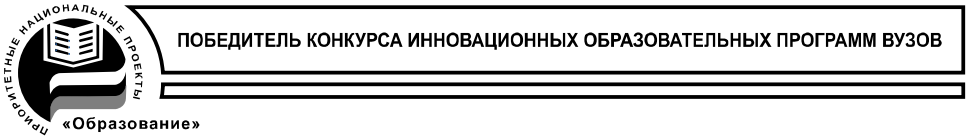
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ



В.А. Безруков

WIN32 API
ПРОГРАММИРОВАНИЕ

Учебное пособие

Санкт-Петербург

2009

УДК 681.3.06(035.5)

Безруков В.А. Win32 API. Программирование /учебное пособие. – СПб: СПбГУ ИТМО, 2009. – 90 с.

Рассмотрены основные принципы программирования в среде Microsoft Windows на языке C++ с применением Win32 API.

Пособие предназначено для студентов, обучающихся по специальностям 210202.65 «Проектирование и технология вычислительных средств» и 0900104.65 «Комплексная защита объектов информации», а также для студентов других специальностей изучающих дисциплину «Программирование на языках высокого уровня».

Рекомендовано к печати Советом факультета компьютерных технологии и управления, протокол № 5 от 08 декабря 2009 г.

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007-2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация этой программы создала основу формирования программы дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

ВВЕДЕНИЕ

В операционной системе Windows реализована объектно-ориентированная идеология. Базовый объект системы – окно, поведение которого определяется методом, называемым функцией окна. Графический образ окна на экране дисплея – прямоугольная рабочая область.

Независимо от своего типа любой объект Windows идентифицируется описателем или дескриптором (handle). Дескриптор – это ссылка на объект. Все взаимоотношения программного кода с объектом осуществляются только через его дескриптор.

Интерфейс прикладного программирования (API – Application Programming Interface) представляет собой совокупность 32-битных функций (Win32 API), которые предназначены для создания приложений (программ), работающих под управлением Microsoft Windows. Функции объявлены в заголовочных файлах. Главный из них − файл windows.h, в котором содержатся ссылки на другие заголовочные файлы.

В Win32 единицей работы компьютера является поток − ход выполнения программы в рамках процесса (в контексте процесса). Поток выполняет программный код, принадлежащий процессу. Процесс − это экземпляр выполняемой программы (но не ход ее выполнения). Он не является динамическим объектом и включает виртуальное адресное пространство, код и данные, файлы, синхронизирующие объекты, динамические библиотеки.

Каждое приложение создает, по меньшей мере, один первичный поток, но может создать и много потоков.

Любое приложение Windows представлено на экране дисплея как минимум одним окном с набором стандартных элементов управления. Различают следующие типы окон:

- перекрывающие (overlapped window);
- всплывающие (pop-up window);
- дочерние (child window);
- слоистые (layered window) – особые окна, которые позволяют улучшить визуальный эффект, включая прозрачность.

Перекрывающие окна создаются функцией CreateWindowEx() со стилем WS_OVERLAPPEDWINDOW. Этот стиль определяет наличие заголовка, системного меню, кнопок минимизации и максимизации, кнопки закрытия окна и «толстой» рамки, позволяющей изменять размеры окна. Перекрывающие окна предназначены для главных окон приложений и могут иметь меню.

Всплывающие окна создаются функцией CreateWindowEx() со стилем WS_POPUP и предназначены для окон диалогов, окон сообщений и других окон временного использования, которые могут находиться вне главного окна приложения. Для того чтобы временное окно имело заго-

ловок, рамку и системное меню, необходимо при его создании использовать комбинацию стилей WS_POPUPWINDOW | WS_CAPTION.

Дочерние окна создаются функцией CreateWindowEx() со стилем WS_CHILD и обычно используются для разделения клиентской области родительского окна на отдельные функциональные области. Дочерние окна могут иметь заголовок, системное меню, кнопки минимизации и максимизации, рамку и полосы прокрутки, но не могут иметь меню. Дочерние окна всегда находятся в пределах клиентской области родительского окна, т.е. их координаты всегда отсчитываются от левого верхнего угла родительского окна. Родительское окно может быть перекрывающим, всплывающим или даже другим дочерним окном.

Виды приложений:

- SDI (Single Document Interface) – приложение с одно–документным интерфейсом;
- MDI (Multiple Document Interface) – приложение с многодокументным интерфейсом;
- диалоговое приложение (Based Dialog) − содержит только диалоговое окно с элементами управления, не имеет главного окна, а значит, не имеет меню.

Windows поддерживает работу с символами как в традиционной ANSI кодировке, так и в кодировке UNICODE. В стандарте UNICODE каждый символ кодируется двумя байтами, что позволяет определить 65536 символов.

Чтобы была возможность компилировать приложение как ANSI версию или как UNICODE версию без изменения приложения необходимо:

1. включить в приложение файл tchar.h;
2. при определении символов и строк использовать типы TCHAR, LPTSTR и LPCTSTR;
3. при определении символьных или строковых литералов использовать макрос _TEXT (или просто _T);
4. необходимо помнить, что sizeof(szBuffer) – размер буфера в байтах, а sizeof(szBuffer)/sizeof(TCHAR) – размер буфера в символах.

Типы данных TCHAR, LPTSTR и LPCTSTR определены следующим образом:

```
typedef  unsigned short  wchar_t;
typedef  wchar_t   WCHAR;
#define  UNICODE
     typedef WCHAR  TCHAR;      //UNICODE кодировка
#else
     typedef char   TCHAR;       //ANSI кодировка
#endif
```

```
typedef  WCHAR*  LPWSTR;
typedef  CHAR*   LPSTR;
#define  UNICODE
     typedef  LPWSTR  LPTSTR;      //UNICODE кодировка
#else
     typedef  LPSTR   LPTSTR;       //ANSI кодировка
#endif

typedef CONST WCHAR* LPCWSTR;
#define  UNICODE
     typedef  LPCWSTR LPCTSTR;      //UNICODE кодировка
#else
     typedef  LPCSTR  LPCTSTR;       //ANSI кодировка
#endif
```

Почти все функции, получающие в качестве аргумента адрес строки, имеют ANSI и UNICODE версии, например, прототип функции lstrcat конкатенации символьных строк имеет вид:

```
LPTSTR lstrcat (
     LPTSTR lpString1,
     LPTSTR lpString2
);
```

Функции Win32 также имеют ANSI и UNICODE версии, например, функция DispatchMessage():

```
# ifdef  UNICODE
#define  DispatchMessage  DispatchMessageW
#else
#define  DispatchMessage  DispatchMessageA
#endif
```

Приложения, приведенные как примеры в пособии, могут компилироваться без изменения исходного текста в ANSI версии или в UNICODE версии, и протестированы на компьютере с операционной системой Microsoft Windows XP Professional в среде Microsoft Visual Studio 2005.

1. ОБЩАЯ СТРУКТУРА ПРИЛОЖЕНИЯ WINDOWS

Windows накладывает жесткие ограничения на структуру приложения, которое, как правило, содержит минимум две принципиально важные функции: главную WinMain() и функцию окна WndPoc().

1.1. Главная функция WinMain()

Функция WinMain() должна быть в каждом приложении. Ее прототип описан в файле winbase.h следующим образом:

```
int WINAPI WinMain(
   HINSTANCE hInstance,   //дескриптор экземпляра
                          //приложения
  HINSTANCE hPrevInstance,    //дескриптор предыдущего
                              //экземпляра приложения
   LPSTR lpszCmdLine,     //указатель на параметры
                          //командной строки
   int nCmdShow      //константа, характеризующая
```

```
                    //начальный вид окна
);
```

Спецификатор WINAPI определяет соглашение о вызове, т.е. принятый в Win32 порядок передачи параметров при вызове функций. Параметры передаются через стек, справа налево, т.е. первый параметр помещается в стек последним, а очистку стека осуществляет вызываемая процедура.

Вызывая функцию WinMain(), Windows передает ей четыре параметра (аргумента).

Первый параметр hInstance типа HINSTANCE представляет собой дескриптор данного экземпляра приложения. Он назначается приложению при его запуске системой Windows и служит для его идентификации. Многие функции Win32 API используют этот дескриптор в качестве одного из параметров. Если значение дескриптора (hInstance) используется в другой функции, т.е. не в главной, то получить значение hInstance можно следующими способами:

- объявить глобальную переменную HINSTANCE hInst;
- значение дескриптора определить при помощи функции GetClassLong(), например:

```
hInst=(HINSTANCE)GetClassLong(hWnd,GCL_HMODULE);
```

- значение дескриптора определить при помощи функции GetModuleHandle() следующим образом:

```
hInst= GetModuleHandle(NULL);.
```

Второй параметр – hPrevInstance – всегда равен нулю и не имеет смысла.

Третий параметр – lpszCmdLine – представляет собой указатель на строку, содержащую параметры командной строки запуска приложения, если они при запуске были указаны.

Четвертый параметр – nCmdShow – характеризует режим запуска.

Внутренние, действующие в программе имена для параметров функции WinMain(), как и для любой другой, можно выбирать по своему усмотрению.

В типичном приложении Windows главная функция WinMain() должна выполнить как минимум три процедуры:

1) зарегистрировать в системе класс главного окна приложения; если при этом необходимо вывести на экран внутренние порожденные окна, то их классы также следует зарегистрировать (Windows выводит на экран и обслуживает только зарегистрированные окна);

2) создать главное и порожденные окна и показать их на экране (порожденные окна можно создать и позже и необязательно в главной функции);

3) организовать главный цикл обработки сообщений, поступающих в приложение.

