.0f; I < iNumOfPies; ++i)
            s += pData[i];

        if (s == 0.0f)
            return;

        k = (float)(2.0f * M_PI / s);

        hPen =      CreatePen(PS_SOLID, 2, 0);
        hOldPen     =       (HPEN)SelectObject(hDC, hPen);

        for (I = 0u, s = 0.0f; I < iNumOfPies; ++i)
        {
            e = s + pData[i] * k;

            if (I == 1 || I == 3 || I == 6)
            {
                dx = int(cos((e + s) / 2) * r * 0.25f);
                dy = int(sin((e + s) / 2) * r * 0.25f);
            }
            else
            {
                dx = dy = 0;
            }

            sprintf(szLabel, "%d - %.02f (%.0f%%)", I + 1u,
(float)pData[i], (float)(pData[i] * k * 50.0f / M_PI));

            if (fShadow)
                DiagramPie(hDC, x + dx, y + dy, r, s, e, 6, 0,
NULL);
```

```
            else
                DiagramPie(hDC, x + dx, y + dy, r, s, e,
pBrushes[i].fnStyle, pBrushes[i].clrref, szLabel);

            s = e;
        }

        SelectObject(hDC, hOldPen);
        DeleteObject(hPen);

        return;
}

void DiagramPie(HDC hDC, int x, int y, int r, float s, float e,
int fnStyle, COLORREF clrref, const char* szLabel)
{
        HBRUSH    hBrush, hOldBrush;
        float     dx, dy;
        float     m;

        hBrush         =    CreateHatchBrush(fnStyle, clrref);
        hOldBrush =     (HBRUSH)SelectObject(hDC, hBrush);

        Pie(
            hDC,
            x - r,
            y - r,
            x + r,
            y + r,
            x + int(cosf(e) * r),
            y + int(sinf(e) * r),
            x + int(cosf(s) * r),
            y + int(sinf(s) * r)
        );

        SelectObject(hDC, hOldBrush);
        DeleteObject(hBrush);

        if (!szLabel)
            return;

        m = (e + s) / 2.0f;

        dx = cosf(m) * r;
        dy = sinf(m) * r;

        MoveToEx(
            hDC,
            x + int(dx * 0.5f),
            y + int(dy * 0.5f),
            NULL
        );
```

```
    LineTo(
         hDC,
         x + int(dx * 1.2f),
         y + int(dy * 1.2f)
    );

    TextOut(
         hDC,
         x + int(dx * 1.25f) + 5,
         y + int(dy * 1.25f) - 15,
         szLabel,
         (int)strlen(szLabel)
    );


     return;
}

void ShadyDiagram(HDC hDC, int x, int y, int r, int
iShadowDepth, const float* pData, const Brush* pBrushes,
unsigned int iNumOfPies)
{
     Diagram(hDC, x + iShadowDepth, y + iShadowDepth, r, true,
pData, pBrushes, iNumOfPies);
     Diagram(hDC, x, y, r, false, pData, pBrushes, iNumOfPies);

     return;
}
```

Подробно рассмотрим работу программы. Вывод на экран осуществляется при обработке сообщения WM_PAINT:

```
case WM_PAINT:
          hDC = BeginPaint(hWnd, &ps);
          ShadyDiagram(hDC, 320, 260, 200, 10, g_data,
g_pBrushes, g_iNumOfPies);
          EndPaint(hWnd, &ps);
          break;
```

Почему нельзя просто вывести данные в произвольном месте программы? Современные операционные системы, как правило, являются многозадачными, что означает возможность одновременной работы нескольких программ. Допустим, что какой-то рисунок отображен в окне программы, потом окно частично перекрывается другим окном или полностью исчезает с экрана. Затем через некоторое время верхнее окно сдвигается. В такой ситуации операционная система самостоятельно не восстановит пропавшую часть изображения.

Существует взаимодействие «документ – вид», используемое при графических построениях с помощью электронных устройств. Документ представляет некие данные, невидимые для пользователя, хранимые в памяти компьютера, которые пользователь может редактировать, загружать и сохранять, используя диск. С другой стороны, вид – временные данные, связанные с документом, которые пользователь ассоциирует с ним. Например, в текстовом документе программы «Блокнот» – массив байт, каждый байт представляет один символ, хранимый в оперативной памяти. Видом этого документа является графическое изображение на экране монитора с помощью шрифта определенного типа, размера, цвета и т. д. Вид можно всегда восстановить по данным документа, используя предопределенный программистом алгоритм, что и осуществляется при обработке сообщения WM_PAINT.

Функция BeginPaint возвращает контекст отображения для окна hWnd:

HDC WINAPI BeginPaint(HWND hWnd, PAINTSTRUCT FAR *lpps);

Перед этим функция подготавливает окно для рисования, заполняя структуру PAINTSTRUCT информацией, которую можно использовать в процессе рисования.

Контекст отображения, полученный с помощью функции BeginPaint, необходимо освободить перед завершением обработки сообщения WM_PAINT, вызвав функцию EndPaint с теми же параметрами:

void WINAPI EndPaint(HWND hWnd, PAINTSTRUCT FAR *lpps);

Функции BeginPaint и EndPaint можно использовать только внутри обработчика сообщения WM_PAINT. Если требуется рисовать при обработке других сообщений, получить контекст отображения можно с помощью функции GetDC() и освободить функцией ReleaseDC().

Функция GetDC возвращает контекст отображения для окна с идентификатором hwnd:

HDC WINAPI GetDC(HWND hWnd);

Функция ReleaseDC освобождает контекст изображения hdc, полученный для окна hwnd:

int WINAPI ReleaseDC(HWND hwnd, HDC hdc);

Каждый раз, когда приложение получает общий контекст отображения, его атрибуты получают значения по умолчанию: белая кисть для заполнения фигур, черное перо для рисования линий и т. п. Если перед выполнением рисования приложение изменит атрибуты контекста отображения, вызвав соответствующие функции GDI, в следующий раз при получении значений эти атрибуты вновь примут значения по умолчанию. Поэтому установка атрибутов должна выполняться каждый раз после получения общего контекста отображения.

Итак, данные в программе определяются следующим образом:

```
const float g_data[] = { 10.0f, 4.0f, 30.0f, 12.0f, 26.0f, 5.0f,
18.0f };

struct Brush
{
     int            fnStyle;
     COLORREF  clrref;
};

unsigned int   g_iNumOfPies;
     Brush*              g_pBrushes;
```

где g_data – массив некоторых данных, по которым строится диаграмма; структура Brush используется для хранения типов кистей (различные виды штриховки) и их цветов; g_iNumOfPies – количество значений, по которым строится диаграмма; g_pBrushes – кисти; массив будет динамически создан при запуске программы и заполнен случайными значениями. Разумеется, размеры массивов g_data и g_pBrushes одинаковы и равны g_iNumOfPies.

После успешного создания окна массив кистей создается и заполняется:

```
case WM_CREATE:
          g_iNumOfPies   =    sizeof(g_data) / sizeof(g_data[0]);

          g_pBrushes          =    new Brush[g_iNumOfPies];

          srand((unsigned int)time(NULL));

          for (I = 0u; I < g_iNumOfPies; ++i)
          {
               g_pBrushes[i].fnStyle    =    rand() % 7;
               g_pBrushes[i].clrref     =    RGB(rand() % 0xFF,
rand() % 0xFF, rand() % 0xFF);
          }
```

Зачем он нужен?

Например, пользователь поменял размер окна, следовательно, всю диаграмму придется перерисовать и при этом желательно, чтобы цвета всех сегментов остались такими же, какими были. Если программу запустить повторно, диаграмма перекрасится.

Все ресурсы, которые мы запрашиваем в программе, например, оперативную память под массив данных, должны быть освобождены при завершении программы. Это удобно сделать при обработке противоположного WM_CREATE сообщения WM_DESTROY:

```
case WM_DESTROY:
          if (g_pBrushes)
               delete[] g_pBrushes;
```

Все действия по построению диаграммы осуществляются при обработке сообщения WM_PAINT:

```
case WM_PAINT:
          hDC = BeginPaint(hWnd, &ps);
          ShadyDiagram(hDC, 320, 260, 200, 10, g_data,
g_pBrushes, g_iNumOfPies);
          EndPaint(hWnd, &ps);
          break;
```

Весь код инкапсулирован в три пользовательские функции:

1) ShadyDiagram – построение полной диаграммы;
2) Diagram – построение части диаграммы (основной или тени);
3) DiagramPie – построение одного сегмента диаграммы.

ShadyDiagram просто вызывает два раза функцию Diagram, передавая немного разные по величине iShadowDepth (глубина тени) координаты центра для диаграммы и ее тени:

```
     void ShadyDiagram(HDC hDC, int x, int y, int r, int
iShadowDepth, const float* pData, const Brush* pBrushes,
unsigned int iNumOfPies)
     {
          Diagram(hDC, x + iShadowDepth, y + iShadowDepth, r,
true, pData, pBrushes, iNumOfPies);
          Diagram(hDC, x, y, r, false, pData, pBrushes,
iNumOfPies);
```

```
        return;
    }
```

Функция построения одного сегмента имеет следующий прототип:

```
    void DiagramPie(HDC hDC, int x, int y, int r, float s, float
e, int fnStyle, COLORREF clrref, const char* szLabel);
```

В функцию передается контекст устройства, координаты центра *x* и *y* (по умолчанию ось *x* направлена вправо, ось *y* – вниз, начало координат – в левом верхнем углу рабочей области окна), радиус, начальный и конечный углы в радианах, между которыми строится сегмент *s* и *e*, стиль кисти и ее цвет, текстовая метка для сегмента. Если **szLabel** равно NULL, метка не отображается (необходимо для сегментов тени).

Создание штриховой кисти осуществляется следующим образом:

```
hBrush      =     CreateHatchBrush(fnStyle, clrref);
```

Первым параметром функции задается стиль штриховки, вторым – цвет. После создания кисть необходимо выбрать в качестве текущей:

```
hOldBrush = (HBRUSH)SelectObject(hDC, hBrush);
```

Функция универсальна для кистей, перьев и т. п., поэтому требуется явное преобразование типа. Возвращает предыдущую используемую кисть. Следует удалить кисть после использования.

```
DeleteObject(hBrush);
```

Существует правило, справедливое для различных областей программирования: нельзя удалять что-то, используемое в данный момент. Поэтому перед удалением нужно задать другую кисть текущей, например, предыдущую. Такой же подход применяется в функции Diagram() при смене пера.

Все последующие команды вывода используют прямоугольную декартовую систему координат, поэтому для преобразования угловых величин в прямоугольные координаты используются тригонометрические функции *cos*() и *sin*().

2.2. Вывод графической информации на принтер

При выводе на принтер отличий немного. Связаны они прежде всего с постраничным графическим выводом на печать. Дело в том, что при вызове функций рисования для контекста принтера эти команды GDI выполняются не сразу, а накапливаются в специальном метафайле. И только после того, как приложение завершит рисование одной страницы документа, созданный метафайл «проигрывается» в контексте принтера. Печать происходит именно в этот момент. Следовательно, требуется сообщить о начале и завершении процесса печати листа.

Функции для работы с принтером

1. **StartDoc** – формирует задание на печать нового документа.
2. **StartPage** – подготавливает контекст устройства вывода для печати новой страницы – готовит метафайл, необходимо вызвать эту функцию перед выводом в контекст устройства.
3. **EndPage** – завершает программный процесс печати одной страницы – формирование метафайла, после чего он выводится непосредственно на принтер.
4. **EndDoc** – завершает процесс печати документа.
5. **AbortDoc** – служит для принудительного завершения процесса печати.
6. **SetAbortProc** – используется для обеспечения возможности фоновой печати и принудительного завершения процесса печати.
7. **ResetDC** – позволяет настроить индивидуальные параметры печати отдельных листов документа.

На первый взгляд, контекст отображения для принтера получить нетрудно – достаточно вызвать функцию CreateDC, указав имя драйвера, имя устройства и имя порта вывода, к которому подключен принтер:

```
HDC WINAPI CreateDC(
LPCSTR lpszDriver,// имя драйвера
LPCSTR lpszDevice,// имя устройства
LPCSTR lpszOutput,// имя файла или порта вывода
const void FAR* lpvInitData); // данные для инициа-
лизации
```

Созданный при помощи функции CreateDC контекст устройства следует удалить после использования, вызвав функцию DeleteDC:

```
BOOL WINAPI DeleteDC(HDC hdc);
```

Параметр lpszDriver является указателем на строку символов, содержащую имя драйвера, обслуживающего физическое устройство. Имя драйвера совпадает с именем файла *.drv, содержащего драйвер. Этот драйвер находится в системном каталоге Windows.

Имя устройства lpszDevice – это название устройства.

Параметр lpszOutput указывает на структуру данных типа DEVMODE, используемую при инициализации устройства вывода. Если при работе с устройством нужно использовать параметры, установленные при помощи приложения Control Panel, параметр lpszOutput следует указать как NULL.

Данные о текущем принтере и всех установленных можно считать из системных ini файлов (windows/win.ini) функцией GetProfileString().

Другим способом получения контекста принтера является использование стандартного диалогового окна печати.

С помощью функции PrintDlg() приложение может вывести на экран диалоговое окно, представленное на рис. 2, с помощью которого пользователь может напечатать документ, выбрать нужный принтер или изменить его параметры.

Прототип функции, описанный в файле commdlg.h (обычно включается в файле windows.h), следующий:

```
BOOL PrintDlg(PRINTDLG FAR* lppd);
```

При успешном завершении функция возвращает значение TRUE. В случае ошибки, отмены печати или выбора принтера (если функция PrintDlg используется только для выбора принтера) функция возвращает значение FALSE.

В качестве параметра функции PrintDlg необходимо передать адрес предварительно подготовленной структуры типа PRINTDLG, описанной в файле commdlg.h:

```
typedef struct tagPD
{
  DWORD   lStructSize;
```

```
  HWND    hwndOwner;
  HGLOBAL hDevMode;
  HGLOBAL hDevNames;
  HDC     hDC;
  DWORD   Flags;
  UINT    nFromPage;
  UINT    nToPage;
  UINT    nMinPage;
  UINT    nMaxPage;
  UINT    nCopies;
  HINSTANCE hInstance;
  LPARAM  lCustData;
  UINT (CALLBACK* lpfnPrintHook)(HWND, UINT,WPARAM,LPARAM);
  UINT (CALLBACK* lpfnSetupHook)(HWND, UINT,WPARAM,LPARAM);
  LPCSTR  lpPrintTemplateName;
  LPCSTR  lpSetupTemplateName;
  HGLOBAL hPrintTemplate;
  HGLOBAL hSetupTemplate;
} PRINTDLG;
typedef PRINTDLG  FAR* LPPRINTDLG;
```

Рассмотрим назначение некоторых полей этой структуры.

lStructSize – размер структуры PRINTDLG в байтах. Это поле НЕОБХОДИМО заполнить перед вызовом функции PrintDlg(), иначе окно даже не появится.

hwndOwner – идентификатор родительского окна. Если в поле Flags не указано значение PD_SHOWHELP, в поле hwndOwner можно указать NULL.

hDevNames – идентификатор глобального блока памяти, имеющего в своем составе структуру типа DEVNAMES из трех текстовых строк. Первая строка определяет имя драйвера принтера, вторая – имя принтера и третья – имя порта вывода, к которому подключен принтер.

Если содержимое этого поля указать как NULL, после возвращения из функции PrintDlg поле будет включать идентификатор глобального блока памяти, заказанного функцией для структуры

DEVNAMES. В структуре будут находиться строки, соответствующие выбранному принтеру.

hDevMode – идентификатор глобального блока памяти, охватывающего структуру типа DEVMODE, которая используется для инициализации параметров принтера.

Если содержимое этого поля указать как NULL, после возвращения из функции PrintDlg поле будет вмещать идентификатор глобального блока памяти, заказанного функцией. В этом блоке памяти будет расположена структура DEVMODE, заполненная выбранными параметрами принтера.

hDC – контекст устройства или информационный контекст.

Это поле заполняется после возвращения из функции PrintDlg, если в поле Flags указано одно из значений: PD_RETURNDC или PD_RETURNIC. В первом случае возвращается контекст принтера, который можно использовать для печати, во втором – информационный контекст, который можно использовать для получения разнообразной информации о принтере.

Flags – это поле должно содержать флаги инициализации. Тут с помощью логической операции можно указать начальное положение переключателей, отключить некоторые пункты диалога (например, печать выделенного фрагмента, если такая возможность отсутствует в программе) и т. п.

Перед вызовом функции PrintDlg() следует проинициализировать нужные поля, остальные установить в ноль:

```
BOOL          fResult;
PRINTDLG      pd;

...