1.2. Класс окна и его характеристики

Оконный класс (window class), или класс окна, – это структура типа WNDCLASSEX, определяющая основные характеристики окна. На базе одного и того же класса можно создать несколько окон, например с разными именами заголовков, и, следовательно, использовать одну и ту же оконную функцию. Для главного окна приложения обычно создается собственный класс окна, учитывающий индивидуальные требования к программе.

Регистрация класса окна заключается в заполнении структурной переменной типа WNDCLASSEX и вызове функции RegisterClassEx(), аргументом которой служит адрес этой переменной. Структура определена в файле winuser.h:

```
typedef struct tagWNDCLASSEX
   {
   UINT cbSize;            //размер структуры в байтах
   UINT style;             //стиль класса окна
   WNDPROC lpfnWndProc;    //указатель на функцию окна
   int cbClsExtra;         //дополнительная память в байтах
                           //для класса окна
   int cbWndExtra;         //дополнительная память в байтах
                           //для каждого окна этого класса
   HINSTANCE hInstance;    //дескриптор экземпляра
                           //приложения
   HICON hIcon;      //дескриптор пиктограммы приложения
   HCURSOR hCursor; //дескриптор курсора приложения
   HBRUSH hbrBackground;       //дескриптор кисти для
                               //закраски фона окна
   LPCTSTR lpszMenuName;       //указатель на строку
                               //с именем меню окна
   LPCTSTR lpszClassName; //указатель на строку
                               //с именем класса окна
   HICON hIconSm; //дескриптор малой пиктограммы
   } WNDCLASSEX;
```

Все элементы в структуре WNDCLASSEX представляют собой 32-разрядные значения.

В большинстве случаев нет необходимости определять все члены этой структуры. При заполнении переменной типа WNDCLASSEX необходимо обеспечить нулевое значение тем элементам структуры, которым не присваиваются конкретные значения. Нулевое значение означает для Windows, что характеристики этого элемента должны устанавливаться по умолчанию. Это правило характерно и для других структур, например OPENFILENAME, служащей для вывода на экран стандартного диалога «Открытие файла», или LOGFONT, позволяющей создать шрифт требуемого начертания.

Предварительное обнуление всей структурной переменной перед ее инициализацией можно осуществить с помощью функций memset() или ZeroMemory() , которые имеют следующие прототипы:

```
VOID ZeroMemory(
    PVOID pBuffer,   //указатель на блок памяти, который
                     //заполняется нулями
```

```
    SIZE_T length    //размер в байтах блока памяти
);
void *memset( void *dest, int c, size_t count );
```
Например: `memset(&wc, 0, sizeof(wc));`

После обнуления следует присвоить значения только тем элементам структуры WNDCLASSEX, которые определяют конкретные свойства класса.

Значение первого поля cbSize структуры WNDCLASSEX должно быть равно ее размеру в байтах. Это поле служит для обеспечения совместимости в случае будущих изменений.

Второе поле style структуры WNDCLASSEX представляет собой целое число (32 бита). Каждый разряд числа закреплен за той или иной характеристикой окна (или окон). При этом каждому биту числа соответствует своя символическая константа. В заголовочных файлах Windows такие идентификаторы начинаются с префикса CS_. Например:

- CS_DBLCLKS – установлен третий бит (0x00000008), что позволяет программе реагировать на двойные щелчки мышью в области окна;
- CS_VREDRAW – установлен нулевой бит (0x00000001), что заставляет перерисовывать окно заново при каждом изменении его размера по вертикали;
- CS_HREDRAW – установлен первый бит (0x00000002), что заставляет перерисовывать окно заново при каждом изменении его размера по горизонтали;
- CS_NOCLOSE – установлен девятый бит (0x00000200), что запрещает закрытие окна пользователем.

Объединение констант операцией «побитовое ИЛИ» языка C++ позволяет набрать требуемый комплект свойств (style).

Наиболее важными для функционирования программы являются следующие поля структуры WNDCLASSEX:

- lpfnWndProc – адрес оконной процедуры (третье поле);
- hInstance – дескриптор данного приложения (шестое поле);
- lpszClassName – указатель на строку с именем класса (11-е поле).

С помощью структурной переменной типа WNDCLASSEX (параметр lpfnWndProc) операционная система определяет адрес оконной функции, которую она должна вызвать при поступлении в окно сообщений.

Седьмое поле структуры – hIcon – содержит дескриптор пиктограммы. Пиктограмма – это маленькая битовая картинка, которая хранится в файле ресурсов приложения. Пиктограммы могут иметь следующие размеры:

- 16x16 пикселей – малые пиктограммы, обычно 16-цветные;
- 32x32 пикселя – стандартные пиктограммы, обычно 16-цветные;
- 48x48 пикселей – могут использовать 256 цветов.

Для получения дескриптора пиктограммы вызывают функцию

```
HICON LoadIcon(
   HINSTANCE hInst,        //дескриптор экземпляра
                           //приложения
   LPCTSTR  lpszIcon       //указатель на строку,
                           //которая содержит имя ресурса
                           // пиктограмм
);
```

Эта функция загружает ресурс пиктограммы из выполняемого файла (`.exe`) – экземпляра приложения (`hInst`), например:

```
wc.hIcon=LoadIcon(hInst,MAKEINTRESOURCE(IDI_APPICON));
```

Второй параметр – `lpszIcon`, определяющий имя ресурса, – строка с завершающим нулевым символом. В файле описания ресурсов, подготовленном с помощью редактора ресурсов, имя ресурса для пиктограммы представляет собой целочисленный идентификатор. Для преобразования целого числа в указатель на строку ресурса используют макрос `MAKEINTRESOURCE`, который определен в файле winuser.h следующим образом:

```
#define MAKEINTRESOURCE(i) \
                           (LPSTR) ((DWORD)((WORD)(i)))
```

Макрос преобразует число в указатель, при этом старшие 16 разрядов устанавливаются в нулевое значение.

Для загрузки предопределенных пиктограмм параметр `hInst` функции `LoadIcon()` должен иметь значение `NULL`. В этом случае второй аргумент функции содержит константу, идентификатор которой начинается с префикса `IDI_` (ID for icon), например:

```
wc.hIcon = LoadIcon(hInst, IDI_APLICATION);
```

Предопределенные идентификаторы пиктограмм см. в MSDN. Возвращаемое значение при успешном завершении функции `LoadIcon()` – дескриптор пиктограммы, а в случае неудачи – `NULL`.

Необходимо отметить, что функция `LoadIcon()` предназначена только для загрузки стандартных пиктограмм, т.е. загрузить с ее помощью пиктограмму размером 16x16 невозможно. Для загрузки пиктограмм произвольных размеров, курсоров или битовых изображений используется функция `LoadImage()`, которая имеет следующий прототип:

```
HANDLE LoadImage(
   HINSTANCE hInst,  //дескриптор экземпляра
                     //приложения
   LPCTSTR lpszName, //указатель на строку, которая
                     //содержит имя изображения,
                     //подлежащее загрузке
   UINT uType,       //тип загружаемого изображения
   int cxDesired,    //желаемая ширина изображения
                     //в пикселях
   int cyDesired,    //желаемая высота изображения
                     //в пикселях
   UINT fuLoad       //способ загрузки изображения
   );
```

Второй параметр – `lpszName` функции определяет загружаемое изображение. Но если первый параметр `hInst` равен `NULL`, а последний параметр `fuLoad` содержит флаг `LR_LOADFROMFILE`, то параметр

lpszName задает имя файла, в котором хранится изображение загружаемого ресурса.

Третий параметр – uType – определяет тип изображения и может принимать следующие значения:

```
IMAGE_BITMAP;
IMAGE_CURSOR;
IMAGE_ICON.
```

Четвертый и пятый параметры – cxDesired, cyDesired – задают ширину и высоту изображения в пикселях. Если оба параметра равны нулю, то функция использует фактическую ширину и высоту изображения.

Шестой параметр – fuLoad – указывает опции загрузки, который может содержать один или несколько следующих флагов:

- LR_DEFAULTCOLOR – флаг по умолчанию, для отображения используется текущий цветовой формат (или нуль);
- LR_DEFAULTSIZE – если значения параметров cxDesired и cyDesired равны нулю, то при загрузке изображения используются размеры по умолчанию;
- LR_LOADTRANSPARENT – загрузка в «прозрачном» режиме (цвет окна по умолчанию – COLOR_WINDOW);
- LR_LOADFROMFILE – функция загружает изображение из файла.

Загрузку маленькой пиктограммы размером 16x16 пикселей можно осуществить следующим образом:

```
wc.hIcon = (HICON)LoadImage( Inst,
                        MAKEINTRESOURCE(IDI_APPICON_SM),
                        IMAGE_ICON,16, 16,0);
```

Загрузка битового изображения из файла:

```
HBITMAP hBitmap (HBITMAP)LoadImage(hInst,
                              FILE_NAME, IMAGE_BITMAP,
                              100, 100, LR_LOADFROMFILE);
```

Восьмое поле hCursor структуры WNDCLASSEX содержит дескриптор курсора мыши. Курсор относится к ресурсам Windows – это битовое изображение размером 32x32 пикселя. Загрузка ресурса курсора производится функцией LoadCursor(), которая имеет следующий прототип:

```
HCURSOR LoadCursor(
   HINSTANCE hInst,       //дескриптор экземпляра
                          //приложения
   LPCTSTR lpszCursor     //указатель на строку,
);                        //содержащую имя ресурса курсора
```

Функция загружает ресурс курсора, заданный вторым параметром lpszCursor из экземпляра приложения, заданного первым параметром hInst.

При загрузке одного из системных (встроенных) курсоров необходимо первому параметру передать значение NULL, а второй должен содержать константу, идентификатор которой начинается с префикса IDC_.