ZeroMemory(&pd, 0, sizeof(PRINTDLG));

pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner   = hWnd;
pd.Flags       = PD_RETURNDC;

fResult        = PrintDlg(&pd);
```

```
if (pd.hDevMode)
  GlobalFree(pd.hDevMode);

if (pd.hDevNames)
  GlobalFree(pd.hDevNames);
```

Следует обратить внимание на необходимость освобождения глобальных блоков памяти (функцией GlobalFree()), а после завершения печати и контекста принтера функцией **DeleteDC(HDC hDC)**.

Внесем в демонстрационную программу следующие изменения:

1. Добавим прототип функции печати после прототипа функции ShadyDiagram:

```
void Print(HWND hWnd);
```

2. В оконную процедуру добавим обработчик отпускания левой кнопки мыши после обработчика функции WM_CREATE:

```
caseWM_LBUTTONUP:
Print(hWnd); break;
```

3. В конец файла добавим описание функции Print().

```
Void Print(HWND hWnd)
{
      DOCINFO           docinfo;
      PRINTDLG          pd;

      ZeroMemory(&pd, sizeof(pd));

      pd.lStructSize = sizeof(PRINTDLG);
      pd.hwndOwner= hWnd;
      pd.Flags= PD_RETURNDC;

      if (PrintDlg(&pd))
      {
            ZeroMemory(&docinfo, sizeof(docinfo));

            docinfo.cbSize = sizeof(docinfo);

            if (StartDoc(pd.hDC, &docinfo) > 0)
            {
                  StartPage(pd.hDC);
                  ShadyDiagram(pd.hDC,   1600,   1300,
```

```
1000, 50, g_data, g_pBrushes, g_iNumOfPies);
                    EndPage(pd.hDC);
                    EndDoc(pd.hDC);
                }
        }
        if(pd.hDevMode)
                GlobalFree(pd.hDevMode);
        if(pd.hDevNames)
                GlobalFree(pd.hDevNames);
        if (pd.hDC)
                DeleteDC(pd.hDC);
        return;
}
```

По щелчку левой кнопкой мыши появляется диалоговое окно печати, в котором можно выбрать принтер и нажать «Печать» (рис. 3).

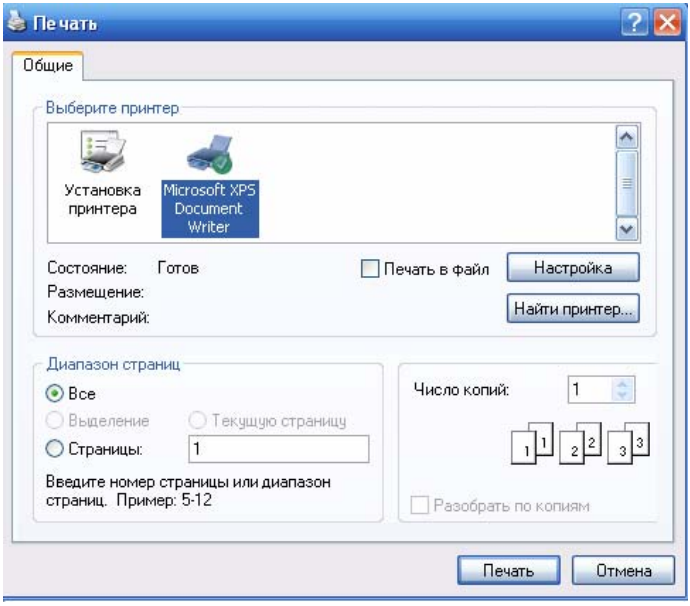

Рис. 3. Настройка печати

Функция Print() работает следующим образом:

Обнуляется память, занимаемая структурой PRINTDLG:

```
ZeroMemory(&pd, sizeof(pd));
```

Затем некоторым полям присваиваются значения:

```
pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner= hWnd;
pd.Flags= PD_RETURNDC;
```

Важным является установка флага PD_RETURNDC.

Следующим шагом вызывается функция PrintDlg() и, если она возвращает истинный результат, что соответствует нажатию кнопки «Печать» (а не «Отмена»), выполняется тело условного оператора if:

```
if (PrintDlg(&pd))
```

При печати заполняется единственное требуемое поле структуры DOCINFO. В целом структура содержит информацию для очереди печати, например количество страниц, которое программа собирается напечатать, название документа. Подробнее можно узнать, обратившись к справке по данной структуре.

Далее вызывается функция начала печати нового документа:

```
if (StartDoc(pd.hDC, &docinfo) > 0)
```

И если не произошло ошибок (результат положительный), начинается печать первой и единственной страницы:

```
StartPage(pd.hDC);
```

Все графические построения на принтере осуществляет такая же функция, что и на экране:

```
ShadyDiagram(pd.hDC, 1600, 1300, 1000, 50, g_data,
g_pBrushes, g_iNumOfPies);
```

Следует обратить внимание на увеличенный в пять раз масштаб (значения координат). Дело в том, что по умолчанию применяются единицы измерения конкретного устройства. Для монитора это пиксели, для принтера – точки. И если 600 пикселей на экране часто занимают половину монитора, то на принтере это чуть больше 2,5 см

(при разрешении 600 точек на дюйм). При печати удобно применять системы координат в миллиметрах или дюймах, что задается функцией SetMapMode().

После вывода изображения сообщаем принтеру о завершении печати страницы и документа в целом:

```
EndPage(pd.hDC);
EndDoc(pd.hDC);
```

Последним шагом освобождаем используемые ресурсы (которых может не быть, если пользователь отменил печать):

```
if(pd.hDevMode)
      GlobalFree(pd.hDevMode);

if(pd.hDevNames)
      GlobalFree(pd.hDevNames);

if (pd.hDC)
      DeleteDC(pd.hDC);
```

Контрольные вопросы

1. Какую функцию выполняет драйвер видеокарты при выполнении в программе команды построения графического примитива, например эллипса?
2. Что такое контекст устройства? Каково его назначение?
3. Можно ли просто изменить байты информации в видеопамяти и сразу увидеть результат на экране монитора в рамках ОС Windows?
4. Последние версии ОС Windows часто используют DirectX с целью вывода информации на экран, например в браузере, а не универсальные функции GDI. С чем это связано?
5. Что такое перо, что такое кисть в терминах GDI?
6. Как в приведенном примере построения круговой диаграммы реализуется эффект тени?
7. Где по умолчанию находится центр координат при выводе информации на экран? Каково направление осей координат? В каких единицах измеряются координаты?

8. Зачем при выборе цвета кисти заливки замкнутой фигуры необходимо сохранять идентификатор предыдущей используемой кисти?
9. Существуют ли отличия в программном выводе информации на экран монитора или на принтер? Почему?
10. Как получить контекст принтера для печати на нем?

3. ОРГАНИЗАЦИЯ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Современные операционные системы с графическим интерфейсом предоставляют приложениям богатый выбор элементов управления, с помощью которых организуется взаимодействие программы с пользователем:

1) меню;
2) диалоговые окна;
3) кнопки;
4) списки;
5) полосы прокрутки;
6) индикаторы;
7) переключатели разных типов и т. п.

Использование стандартных элементов позволяет любой программе выглядеть, как все остальные программы под эту ОС.

На первый взгляд, создание программы с такими элементами управления может показаться сложной задачей. Однако это не так. Напротив, организация интерфейса даже почти не связана с программированием, а ближе к дизайну программного обеспечения. В создании интерфейса с помощью стандартных элементов управления можно выделить два основных момента:

1) разработку внешнего вида программы, заключающуюся в размещении элементов управления в окне программы, диалоговых окнах;
2) написание обработчиков команд элементов управления.

Первый шаг осуществляется с помощью специальных редакторов интерфейса, входящих в среду разработки программ. Разработчик создает меню, диалоговые окна, мышкой размещает элементы управления. Затем интерфейс несколькими командами (функциям *Cu*)

подключается к программе. После этого все элементы присутствуют в программе, однако при нажатии на них не происходит ничего.

Вторым шагом является написание обработчиков событий, которые генерируются при нажатии на соответствующие элементы управления, чтобы они «заработали».

Такой подход называется визуальным программированием, от чего, например, и происходит название Visual Studio, Visual C++, Visual Basic и т. д.

На простом примере рассмотрим подключение меню к программе.

```
// Стандартный включаемый файл Windows
#include <windows.h>

#include «resource.h»

// Прототип функции обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int g_iShape = 0;

// Функция вызывается автоматически, когда программа запускается
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow)
{
   HWND         hWnd;
   MSG          msg;
   WNDCLASSEX   wndclass;

   // Настройка класса окна
   wndclass.cbSize        = sizeof(WNDCLASSEX);
   wndclass.style         = CS_HREDRAW | CS_VREDRAW;
   wndclass.lpfnWndProc   = WndProc;
   wndclass.cbClsExtra    = 0;
   wndclass.cbWndExtra    = 0;
   wndclass.hInstance     = hInstance;
   wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
   wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
   wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   wndclass.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
   wndclass.lpszClassName = “Window Class”; // Имя класса
   wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

   // Регистрация класса окна
   if (RegisterClassEx(&wndclass) == 0)
   {
       // Сбой программы, выход
       return 0;
   }
```

```
    // Создание окна
    hWnd = CreateWindowEx(
       WS_EX_OVERLAPPEDWINDOW,
       «Window Class»,              // Имя класса
       «Приложение Windows»,          // Текст заголовка
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT,
       CW_USEDEFAULT,
       320,
       320,
       NULL,
       NULL,
       hInstance,
       NULL);

    // Отображение окна
    ShowWindow(hWnd, iCmdShow);

    // Обработка сообщений, пока программа не будет прервана
    while(GetMessage(&msg, NULL, 0, 0))
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }

    return (int)msg.wParam;
}

// Функция обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg, WPARAM wParam,
LPARAM lParam)
{
     int            wmId;
     HDC         hDC;
     PAINTSTRUCT ps;

     switch(iMsg)
     {
     // Вызывается, когда пользователь выполняет некую команду,
например из главного меню
     case WM_COMMAND:
          wmId    = LOWORD(wParam);

          switch (wmId)
          {
          case ID_FILE_EXIT:
               SendMessage(hWnd, WM_CLOSE, 0, 0);
               break;
          case ID_SHAPE_SQUARE:
               g_iShape = 0;
               InvalidateRect(hWnd, NULL, TRUE);
               break;
```

```
            case ID_SHAPE_CIRCLE:
                 g_iShape = 1;
                 InvalidateRect(hWnd, NULL, TRUE);
                 break;
            }
            break;
            // Вызывается, когда пользователь отпускает левую
кнопку мыши
      case WM_LBUTTONUP:
            MessageBox(hWnd, TEXT(«Вы кликнули!»), TEXT(«событие»),
MB_OK);
            break;
            // Вызывается, когда окно обновляется
      case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);


             if (g_iShape)
                  Ellipse(hDC, 50, 50, 200, 200);
             else
                  Rectangle(hDC, 50, 50, 200, 200);

             EndPaint(hWnd, &ps);
             break;
             // Вызывается, когда пользователь закрывает окно
       case WM_DESTROY:
             PostQuitMessage(0);
             break;
       default:
             return DefWindowProc(hWnd, iMsg, wParam, lParam);
       }

       return 0;
 }
```

Как видно, отличий от базового примера очень немного:

1) добавлено подключение заголовочного файла ресурсов resource.h;
2) добавлена глобальная переменная, задающая выводимую фигуру;
3) заполнено поле lpszMenuName класса окна, что подключает меню;
4) добавлен обработчик сообщения WM_COMMAND с внутренним переключателем switch для трех разных пунктов меню;
5) немного модифицирован обработчик WM_PAINT.

Помимо файла на языке программирования C++, в exe-файле должен быть слинкован специальный файл ресурсов с расширением **rc (resources cript)**:

```
// Microsoft Visual C++ generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
/////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
/////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
/////
// Russian resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////////
/////
//
// Menu
//

IDR_MENU1 MENU
BEGIN
    POPUP "&Файл"
    BEGIN
        MENUITEM "В&ыход",                      ID_FILE_EXIT
    END
    POPUP "Ф&игура"
    BEGIN
        MENUITEM "&Квадрат",                    ID_SHAPE_SQUARE
        MENUITEM "&Окружность",                 104
    END
```

```
END

#endif    // Russian resources
/////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
```

```
    #endif    // English (U.S.) resources
    ////////////////////////////////////////////////////////////////////////
/////



    #ifndef APSTUDIO_INVOKED
    ////////////////////////////////////////////////////////////////////////
/////
    //
    // Generated from the TEXTINCLUDE 3 resource.
    //


    ////////////////////////////////////////////////////////////////////////
/////
    #endif    // not APSTUDIO_INVOKED
```

Это обычный текстовый файл. Зная синтаксис файлов, можно создать его в редакторе текстов. Однако на практике этот файл автоматически генерируется визуальным редактором. Вторым сгенерированным файлом является resource.h для связи ресурсов с программой на *Cu*:

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by menu.rc
//
#define IDR_MENU1                       101
#define ID_FILE_EXIT                    102
#define ID_SHAPE_SQUARE                 103
#define ID_SHAPE_CIRCLE                 104

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        105
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1001
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Результат работы программы показан на рис. 4. Следует отметить, что программы, обладающие сложным интерфейсом пользователя с большим количеством элементов управления, редко разрабатываются на *Cu* с использованием исключительно функций, предоставляемых операционной системой. Чаще всего для этого используются или специальные библиотеки классов, например, MFC (Microsoft Foundation Classes отсутствует в бесплатной express-версии студии), Qt или специализированные среды быстрой разработки, например, Borland C++ Builder, MATLAB Guide и т. д.

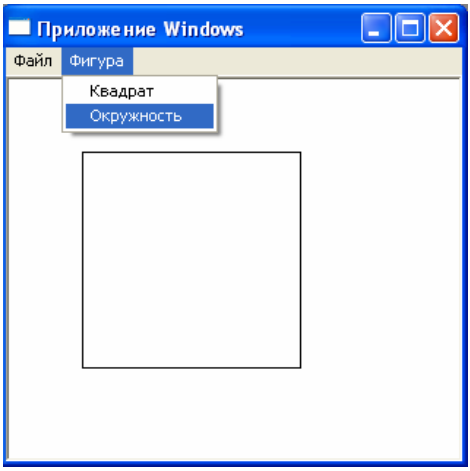

Рис. 4. Подключение главного меню к программе

Контрольные вопросы

1. Что такое визуальное программирование?
2. Что такое файл ресурсов?
3. Зачем нужен символ & в названии пункта меню «В&ыход»?
4. Как подключить главное меню к программе?
5. Какие сообщения генерируются при выборе пользователем пункта меню? Как определить, какой конкретно пункт был выбран?

4. СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

4.1. Сетевые модели, протоколы и архитектура «клиент – сервер»

Сеть – это структура, позволяющая вычислительным машинам взаимодействовать между собой, отправлять и принимать данные и команды. Сети включают компьютеры, среду передачи информации и, как правило, дополнительное сетевое оборудование, делятся на локальные (LAN, Local Area Network) и глобальные (WAN, Wide Area Network). Для того чтобы компьютеры и сетевое оборудование могли понимать друг друга, были разработаны протоколы компьютерных сетей.

Сетевой протокол – набор правил и действий (очередности действий), позволяющих осуществлять соединение и обмен данными между устройствами, включенными в сеть. Это похоже на использование языка: следуя правилам орфографии, люди пишут слова без ошибок и другой человек может их прочитать.

Модель сети – теоретическое описание принципов работы набора сетевых протоколов, взаимодействующих друг с другом. Модель обычно делится на уровни так, чтобы протоколы вышестоящего уровня использовали протоколы нижестоящего (точнее, данные протокола вышестоящего уровня передавались бы с помощью нижележащих протоколов – этот процесс называют инкапсуляцией, процесс извлечения данных вышестоящего уровня из данных нижестоящего – деинкапсуляцией). Модели бывают как практические (использующиеся в сетях, иногда запутанные и/или неполные, но решающие поставленные задачи), так и теоретические (показывающие принципы реализации сетевых моделей, приносящие в жертву наглядности производительность/ возможности).

Наиболее известные сетевые модели:

1) модель OSI – теоретическая, эталонная модель, описанная в международных стандартах;
2) модель DOD (модель TCP/ IP) – практически использующаяся модель, принятая для работы в Интернете;
3) модель AppleTalk – модель работы сетей с оборудованием фирмы Apple;
4) модель SPX/ IPX – модель стека SPX/ IPX (семейство протоколов для локальных вычислительных сетей).

Международная организация по стандартизации (International Organization for Standardization, ISO) приняла в качестве эталонной сетевой модели OSI (Open Systems Interconnection Basic Reference Model – базовая эталонная модель взаимодействия открытых систем, 1978 г.). Модель OSI предлагает взгляд на компьютерную сеть с точки зрения измерений. Каждое измерение обслуживает свою часть процесса взаимодействия. Благодаря такой структуре совместная работа сетевого оборудования и программного обеспечения становится гораздо проще и прозрачнее.

Состоит из семи уровней протоколов. Приложения обращаются с сетью с помощью седьмого уровня, называемого прикладным. Модель OSI заканчивается первым уровнем – физическим, на котором определены стандарты, предъявляемые независимыми производителями к средам передачи данных:

1) тип передающей среды (медный кабель, оптоволокно, радиоэфир и др.);
2) тип модуляции сигнала;
3) сигнальные уровни логических дискретных состояний (нуля и единицы).

Соответственно, информация последовательно передается от прикладного уровня к физическому – при передаче и, наоборот, от физического к прикладному – при приеме. Это можно сравнить с работой почты. Письмо – пакет информации самого высокого уровня, ближе всего к адресату и отправителю, вкладывается, то есть инкапсулируется, в конверт. Конверт с адресом и индексом – более низкий уровень, дальше от пользователей почты. Он упаковывается в контейнер вместе с другими конвертами, которые уйдут в том же направлении.

Затем контейнер помещается в почтовый вагон поезда. Поезд доставляет контейнер в нужный город, там его разгружают, вынимают из него конверт и доставляют по нужному адресу. А получатель может вынуть из конверта само письмо и прочесть. Примерно так же происходит с пакетами информации.

Любой протокол модели OSI должен взаимодействовать либо с протоколами своего уровня, либо с протоколами на единицу выше и/или ниже своего уровня. Взаимодействия с протоколами своего уровня называются горизонтальными, а с уровнями на единицу выше или ниже – вертикальными. Любой протокол модели OSI может выполнять только функции своего уровня и не может выполнять функций другого уровня, что не предусмотрены в протоколах альтернативных моделей.

Уровни модели OSI:

1) физический (physical) – на этом уровне осуществляется работа со средой передачи, сигналами и двоичными данными;
2) канальный (data link) – производится физическая адресация устройств в сети;
3) сетевой (network) – определяются маршрут и логическая адресация устройств;
4) транспортный (transport) – реализуется прямая связь между конечными пунктами и обеспечивается надежность передачи данных;
5) сеансовый (session) – происходит управление сеансом связи;
6) представительский (presentation) – обеспечивается представление и шифрование данных;
7) прикладной (application) – организуется доступ приложения к сетевым службам.

Каждому уровню с некоторой долей условности соответствует свой операнд – логически неделимый элемент данных, которым на отдельном уровне можно оперировать в рамках модели и используемых протоколов: на физическом уровне мельчайшая единица – бит, на канальном уровне информация объединена в кадры, на сетевом – в пакеты (датаграммы), на транспортном – в сегменты. Любой фрагмент данных, логически объединённых для передачи (кадр, пакет, датаграмма), считается сообщением. Именно сообщения в общем виде являются операндами сеансового, представительского и прикладного уровней.

К базовым сетевым технологиям относятся физический и канальный уровни.

Для запоминания названий семи уровней модели OSI на русском языке рекомендуют использовать фразу: «**П**росто **п**редставь **с**ебе **т**ачку, **с**тремящуюся **к ф**инишу», в которой первые буквы слов соответствуют первым буквам названий уровней (от 7-го к 1-му).

Несмотря на то что эталонная модель OSI в настоящее время является общепризнанной, исторически и технически открытым стандартом сети Internet являются протокол управления передачей (Transmission Control Protocol – TCP) и Internet-протокол (IP), которые обычно рассматриваются как одно целое и обозначаются TCP/IP. Эталонная модель TCP/IP, разработанная ещё до принятия модели OSI и вне связи с ней, и стек протоколов TCP/IP позволяют организовать связь между двумя компьютерами, расположенными в любых точках земного шара, со скоростью, близкой к скорости света.

Министерство обороны США (Department of Defence – DoD) создало почву для разработки эталонной модели TCP/IP, поскольку оно требовало, чтобы сеть продолжала функционировать в любых условиях, даже в случае ядерной войны.

Для более наглядной иллюстрации представим себе мир, находящийся в состоянии ядерной войны и пронизанный самыми разными типами соединений, включая проводные, микроволновые соединения, оптоволоконные кабели и спутниковую связь. Далее предположим, что требуется, чтобы информация и данные (в виде пакетов) надежно передавались по этой сети независимо от состояния любого конкретного узла этой сети или другой сети (которая в данном случае может быть уничтожена в ходе военных действий). Министерство обороны требовало, чтобы в любых условиях его данные продолжали передаваться по сети между любыми точками. Эта весьма сложная задача проектирования устойчивой сети привела к созданию сетевой модели TCP/IP (модели DoD), ставшей с тех пор стандартом, на базе которого выросла глобальная сеть Internet. При изучении уровней модели TCP/IP следует помнить о первоначальных целях, которые ставились перед сетью Internet. Это поможет понять некоторые, возможно, неясные аспекты проблемы.

Уровни сетевой модели TCP/IP (модели DoD)

Сетевая модель TCP/IP имеет четыре уровня:

- четвертый – прикладной уровень, или уровень приложений (application);
- третий – транспортный уровень (transport);
- второй – межсетевой, или Internet-уровень (network);
- первый – канальный, или уровень доступа к сети (data link).

Необходимо отметить, что некоторые уровни модели TCP/IP имеют те же названия, что и уровни эталонной модели OSI. Однако не следует отождествлять одноименные уровни этих двух моделей. Функции одноименных уровней обеих моделей могут совпадать, но могут и различаться.

Подобное деление, на наш взгляд, упростило понимание того, как организованы компьютерные сети. Такое упрощение никак не затронуло функциональные возможности этой модели. Все функции, присущие модели OSI, полностью поддерживаются этой моделью. Эта сетевая модель также не зависит от среды передачи данных.

Самый нижний уровень модели DoD называется уровнем сетевого доступа (Network Access). Он соответствует двум уровням модели OSI: физическому и канальному. Уровень сетевого доступа отвечает за доставку данных к физическим сетевым устройствам, таким как сетевые адаптеры, в виде кадров (фреймов, от англ. *frame*). На нем работают такие протоколы, как Ethernet, IEEE 802.11 (известный как Wi-Fi) и 802.16 (WiMAX), а также некоторые другие, не столь привычные для большинства пользователей. Они служат для того же, для чего обыкновенной почте нужны поезда, самолеты и курьеры, – то есть выступают в качестве транспорта.

На следующем, втором уровне модели DoD происходит передача сообщений между сетями, в том числе разнородными по устройству. Этот уровень называется Internet (межсетевой) и сопоставляется с сетевым уровнем модели OSI. Именно здесь используется протокол IP, отвечающий за доставку пакетов информации в современных компьютерных сетях, в том числе в глобальной сети Интернет. Протокол IP не гарантирует доставку всех пакетов данных в той последовательности, в какой они были отправлены. Пакет может потеряться по дороге, про-

дублироваться, или прийти не в свою очередь – протокол IP не имеет контроля над такими ошибками.

Поэтому выше располагаются протоколы транспортного уровня (Transport) модели DoD, который соответствует одноименному уровню OSI. Два самых известных протокола транспортного уровня – TCP и UDP.

Верхний, четвертый уровень DoD охватывает целых три уровня модели OSI: сеансовый, уровень представления и прикладной. Это прикладной уровень, здесь расположены протоколы доставки веб-контента (http, HTTPS), файлов (FTP, BitTorrent), электронной почты (POP3, IMAP), а также удаленного администрирования (SSH, Telnet). Все эти, а также многие другие протоколы инкапсулируются в TCP и UDP. Для определения протокола прикладного уровня, по которому передается конкретный сегмент данных TCP или UDP, каждому протоколу верхнего уровня приписан определенный порт TCP и/или UDP, иногда несколько.

Порт – это номер, который по возможности однозначно сопоставляется с протоколом верхнего уровня. Сопоставлением портов TCP и UDP с определенными протоколами прикладного уровня занимается IANA (Internet Assigned Numbers Authority – Администрация адресного пространства Интернет). Примеры некоторых портов приведены в табл. 4.

Таблица 4

Примеры некоторых портов для протоколов прикладного уровня

Протокол прикладного уровня	Используемые порты
HTTP	80, 8080 TCP
HTTPS	443 TCP
FTP	20 (данные), 21 (команды) TCP
SSH, SFTP	22 TCP, UDP
DNS	53 TCP, UDP
POP3	110 TCP
ICQ	5190 TCP

Каждый сегмент данных TCP или UDP обозначается номерами портов получателя и отправителя – «обратным адресом», на который приходят ответные пакеты. Например, отправляя пакеты данных по http, программы помечают их номером порта получателя 80. Аналогичным образом проставляются в IP-пакетах номера-идентификаторы TCP и UDP для определения того, какой конкретно протокол используется. К открытию или закрытию тех или иных портов TCP и UDP сводится по большей части деятельность файрволлов (брандмауэров) – специальных программ для защиты от ненужных данных в сети. Можно закрывать и открывать порты для определенных IP-адресов своей сети или их диапазонов.

Для попадания пакета информации на нужный компьютер (правильнее – на сетевой интерфейс) служит протокол IP (Internet Protocol). Это протокол сетевого уровня модели OSI и межсетевого – модели DoD. На данный момент больше всего распространена его четвертая версия, однако мир медленно, но верно готовится переходить на более новую – шестую. В четвертой версии протокола адрес сетевого интерфейса (компьютера) имеет вид наподобие 213.192.111.5: четыре числа от 0 до 255, разделенных точками. На самом деле каждое число от 0 до 255 – это 8 бит, то есть IP-адрес состоит из четырех фрагментов по 8 бит (октетов), а всего из 32 бит. Соответственно, пакет IP содержит IP-адреса получателя и отправителя. В пределах сети каждый сетевой интерфейс снабжается уникальным IP-адресом. То есть в Интернете не может быть двух компьютеров с одинаковыми IP-адресами, так же как их не может быть в отдельно взятой локальной сети. При этом в разных локальных сетях могут встречаться сетевые карты с одинаковыми IP-адресами. В Интернете они напрямую не видны. Для передачи пакетов IP между разными сетями, например между локальной и Интернетом, используются маршрутизаторы (они же роутеры).

Для использования в локальных сетях выделено три специальных IP-диапазона: 192.168.0.0–192.168.255.255, 172.16.0.0–172.31.255.255, 10.0.0.0–10.255.255.255. Адреса из этих множеств отличаются тем, что в адресном пространстве глобальной сети Интернет они отсутствуют. Это было сделано для того, чтобы при подключении нескольких машин одной локальной сети через общий шлюз не выделять Интернет-адрес на каждую. Благодаря этому адресов Ipv4 хватало на всех до само-

го недавнего времени. Таким образом, при построении локальной сети адреса ее устройствам назначаются из одного из этих множеств в зависимости от размера локальной вычислительной сети. В малых офисных и домашних сетях используется диапазон 192.168.0.0–192.168.255.255, в районных – адреса от 172.16.0.0 до 172.31.255.255.

В Интернете адреса выдаются согласно решениям IANA. Эта организация упоминалась выше в связи с припиской портов определенным протоколам прикладного уровня, она же выдает IP-адреса крупным региональным регистраторам, которые раздают их более мелким компаниям, а они уже приписывают IP-адреса конкретным ресурсам. Запас свободных Ipv4-адресов подошел к концу, и именно из-за этого постепенно внедряют Ipv6, в котором адреса имеют длину в 128 бит и пишутся обычно восемью группами по четыре шестнадцатеричных цифры (от 0 до F).

Часто при работе в сети обращение к ресурсам происходит по буквенным адресам, а не по IP-адресам. Существует система, которая переводит близкие людям буквосочетания в понятные машинам цифровые коды, то есть URL (Uniform Resource Locator, «единообразный адрес ресурса») в IP-адрес. Она называется DNS, то есть Domain Name System, система доменных имен. Работой службы DNS в глобальном масштабе управляет ICANN (Internet Corporation for Assigned Names and Numbers, Международная корпорация по присвоенным именам и номерам). Тут господствует принцип той же иерархии, что и при раздаче IP-адресов. ICANN создает домен верхнего уровня и дает региональным регистраторам доменов право присваивать имена более низких уровней в этом домене (начиная со второго) определенным IP-адресам. Что же такое домен? Рассмотрим какой-нибудь адрес сетевого ресурса, например edu.tltsu.ru. Это домен третьего уровня. Домен первого уровня –.ru. Домен второго уровня –tltsu.ru. Домен некоего уровня интересен тем, что в нем можно разместить несколько доменов более низких уровней, причем тут фантазия ограничена только практическими соображениями, то есть удобством запоминания интернет-адреса.

Таким образом, чтобы выдавать на каждое зарегистрированное в Интернете доменное имя соответствующий ему IP-адрес, есть служба DNS. По одноименному с ней протоколу компьютер при вводе URL запрашивает специальный прописанный в его настройках сервер с целью

выяснить, к какому IP-адресу обращаться. Сервер DNS ему отвечает, и он может установить соединение. У каждого провайдера есть свой DNS-сервер. Разумеется, он не может держать на своих дисках таблицу соответствия IP-адресам для всех URL в мире. Для этого есть специальные, очень большие DNS-серверы. Они называются корневыми, и их всего 13 во всем мире. Большинство находится в США. Если в КЭШе DNS-сервера провайдера нет нужной записи, он обращается к серверу более высокого ранга с запросом и получает ее. Иногда запрос доходит по цепочке и до корневых серверов.

Итак, для выхода в Интернет компьютеру необходимо знать IP-адрес DNS-сервера. Провайдер может выдавать его динамически по протоколу DHCP, так же как IP-адрес и некоторые другие настройки, а может просто написать в инструкции в явном виде. Тогда его нужно вбить в подходящее поле настройки сети на роутере (интернет-шлюзе) или конкретном компьютере. Чаще всего провайдер дает адреса двух DNS-серверов. Можно легко проверить, все ли в порядке с настройками DNS. Для этого надо обратиться к какому-нибудь ресурсу сначала по имени, а потом по IP-адресу. Например, у yandex.ru IP-адрес такой: 213.180.204.11. И если команда ping.yandex.ru выдает ошибку, а ping 213.180.204.11 без проблем осуществляется, значит, что-то не так со службой DNS. Либо настройки неактуальны (например, провайдер поменял адрес сервера DNS), либо с самим этим сервером проблемы.

Существует концепция взаимодействия по вычислительной сети «клиент – сервер». Она сводится к тому, что клиент отправляет серверу запрос, а сервер на него отвечает. То есть клиент заказывает у сервера некую услугу (отсюда название *server*, то есть «обслуживающий»). Причем этой услугой может являться что угодно, в зависимости от специализации сервера: IP-адрес в обмен на URL у DNS-сервера, страница сайта в обмен на ее адрес у веб-сервера, файл в обмен на его путь у FTP-сервера. Для разработки сетевых программ согласно данной концепции необходимо четко определить, кто запрашивает услугу, а кто предоставляет. И то и другое происходит на уровне программного обеспечения. Веб-страницы у веб-сервера (программа Apache) запрашивает браузер (например, Internet Explorer или Google Chrome), а файлы у FTP-сервера – клиент FTP. Программа, отвечающая на эти запросы с того конца канала, и называется собственно сервером. Однако сервер

не просто класс программного обеспечения. Так как востребованные клиентами услуги надо предоставлять бесперебойно, в режиме «24/7», то сеть должна функционировать постоянно. Для этого компьютеры, на которых работают предоставляющие услуги программы, должны быть более надежными, чем обычные клиентские машины. К тому же к популярным сетевым сервисам вроде поисковых машин или онлайн-хранилищ видео и музыки одновременно обращаются со всех концов Интернета множество клиентов. Следовательно, их вычислительная мощность должна быть достаточно большой. В связи с этим был выделен отдельный класс надежных, мощных и легко масштабируемых (объединяемых друг с другом для увеличения производительности) вычислительных машин. Поскольку они также называются серверами, возможна путаница. Осталось добавить, что серверы (программы) можно устанавливать и на обычные персональные компьютеры.

4.2. Windows Sockets

Winsock, или Windows Sockets, – это интерфейс программирования приложений (API), созданный для реализации программ в сети на основе протоколов TCP/IP.

При взаимодействии «клиент – сервер» в сети каждого участника взаимодействия можно рассматривать как конечную точку, или сокет. Windows Sockets разрабатывался на основе интерфейса Беркли для UNIX (или BSD-сокетов), но к ним добавлены функции поддержки событий Windows. Таким образом, несмотря на то что рассмотрены только Windows Sockets, в целом программирование сокетов в системах UNIX и Windows похоже.

В настоящее время существует две основные версии Winsock API:

1) WinSock 1.1 –поддержка только протоколов TCP/IP;
2) WinSock 2.0 – введена возможность работы с самыми разными сетевыми протоколами и моделями, например SPX/ IPX.

Официальная спецификация Winsock выделяет три типа функций:

1) функции Беркли;
2) информационные функции (получение информации о наименовании доменов, службах, протоколах Интернета);
3) расширения Windows для функций Беркли.

Все функции могут быть блокирующими и неблокирующими. Обычно блокирующие – это функции Беркли. То есть при работе такой функции нельзя выполнять другие функции WinSock.

Код программы, осуществляющей инициализацию интерфейса Winsock API (WSA) и его деинициализацию, следующий:

```
#include <stdio.h>
#include <winsock.h>

const int WINSOCK_VERSION = 0x0101;

void main()
{
     WSADATA wsaData;

     if (WSAStartup(WINSOCK_VERSION, &wsaData))
          printf("Winsock startup FAILED!\n");
     else
          printf("Winsock startup is successful.\n");

     if (WSACleanup())
          printf("Winsock cleanup FAILED!\n");
     else
          printf("Winsock cleanup is successful.\n");

     system("pause");

     return;
}
```

Программа скомпилирована как консольный проект Win32. Для успешной линковки необходимо добавить в список зависимостей приложения файл wsock32.lib, входящий в состав любого современного компилятора C++ для Windows.

С помощью **#include <winsock.h>** подключаются библиотечные функции. Далее объявляется константа с номером версии, с которой будет работать приложение – WINSOCK_VERSION.

Функция **WSAStartup()** инициализирует Winsock. Эта функция всегда вызывается самой первой при начале работы с Winsock. Ее прототип следующий:

```
int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData);
```

Первый параметр – это версия, которая будет использоваться. Младший байт – основная версия, старший байт – расширение вер-

сии. То есть в примере используется версия 1.1. Если инициализация состоялась, то вернется нулевое значение. Инициализация заключается в сопоставлении номера версии и реально существующей библиотеки динамической компоновки (файла с расширением DLL) в системной папке Windows.

Второй параметр – это указатель на структуру WSADATA, в которую возвратятся параметры инициализации. Структура имеет следующее определение:

```
typedef struct WSAData {
        WORD wVersion;
        WORD wHighVersion;
        char szDescription[WSADESCRIPTION_LEN+1];
        char szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short iMaxSockets;
        unsigned short iMaxUdpDg;
        char FAR * lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

WSACleanup() завершает использование данного DLL-файла и прерывает обращение к функциям Winsock. При удачном выполнении вернется ноль. Результат успешной работы программы приведен на рис. 5.

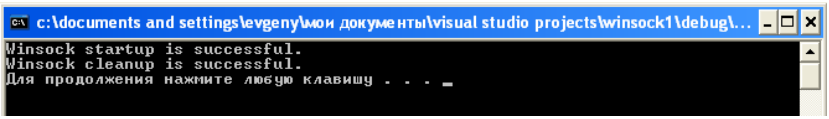

Рис. 5. Инициализация и деинициализация Winsock API

В состав Winsock входит ряд информационных функций, например, для получения сведений об имени компьютера в сети или IP-адреса. Рассмотрим простой пример, выводящий на экран имя локальной машины.

```
#include <stdio.h>
#include <winsock.h>

const int WINSOCK_VERSION = 0x0101;
```

```
void main()
{
     char    szInfo[BUFSIZ];
     WSADATA wsaData;

     if (WSAStartup(WINSOCK_VERSION, &wsaData))
          printf("Winsock startup FAILED!\n");
     else
          printf("Winsock startup is successful.\n");

     if (gethostname(szInfo,sizeof(szInfo)))
          printf("Local host name has not been configured\n");
     else
          printf("Host name is %s.\n", szInfo);

     if (WSACleanup())
          printf("Winsock cleanup FAILED!\n");
     else
          printf("Winsock cleanup is successful.\n");

     system(«pause»);

     return;
}
```

Отличие этого примера от предыдущего заключается в использовании информационной функции gethostname(), имеющей следующий прототип:

```
int gethostname (
       char FAR * name,
       int namelen
);
```

В эту функцию передается буфер и его длина для возврата имени. При отсутствии ошибок эта функция вернет 0. Результат штатной работы программы приведен на рис. 6.

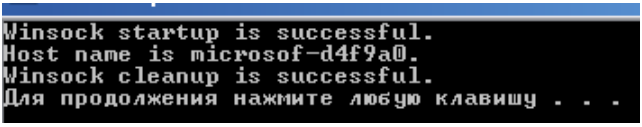

Рис. 6. Работа функции gethostname()

Рассмотрим процедуру создания и закрытия сокетов, изменив предыдущий пример:

```
#include <stdio.h>
#include <winsock.h>

const int WINSOCK_VERSION = 0x0101;

void main()
{
     SOCKET  serverSocket;
     Char    szInfo[BUFSIZ];
     WSADATA wsaData;

     if (WSAStartup(WINSOCK_VERSION, &wsaData))
          printf("Winsock startup FAILED!\n");
     else
          printf("Winsock startup is successful.\n");

     if (gethostname(szInfo,sizeof(szInfo)))
          printf("Local host name has not been configured\n");
     else
          printf("Host name is %s.\n", szInfo);

     serverSocket = socket(PF_INET, SOCK_STREAM, 0);

     if (serverSocket == INVALID_SOCKET)
          printf("Socket creation FAILED!\n");

     closesocket(serverSocket);

     if (WSACleanup())
          printf("Winsock cleanup FAILED!\n");
     else
          printf("Winsock cleanup is successful.\n");

     system(«pause»);

     return;
}
```

Добавлена следующая переменная:

```
SOCKET serverSocket;
```

Тип SOCKET определяется в файле winsock.h следующим образом:

```
/*
 * The new type to be used in all
 * instances which refer to sockets.
 */

typedef UINT_PTR        SOCKET;
```

UINT_PTR в файле basetsd.h определяется как

```
typedef _W64 unsigned int UINT_PTR, *PUINT_PTR;
```

То есть, по сути, является замаскированным целым числом без знака типа unsigned int и содержит идентификатор сокета в системе.

За создание сокета отвечает следующая строчка:

```
serverSocket = socket(PF_INET, SOCK_STREAM, 0);
```

Функция socket() создает сокет заданного типа и возвращает его идентификатор в случае успеха или константу INVALID_SOCKET (также определена в файле winsock.h) – в противном случае. Все три аргумента функции являются целочисленными, задают тип сокета указанием нужных констант, список которых можно найти в заголовочном файле, в документации среды разработки или в RFC 790 от сентября 1981 года. Первый аргумент – тип сетевой модели, PF_INET соответствует сетевой модели Интернета (TCP/IP), PF_OSI – модели OSI, PF_APPLETALK – модели AppleTalk и т. д. Однако указание, например, PF_OSI в качестве сетевой модели приведет к ошибке создания сокета, поскольку в рамках Windows данная модель не поддерживается, а аргумент функции и константы введены для возможности расширения интерфейса и, возможно, использования в будущих операционных системах. Дальше указывается тип потока: SOCK_STREAM используется для TCP, а SOCK_DGRAM – для UDP. Третий аргумент – собственно протокол, где 0 по умолчанию соответствует протоколу IP.

После работы сокет необходимо закрыть:

```
closesocket(serverSocket);
```

Единственным параметром функции является сокет, который требуется закрыть.

Попробуем реализовать простой сервер, работающий по протоколу http и использующий, соответственно, порт 80.

Рассмотрим, как производится подключение к серверу. Сначала программа подключается к адресу IP с созданием сокета. Программа будет ждать подключения. Для подключения программы клиент тоже создает сокет и пытается подключиться к сокету сервера. Как только сервер зарегистрирует попытку подключения, он создаст новый сокет. И этот новый сокет будет использоваться для взаимодействия с клиентом. А тот сокет, к которому была попытка подключения, будет ждать следующего. Сокет может быть создан на основе TCP или UDP. На этой основе производится взаимодействие сервера со многими программами.

Для организации связи серверу необходимо вызвать функцию bind() для действительного сокета и связать его с номером порта, который будет «прослушиваться» для ожидания подключения. Функция требует заполненную структуру SOCKADDR_IN с такими параметрами связи, как порт и атрибуты. Ее прототип:

```
struct sockaddr_in{
        short sin_family;
        unsigned short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];};
```

В данной структуре присутствует вложенная структура sin_addr. Она описана следующим образом:

```
struct in_addr {
        union {
                struct{
                unsigned char
                        s_b1,
                        s_b2,
                        s_b3,
                        s_b4;
                } S_un_b;
                struct{
                unsigned short
                        s_w1,
                        s_w2;
                } S_un_w;
```

```
            unsigned long S_addr;
        } S_un;
};
```

Смысл полей достаточно прозрачен: тип сетевого адреса (соответствующий разным сетевым моделям), адрес, порт.

После описания структур и заполнения данными можно вызывать bind() с проверкой результата на ошибку:

```
SOCKADDR_IN socketaddr;
socketaddr.sin_family = AF_INET;
socketaddr.sin_addr.s_addr= INADDR_ANY;
socketaddr.sin_port = htons(80);

if (bind(serverSocket,(LPSOCKADDR)&socketaddr,sizeof(socketaddr)) ==
SOCKET_ERROR)
printf("Socket binding FAILED!");
else
printf("Socked binding is successful.\n");
```

Ее прототип:

```
int bind (
        SOCKET s,
        const struct sockaddr FAR*name,
        int namelen
);
```

Если всё нормально, то данная функция вернет 0, в противном случае – SOCKET_ERROR. Как видно, адрес не указывается (INADDR_ANY – это константа нуля), поскольку создается сокет для сервера. Полный текст программы выглядит следующим образом:

```
#include <stdio.h>
#include <winsock.h>

const int WINSOCK_VERSION = 0x0101;

void main()
{
     SOCKET          serverSocket;
     SOCKADDR_IN     socketaddr;
     char            szInfo[BUFSIZ];
     WSADATA         wsaData;
```

```
    if (WSAStartup(WINSOCK_VERSION, &wsaData))
        printf("Winsock startup FAILED!\n");
    else
        printf("Winsock startup is successful.\n");

    if (gethostname(szInfo,sizeof(szInfo)))
        printf("Local host name has not been configured\n");
    else
        printf("Host name is %s.\n", szInfo);

    serverSocket = socket(PF_INET, SOCK_STREAM, 0);

    if (serverSocket == INVALID_SOCKET)
        printf("Socket creation FAILED!\n");

    socketaddr.sin_family         =    AF_INET;
    socketaddr.sin_addr.s_addr    =    INADDR_ANY;
    socketaddr.sin_port           =    htons(80);

    if (bind(serverSocket, (LPSOCKADDR)&socketaddr,
sizeof(socketaddr)) == SOCKET_ERROR)
        printf("Socket binding FAILED!\n");
    else
        printf("Socked binding is successful.\n");

    closesocket(serverSocket);

    if (WSACleanup())
        printf("Winsock cleanup FAILED!\n");
    else
        printf("Winsock cleanup is successful.\n");

    system(«pause»);

    return;
}
```

В компьютерах архитектуры Intel 80×86 и совместимых с ними слово «двухбайтовое число» хранится следующим образом:

младший байт – n;

старший байт – n+1,

где n – номер ячейки в памяти.

К сожалению, в Интернете наоборот:

младший байт – n+1;

старший байт – n.