Очевидно, что для главного окна приложения целесообразно выбирать стандартный курсор IDC_ARROW. Например, загрузку курсора, используемого по умолчанию, можно выполнить следующим образом:

```
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Встроенные курсоры и их символические имена:

IDC_ARROW – стандартная стрелка;

IDC_CROSS – перекрестие;

IDC_SIZEALL – четырехконечная стрелка;

IDC_SIZENS – двухконечная стрелка (север–юг);

IDC_IBEAM – текстовый двутавр;

IDC_WAIT – песочные часы.

Для динамического изменения формы курсора в зависимости от его местонахождения применяется функция SetCursor(), которая имеет следующий прототип:

```
HCURSOR SetCursor(HCURSOR hCursor);
```

Функция SetCursor делает текущим курсор, который передается ей в качестве параметра, и при этом возвращается дескриптор предшествующего курсора.

Положение курсора отслеживается при обработке сообщения WM_MOUSMOVE. Позиция курсора определяется относительно левого верхнего угла клиентской области.

Например, для изменения формы курсора в верхней и нижней областях окна необходимо обработать сообщения WM_CREATE, WM_SIZE и WM_MOUSMOVE следующим образом (переменные hCursor1, hCursor2, wClient, hClient, xPos и yPos должны быть объявлены в оконной процедуре с модификатором static):

```
Case WM_CREATE:
//…
hCursor1 = LoadCursor(NULL,IDC_SIZEALL);
hCursor2 = LoadCursor(NULL, IDC_WAIT);
//…
case WM_SIZE:
//определяем размеры клиентской области окна
wClient = LOWORD(lParam);
hClient = HIWORD(lParam);
break;
case WM_MOUSMOVE:
//определяем координаты курсора
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
if(yPos < 32)
//изменяем форму курсора
SetCursor(hCursor1);
if(yPos > hClient – 32)
//изменяем форму курсора
SetCursor(hCursor2);
break;
//…
```

Цвет фона окна определяется дескриптором кисти, записанным в поле hbrBackground структуры WNDCLASSEX.

Кисть (brush) – это графический объект, который представляет собой шаблон пикселей различных цветов.

Для описания графических объектов, таких как перо, кисть, растровое изображение, палитра, шрифт или регион, в Windows используется обобщенный тип HGDIOBJ. Функция GetStockObject() возвращает такой обобщенный дескриптор объекта и имеет следующий прототип:

```
HGDIOBJ GetStockObject(int nObjectTyp);
```

Поэтому при использовании функции GetStockObject() необходимо явное преобразование типа (HPEN,HBRUSH, HBITMAP, HPALETTE, HFONT, HRGN). Например, загрузка встроенной белой кисти может быть осуществлена следующим образом:

```
ws.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH).
```

В файле windowsx.h включены макросы GetStockBrush, GetStockPen и GetStockFont, выполняющие такое преобразование типа, например:

```
#define GetStockBrush(i)(HBRUSH)GetStockObject(i))
```

Эти макросы можно использовать вместо функции GetStockObject() следующим образом:

```
ws.hbrBackground = GetStockBrush(WHITE_BRUSH);
```

В Windows имеется несколько встроенных (предопределенных) кистей:

WHITE_BRUSH – белый цвет;

DKGRAY_BRUSH – темно-серый цвет;

LTGRAY_BRUSH – светло-серый цвет;

GRAY_BRUSH – серый цвет;

BLACK_BRUSH – черный цвет;

NULL_BRUSH – прозрачный.

Создание кисти произвольного цвета осуществляется вызовом функции CreateSolidBrush(), которая имеет следующий прототип:

```
HBRUSH CreateSolidBrush(COLORREF  crColor);
```

Аргумент crColor – значение RGB для цвета кисти. Цвет задается в виде трех целых чисел, характеризующих интенсивность красной, зеленой и синей составляющих цвета. Для упаковки этих трех чисел в двойное слово (4 байта) служит макрос RGB, например:

```
ws.hbrBackground= CreateSolidBrush(RGB(0,255,255));
```

Регистрация класса окна производится вызовом функции RegisterClassEx(), которая имеет следующий прототип:

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

Аргумент функции lpwc – адрес структурной переменной типа WNDCLASSEX.

Функция RegisterClassEx() возвращает атомарное значение ATOM (WORD – 16 бит), которое является уникальным идентификатором зарегистрированного класса.

Модификация характеристик оконного класса производится всегда только при обработке сообщения WM_CREATE функцией

SetClassLong() или SetWindowLong(), которые имеют следующие прототипы:

```
DWORD SetClassLong(
      HWND hWnd,//дескриптор окна
      int nIndex,     //индекс значения, которое
                      //необходимо изменить
      LONG dwNewLong  //новое значение
);
LONG SetWindowLong(
      HWND hWnd,//дескриптор окна
      int nIndex,     //индекс значения, которое
                      //необходимо изменить
      LONG dwNewLong  //новое значение
);
```

Параметр nIndex функции SetClassLong() может принимать следующие значения:

- GCL_HBRBACKGROUND – изменить дескриптор кисти цвета фона;
- GCL_HCURSOR – изменить дескриптор курсора;
- GCL_HICON – изменить дескриптор пиктограммы;
- GCL – HICONSM – изменить дескриптор малой пиктограммы;
- GCL_MENUNAME – изменить адрес ресурса меню;
- GCL_STYLE – изменить стиль класса окна;
- GCL_WNDPROC –изменить адрес оконной процедуры, связанной с окном.

Например, для изменения дескриптора кисти цвета фона необходимо ввести следующие строки:

```
Case WM_CREATE:
    SeClassLong(hWnd,GCL_HBRBACKGROUND,
              (LONG)CreateSolidBrush(RGB(0,250,250)));
    return TRUE;
   //…
```

1.3. Создание и показ окна

Для создания любых окон, в том числе и окон элементов управления, используется функция Windows CreateWindowEx(), прототип которой приведен ниже:

```
HWND CreateWindowEx(
    DWORD dwExStyle,      //расширенный стиль окна,
                          //применяется совместно
                          //со стилем dwStyle
    LPCTSTR lpClassName,  //адрес строки с именем
                          //зарегистрированного класса,
                          // либо имя одного из
                          //предопределенных классов
    LPCTSTR lpWindowName  //адрес строки с
                          //заголовком главного окна
    DWORD dwStyle,   //стиль окна
    int x            //горизонтальная позиция верхнего
                     //левого угла окна
    int y,           //вертикальная позиция верхнего
                     //левого угла окна
    int nWidth, //ширина окна в пикселях
```

```
int nHeight,     //высота окна в пикселях
HWND hWndParent,//дескриптор родительского окна
HMENU hMenu,     //дескриптор меню окна или
                 //идентификатор элемента управления
HINSTANCE hInstance,  //дескриптор
                      //экземпляра приложения
LPVOID lpParam        //указатель на
                      //дополнительные данные
);
```

Первый параметр – dwExStyle – задает расширенный стиль окна, который можно задать одним или несколькими флагами, используя побитовую операцию ИЛИ. Флаги расширенных стилей:

- WS_EX_MDICHILD – создать дочернее окно многодокументного интерфейса;
- WS_EX_TOOLWINDOW – создать окно с инструментами, предназначенное для реализации плавающих панелей инструментов;
- WS_EX_ACCEPTFILES – создать окно, которое принимает перетаскиваемые файлы.

Расширенный стиль dwExStyle применяется совместно со стилем, который определяется четвертым параметром dwStyle. Полный список расширенных стилей см. в MSDN.

Второй параметр – lpClassName – указатель на строку, содержащую имя зарегистрированного функцией RegisterClassEx() класса, либо имя одного из предопределенных классов.

Вызывая функцию CreateWindowEx() многократно, можно создать много окон данного класса, различающихся, например, размерами и местоположением на экране. Окна предопределенных классов необходимо создавать как дочерние. Предопределенные классы окон, реализующие различные элементы управления:

- BUTTON – кнопки, группы, флажки, переключатели или пиктограммы (префикс в обозначении стиля – BS_);
- COMBOBOX – комбинированный список с полем редактирования в верхней части или выпадающий список выбора (элементы LISTBOX и EDIT) (префикс в обозначении стиля – CBS_);
- EDIT– поле редактирования , предназначенное для ввода текста с клавиатуры (префикс в обозначении стиля –ES_);
- LISTBOX – элемент управления списком строк, из которого можно выбрать любую строку или несколько строк (стили LBS_OWNERDRAWFIXED и LBS_OWNERDRAW позволяют управлять видом строки) (префикс в обозначении стиля –LBS_);
- MDICLIENT – клиентское окно многодокументного интерфейса;
- RICHEDIT – элемент управления в дополнение к EDIT – позволяет редактировать текст с разными шрифтами и стилями (префикс в обозначении стиля – ES_);
- RICHEDIT_CLASS – усовершенствованная версия RICHEDIT (префикс в обозначении стиля – ES_);

- SCROLBAR – элемент управления полосой прокрутки (слайдер) (префикс в обозначении стиля – SBS_);
- STATIC – элемент управления статическим текстом – используется для размещения в окне текста, рамок или как контейнер для других элементов управления (префикс в обозначении стиля – SS_).

Третий параметр – lpWindowName функции CreateWindowEx() – определяет адрес строки с именем окна. Место отображения имени зависит от вида окна, например:

- главное окно приложения – имя выводится в верхнюю часть окна как заголовок окна;
- дочернее окно класса BUTTON – имя размещается по центру кнопки.

Четвертый параметр dwStyle определяет стиль окна. С помощью стиля задаются такие характеристики окна, как вид окружающей его рамки, наличие системного меню, присутствие линеек вертикальной и горизонтальной прокрутки и т.п. Побитовая операция ИЛИ (знак |) позволяет набрать требуемый комплект свойств. Обычно главное окно описывается константой WS_OVERLAPPEDWINDOW(0x00CF0000). Это перекрывающееся окно с заголовком, системным меню, кнопками минимизации и максимизации, кнопкой закрытия окна и «толстой» рамкой, позволяющей изменять размеры окна.

Пятый параметр x и шестой параметр y – экранные координаты верхнего левого угла главного окна, т.е. координаты относительно начала экрана в пикселях. Для дочерних окон и элементов управления эти координаты отсчитываются относительно левого верхнего угла родительского окна и измеряются в пикселях. Если позиция x не важна, то можно установить значение x равным CW_USEDEFAULT – значение по умолчанию. В этом случае параметр y игнорируется.

Седьмой и восьмой параметры nWidth и nHeight – ширина и высота окна в пикселях. Если параметрам nWidth и nHeight присвоено значение CW_USEDEFAULT, то для них будут использоваться значения по умолчанию.

Девятый параметр hWndParent – это дескриптор родительского окна; для главного окна, у которого нет родителя, используется константа HWND_DESKTOP или NULL.

Десятый параметр hMenu функции CreateWindowEx() содержит дескриптор меню окна или *идентификатор элемента управления*. Интерпретация значения параметра hMenu зависит от вида окна:

- приложение использует меню, определенное полем lpszMenu структурной переменной типа WNDCLASSEX (класс окна). В этом случае параметру hMenu необходимо присвоить значение NULL;

- приложение создает элемент управления как дочернее окно. В этом случае параметру hMenu присваивается целочисленное значение, используемое далее как *идентификатор* созданного элемента (этот идентификатор будет содержаться в сообщении WM_COMMAND, поступающего от созданного элемента управления).

Одиннадцатому параметру hInstance должно быть присвоено значение дескриптора экземпляра приложения (значение аргумента hInstance главной функции WinMain).

Последний параметр lpParam функции CreateWindowEx() – указатель на дополнительные данные, передаваемые окну в момент его создания при обработке сообщения WM_CREATE, или значение NULL.

В сообщениях WM_CREATE и WM_NCCREATE параметр lpParam содержит указатель на структуру CREATESTRUCT, в которую будут занесены параметры функции CreateWindowEx().

Символическое обозначение NULL ((void*)0) используется в тех случаях, когда для некоторой функции надо указать нулевое значение параметра, являющегося указателем.

Функция CreateWindowEx() при успешном завершении возвращает дескриптор созданного окна, т.е. локальную переменную hWnd. Дескриптор окна hWnd передается как параметр в функцию ShowWindow(), которая организует вывод созданного окна на экран. Функция ShowWindow() имеет следующий прототип:

```
BOOL ShowWindow(
       HWND hWnd,     // дескриптор окна
       int nCmdShow   // вид показанного окна
);
```

Параметр *nCmdShow* определяет, в каком виде будет показано окно. При начальном отображении главного окна рекомендуется присваивать второму параметру значение, которое было получено через параметр *nCmdShow* главной функции WinMain.

Вызов ShowWindow() влечет за собой генерацию сообщений WM_SIZE и WM_MOVE. При обработке сообщения WM_SIZE система всегда генерирует сообщение WM_PAINT.

1.4. Обработка сообщений

В ООП управляющие данные принято называть сообщениями. Эта терминология используется и в Windows. Сообщения являются реакцией системы на происходящие в ней события и единственным средством связи окна (и всего приложения) с операционной системой.

Коды сообщений Windows имеют единый формат. Это 32-битные целые без знака (для системных сообщений они принимают значения от 1 до 0x3FF), которым для лучшей читаемости присвоены параметрические имена, т.е. каждому коду соответствует символическая константа. Например, аппаратные сообщения:

- WM_MOUSMOUVE – 200h (реакция на перемещение мыши);
- WM_LBUTTONDOWN – 201h (реакция на нажатие левой клавиши мыши);
- WM_TIMER – 113h (реакция на срабатывания таймера);
- WM_MOVSEACTIVATE–0021h (активизировать мышь).

Сообщения могут возникать и в результате программных действий системы. Например, при создании и выводе главного окна система посылает в приложение следующие сообщения:

- WM_GETMINMAXINFO (для уточнения размеров окна);
- WM_ERASEBKGND (при заполнении окна цветом фона);
- WM_SIZE (оценка размера рабочей области);
- WM_PAINT (перерисовка окна).

Сообщения могут создаваться в прикладной программе и посылаться в Windows для того, чтобы система выполнила некоторые действия, например, заполнила информацией окно со списком или сообщила о состоянии элемента управления. Номера таких сообщений превышают значение 0x3FF.

Сообщения могут создаваться и непосредственно программистом.

Для каждого потока в Win32 создается своя очередь сообщений, которая не имеет фиксированной длины. Сообщения из системной очереди не передаются в приложение в целом, а распределяются по его потокам.

Главный цикл обработки сообщений в простейшем виде состоит из следующих предложений языка:

```
2while(GetMessage(&Msg, NULL, 0, 0))
   {
      TranslateMessage(&Msg);
      DispatchMessage(&Msg);
   }
```

Назначение цикла обработки сообщений – получить сообщения, поступающие в приложение, и вызвать в ответ на каждое сообщение оконную функцию. Задача оконной функции – выполнить требуемую обработку поступившего сообщения.

В цикле вызывается функция Windows GetMessage(), которая анализирует очередь сообщений своего потока и, если в очереди обнаруживается сообщение, изымает его и переносит в структурную переменную типа MSG, предназначенную для приема сообщений.

Структура типа MSG описана в файле winuser.h следующим образом:

```
typedef struct tagMSG{
   HWND hWnd;           //дескриптор (хэндл) окна,
                        //которому передано сообщение
   UINT message;        //код данного сообщения
                        //(идентификатор)
   WPARAM wParam;       //дополнительная информация
   LPARAM lParam;       //дополнительная информация
   DWORD time;          //время отправления сообщения
   POINT pt;            //позиция курсора мыши на момент
```

```
                        //отправления сообщения
}Msg; //новое имя для типа tagMSG.
```

Интерпретация параметров wParam и lParam зависит от кода сообщения (message).

Тип данных POINT используется для представления точки парой ее координат и описан следующим образом:

```
typedef struct tagPOINT{
     LONG x;
     LONG y;
     } POINT;
```

Из структуры MSG следует, что содержимое сообщения представляет собой пакет из шести данных. Прототип функции GetMessage() имеет следующий вид:

```
BOOL GetMessage(
   LPMSG lpMsg,  //указатель на структуру переменную типа MSG
   HWND hWnd,    //дескриптор окна, принимающего сообщение
   UINT uMsgFilterMin,    //максимальный номер сообщения
   INT uMsgFilterMax      //минимальный номер сообщения
    );
```

Параметр lpMsg задает адрес структурной переменной типа MSG, в которую помещается выбранное из очереди сообщение.

Параметр hWnd содержит дескриптор окна, принимающего сообщение. Если этот параметр равен NULL, функция GetMessage() работает со всеми сообщениями данного приложения.

Параметры uMsgFilterMin и uMsgFilterMax указывают соответственно минимальный и максимальный номера принимаемого сообщения. Если в качестве этих параметров указать константы WM_KEYFIRST и WM_KEYLAST, функция будет забирать из очереди только сообщения, генерируемые клавиатурой. Константы WM_MOUSFIRST и WM_MOUSLAST позволяют работать только с сообщениями от мыши. Чтобы исключить фильтрацию сообщений, необходимо, чтобы оба параметра были равны нулю.

Функция GetMessage() завершится с возвратом значения TRUE лишь после того, как очередное сообщение попадет в структурную переменную Msg.

Далее вызываются функции Windows TranslateMessage() и DispatchMessage(), которые имеют следующие прототипы:

```
BOOL  TranslateMessage(CONST MSG* lpMsg);
LRESULT DispatchMessage(CONST MSG* lpMsg);
```

Функция TranslateMessage() преобразует аппаратные сообщения от клавиатуры WM_KEYDOWN (клавиша нажата) и WM_KEYUP (клавиша отпущена) в символьное сообщение WM_CHAR. Параметр wParam этого сообщения содержит код символа. Сообщение WM_CHAR помещается в очередь, и на следующей итерации цикла функция GetMessage() извлекает его для последующей обработки. Вызов функции TranslateMessage() необходим только в тех приложениях, которые обрабатывают ввод данных с клавиатуры.

Далее функция DispatchMessage() вызывает оконную функцию того окна, которому предназначено данное сообщение, и передает ей содержимое сообщения из структурной переменной типа Msg (первые четыре поля). Если программе необходимы элементы time и pt, то их можно извлечь непосредственно из структурной переменной. После того как оконная функция обработает сообщение, возврат из нее приводит к возврату из функции DispatchMessage() на продолжение цикла while.

Если функция GetMessage() обнаруживает в очереди сообщение WM_QUIT (код 0x12), то она завершается с возвратом значения FALSE, что приводит к завершению цикла и переходу на предложение return Msg.wParam; т.е. к завершению главной функции и всего приложения.

Если функция GetMessage() не обнаруживает сообщений в очереди потока, система останавливает выполнение данного потока и переводит его в «спящее» состояние. Такое состояние потока не потребляет процессорного времени и не тормозит работу системы. Поток возобновит свою работу, как только в очереди появится сообщение.

Вместо функции GetMessage() часто используют функцию PeekMessage(), которая имеет следующий прототип:

```
BOOL PeekMessage(
    LPMSG lpMsg,        //адрес структурной переменной
                        //с сообщением
    HWND hwnd,          //дескриптор окна
    UINT wMsgFilterMin,     //Min номер выбираемого
                            //сообщения
    UINT wMsgFilterMax,     //Max номер выбираемого
                            //сообщения
    UINT wRemoveMsg         //флаг удаления сообщения
);
```

Первые четыре параметра функции PeekMessage() совпадают с параметрами функции GetMessage().

Пятый параметр wRemoveMsg определяет, как должно быть выбрано сообщение из очереди, а именно:

- PM_NOREMOV – сообщение выбирается и не удаляется из очереди;
- PM_REMOV – сообщение выбирается и удаляется из очереди.

Функция PeekMessage() не ждет, когда сообщение поступит в очередь сообщений, а сразу возвращает управление. Если сообщения в очереди нет, то функция возвращает нуль, а если сообщение находится в очереди, оно помещается в структурную переменную типа MSG и удаляется из очереди сообщений (флаг PM_REMOV) или не удаляется (флаг PM_NOREMOV).

Функция PeekMessage() часто используется для оптимизации работы программы. Если сообщение не находится в очереди сообщений, то в свободное время можно выполнять какую-либо другую работу. Этот прием часто встречается в программах, выполняющих графическую анимацию, и более эффективен, чем использование сообщений от таймера.

Главный цикл обработки сообщений с применение функции PeekMessage() может выглядеть следующим образом:

```
while(1)
   {
     if(PeekMessage(&Msg,NULL,0,0, PM_REMOV))
     {
        if(Msg.message==WM_QUIT) break;
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
     }
   }
```

1.5. Оконная функция

Оконная функция – это функция обратного вызова, предназначенная для обработки сообщений, адресованных окну того оконного класса, в котором содержится ссылка на данную процедуру.

Функции обратного вызова – это функции, которые вызывает сама операционная система. Компилятор определяет их по спецификатору CALLBACK. Оконная функция получает четыре параметра, а ее заголовок имеет стандартный синтаксис:

```
LRESULT CALLBACK Имя_функции(
    HWND hWnd,          //дескриптор (хэндл) окна, которому
                        //предназначено данное сообщение
    UINT uMsg           //код пришедшего сообщения
    WPARAM wParam    //дополнительная информация
                        //о сообщении
    LPARAM lParam    //дополнительная информация
                        //о сообщении
);
```

Имя функции может быть произвольным, но для главного окна приложения обычно используется имя WndProc. В теле оконной функции после объявления необходимых локальных переменных используется оператор switch…case. Например:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg,
                         WPARAM wParam, LPARAM lParam)
     {
          switch(Msg)
          {
               case WM_CREATE:
               //…
               return TRUE;
               case WM_KEYDOWN:
               /*
                Обрабатываем сообщение WM_KEYDOWN
                (нажатие клавиши)
               */
                    switch(wParam)
                    {
                    /* Клавиша Esc */
                    case VK_ESCAPE:
```

```
                    SendMessage(hWnd,WM_CLOSE,0,0);
                    break;
                }
                break;
            case WM_DESTROY:
            /*
            Функция PostQuitMessage выполняет только одно
            действие, ставит в очередь сообщение WM_QUIT.
            Параметр у этой функции - код возврата, кото-
            рый помещается в Msg.wParam и возвращается
            главной функцией
            */
                PostQuitMessage(0);
                break;
            default:
            /*Обработка прочих сообщений по умолчанию*/
          return DefWindowProc(hWnd,msg,wParam,lParam);
  }
    return 0L;
}
```

Реально в программе требуется обрабатывать конкретные сообщения, а для обработки прочих сообщений предназначена функция DefWindowProc() (Default Windows Procedure).

Единственное сообщение, не входящее в «юрисдикцию» DefWindowProc, –сообщение об уничтожении окна WM_DESTROY, поэтому его обработка должна всегда присутствовать в оконной процедуре.

Обычно приложения завершают свою работу по команде пользователя, например ALT+F4. В этом случае Windows убирает с экрана окно приложения и посылает сообщение WM_DESTROY не в очередь приложения, а в оконную функцию с непосредственным ее вызовом. Обработка этого сообщения должна состоять в освобождении ресурсов Windows. Например, при обработке сообщения WM_CREATE приложение могло создать и использовать в программе кисти, перья, шрифты, установить таймеры, динамически выделить память т.п. При завершении приложения эти ресурсы необходимо освободить. Далее необходимо вызвать функцию PostQuitMessage(0);

Все сообщения, обработанные в оконной процедуре, должны возвращать стандартное значение – нуль по выходу из процедуры. Однако бывают случаи, когда требуется возвратить единицу или -1. Например, после обработки сообщения WM_NCCREATE, которое приходит перед сообщением WM_CREANT, следует вернуть единицу.

Оконная функция может вызываться двумя способами: функцией DispatchMessage() и непосредственно программами Windows.

1.6. Сообщения Windows

Процедура формирования аппаратного сообщения включает следующие этапы:

1) аппаратное прерывание;

2) формирование драйвером пакета данных;

3) установка сообщения в системную очередь;

4) пересылка сообщения в очередь потока;

5) вызов оконной функции.

Сообщение WM_MOUSMOVE генерируется системой при перемещении мыши по клиентской области окна.

Структурная переменная Msg (тип MSG) при формировании сообщения WM_MOUSEMOVE заполнится следующей информацией:

1. Msg.hWnd – дескриптор окна под курсором мыши;
2. Msg.message – код сообщения (0x200);
3. Msg.wParam – комбинация битовых флагов, индицирующих состояния клавиш мыши (нажаты/не нажаты), а также клавиш Ctrl и Shift ;
4. Msg.lParam – позиция курсора мыши относительно клиенской области окна;
5. Msg.time – время отправления сообщения;
6. Msg.pt – позиция курсора мыши относительно границ экрана.

Вообще манипуляции с мышью порождают следующие сообщения:

- WM_LBUTTONDOWN (201h) – нажатие левой кнопки мыши;
- WM_LBUTTONUP (202h) – отпускание левой кнопки мыши;
- WM_RBUTTONDOWN (204h) – нажатие правой кнопки мыши;
- WM_RBUTTONUP – отпускание правой кнопки мыши;
- WM_LBUTTONDBLCLK (203h) – двойное нажатие левой кнопки мыши.

Для всех вышеперечисленных сообщений структурная переменная типа MSG заполняется одинаковой информацией, т.е. пакеты данных этих сообщений не различаются.

Параметр lParam содержит координаты курсора мыши, а именно, младшее слово lParam содержит координату **x**, а старшее слово lParam – координату **y**. Отсчет координат ведется относительно левого верхнего угла клиентской области окна. Для выделения значений, находящихся в старшем и младшем словах параметра lParam, предназначены макрокоманды LOWORD и HIWORD.

Значение параметра wParam показывает состояние кнопок мыши, а также клавиш Shift и Ctrl. Если кнопка мыши или клавиша нажата, то значения соответствующих им полей устанавливаются равными единице, иначе – нулю. Параметр wParam можно проверить при помощи соответствующих битовых масок, используя побитовую операцию ИЛИ. Битовые маски:

MK_LBUTTON – левая клавиша нажата;

MK_MBUTTON – средняя клавиша нажата;

MK_RBUTTON – правая клавиша нажата ;

MK_SHIFT – клавиша Shift нажата;

MK_CONTROL – клавиша Ctrl нажата.

Например, как можно определить, была ли нажата клавиша Shift во время нажатия левой кнопки мыши (сообщение WM_LBUTTONDOWM):

```
Case WM_LBUTTONDOWM:
    //…
    if(wParam & MK_SHIFT)
       {
          /*нажата клавиша Shift*/
       }
    //…
```

Двойной щелчок левой клавишей мыши порождает четыре сообщения:

```
WM_LBUTTONDOWN;
WM_LBUTTONUP;
WM_LBUTTONDBLCLK;
WM_LBUTTONUP.
```

Можно обрабатывать все четыре, а можно только WM_LBUTTONDBLCLK. Необходимо отметить, что окно будет получать сообщение о двойном щелчке (DBLCLK) только в том случае, если стиль класса окна содержит флаг CS_DBLCLK.

Сообщения от клавиатуры:

- WM_KEYDOWN – нажатие несистемной клавиши (любая клавиша без сопровождения с Alt);
- WM_KEYUP – отпускание несистемной клавиши;
- WM_SYSKEYDOWN – нажатие системной клавиши (любая клавиша с сопровождением клавиши Alt);
- WM_SYSKEYUP – отпускание системной клавиши.

Для всех сообщений от клавиатуры параметр lParam содержит семь полей:

1) поле (от 0 до15 бита) – содержит счетчик повторений данного сообщения: равен либо единице, либо числу нажатий в режиме автоповтора, а для WM_KEYUP всегда равен единице;

2) поле (от 16 до 23 бита) – содержит скан-код;

3) поле (24-й бит) – флаг расширенной клавиатуры, бит устанавливается в 1 для функциональных клавиш и клавиш Alt и Ctrl на правой стороне клавиатуры;

4) поле (от 25 до 28 бита) – не используется;

5) поле (29-й бит) – установленный бит означает, что клавиша Alt нажата (бит всегда сброшен для сообщений WM_KEYDOWN и WM_KEYUP, т.е. 0);

6) поле (30-й бит) – установленный бит означает, что клавиша была нажата до отправки сообщения, а 0 – отпущена;

7) поле (31-й бит) – флаг нового состояния клавиатуры, т.е. 0 – клавиша нажата, 1 – отпущена.