Для решения этой проблемы WinSock API предоставляет следующие функции:

1) htohl – преобразует 32-битные локальные числа к сетевым, сортируя байты;
2) htohs –преобразует 16-битные локальные числа к сетевым, сортируя байты;
3) ntonl –преобразует 32-битные сетевые числа к локальным, сортируя байты;
4) ntons –преобразует 16-битные сетевые числа к локальным, сортируя байты.

На данном этапе программа почти готова принимать сообщения по сети. Чтобы узнать об установлении соединения клиента с сервером, используются события и сообщения ОС Windows. Как известно, обработчиком сообщений является окно, а следовательно, необходимо связать сокет с дескриптором окна. Для этого служит функция **WSAAsyncSelect()**. Как следует из приставки WSA, она относится к расширениям Windows для функций Беркли, то есть специфична для этой ОС.

```
     Int error;
     error = WSAAsyncSelect(serverSocket, hWnd, WM_SERVER_ACCEPT,
FD_ACCEPT);

     if (error == SOCKET_ERROR)
          printf("WSAAsyncSelect() FAILED!\n");
```

Прототип функции имеет следующий вид:

```
int WSAAsyncSelect (
        SOCKET s,
        HWND hWnd,
        unsigned int wMsg,
        long lEvent
);
```

Первым аргументом передается сокет, вторым – дескриптор окна, третьим – сообщение, которое будет послано, четвертым – событие, при котором генерируется сообщение. FD_ACCEPT означает, что сообщение сгенерируется при запросе от клиента.

Здесь интересен параметр unsigned int wMsg, который говорит о том, какое сообщение будет послано в случае подключения к серверу. Сообщения WM_SERVER_ACCEPT в Windows не существует, это пользовательское сообщение, его можно описать следующим образом:

```
const int WM_SERVER_ACCEPT = WM_USER + 1;
```

Включается «прослушивание» функцией listen().

```
Error = listen(serverSocket, 10);

if (error == SOCKET_ERROR)
      printf(«listen() FAILED!\n»);
```

Прототип функции listen() выглядит следующим образом:

```
int listen (
      SOCKET s,
      int backlog
);
```

Первым аргументом является сокет, вторым – максимальное количество подключений.

К сожалению, просто добавить в текст последней программы функцию WSAAsyncSelect не получится. Первая проблема очевидна – это отсутствие дескриптора окна у консольного приложения, вторая – отсутствие обработки поступающих сообщений. И если первую еще можно решить, например, поиском дескриптора окна по его заголовку функцией FindWindow(), то вторая приводит к необходимости создания оконной процедуры и полноценного Windows-приложения.

Ниже приведен текст Winsock-программы, выступающей в роли сервера HTTP:

```
// Стандартный включаемый файл Windows
#include <windows.h>
#include <winsock.h>

const int WINSOCK_VERSION     =     0x0101;
const int WM_SERVER_ACCEPT    =     WM_USER + 1;

// Прототип функции обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

bool Start(HWND);
void Stop();
```

```
char      g_szStatus[512];
SOCKET    g_serverSocket;

// Функция вызывается автоматически, когда программа запускается
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow)
{
     HWND         hWnd;
     MSG          msg;
     WNDCLASSEX   wndclass;

     // Настройка класса окна
     wndclass.cbSize         = sizeof(WNDCLASSEX);
     wndclass.style          = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc    = WndProc;
     wndclass.cbClsExtra     = 0;
     wndclass.cbWndExtra     = 0;
     wndclass.hInstance      = hInstance;
     wndclass.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor        = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =
(HBRUSH)GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName   = NULL;
     wndclass.lpszClassName = "Window Class"; // Имя класса
     wndclass.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);

     // Регистрация класса окна
     if(RegisterClassEx(&wndclass) == 0)
     {
          // Сбой программы, выход
          return 0;
     }

     // Создание окна
     hWnd = CreateWindowEx(
          WS_EX_OVERLAPPEDWINDOW,
          «Window Class»,             // Имя класса
          «Сервер»,                   // Текст заголовка
          WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
WS_MINIMIZEBOX,

          CW_USEDEFAULT,
          CW_USEDEFAULT,
          600,
          100,
          NULL,
          NULL,
          hInstance,
          NULL);

     // Отображение окна
     ShowWindow(hWnd, iCmdShow);
```

```
     // Обработка сообщений, пока программа не будет прервана
     while(GetMessage(&msg, NULL, 0, 0))
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     return (int)msg.wParam;
}

// Функция обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg, WPARAM wParam,
LPARAM lParam)
{
     HDC         hDC;
     PAINTSTRUCT ps;

     switch(iMsg)
     {
          // Вызывается сразу же при создании окна функцией
CreateWindow() (или CreateWindowEx())
     case WM_CREATE:
          strcpy(g_szStatus, «Щелкните левой кнопкой мыши для
запуска сервера, правой для его остановки»);
          // g_serverSocket инициализируется значением
недействительного сокета
          g_serverSocket = INVALID_SOCKET;
          break;
          // Вызывается, когда пользователь отпускает левую
кнопку мыши
     case WM_LBUTTONUP:
          // Запуск сервера
          Start(hWnd);
          // Перерисовка окна (генерация сообщения WM_PAINT)
          InvalidateRect(hWnd, NULL, TRUE);
          break;
          // Вызывается, когда пользователь отпускает правую
кнопку мыши
     case WM_RBUTTONUP:
          // Остановка сервера
          Stop();

          InvalidateRect(hWnd, NULL, TRUE);
          break;
      case WM_SERVER_ACCEPT:
          strcpy(g_szStatus, “Клиент подключился”);
          InvalidateRect(hWnd, NULL, TRUE);
          break;
          // Вызывается, когда окно обновляется
      case WM_PAINT:
          hDC = BeginPaint(hWnd, &ps);
```

```
            TextOut(hDC, 20, 20, g_szStatus,
(int)strlen(g_szStatus));
            EndPaint(hWnd, &ps);
            break;
            // Вызывается, когда пользователь закрывает окно
      case WM_DESTROY:
            // Остановка сервера при закрытии окна
            Stop();
            PostQuitMessage(0);
            break;
      default:
            return DefWindowProc(hWnd, iMsg, wParam, lParam);
      }

      return 0;
}


bool Start(HWND hWnd)
{
      int            error;
      SOCKADDR_IN    socketaddr;
      WSADATA        wsaData;

      if (g_serverSocket != INVALID_SOCKET)
      {
            strcpy(g_szStatus, "Сервер уже запущен");
            return false;
      }

      if (WSAStartup(WINSOCK_VERSION, &wsaData))
      {
            strcpy(g_szStatus, "Ошибка инициализации Winsock");
            return false;
      }

      g_serverSocket = socket(PF_INET, SOCK_STREAM, 0);

      if (g_serverSocket == INVALID_SOCKET)
      {
            strcpy(g_szStatus, «Ошибка создания сокета»);
            return false;
      }
      socketaddr.sin_family         =    AF_INET;
      socketaddr.sin_addr.s_addr    =    INADDR_ANY;
      socketaddr.sin_port           =    htons(80);

      error = bind(g_serverSocket, (LPSOCKADDR)&socketaddr,
sizeof(socketaddr));
```

```
     if (error == SOCKET_ERROR)
     {
          strcpy(g_szStatus, "Ошибка связи сокета с портом");
          return false;
     }

     error = WSAAsyncSelect(g_serverSocket, hWnd,
WM_SERVER_ACCEPT, FD_ACCEPT);

     if (error == SOCKET_ERROR)
     {
          strcpy(g_szStatus, "Ошибка связи сокета с окном");
          return false;
     }

     error = listen(g_serverSocket, 10);

     if (error == SOCKET_ERROR)
     {
          strcpy(g_szStatus, "Ошибка прослушивания сокета");
          return false;
     }

     strcpy(g_szStatus, "Сервер успешно запущен");

     return true;
}

void Stop()
{
     if (g_serverSocket == INVALID_SOCKET)
     {
          strcpy(g_szStatus, «Сервер не запущен или уже
остановлен»);
          return;
     }

     closesocket(g_serverSocket);

     g_serverSocket = INVALID_SOCKET;

     if (WSACleanup())
     {
          strcpy(g_szStatus, "Ошибка освобождения Winsock");
          return;
     }
     strcpy(g_szStatus, "Сервер успешно остановлен");

     return;
}
```

Итак, запуск сервера будет происходить по нажатию левой кнопки мыши, остановка – правой. В текст простейшего приложения Windows внесены следующие изменения:

1) добавлено подключение заголовочного файла winsock.h;
2) определены константы для номера используемой версии winsock и пользовательского сообщения, которое будет генерироваться при поступлении запроса с клиентской программы;
3) введены функции Start() и Stop() для запуска и остановки сервера;
4) добавлены две глобальные переменные **`g_szStatus`** *и* **`g_serverSocket`** для хранения выводимой на экран текстовой информации о состоянии сервера и сокет; запись и чтение в этих переменных осуществляется как в оконной процедуре, так и в Start() и Stop();
5) в функцию CreateWindow внесены косметические изменения: названия программы, стиль окна без изменения размера, произвольное начальное положение, описываемое константой CW_USEDEFAULT, размеры окна, опытным путем подобранные под выводимый текст состояния;
6) новая оконная процедура теперь обрабатывает сообщения о создании окна, отпускании левой кнопки мыши, отпускании правой кнопки мыши, поступлении запроса клиента, уничтожении окна.

Результат запуска приложения показан на рис. 7, 8.

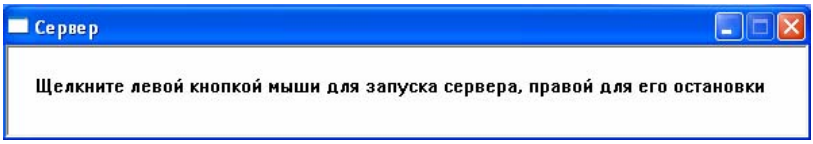

Рис. 7. Запуск серверной программы

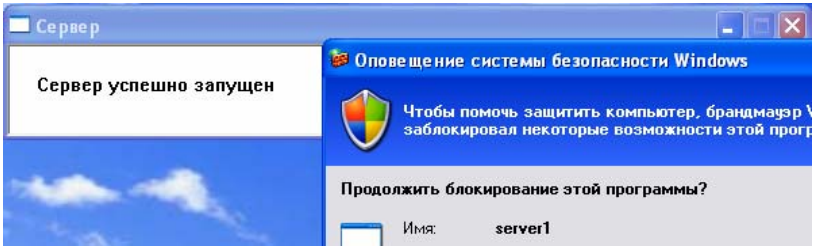

Рис. 8. Запуск сервера по щелчку левой кнопки мыши

При запуске сервера может появиться оповещение файрволла о сетевой активности приложения. Как подключиться к серверу? Для этого необходима другая программа, называемая клиентом. Поскольку при связи сокета с портом указан порт 80, используемый для HTTP-протокола, можно воспользоваться любым интернет-браузером: Internet Explorer, Firefox, Opera, Google Chromeи т. д. Чтобы узнать IP-адрес сервера в сети, можно воспользоваться командой ipconfig. Для этого необходимо запустить программу «Командная строка» (Пуск – Все программы – Стандартные – Командная строка или Пуск – Выполнить... – набрать cmd и нажать enter), набрать ipconfig и нажать enter. Появится информация, подобная показанной на рис. 9.

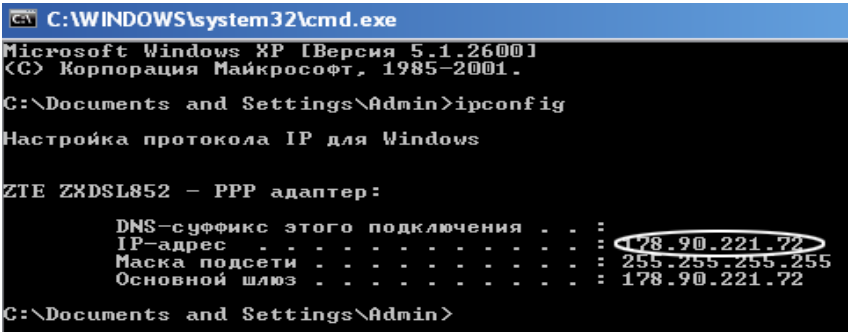

Рис. 9. Команда ipconfig

Соответственно, зная IP-адрес компьютера, на котором запущена приведенная программа, с помощью браузера можно к ней подключиться с любого компьютера сети. Другим вариантом является использование уникального IP-адреса 127.0.0.1, что соответствует адресу локальной машины. То есть можно запустить сервер и подключиться к нему с этого же компьютера, используя приведенный IP-адрес или URL **localhost**, соответствующий такому IP-адресу. Это очень удобно при отладке сетевых программ на одном компьютере без подключения к локальной или глобальной сети.

Результат подключения клиента показан на рис. 10.

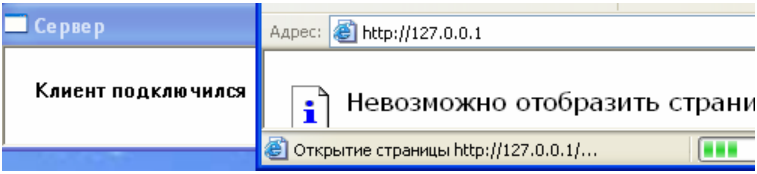

Рис. 10. Подключение клиента к серверу

Сервер успешно среагировал на подключение клиента обработкой введенного сообщения WM_SERVER_ACCEPT. Однако для полноценной поддержки взаимодействия «клиент – сервер» необходимо, чтобы сервер «ответил» клиенту. Пока этого не происходит, что видно по состоянию браузера, который подключился к серверу, но «завис» на этапе открытия страницы, ожидая ответа.

Окончательный текст демонстрационной программы-сервера следующий.

```
// Стандартный включаемый файл Windows
#include <windows.h>
#include <winsock.h>

const int WINSOCK_VERSION     =     0x0101;
const int WM_SERVER_ACCEPT    =     WM_USER + 1;

// Прототип функции обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void OnServerAccept(WPARAM, LPARAM);
bool Start(HWND);
void Stop();

char           g_szStatus[512];
SOCKET         g_serverSocket;

// Функция вызывается автоматически, когда программа запускается
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
PSTR szCmdLine, int iCmdShow)
{
     HWND          hWnd;
     MSG           msg;
     WNDCLASSEX    wndclass;

     // Настройка класса окна
     wndclass.cbSize         = sizeof(WNDCLASSEX);
     wndclass.style          = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc    = WndProc;
     wndclass.cbClsExtra     = 0;
```

```
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =
(HBRUSH)GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = "Window Class"; // Имя класса
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     // Регистрация класса окна
     if(RegisterClassEx(&wndclass) == 0)
     {
          // Сбой программы, выход
          return 0;
     }

     // Создание окна
     hWnd = CreateWindowEx(
          WS_EX_OVERLAPPEDWINDOW,
          «Window Class»,            // Имя класса
          «Сервер»,                  // Текст заголовка

           WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
WS_MINIMIZEBOX,
           CW_USEDEFAULT,
           CW_USEDEFAULT,
           600,
           100,
           NULL,
           NULL,
           hInstance,
           NULL);

     // Отображение окна
     ShowWindow(hWnd, iCmdShow);

     // Обработка сообщений, пока программа не будет прервана
     while(GetMessage(&msg, NULL, 0, 0))
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     return (int)msg.wParam;
}

// Функция обратного вызова для обработки сообщений
LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg, WPARAM wParam,
LPARAM lParam)
{
```

```
    HDC         hDC;
    PAINTSTRUCT ps;

    switch(iMsg)
    {
        // Вызывается сразу же при создании окна функцией
CreateWindow() (или CreateWindowEx())
    case WM_CREATE:
        strcpy(g_szStatus, «Щелкните левой кнопкой мыши для
запуска сервера, правой для его остановки»);
        // g_serverSocket инициализируется значением
недействительного сокета
        g_serverSocket = INVALID_SOCKET;
        break;
        // Вызывается, когда пользователь отпускает левую
кнопку мыши
    case WM_LBUTTONUP:
        // Запуск сервера
        Start(hWnd);
        // Перерисовка окна (генерация сообщения WM_PAINT)
        InvalidateRect(hWnd, NULL, TRUE);
        break;
        // Вызывается, когда пользователь отпускает правую
кнопку мыши
    case WM_RBUTTONUP:

        // Остановка сервера
        Stop();
        InvalidateRect(hWnd, NULL, TRUE);
        break;
    case WM_SERVER_ACCEPT:
        OnServerAccept(wParam, lParam);
        InvalidateRect(hWnd, NULL, TRUE);
        break;
        // Вызывается, когда окно обновляется
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        TextOut(hDC, 20, 20, g_szStatus,
(int)strlen(g_szStatus));
        EndPaint(hWnd, &ps);
        break;
        // Вызывается, когда пользователь закрывает окно
    case WM_DESTROY:
        // Остановка сервера при закрытии окна
        Stop();
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, iMsg, wParam, lParam);
    }

    return 0;
```

```
}

void OnServerAccept(WPARAM wParam, LPARAM lParam)
{
     int            error, length;
     SOCKET         clientSocket;
     SOCKADDR  socketclientaddr;

     static char html[] =
          "HTTP/1.0 200 OK\r\nContent-length:87\r\n\r\n"
          "<html><body><b><span style='font-size:72.0pt'>Сервер
работает!</span></b></body></html>";

     if (WSAGETASYNCERROR(lParam))
     {
          strcpy(g_szStatus, "Ошибка определения клиента");
          return;
     }

     if (WSAGETSELECTEVENT(lParam) == FD_ACCEPT)
     {
          length = sizeof(SOCKADDR);

          clientSocket = accept(g_serverSocket,
(LPSOCKADDR)&socketclientaddr, &length);

          if (clientSocket == INVALID_SOCKET)
          {
               strcpy(g_szStatus, "Ошибка создания сокета
клиента");
               return;
          }
     }
     else if (WSAGETSELECTEVENT(lParam) == FD_READ)
     {
          clientSocket = (SOCKET)wParam;

          ZeroMemory(g_szStatus, sizeof(g_szStatus));

          error = recv(clientSocket, g_szStatus,
sizeof(g_szStatus), 0);

          if (error == SOCKET_ERROR)
          {
               strcpy(g_szStatus, "Ошибка в получении запроса
клиента");
               return;
          }

          error = send(clientSocket, html, (int)strlen(html), 0);
```

```
          if (error == SOCKET_ERROR)
          {
               strcpy(g_szStatus, "Ошибка в ответе клиенту");
               return;
          }

          closesocket(clientSocket);
     }

     strcpy(g_szStatus, "Клиент подключился");

     return;
}

bool Start(HWND hWnd)
{
     int            error;
     SOCKADDR_IN    socketaddr;
     WSADATA        wsaData;

     if (g_serverSocket != INVALID_SOCKET)
     {
          strcpy(g_szStatus, "Сервер уже запущен");
          return false;
     }

     if (WSAStartup(WINSOCK_VERSION, &wsaData))
     {
          strcpy(g_szStatus, "Ошибка инициализации Winsock");

          return false;
     }

     g_serverSocket = socket(PF_INET, SOCK_STREAM, 0);

     if (g_serverSocket == INVALID_SOCKET)
     {
          strcpy(g_szStatus, «Ошибка создания сокета»);
          return false;
     }

     socketaddr.sin_family          =    AF_INET;
     socketaddr.sin_addr.s_addr     =    INADDR_ANY;
     socketaddr.sin_port            =    htons(80);

     error = bind(g_serverSocket, (LPSOCKADDR)&socketaddr,
sizeof(socketaddr));

     if (error == SOCKET_ERROR)
     {
          wsprintf(g_szStatus, "Ошибка связи сокета с портом:
%d", WSAGetLastError());
```

```
            return false;
        }

        error = WSAAsyncSelect(g_serverSocket, hWnd,
WM_SERVER_ACCEPT, FD_ACCEPT | FD_READ);

        if (error == SOCKET_ERROR)
        {
            strcpy(g_szStatus, "Ошибка связи сокета с окном");
            return false;
        }

        error = listen(g_serverSocket, 10);

        if (error == SOCKET_ERROR)
        {
            strcpy(g_szStatus, "Ошибка прослушивания сокета");
            return false;
        }

        strcpy(g_szStatus, "Сервер успешно запущен");

        return true;
}

void Stop()
{
        if (g_serverSocket == INVALID_SOCKET)
        {
            strcpy(g_szStatus, «Сервер не запущен или уже
остановлен»);

             return;
        }

        closesocket(g_serverSocket);

        g_serverSocket = INVALID_SOCKET;

        if (WSACleanup())
        {
             strcpy(g_szStatus, "Ошибка освобождения Winsock");
             return;
        }

        strcpy(g_szStatus, "Сервер успешно остановлен");

        return;
 }
```

Изменения следующие: код обработки сообщения WM_SERVER_ACCEPT вынесен в отдельную функцию OnServerAccept() для упрощения оконной процедуры.

В вызове функции WSAAsyncSelect флаг заменен на **FD_ACCEPT | FD_READ**, что означает генерацию сообщения при подключении клиента и передаче клиентом данных. Дело в том, что послать клиенту данные можно сразу после установки соединения функцией accept(), однако в случае использования web-браузера это не получится. Web-браузер после установки соединения посылает серверу запрос, например, с именем страницы и требует, чтобы сервер принял этот запрос. Без этого обратная передача данных не будет осуществляться. Следовательно, в нашем случае требуется не только установить соединение, но и сначала подождать запрос от клиента, то есть сообщения FD_READ. Для произвольной передачи данных с целью тестирования сервера удобно воспользоваться программой telnet в качестве клиентской. Для ее запуска достаточно набрать в командной строке «telnet адрес порт» через пробел и нажать enter.

Приведенную функцию обработки запросов клиентов можно мысленно разделить на две части: установку соединения, обработку запроса клиента. Текст первой части следующий:

```
if (WSAGETSELECTEVENT(lParam) == FD_ACCEPT)
     {
          length = sizeof(SOCKADDR);

          clientSocket = accept(g_serverSocket,
(LPSOCKADDR)&socketclientaddr, &length);

          if (clientSocket == INVALID_SOCKET)
          {
               strcpy(g_szStatus, "Ошибка создания сокета
клиента");
               return;
          }
     }
```

То есть, если произошла попытка установки соединения, то вызывается функция accept(), которая его устанавливает. Имеет три аргумента: сокет сервера, необязательный (можно задать NULL) указатель на структуру, куда запишется IP-адрес клиента и указатель на ячейку, где хранится размер этой структуры, тоже необязательный. Из-за существования разных версий Winsock требуется явное преобразование

второго аргумента. Accept() автоматически создает новый сокет, выполняет связывание и возвращает его дескриптор. Если в момент вызова accept() очередь пуста, функция не возвращает управление до тех пор, пока с сервером не будет установлено хотя бы одно соединение. Следовательно, сервер можно создать и в консольном режиме без использования сообщений Windows, путем постоянного вызова accept() в бесконечном цикле. В случае возникновения ошибки функция возвращает отрицательное значение.

Текст второй части, отвечающей за ответ клиенту, следующий.

```
else if (WSAGETSELECTEVENT(lParam) == FD_READ)
     {
          clientSocket = (SOCKET)wParam;

          ZeroMemory(g_szStatus, sizeof(g_szStatus));

          error = recv(clientSocket, g_szStatus,
sizeof(g_szStatus), 0);

          if (error == SOCKET_ERROR)
          {
               strcpy(g_szStatus, "Ошибка в получении запроса
клиента");
               return;
          }

          error = send(clientSocket, html, (int)strlen(html), 0);

          if (error == SOCKET_ERROR)
          {
               strcpy(g_szStatus, "Ошибка в ответе клиенту");
               return;
          }

          closesocket(clientSocket);
     }
```

Дескриптор сокета клиента, запросившего данные, содержится в переменной wParam. Использовать дескриптор, полученный ранее от функций accept(), не рекомендуется, так как одновременно могут поступать запросы с разных клиентов.

После того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными, вызывая функции «*int* ***send*** *(SOCKET s, const char FAR* buf, int len, int flags)*» и «*int* ***recv*** *(SOCKET s, char FAR* buf, int len, int flags)*» для посылки и приема данных соответственно.

Функция же recv() (**rec**ei**v**e, получить) возвращает управление после того, как получит датаграмму. Датаграмма – это совокупность одного или нескольких IP-пакетов, посланных вызовом send(). Первый аргумент – сокет клиента, второй – буфер, в который записываются получаемые данные, третий – размер этого буфера, четвертый – дополнительные флаги настроек. Подразумевается, что функции recv() предоставлен буфер достаточных размеров, – в противном случае ее придется вызвать несколько раз. Однако при всех последующих обращениях данные будут браться из локального буфера сетевого интерфейса, а не приниматься из сети, так как TCP-провайдер не может получить «кусочек» датаграммы, а только её всю целиком.

Функция send() возвращает управление сразу же после ее выполнения независимо от того, получила ли принимающая сторона наши данные или нет. Аргументы функции аналогичны аргументам функции recv(). При успешном завершении функция возвращает количество передаваемых (не переданных) данных, т. е. успешное завершение еще не свидетельствует об успешной доставке! Протокол TCP гарантирует успешную доставку данных получателю, но лишь при условии, что соединение не будет преждевременно разорвано. Если связь прервется до окончания пересылки, данные останутся не переданными, но вызывающий код не получит об этом никакого уведомления. Ошибка возвращается лишь в случае, если соединение разорвано до вызова функции send().

Результат работы сервера показан на рис. 11.

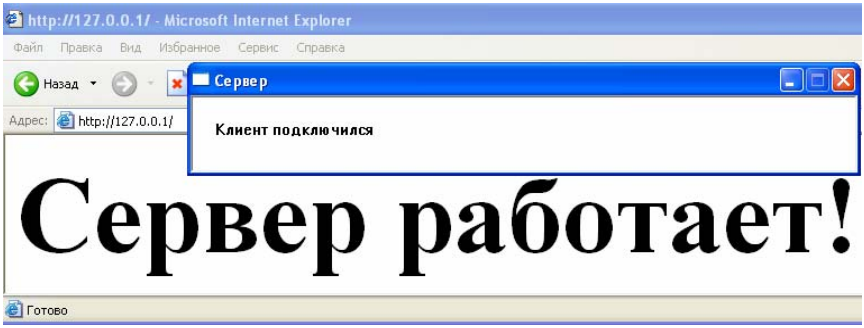

Рис. 11. Работа сервера по протоколу HTTP

Реализация клиентских программ часто гораздо проще. В этом случае вызов функции bind() заменяется на connect() с указанием адреса сервера, функцией send() посылается запрос, далее ожидается ответ –recv().

```
#include <stdio.h>
#include <winsock.h>

const int WINSOCK_VERSION = 0x0101;

void main()
{
     char      buffer[512];
     SOCKET         clientSocket;
     SOCKADDR_IN    socketaddr;
     WSADATA        wsaData;

     static char request[] = "GET / \r\nHTTP 1.0 \r\n\r\n";

     printf("Enter IP - address: ");
     scanf("%s", buffer);

     if (WSAStartup(WINSOCK_VERSION, &wsaData))
          printf("Winsock startup FAILED!\n");
     else
          printf("Winsock startup is successful.\n");

     clientSocket = socket(PF_INET, SOCK_STREAM, 0);

     if (clientSocket == INVALID_SOCKET)
          printf("Socket creation FAILED!\n");

     socketaddr.sin_family         =    AF_INET;
     socketaddr.sin_addr.s_addr    =    inet_addr(buffer);
     socketaddr.sin_port           =    htons(80);

     if (connect(clientSocket, (LPSOCKADDR)&socketaddr,
sizeof(socketaddr)) == SOCKET_ERROR)
          printf("Socket connecting FAILED!\n");
     else
          printf("Socked connecting is successful.\n");

     send(clientSocket, request, (int)strlen(request), 0);

     printf("\n");

     while (1)
     {
          ZeroMemory(buffer, sizeof(buffer));
```

```
            if (recv(clientSocket, buffer, sizeof(buffer) - 1, 0)
<= 0)
                 break;

            printf("%s", buffer);
      }

      printf("\n\n");


       closesocket(clientSocket);

       if (WSACleanup())
            printf("Winsock cleanup FAILED!\n");
       else
            printf("Winsock cleanup is successful.\n");

       system("pause");

       return;
 }
```

В строчке

```
socketaddr.sin_addr.s_addr =      inet_addr(buffer);
```

функцией inet_addr() преобразовывается адрес в текстовом виде во внутреннее представление.

Аргументы функции connect() аналогичны аргументам функции bind(): сокет, указатель на структуру с адресом и портом, размер структуры.

Следует заметить, что функция recv() – блокирующая. Это означает, что управление программе не вернется до получения данных (аналогично accept()) и сложные клиентские программы следует реализовать по принципу рассмотренного сервера: принимать данные исключительно при обработке соответствующего сообщения (FD_READ). Результат работы программы при подключении к финальной версии предложенного сервера показан на рис. 12.

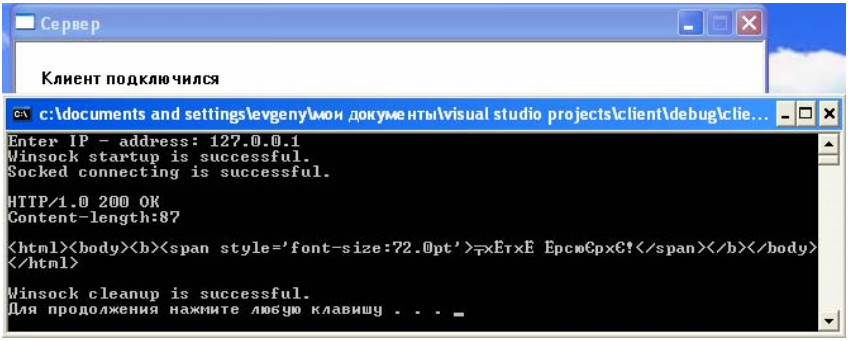

Рис. 12. Работа клиента в консольном режиме

Таким образом, в этом разделе рассмотрены:

1) основные понятия, необходимые для разработки программ, обеспечивающих сетевое взаимодействие электронных устройств: сетевые модели, протоколы, адреса, порты;
2) важнейшие функции Беркли, обеспечивающие приложения в сети;
3) принципы построения серверных программ на языке программирования *Cи*;
4) принципы построения клиентских программ на языке программирования *Cи*.

Приведенные примеры программ работают на 80 порте, на практике необходимо выбирать неиспользуемый другими программами порт.

Следует отметить, что в коде приведенных программ достаточно большое количество условных операторов: при написании сетевых программ следует уделять пристальное внимание результатам выполнения функций, а также принимаемым по сети данным. Ошибки могут возникать на любом этапе работы. Например, функция bind() вернет ошибку, если желаемый порт уже занят. Это может быть из-за запущенного другого экземпляра сервера или программы фильтрации HTTP-трафика (антивируса). Ошибки могут привести к серьезным нарушениям в сетевой безопасности и к уязвимости данных и оборудования.