Параметр wParam содержит виртуальный код клавиши, который идентифицирует нажатую или отпущенную клавишу. Например, виртуальные десятичные коды для следующих клавиш (полный список см. в MSDN.):

- Shift – идентификатор VK_SHIFT (код – 16);
- Ctrl – идентификатор VK_CONTROL (код – 17);
- Alt – идентификатор VK_MENU (код – 18);
- Esc – идентификатор VK_ESCAPE (код – 27);
- F1…F12 – идентификаторы VK_F1…VK_F12, (коды – 112…123).

Сообщение WM_CHAR обрабатывается при вводе информации от символьных клавиш. Функция TranslateMessage() преобразует аппаратное сообщение от клавиатуры в символьное сообщение WM_CHAR, которое содержит код символа (wParam). Сообщение WM_CHAR помещается в очередь, а на следующей итерации цикла функция GetMessage() извлекает его для последующей обработки.

Параметр lParam сообщения WM_CHAR имеет то же значение, что и параметр lParam породившего его аппаратного сообщения.

Параметр wParam сообщения WM_CHAR содержит код символа. Например, если нажимается и отпускается клавиша A в нижнем регистре, то генерируются три сообщения в следующем порядке:

1. WM_KEYDOWN – виртуальная клавиша A;
2. WM_CHAR – ANSI код символа ‘a’;
3. WM_KEYUP – виртуальная клавиша A.

Сообщение WM_COMMAND формируется во время нажатия левой клавиши мыши над строкой меню, т.е. аппаратное прерывание поглощается системой и вместо сообщения WM_LBUTTONDOWN формируется сообщение WM_COMMAND. При выборе пунктов системного меню вместо сообщения WM_COMMAND системой генерируется сообщение WM_SYSCOMMAND.

Непосредственно перед активизацией главного меню или всплывающего меню системой посылаются сообщения WM_INITMENU или WM_INITMENUPOPUP соответственно. Они позволяют приложению изменить меню перед тем, как оно будет отображено на экране. Для обоих сообщений параметр wParam содержит дескриптор активизируемого меню, а параметр lParam – следующую информацию:

- в младшем слове lParam – относительная позиция пункта меню (отсчет от нуля), который открывает подменю;
- в старшем слове lParam – TRUE, если раскрывающее меню – меню окна.

Источниками сообщения WM_COMMAND могут быть:

1) пункты меню, определяющие команды, при этом параметры wParam и lParam принимают следующие значения:

- младшее слово wParam содержит идентификатор пункта меню (wid);
- старшее слово wParam содержит нуль (wCmd);
- параметр lParam содержит нуль.

2) быстрые клавиши (keyboard accelerator), при этом параметры wParam и lParam принимают следующие значения:

- младшее слово wParam содержит идентификатор быстрой клавиши (wid);
- старшее слово wParam содержит единицу (wCmd);
- параметр lParam содержит нуль.

3) элементы управления (кнопки, списки, текстовые поля и т.д.), при этом параметры wParam и lParam принимают следующие значения:

- младшее слово wParam содержит идентификатор элемента управления (wid);
- старшее слово wParam содержит код извещения, т.е. действия, выполненные над элементом управления (wCmd);
- параметр lParam содержит дескриптор элемента управления (hWnd).

Приложение должно разбирать сообщение WM_COMMAND на составные части следующим образом:

```
case WM_COMMAND:
   wid = LOWORD(wParam);
   hWnd = (HWND)(UINT)lParam;
   wCmd = HIWORD(wParam);
   //…
```

Сообщение WM_CLOSE генерируется, когда пользователь щелкает мышью по кнопке закрытия окна или нажимает комбинацию клавиш Alt+F4, при этом окно еще не разрушено. Если обработка этого сообщения отсутствует, то функция DefWindowPro() вызывает по умолчанию функцию DestroyWindow(), которая посылает окну (потом дочерним окнам, если они есть у этого окна) сообщение WM_DESTROY. Функция DestroyWindow() завершит свою работу только после уничтожения всех дочерних окон. Но если необходимо предусмотреть вывод предупреждающих сообщений типа «Вы уверены?», то это можно сделать, когда окно еще не разрушено, т.е. при обработке сообщения WM_CLOSE, например:

```
case WM_CLOSE:
   if(MessageBox(hWnd," Вы уверены?", " ",
               MB_YESNO | ICONQUESTION) == IDYES)
      DestroyWindow(hWnd);
   break;
```

Сообщение WM_SIZE посылается окну после изменения его размера. Параметр wParam этого сообщения содержит следующие значения (полный список см. в MSDN):

- SIZE_MAXIMIZED – окно было развернуто;
- SIZE_MINIMIZED – окно было свернуто;
- SIZE_RESTORED – окно было изменено, но не SIZE_MAXIMIZED и не SIZE_MINIMIZED.

Параметр lParam в младшем слове содержит новую ширину клиентской области окна, в старшем слове – новую высоту.

Сообщение WM_MOVE посылается окну после его перемещения. Параметр lParam этого сообщения содержит следующие значения:

- младшее слово lParam содержит x-координату левого верхнего угла клиентской области окна;
- старшее слово lParam содержит y- координату левого верхнего угла клиентской области окна.

Сообщение WM_CREATE генерируется системой в процессе создания окна. Перехватив это сообщение, можно выполнить некоторые инициализирующие действия, например: установить системный таймер, загрузить требуемые ресурсы (шрифты, растровые изображения), открыть файлы с данными и т.д.

Параметр lParam сообщения WM_CREATE содержит указатель на структуру типа CREATESTRUCT, параметры которой аналогичны параметрам функции CreateWindowEx().

Сообщение WM_CREATE не поступает в очередь сообщений, т.е. Windows непосредственно вызывает оконную функцию WndProc и передает ей необходимые параметры. Следует иметь в виду, что во время обработки сообщения, например WM_MOUSMOVE, все его содержимое находится в структурной переменной типа MSG. При обработке же сообщения WM_CREATE мы имеем дело только с параметрами, переданными Windows в оконную процедуру.

Аналогично, помимо очереди сообщений, обрабатываются другие сообщения, например:

- WM_INITDIALOG (инициализация диалога);
- WM_SYSCOMMAND (выбор пунктов системного меню);
- WM_DESTROY (уничтожение окна).

Сообщение WM_PAINT уведомляет программу, что часть или вся клиентская область окна недействительна (invalid) и ее следует перерисовать. Клиентская область окна становится недействительной, и система генерирует сообщение WM_PAINT в следующих случаях:

- при создании окна;
- при минимизации окна и последующем его разворачивании;
- при перемещении порожденного окна по пространству главного окна;
- при изменении размеров и местоположения окна и т.д.

Общее правило рисования заключается в том, что вывод в окно приложения любых графических объектов должен выполняться исключительно в блоке обработки сообщения WM_PAINT. Только в этом случае графическое содержимое клиентской области окна не будет теряться, например, при перечисленных выше случаях. При этом восстановление нерабочей области окна берет на себя система.

Обработка сообщения WM_PAINT связана с так называемым контекстом устройства. Контекст устройства (device context) описывает физическое устройство вывода информации, например дисплей или

принтер. Это некоторая внутренняя структура данных, в которой хранятся, например, текущее значение режимов рисования, дескрипторы инструментов рисования: кисти, перья, шрифты, и др. Все графические функции используют контекст устройства для опредения режимов рисования и характеристик применяемых ими инструментов.

Обработку сообщения WM_PAINT рекомендуется всегда начинать с вызова функции BeginPaint(), которая имеет следующий прототип:

```
HDC BeginPaint(HWND hWnd,
               LPPAINSTRUCT lpPaintStruct);
```

Первый параметр hWnd представляет собой дескриптор того устройства вывода, в котором предполагается рисовать, т.е. для которого требуется контекст устройства. Второй параметр lpPaintStruct – адрес структурной переменной типа PAINTSTRUCT, которая заполняется Windows всякий раз, когда приложение перехватывает обработку сообщения WM_PAINT.

Структура типа PAINTSTRUCT описана в файле winuser.h и содержит следующие элементы:

```
typedef struct tagPAINTSTRUCT
{
   HDC hDc;     //дескриптор контекста устройства
   BOOL fErase; //флаг перерисовки фона клиентской
                //области окна
   RECT rcPaint;     //область вырезки, т.е.границы
                     //недействительного прямоугольника
   BOOL fRestor;     //зарезервировано
   BOOL fIncUpdate;  //зарезервировано
   BYTE rgbReserved[32];  //зарезервировано
} PAINSTRUCT;
```

Флаг перерисовки окна fErase обычно равен TRUE. Это означает, что обновляемый регион помечен для стирания и система Windos обновляет фон клиентской области. По умолчанию для этого используется кисть, заданная полем wc.hbrBackground.

Область вырезки rcPaint представляет собой структуру типа RECT, которая содержит координаты обновляемого прямоугольника в пикселях относительно левого верхнего угла клиентской области.

Координаты прямоугольника определяются системой или задаются при вызове функций InvalidateRect() или InvalidateRgn(). Один из параметров этих функций разрешает или запрещает стирание фона. Если задано стирание фона, то функция BeginPaint() посылает оконной процедуре сообщение WM_ERASEBKGND. Приложение может обрабатывать это сообщение, чтобы отобразить однородный или растровый фон. Обычно это сообщение обрабатывается по умолчанию функцией DefWindowProc(). До своего завершения функция BeginPaint() посылает оконной процедуре сообщение WM_NCPAINT, которое определяет обновление неклиентской области окна.

Структура, описывающая прямоугольник (rectangle), определена в заголовочном файле следующим образом:

```
typedef struct tagRECT {
    LONG left; //x-координата верхнего левого угла
     //прямоугольнка
    LONG top;  //y-координата верхнего левого угла
     //прямоугольника
    LONG right;//x-координата правого нижнего угла
     //прямоугольника
    LONG bottom//y-координата правого нижнего
     //угла прямоугольника
    }RECT;
```

Переменные типа RECT часто используются в приложениях, так как области экрана в Windows всегда имеют прямоугольную форму.