Контрольные вопросы

1. В чем различие между сетевым протоколом и сетевой моделью? Зачем сетевые модели имеют многоуровневую структуру?
2. Каково назначение портов в модели TCP/IP? Могут ли две программы, запущенные на одном компьютере, прослушивать в ожидании данных один порт?
3. Что такое сервер?
4. Каково назначение DNS-сервера? Где обычно он располагается? Что произойдет, если такой сервер выйдет из строя?
5. Можно ли передавать информацию между устройствами по сети, если им не назначены IP-адреса?
6. Вызов каких функций Беркли необходим для прослушивания порта 80? Почему при указании порта сокету запись socketaddr.sin_port = 80; была бы ошибочной?
7. Как программе среагировать на подключение клиента после установки сокета в режим прослушивания?
8. Какую стандартную программу Windows можно использовать в качестве универсального клиента при тестировании разработанных серверных приложений?
9. Почему не следует использовать дескриптор сокета, полученный сервером при подключении клиента в процессе обмена данными?

5. ОСНОВЫ ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих одновременно.

Существуют различные способы реализации параллельных вычислений: каждый вычислительный процесс может быть реализован в виде процесса операционной системы либо вычислительные процессы могут представлять собой набор потоков выполнения внутри одного процесса. Поток (или правильнее поток выполнения) – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, например память, тогда как процессы не разделяют этих ресурсов. Параллельные программы могут физически исполняться либо последовательно на единственном процессоре – перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно – выделяя каждому вычислительному процессу один или несколько процессоров (находящихся рядом или распределённых в компьютерную сеть).

Основная сложность при проектировании параллельных программ – обеспечить правильную последовательность взаимодействий между различными вычислительными процессами, а также разделение таких ресурсов, как оперативная память или периферийные устройства.

В некоторых параллельных системах программирования передача данных между компонентами скрыта от программиста, тогда как в других она должна указываться явно. Явные взаимодействия могут быть разделены на два типа.

1. Взаимодействие через разделяемую память (например, в Java или C#). Данный вид параллельного программирования обычно требует какой-то формы захвата управления для координации потоков между собой.

2. Взаимодействие с помощью передачи сообщений. Обмен сообщениями может происходить асинхронно либо с использованием метода «рандеву», при котором отправитель блокирован до тех пор, пока его сообщение не будет доставлено. Асинхронная передача сообщений может быть надёжной (с гарантией доставки) либо ненадёжной. Параллельные системы, основанные на обмене сообщениями, зачастую более просты для понимания, чем системы с разделяемой памятью, и обычно рассматриваются как более совершенный метод параллельного программирования. Обмен сообщениями может быть эффективно реализован на симметричных мультипроцессорах как с разделяемой когерентной памятью, так и без неё.

Существует довольно много разных технологий параллельного программирования. Причем эти технологии отличаются не столько языками программирования, сколько архитектурными подходами к построению параллельных систем. Например, какие-то технологии предполагают построение параллельных решений на основе нескольких компьютеров (как одного, так и разных типов), другие же предполагают именно работу на одной машине с несколькими процессорными ядрами. В настоящее время основными программными инструментами создания параллельных программ являются:

1) **OpenMP** – используется в параллельных системах с общей памятью (например, современные компьютеры с многоядерными процессорами);
2) **MPI (Message Passing Interface)** – стандартная система передачи сообщений между параллельно исполняемыми процессами, используемая при разработке программ для суперкомпьютеров;
3) **POSIX Threads** – стандарт реализации потоков выполнения;
4) операционная система Windows – имеет встроенную поддержку многопоточных приложений для C++ на уровне API;
5) **PVM (Parallel Virtual Machine)** – позволяет объединять разнородные связанные сетью компьютеры в общий вычислительный ресурс.

Системы на базе нескольких компьютеров относят к классу систем для распределенных вычислений. Подобные решения используются довольно давно. Наиболее яркий пример технологии распределенных вычислений – MPI (Message Passing Interface – интерфейс передачи сообщений). MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для огромнейшего числа компьютерных платформ. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные/ многопроцессорные с общей/ раздельной памятью), взаимного расположения ветвей (на одном процессоре или на разных).

Так как MPI предназначен в первую очередь для систем с раздельной памятью, то использование его для организации параллельного процесса в системе с общей памятью является крайне сложным и нецелесообразным. Тем не менее ничего не мешает делать MPI-решения для одной машины.

Системы параллельного программирования для работы на одной машине начали развиваться относительно недавно. Конечно, это не принципиально новые идеи, но именно с приходом многоядерных систем на рынок персональных компьютеров, мобильных устройств такие технологии, как OpenMP, получили значительное развитие.

Очень важно, чтобы технология параллельного программирования поддерживала возможность делать программу параллельной постепенно. Разумеется, идеальную параллельную программу следует сразу писать параллельной, возможно, на каком-нибудь функциональном языке, где вопрос распараллеливания вообще не стоит. Но на практике приходится постепенно распараллеливать написанную последовательную с целью повышения быстродействия. В этом случае технология OpenMP будет очень удачным выбором. Она позволяет, выбрав в приложении наиболее нуждающиеся в параллелизации места, в первую очередь сделать параллельными именно их. Процесс разработки параллельной версии можно прерывать, выпускать промежуточные версии программы, возвращаться к нему по мере необходимости. Именно поэтому, в частности, технология OpenMP стала довольно популярной.

OpenMP (Open Multi-Processing) – это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения, чья работа регулируется некоммерческой организацией, называемой OpenMP Architecture Review Board (ARB).

Первая версия появилась в 1997 году, предназначалась для языка Fortran. Для C/C++ версия разработана в 1998 году. В 2008 году вышла версия OpenMP 3.0. Интерфейс OpenMP стал одной из наиболее популярных технологий параллельного программирования. OpenMP успешно используется как при программировании суперкомпьютерных систем с большим количеством процессоров, так и в настольных пользовательских системах или, например, в Xbox 360.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

Задачи, выполняемые потоками параллельно, так же как данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка – прагм. Например, участок кода на языке Fortran, который должен исполняться несколькими потоками, каждый из которых имеет свою копию переменной *N*, предваряется следующей директивой: !$OMP PARALLEL PRIVATE(N).

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне – при помощи переменных окружения.

Ключевыми элементами OpenMP являются:

1) конструкции для создания потоков (директива parallel);
2) конструкции распределения работы между потоками (директивы DO/for и section);

3) конструкции для управления работой с данными (выражения shared и private для определения класса памяти переменных);
4) конструкции для синхронизации потоков (директивы critical, atomic и barrier);
5) процедуры библиотеки поддержки времени выполнения (например, omp_get_thread_num);
6) переменные окружения (например, OMP_NUM_THREADS).

В OpenMP используется модель параллельного выполнения «ветвление – слияние». Программа OpenMP начинается как единственный поток выполнения, называемый начальным потоком. Когда поток встречает параллельную конструкцию, он создает новую группу потоков, состоящую из себя и некоторого числа дополнительных потоков, и становится главным в новой группе. Все члены новой группы (включая главный) выполняют код внутри параллельной конструкции. В конце параллельной конструкции имеется неявный барьер. После параллельной конструкции выполнение пользовательского кода продолжает только главный поток. В параллельный регион могут быть вложены другие параллельные регионы, в которых каждый поток первоначального региона становится основным для своей группы потоков. Вложенные регионы могут, в свою очередь, включать регионы более глубокого уровня вложенности.

Число потоков в группе, выполняющихся параллельно, можно контролировать несколькими способами. Один из них – использование переменной окружения OMP_NUM_THREADS. Другой способ – вызов процедуры omp_set_num_threads(). Еще один способ – использование выражения num_threads в сочетании с директивой parallel.

В этой программе два массива (*a* и *b*) складываются параллельно десятью потоками.

```
#include <stdio.h>
#include <omp.h>

#define N 100

int main(int argc, char *argv[])
{
```

```
        float a[N], b[N], c[N];
        int i;
        omp_set_dynamic(0);      // запретить библиотеке openmp
менять число потоков во время исполнения
        omp_set_num_threads(10); // установить число потоков в 10

        // инициализируем массивы
        for (I = 0; I < N; i++)
        {
            a[i] = I * 1.0;
            b[i] = i * 2.0;
        }

        // вычисляем сумму массивов
    #pragma omp parallel shared(a, b, c) private(i)
        {
    #pragma omp for
          for (I = 0; I < N; i++)
            c[i] = a[i] + b[i];
        }
        printf ("%f\n", c[10]);
        return 0;
    }
```

Эту программу можно скомпилировать, используя gcc-4.4 и более новые с флагом –fopenmp. Очевидно, если убрать подключение заголовочного файла omp.h, а также вызовы функции настройки OpenMP, программу можно скомпилировать на любом компиляторе C как обычную последовательную программу.

OpenMP поддерживается многими современными компиляторами.

1. Компиляторы Sun Studio поддерживают официальную спецификацию OpenMP 2.5 – с улучшенной производительностью под ОС Solaris; поддержка Linux запланирована на следующий релиз.

2. Visual C++ 2005 и выше поддерживает OpenMP в редакциях Professional и Team System.

3. GCC 4.2 поддерживает OpenMP, а некоторые дистрибутивы (такие как Fedora Core 5 gcc) включили поддержку в свои версии GCC 4.1.

4. Intel C++ Compiler, включая версию Intel Cluster OpenMP для программирования в системах с распределённой памятью.

Message Passing Interface (**MPI,** интерфейс передачи сообщений) – программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом, Эвином Ласком (англ.) и др.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в приложениях пользователя. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI. Существуют реализации для языков Фортран 77/90, Си и Си++.

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кэшем). Обе технологии могут использоваться совместно, для оптимального функционирования в кластере многоядерных систем.

Первая версия MPI разрабатывалась в 1993–1994 годах, и MPI 1 вышла в 1994 году.

Большинство современных реализаций MPI поддерживают версию 1.1. Стандарт MPI версии 2.0 поддерживается большинством современных реализаций, однако некоторые функции могут быть реализованы не до конца.

В MPI 1.1 (опубликован 12 июня 1995 года, первая реализация появилась в 2002 году) поддерживаются следующие функции:

– передача и получение сообщений между отдельными процессами;
– коллективные взаимодействия процессов;

– взаимодействия в группах процессов;
– реализация топологий процессов.

В MPI 2.0 (опубликован 18 июля 1997 года) дополнительно поддерживаются следующие функции:
– динамическое порождение процессов и управление процессами;
– односторонние коммуникации (Get/Put);
– параллельный ввод и вывод;
– расширенные коллективные операции (процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов).

Версия MPI 2.1 вышла в начале сентября 2008 года.

Базовым механизмом связи между MPI-процессами является передача и приём сообщений. Сообщение несёт передаваемые данные и информацию, позволяющую принимающей стороне осуществлять их выборочный приём:
1) отправитель – ранг (номер в группе) отправителя сообщения;
2) получатель – ранг получателя;
3) признак – может использоваться для разделения различных видов сообщений;
4) коммуникатор – код группы процессов.

Операции приёма и передачи могут быть блокирующимися и неблокирующимися. Для неблокирующихся операций определены функции проверки готовности и ожидания выполнения операции.

Другим способом связи является удалённый доступ к памяти (RMA), позволяющий читать и изменять область памяти удалённого процесса. Локальный процесс может переносить область памяти удалённого процесса (внутри указанного процессами окна) в свою память и обратно, а также комбинировать данные, передаваемые в удалённый процесс, с имеющимися в его памяти данными (например, путём суммирования). Все операции удалённого доступа к памяти неблокирующиеся, однако до и после их выполнения необходимо вызывать блокирующиеся функции синхронизации.

Ниже приведён пример программы нахождения числа π на языке *C* с использованием MPI.

```
// Подключение необходимых заголовков
#include <stdio.h>
#include <math.h>
// Подключение заголовочного файла MPI
#include «mpi.h»

// Функция для промежуточных вычислений
double f(double a)
{
    return (4.0 / (1.0+ a*a));
}

// Главная функция программы
int main(int argc, char **argv)
{
    // Объявление переменных
    int done = 0, n, myid, numprocs, I;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    // Инициализация подсистемы MPI
    MPI_Init(&argc, &argv);
    // Получить размер коммуникатора MPI_COMM_WORLD
    // (общее число процессов в рамках задачи)
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    // Получить номер текущего процесса в рамках
    // коммуникатора MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
```

```
        // Вывод номера потока в общем пуле
        fprintf(stdout, "Process %d of %d is on %s\n",
myid,numprocs,processor_name);
        fflush(stdout);

        while(!done)
        {
            // количество интервалов
            if(myid==0)
            {
                fprintf(stdout, "Enter the number of intervals:
(0 quits) ");
                fflush(stdout);
                if(scanf("%d",&n) != 1)
                {
                    fprintf(stdout, "No number entered;
quitting\n");
                    n = 0;
                }
                startwtime = MPI_Wtime();
            }
            // Рассылка количества интервалов всем процессам (в
том числе и себе)
            MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
            if(n==0)
                done = 1;
            else
            {
                h = 1.0 / (double) n;
                sum = 0.0;
                // Обсчитывание точки, закрепленной за процессом
                for(I = myid + 1 ; (I <= n) ; I += numprocs)
                {
                    x = h * ((double)I - 0.5);
                    sum += f(x);
                }
```

```
                mypi = h * sum;

                // Сброс результатов со всех процессов и
сложение
                MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);

                // Если это главный процесс, вывод полученного
результата
                if(myid==0)
                {
                    printf("PI is approximately %.16f, Error is
%.16f\n", pi, fabs(pi - PI25DT));
                    endwtime = MPI_Wtime();
                    printf("wall clock time = %f\n", endwtime-
startwtime);
                    fflush(stdout);
                }
            }
        }

        // Освобождение подсистемы MPI
        MPI_Finalize();
        return 0;
    }
```

Наиболее распространенными реализациями MPI на сегодняшний день являются:

MPICH – самая распространённая бесплатная реализация, работает на UNIX-системах и Windows NT.

LAM/MPI – ещё одна бесплатная реализация MPI. Поддерживает гетерогенные конфигурации, LAM (http://www.lam-mpi.org) поддерживает гетерогенные конфигурации, пакет Globus и удовлетворяет IMPI (Interoperable MPI).

Поддерживаются различные коммуникационные системы (в том числе Myrinet).

WMPI – реализация MPI для Windows.

MPI/PRO for Windows NT – коммерческая реализация для Windows NT.

Intel MPI – коммерческая реализация для Windows/ Linux.

Microsoft MPI входит в состав Compute Cluster Pack SDK.Основан на MPICH2, но включает дополнительные средства управления заданиями. Поддерживается спецификация MPI-2.

HP-MPI – коммерческая реализация от HP.

SGI MPT – платная библиотека MPI от SGI.

Mvapich – бесплатная реализация MPI для Infiniband.

Open MPI – бесплатная реализация MPI, наследник LAM/MPI.

Oracle HPC ClusterTools – бесплатная реализация для Solaris SPARC/x86 и Linux на основе Open MPI.

MPJ – MPI for Java.

POSIX Threads – стандарт POSIX реализации потоков выполнения, определяющий API для создания и управления ими.

Библиотеки, реализующие этот стандарт (и функции этого стандарта), обычно называются Pthreads (функции имеют приставку «pthread_»). Хотя наиболее известны варианты для Unix-подобных операционных систем, таких как Linux или Solaris, но существует и реализация для Microsoft Windows (Pthreads-w32).

Pthreads определяет набор типов и функций на языке программирования *Си*. Заголовочный файл – pthread.h.

Типы данных:

1) pthread_t – дескриптор потока;
2) pthread_attr_t – перечень атрибутов потока.

Функции управления потоками:

1) pthread_create() –создание потока;
2) pthread_exit() – завершение потока (должна вызываться функцией потока при завершении);
3) pthread_cancel() – отмена потока;
4) pthread_join() – заблокировать выполнение потока до прекращения другого потока, указанного в вызове функции;
5) pthread_detach() – освободить ресурсы, занимаемые потоком (если поток выполняется, то освобождение ресурсов произойдёт после его завершения);

6) pthread_attr_init() – инициализировать структуру атрибутов потока;
7) pthread_attr_setdetachstate() – указать системе, что после завершения потока она может автоматически освободить ресурсы, занимаемые потоком;
8) pthread_attr_destroy() – освободить память от структуры атрибутов потока (уничтожить дескриптор).

Функции синхронизации потоков:

1) pthread_mutex_init(), pthread_mutex_destroy(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock();
2) pthread_cond_init(), pthread_cond_signal(), pthread_cond_wait().

Пример использования потоков на языке С.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

static void wait_thread(void)
{
    time_t start_time = time(NULL);

    while (time(NULL) == start_time)
    {
        /* do nothing except chew CPU slices for up to one
second. */
    }
}

static void *thread_func(void *vptr_args)
{
    int I;

    for (I = 0; I < 20; i++)
    {
        fputs("  b\n", stderr);
        wait_thread();
```

```
        }

        return NULL;
    }

    int main(void)
    {
        int I;
        pthread_t thread;

        if (pthread_create(&thread, NULL, thread_func, NULL) !=
0)
        {
            return EXIT_FAILURE;
        }

        for (I = 0; I < 20; i++)
        {
            puts("a");
            wait_thread();
        }

        if (pthread_join(thread, NULL) != 0)
        {
            return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
    }
```

Представленная программа использует два потока, печатающих в консоль сообщения: один, печатающий ‘a’, второй – ‘b’. Вывод сообщений смешивается в результате переключения выполнения между потоками или при одновременном выполнении на мультипроцессорных системах.

Программа на *C* создает один новый поток для печати ‘b’, а основной поток печатает ‘a’. Основной поток (после печати ‘aaaaa....’) ждёт завершения дочернего потока.

Контрольные вопросы

1. Что такое параллельная программа?
2. В чем отличие между процессом и потоком выполнения?
3. Может ли программа создать пять потоков при работе на четырехъядерном процессоре?
4. Каковы особенности параллельных программ с разделяемой памятью?
5. Какие существуют программные средства для разработки параллельных программ?
6. Почему большое распространение при создании программ для ПК получил именно OpenMP, а не, например, MPI?

6. ОСНОВЫ РАЗРАБОТКИ И ПРОГРАММИРОВАНИЯ ПРОСТЕЙШИХ USB-УСТРОЙСТВ

Шина USB (Universal Serial Bus – универсальная последовательная шина) – появилась 15 января 1996 года при утверждении первого варианта стандарта фирмами Intel, DEC, IBM, NEC, Northen Telecom и Compaq.

Основная цель стандарта, поставленная перед его разработчиками, – создать возможность пользователям работать в режиме Plug&Play с периферийными устройствами. Это означает, что должно быть предусмотрено подключение устройства к работающему компьютеру, автоматическое распознавание его немедленно после подключения и последующей установки соответствующих драйверов. Кроме этого, желательно питание маломощных устройств подавать с самой шины. Скорость шины должна быть достаточной для подавляющего большинства периферийных устройств. Контроллер USB должен занимать только одно прерывание независимо от количества подключенных к шине устройств, то есть решить проблему нехватки ресурсов на внутренних шинах IBM PC совместимого компьютера.

Практически все поставленные задачи были решены в стандарте на USB, и весной 1997 года стали появляться компьютеры, оборудованные разъемами для подключения USB-устройств. Сейчас USB стала настолько активно внедряться производителями компьютерной периферии, что, например, в компьютере iMAC фирмы Apple Computers присутствует только USB в качестве внешней шины.

Возможности USB 1.0:

1) высокая скорость обмена данными (full-speed) – 12 Мб**ит**/с;

2) максимальная длина кабеля для высокой скорости обмена – 5 метров;

3) низкая скорость обмена данными (low-speed) – 1,5 Мб**ит**/с;
4) максимальная длина кабеля для низкой скорости обмена – 3 метра;
5) максимальное количество подключенных устройств – 127;
6) возможное одновременное подключение устройств с различными скоростями обмена;
7) напряжение питания для периферийных устройств – 5 В;
8) максимальный ток потребления на одно устройство – 500 мА.

Поэтому целесообразно подключать к USB 1.0 практически любые периферийные устройства, кроме цифровых видеокамер и высокоскоростных жестких дисков. Особенно удобен этот интерфейс для подключения часто подключаемых/ отключаемых приборов, таких как цифровые фотокамеры. Возможность использования только двух скоростей обмена данными ограничивает применяемость шины, но существенно уменьшает количество линий интерфейса и упрощает аппаратную реализацию. Питание непосредственно от USB возможно только для устройств с малым потреблением, таких как клавиатуры, мыши, джойстики и т. п.

Сигналы USB передаются по 4-проводному кабелю, схематично показанному на рис. 13.

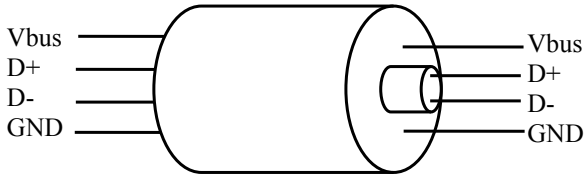

Рис. 13. Сигнальные провода USB

Здесь GND – цепь общего провода для питания периферийных устройств, Vbus – +5 В также для цепей питания. Шина D+ предназначена для передачи данных по шине, а шина D – для приема данных. Кабель для поддержки полной скорости шины (full-speed) выполняется как витая пара, защищается экраном и может также использоваться для работы в режиме минимальной скорости (low-speed). Кабель для работы только на минимальной скорости (например, для подключения мыши) может быть любым и неэкранированным. Разъемы, используемые для подклю-

чения периферийных устройств, делятся на серии: разъемы серии «А» (вилка и розетка) предназначены только для подключения к источнику, например, компьютеру; разъемы серии «В» (вилка и розетка) – только для подключения к периферийному устройству.

USB-разъемы имеют следующую нумерацию контактов, показанную в табл. 5.

Таблица 5

Назначение и маркировка контактов USB

Номер контакта	Назначение	Вид провода
1	V BUS	Красный
2	D –	Белый
3	D +	Зеленый
4	GND	Черный
Оплетка	Экранирование сигналов	Металлическая оплетка

В 1999 году тот же консорциум компьютерных компаний, который инициировал разработку первой версии стандарта на шину USB, начал активно разрабатывать версию 2.0 USB, которая отличается введением дополнительного высокоскоростного (Hi-speed) режима. Полоса пропускания шины увеличена в 40 раз, до 480 Мбит/с, что сделало возможным передачу видеоданных по USB. Совместимость всей ранее выпущенной периферии и высокоскоростных кабелей полностью сохраняется. Контроллер стандарта 2.0 уже интегрирован в набор системной логики программируемых устройств (например, материнская плата персонального компьютера).

В 2008 году компаниями Intel, Microsoft, Hewlett-Packard, Texas Instruments, NEC и NXP Semiconductors создана спецификация стандарта USB 3.0. В спецификации USB 3.0 разъёмы и кабели обновлённого стандарта физически и функционально совместимы с USB 2.0, однако в дополнение к четырем линиям связи добавлены ещё четыре. Тем не менее новые контакты в разъёмах USB 3.0 расположены отдельно от старых на другом контактном ряду. Спецификация USB 3.0 повышает максимальную скорость передачи информации до 5 Гбит/с –что на порядок больше 480 Мбит/с, которые может обеспечить USB 2.0. Кроме

того, увеличена максимальная сила тока с 500 до 900 мА на одно устройство, что позволяет питать некоторые устройства, требующие ранее отдельного блока питания.

Предположим, разработано устройство USB, с которым необходимо работать с помощью компьютера. Этого можно достигнуть двумя способами:

1) разработкой полнофункционального драйвера операционной системы;
2) использованием интерфейса специального класса USB-устройств, называемыхHID (Human Interface Device).

Первый способ универсален: владея достаточными познаниями в области написания драйверов, можно запрограммировать работу с любым устройством на любой скорости, поддерживаемой USB. Но это достаточно непростая задача.

Второй способ заключается в следующем. Существует поддерживаемый современными операционными системами интерфейс для устройств взаимодействия компьютера и человека или HID-устройств, таких как:

1) клавиатуры, мыши, джойстики;
2) различные датчики и считыватели;
3) игровые рулевое управление и педали;
4) кнопки, переключатели, регуляторы.

Любое такое устройство, если оно выполняет требования к HID-устройствам, будет автоматически распознано системой и не потребует написания специальных драйверов. Кроме того, их программирование, как правило, намного проще написания специализированного драйвера. К сожалению, данный способ имеет существенный недостаток: скорость обмена информацией с HID-устройством сильно ограничена и составляет максимум 64 кБ/с.

На основе HID-технологии можно организовать взаимодействие с любым устройством, даже если оно не является в строгом смысле интерфейсным устройством. Это позволяет отказаться от трудоемкой разработки уникального драйвера устройства и сэкономить время на разработку нового USB-устройства. На стороне хоста обменом с устройством будет руководить стандартный HID-драйвер, включенный

в поставку операционной системы. Нужно лишь выполнить со стороны устройства минимальные требования USB-HID протокола.

Стоит отметить, что многие USB-приборы, с первого взгляда не попадающие под определение устройств взаимодействия с человеком, логичнее все же реализовать как HID-устройства. Такое явление часто встречается в области производственного оборудования, которая в последнее время переживает массовое внедрение USB-технологий. Рассмотрим, к примеру, лабораторный источник питания с возможностью задания параметров его выходных сигналов с компьютера с помощью USB-интерфейса. Непосредственно источник питания не является средством взаимодействия с человеком. Однако в данном случае функции, реализуемые посредством USB-подключения, дублируют клавиатуру, регуляторы и индикаторы, установленные на самом приборе. А эти органы управления как раз попадают под определение HID. Соответственно, блок питания с этими USB-функциями логичнее всего организовать как HID-устройство.

В рассмотренном примере для нормальной работы достаточно будет небольшой скорости передачи данных, в других же случаях приборы могут быть весьма требовательны к скорости обмена. Низкая скорость передачи является главным ограничением HID-варианта построения устройства, что в сравнении с 12 Мбит/с полной скорости шины USB 1.0 выглядит большим минусом HID-технологии в вопросе выбора конкретной USB-реализации. Однако для многих задач коммуникации указанной скорости вполне хватает и HID-архитектура как специализированный инструмент занимает достойное место среди способов организации обмена данными.

HID-устройства бывают двух типов: участвующие (загрузочные) и не участвующие в начальной загрузке компьютера. Наиболее ярким примером загрузочного USB-HID устройства является клавиатура, работа которой начинается с запуска компьютера.

При разработке HID-устройства необходимо обеспечить следующие требования, налагаемые спецификацией:

1) полноскоростное HID-устройство может передавать 64000 байт каждую секунду, или по 64 байта каждые 1 мс; низкоскоростное HID-устройство имеет возможность передать вплоть до 800 байт в секунду, или по 8 байт каждые 10 мс;

2) HID-устройство может назначить частоту своего опроса для определения того, есть ли у него свежие данные для передачи;
3) обмен данными с HID-устройством осуществляется посредством специальной структуры, называемой репортом (report). Каждый определенный репорт может содержать до 65535 байт данных. Структура репорта имеет весьма гибкую организацию, позволяющую описать любой формат передачи данных. Для того чтобы конкретный формат репорта стал известен хосту, микроконтроллер должен содержать специальное описание – дескриптор репорта.

Реализуется USB-взаимодействие непосредственно на микроконтроллере несколькими способами:
1) использованием контроллера с аппаратной поддержкой, например AT90USB* фирмы Atmega;
2) использованием программной эмуляции USB-интерфейса на любом микроконтроллере.

Для программной реализации в настоящее время существует ряд готовых решений под различные семейства микроконтроллеров. Для AVR-микроконтроллеров, например Atmega8, можно использовать следующие свободные библиотеки на языке *Си*:
1) V-USB;
2) USBtiny.

Обе достаточно простые в использовании, обеспечивают полную эмуляцию USB 1.1 low-speed устройств (за исключением обработки ошибок связи и электрических характеристик) и запускаются практически на всех AVR-контроллерах с минимум 2 килобайтами flash-памяти, 128 байтами RAM и частотой от 12 до 20 МГц.

Для написания приложений с поддержкой Windows USB HID устройств требуются заголовочные файлы hid*, входящие в состав WDK (Windows Driver Kit), можно использовать свободно распространяемую библиотеку Hidlibrary или другую аналогичную.

Таким образом, программирование USB – достаточно сложная задача, требующая специального микроконтроллера с аппаратной поддержкой и написания драйвера операционной системы. Однако на практике при разработке устройств можно использовать значительно более простой интерфейс HID-устройств, поддержка которого реализована на уровне стандартного драйвера системы, а программирование упрощается использованием существующих библиотек функций.

Контрольные вопросы

1. В чем отличие провода D- и GND в USB? Почему нельзя использовать один общий провод для питания и сигнала?
2. Сколько режимов скорости работы USB существует на сегодняшний день (включая версию 3.0)?
3. Что такое HID-устройство? Почему для их работы в современных ОС не требуется написание драйверов?
4. Можно ли реализовать USB-устройства с помощью микропроцессора, не имеющего встроенной поддержки интерфейса?
5. Каковы основные отличия USB 3.0 от предыдущих версий?

ЗАКЛЮЧЕНИЕ

В состав современных электронных устройств зачастую входят программируемые элементы – микропроцессоры. Функционирование таких устройств определяется не только схемотехническим решением, но в значительной мере и алгоритмами, заложенными в микропроцессоры. С течением времени быстродействие процессоров, применяемых в бытовой и промышленной электронике, становится сравнимо с быстродействием процессоров, использовавшихся в персональных компьютерах несколько лет назад, а иногда и превосходит его. Стоимость микросхем с низким энергопотреблением, например многоядерных процессоров ARM-архитектуры, работающих на частотах до 1,2 ГГц и более, постоянно снижается. С одной стороны, это позволит разрабатывать более сложные электронные устройства, чем существуют в настоящее время, с другой – повысит требования к рациональному использованию вычислительных ресурсов. Эффективное программирование таких быстродействующих устройств невозможно без многократного использования отлаженного существующего кода: операционных систем, библиотек функций, драйверов. Это приводит к необходимости понимания принципов унификации в программировании, стиля программирования, изучения существующих приемов и подходов, различных концепций, используемых в современном программном обеспечении. Рассмотрение именно таких общих подходов (событийно-управляемое программирование, абстрагирование от аппаратуры, многоуровневые системы, концепция измерений, визуальное программирование и т. д.) на фоне многочисленных примеров – основная задача данного пособия.