По окончании работы с графическими функциями в блоке обработки сообщения WM_PAINT необходимо освободить полученный контекст устройства с помощью функции EndPaint(), которая имеет следующий прототип:

```
BOOL EndPaint(HWND hWnd,CONST PAINTSTRUCT* lpPaintStruct);
```

Типичная обработка сообщения WM_PAINT:

```
Case WM_PAINT:
    hDc=BeginPaint(hWnd, &ps);

    //использование функций GDI
    //…
    EndPaint(hWnd, &ps);
    //…
```

Основным средством программного взаимодействия между разными окнами приложения является посылка сообщений. Для отправки сообщений используется функция SendMessage(), которая имеет следующий прототип:

```
LRESULT SendMessage(
HWND hWnd,      //дескриптор окна получателя
UINT Msg,        //код сообщения
WPARAM wParam,   //первый параметр сообщения
LPARAM lParam    //второй параметр сообщения
);
```

Функция SendMessage() посылает сообщение указанному окну или нескольким окнам. Параметры функции те же, что и параметры, передаваемые в оконную функцию. Когда приложение вызывает функцию SendMessage(), то в свою очередь Windows вызывает оконную процедуру с этими четырьмя параметрами. После того, как оконная процедура завершит обработку сообщения, система передаст управление инструкции следующей за вызовом функции SendMessage().

1.7. Создание приложения с главным окном

Текст программы размещается в листинге 1.1, в котором расположены две функции: WinMain и WndProc. Данное приложение создает окно со строкой заголовка, стандартным набором управляющих кнопок и выводит в нем текущую дату и время и периодически, каждую секунду, обновляет эту информацию.

1.7.1. Окна сообщений

Окна сообщений – это средства диалога системы и прикладной программы с пользователем. В прикладных программах окна сообщений удобно использовать, например, для вывода результатов вычислений или информации о ходе выполнении программы.

Для исследования *адресного пространства программы* воспользуемся окном сообщений. Выведем содержимое сегментных регистров: команд (CS), данных (DS) и стека (SS), а также смещение главной функции WinMain и смещение строки с именем класса.

В главную функцию вставим следующий фрагмент (см. листинг 1.1):

```
TCHAR szText [80];
USHORT regCS,regDS,regSS;
__asm{
       mov regCS,CS
       mov regDS,DS
       mov regSS,SS
      }
wsprintf(szText, _T(" CS=%X, DS=%X, SS=%X\n
          WinMain=%X\n szClassName=%X"),
          regCS, regDS, regSS, WinMain, szClassName);
MessageBox(NULL, szText, _T(" Регистры"),
                                        MB_ICONINFORMATION);
```

Результат выполнения этого фрагмента – окно сообщения (рис. 1)

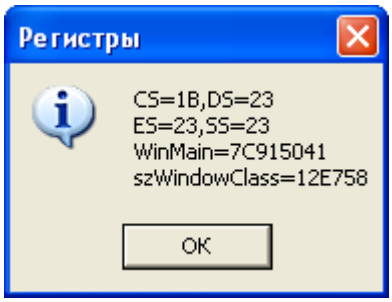


Рис. 1

Функция wsprintf() имеет следущий прототип:

```
int wsprintf(LPSTR lpBuffer,
             LPCTSTR lpszFormatString,
             [arguments]
            );
```

Функция wsprintf() форматирует данные ([arguments]) с использованием заданной строки формата (lpszFormatString) и размещает полученные данные в символьном буфере (lpBuffer).

Второй аргумент lpszFormatString – это указатель на строку с нулевым символом в конце, которая содержит формат выходного результата. Синтаксис произвольной строки формата:

```
%[-][#]<field_width>[.<field_precesion>]<field_type>,
```

где [–] – префикс, изменяет выравнивание поля по правому краю на выравнивание поля по левому краю; [#] – модификатор шестнадцатеричных данных – добавляет перед числом 0X; [<field_width>] – ширина поля, т.е. минимальное число символов, которые копируются в выходной буфер; <.field_precession> – точка и число – определяет точность представления данных, т.е. минимальное количество цифр, которые копируются в выходной буфер: <field_type> – каким образом интерпретируются предыдущие два поля, например:

```
(szText,"Это строка %S, а это ее адрес %#X",lpStr, lpStr).
```

Третий параметр [arguments] – это список элементов данных, которые должны форматироваться и пересылаться в символьный буфер.

Функция wsprintf() возвращает количество символов, сохраненных в буфере.

Функция MessageBox() создает и выводит на экран в требуемой точке служебное окно с заданным текстом. Прототип этой функции:

```
int MessageBox(
HWND hWnd,      //дескриптор родительского окна
LPCTSTR lpText, //выводимый текст
LPCTSTR lpCaption,   //заголовок окна сообщения
UINT uType           //стиль окна сообщения
);
```

Первый параметр hWnd идентифицирует окно, которое является для окна сообщения родительским. В большинстве случаев этому параметру можно присвоить значение NULL.

Второй параметр lpText представляет собой адрес строки с текстом, выводимым в окно, или саму строку, заключенную в кавычки.

Третий параметр lpCaption – это адрес строки или сама строка с текстом заголовка сообщения.

Четвертый параметр uType определяет вид и поведение окна сообщения, т.е. вид пиктограммы, характеризующей тип сообщения, набор управляющих кнопок, а также кнопку, выбранную по умолчанию. Этот параметр может принимать следующие значения:

- MB_YESNO – окно сообщения содержит две кнопки: Yes и No;
- MB_OKCANCEL – окно сообщения содержит две кнопки: Ok и Cancel;
- MB_YESNOCANCEL – окно сообщения содержит три кнопки: Yes, No и Cansel.
- MB_ICONINFORMATION – пиктограмма со знаком вопроса;
- MB_ICONSTOP – пиктограмма со знаком крестика.

Константы, относящиеся к разным элементам окна, могут объединяться с помощью операции побитового ИЛИ.

Функция MessageBox() возвращает нуль, если недостаточно памяти, в ином случае:

- IDCANCEL – была нажата кнопка Cancel;
- IDNO – была нажата кнопка NO;
- IDOK – была нажата кнопка OK;

- IDES – была нажата кнопка YES.

Полный список констант см. в MSDN.

Windows XP выделяет сегменту команд (CS) селектор 0x1B, а сегментам данных (DS, ES) – селектор 0x23. Необходимо отметить, что базовые линейные адреса сегментов команд и данных равны нулю, виртуальные смещения с которыми работает программа, совпадают с абсолютными линейными адресами. Следовательно, плоское виртуальное адресное пространство программы совпадает с плоским линейным адресным пространством.

1.7.2. Вывод текстовых строк

В рассматриваемом примере (см. п. 1.7.4) рисование на экране сводится к выводу текста текущая дата и время с помощью функции DrawText(), которая имеет следующий прототип:

```
int DrawText(
   HDC hDc,          //дескриптор контекста устройства
   LPCTSTR lpString, //указатель на символьную строку
   int nLenght,      //длина текста
   LPRECT lpRect, //указатель на ограничивающий прямоугольник
   UINT uFormat   //флаги форматирования текста
    );
```

Функция выводит текст из строки lpString в прямоугольную область, заданную структурой типа RECT. Если значение параметра nLenght установить в −1, то функция сама определит длину строки по завершающему нулевому символу.

Для получения размеров клиентской области окна (ограничивающий прямоугольник) следует вызвать функцию GetClientRect():

```
BOOL GetClientRect(
    HWND hWnd,      //дескриптор окна
    LPRECT lpRect   //адрес структурной переменной
                    //типа RECT
    );
```

Функция GetClientRect() копирует параметры прямоугольника, ограничивающего клиентскую часть окна, в структурную переменную типа RECT. Параметры даются относительно левого верхнего угла клиентской области окна, поэтому поля left и top всегда равны нулю, а поля right и bottom содержат ширину и высоту клиентской области в пикселях.

Метод форматирования задается параметром uFormat, который может быть задан, например, комбинацией следующих флагов:

```
DT_SINGLELINE | DT_CENTR | DT_VCENTR
```

где DT_SINGLELINE показывает, что текст будет выводиться в одну строку; DT_CENTR – текст будет выводиться по центру относительно горизонтали прямоугольной области; DT_VCENTR – относительно вертикали

1.7.3. Атрибуты цвета и фона выводимого текста

Цвет объектов задается 32-битным числом (тип COLORREF), например: 0x00bbggrr, где b – синяя, g – зеленая, r – красная составляющая цвета. Для создания цвета можно использовать макрос RGB(R,G,B).

Для установки цвета текста (text color) применяется функция SetTextColor(), прототип которой имеет вид

```
COLORREF SetTextColor(HDC hDc, COLORREF crColor);
```

Для установки цвета фона графического элемента (background color), т.е. цвета, который отображается под каждым символом, используется функция SetBkColor(), имеющая следующий прототип:

```
COLORREF  SetBkColor(HDC hDc, COLORREF crColor);
```

Для обеих функций первый параметр hDc – контекст устройства, второй crColor – цвет текста или фона соответственно. Обе функции возвращают ссылку на предыдущий цвет.

Для установки режима смешивания фона (background mix mode) используется функция SetBkMode, прототип которой

```
int SetBkMode(HDC hDc, int uMode);
```

Второй параметр uMode функции может принимать следующие значения: OPAQUE – непрозрачный режим, т.е. цвет фона графического элемента выводится поверх существующего фона окна; TRANSPARENT – прозрачный, т.е. цвет фона графического элемента игнорируется, а символ выводится на существующем фоне окна.

Если перечисленные выше функции не вызывались, то в контексте устройства будут использоваться по умолчанию следующие значения: цвет текста – черный, цвет фона графического элемента – белый, режим смешивания фона – OPAQUE.

1.7.4.Таймеры Windows

Операционная система Windows предоставляет программисту функции, позволяющие установить в приложении требуемое количество программных таймеров. С помощью таймеров приложение, например, может обеспечить временную синхронизацию и задание временных интервалов.

Функция SetTimer() создает или видоизменяет системный таймер, который формирует сообщение WM_TIMER.

```
UINT SetTimer(
     HWND hWnd,     //Дескриптор окна для которого
                    //создается таймер
     UINT uTimerID, //Идентификатор таймера
     UINT uInterval,//Значение временного интервала
                    //в милисекундах
     TIMERPROC fnTimerProc    //Указатель на функцию
                              //обратного вызова
);
```

Таймер после своей установки начинает периодически с интервалом uInterval генерировать сообщение WM_TIMER, которое поступает в окно(hWnd).

Параметр fnTimerProc представляет собой CALLBACK–функцию, которая должна быть определена в программе и содержать процедуру прикладной обработки прерывания от таймера. Если этот параметр не NULL , то при каждом срабатывании таймера операционная система будет непосредственно, в обход оконной функции, вызывать функцию fnTimerProc. Если учесть, что сообщение WM_TIMER посылается функцией DispatchMessage() в оконную функцию в последнюю очередь, т.е. тогда когда в очереди сообщений приложения нет других сообщений, а CALLBACK– функция вызывается непосредственно. Следовательно, CALLBACK– функция обеспечивает более оперативную обработку сигналов от таймера. Ее прототип:

```
VOID CALLBACK TimerProc(
     HWND hWnd,     //Дескриптор окна
     UINT uMsg,     //WM_TIMER
     UINT_PTR idEvent,   //Идентификатор таймера
     DWORD dwTime        //Количество миллисекунд, прошедее
                         //с момента запуска Windows
);
```

Если сообщение от таймера предполагается обрабатывать посредством оконной функции, то на месте последнего параметра функции SetTimer() указывается NULL.

Функция KillTimer()удаляет таймер.

```
BOOL KillTimer(
     HWND hWnd,          //Дескриптор окна
     UINT_PTR idEvent    //Идентификатор таймера
     );
```

1.7.5. Приложение с главным окном

Эту программу можно использовать в качестве каркаса Windows-приложения с главным окном.

```
/*Операторы препроцессора*/
/*
#define UNICODE
#ifdef UNICODE
#define _UNICODE
#endif
*/
#define STRICT
#include <windows.h>
#include <windowsx.h>
/*
Файл tchar.h состоит из макросов, которые ссылаются на UNICODE
данные и функции, если определен макрос UNICODE, или на ANSI
```

```
данные и функции, если этот макрос не определен, кроме того он
полностью заменяет файл string.h
*/
#include <tchar.h>

/*Прототип оконной функции*/
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
/*
Прототип функции получения текущего времени
и преобразование его в символы
*/
void OutTimeDate(HWND);
/*
Массив для формирования строки - текущие дата и время
*/
TCHAR szCurrentTime[40];
int WINAPI WinMain(HINSTANCE hInstance,
                                HINSTANCE hPrevInstance,
                                LPSTR lpCmdLine, int nCmdShow)
{

     /*Произвольный заголовок окна*/
     TCHAR szTitle[]=_TEXT("ИТМО");
     /*Произвольное имя класса*/
     TCHAR szWindowClass[]=_TEXT("QWERTY");

     /*
     Структурная переменная msg типа MSG для
     получения сообщений
     */
     MSG msg;
     /*
     Структурная переменная wcex типа WNDCLASSEX для задания
     характеристик окна
     */
     WNDCLASSEX wcex;
     HWND hWnd; //Дескриптор главного окна
     /*Проверяем, было ли это проложение запущено ранее*/
     if(hWnd=FindWindow(szWindowClass,NULL))
     {
     /*Проверяем, было ли это окно свернуто в пиктограмму*/
          if(IsIconic(hWnd))
               ShowWindow(hWnd,SW_RESTORE);
          /*Выдвигаем окно приложения на передний план*/
          SetForegroundWindow(hWnd);
          return FALSE;
     }
     /*Обнуление всех членов структуры wcex*/
     memset(&wcex,0,sizeof(WNDCLASSEX));
     /*Регистрируем класс главного окна*/
     wcex.cbSize = sizeof(WNDCLASSEX);
     wcex.style  = CS_HREDRAW | CS_VREDRAW;
     /*Определяем оконную процедуру для главного окна*/
     wcex.lpfnWndProc    = (WNDPROC)WndProc;
     //wcex.cbClsExtra    = 0;
     //wcex.cbWndExtra    = 0;
     wcex.hInstance      = hInstance;//Дескриптор приложения
     /*
     Стандартная пиктограмма, которую можно загрузить функцией
     LoadIcon(hInstance, MAKEINTRESOURCE(IDI_?_?))
```

```
*/
wcex.hIcon = (HICON)LoadImage(hInstance,
                              IDI_APPLICATION,
                          IMAGE_ICON,32,32,0);
/*Стандартный курсор мыши*/
wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
/*
Кисть фона и ее цвет можно определить выражениями:
wcex.hbrBackground=(HBRUSH)(COLOR_APPWORKSPACE+1);
wcex.hbrBackground=(HBRUSH)GetStockObject(LTGRAY_BRUSH);
или с помощью  макроса  GetStockBrush(), в этом случае
необходимо подключить файл windowsx.h
*/
wcex.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
/*wcex.lpszMenuName  = MAKEINTRESOURCE(IDC_MSG_1);*/
/*
Имя класса главного окна
*/
wcex.lpszClassName  = szWindowClass;
/*
Маленькая пиктограмма, wcex.hIconSm, которую
можно загрузить функцией LoadImage()
*/
wcex.hIconSm  = NULL;
if(!RegisterClassEx(&wcex))
     return FALSE;
/*Создаем главное окно*/
hWnd = CreateWindowEx(WS_EX_WINDOWEDGE,szWindowClass,
                    szTitle,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,0,CW_USEDEFAULT,0,
                    NULL, NULL, hInstance, NULL);
if (!hWnd)
     return FALSE;
/*
Исследуем адресное пространство приложения. Выведем со-
держимое сегментных регистров команд, данных и стека, а
также смещение главной функции и строки с именем класса
*/
TCHAR szAsm[80];
USHORT regCS,regDS,regES,regSS;
__asm{
     mov regCS,CS
     mov regDS,DS
     mov regES,ES
     mov regSS,SS
}
wsprintf((LPTSTR)szAsm,_T("CS=%X,DS=%X\nES=%X,SS=%X\n
                    WinMain=%X\nszWindowClass=%X"),
                    regCS,regDS,regES,regSS);
MessageBox(NULL,(LPCTSTR)szAsm,_T("Регистры"),
                                MB_ICONINFORMATION);
/*Делаем окно видимым на экране*/
ShowWindow(hWnd, SW_SHOWNORMAL);
/*
Функция UpdateWindow() вызывает передачу сообщения
WM_PAINT непосредственно оконной процедуре, а
функция InvalidateRect()вызывает постановку
сообщения WM_PAINT в очередь приложения, а там оно обра-
батывается с самым низким приоритетом
```

```
     */
     UpdateWindow(hWnd);
     /*Цикл обработки сообщений*/
     while (GetMessage(&msg, NULL, 0, 0))
          {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }
     /*Код возврата*/
     return (int)msg.wParam;
}
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc (HWND hWnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
     /*Горизонтальный размер главного окна*/
     int xSize=500;
     /*Верикальный размер главного окна*/
     int ySize=300;
     /*Структура PAINTSTRUCT с характеристиками рабочей
     области,заполняется функцией BeginPaint
     */
     PAINTSTRUCT ps;
     /*
     TEXTMETRIC - структура для получения
     характеристик шрифта
     */
     TEXTMETRIC tm;
     /*
     LOGFONT - структура для для создания
     логических шрифтов
     */
     LOGFONT lf;
     static HFONT hFont, hOldFont;
     /*
     RECT - структура описывает прямоугольник
     */
     RECT rect;
     LPMINMAXINFO lpmmi;
     /*Дескриптор контекста устройства*/
     HDC hDc;
     UINT width, height;
     /*Имя шрифта*/
     LPTSTR lpszFace=_TEXT("Times New Roman Cyr");
     switch (message)
     {
      /*Переход по значению кода сообщения(msg)*/
      case WM_CREATE:
     /*
     Только здесь можно произвести модификацию класса
     окна. Например, SetClassLong(hWnd, GCL_HBRBACKGROUND,
               (LONG)CreateSolidBrush(RGB(200,160,255));
     Значение дескриптор экземпляра приложения (hInstance)
     определяется, вызовом одной из следующих функций:
     hInst = GetModuleHandle(NULL);
     hInst = (HINSTANCE)GetClassLong(hWnd,GCL_HMODULE);
     */
          /*Обнуление всех членов структуры lf*/
          memset(&lf,0,sizeof(lf));
          /*Устанавливаем размер шрифта*/
```

```
      lf.lfHeight=30;
      /*Копируем в структуру имя шрифта*/
      lstrcpy(lf.lfFaceName,lpszFace);
      /*Создаем шрифт*/
      hFont = CreateFontIndirect(&lf);

      /*Первый немедленный вывод текущего времени*/
      OutTimeDate(hWnd);
      /*
      Функция SetTimer создает системный таймер
      с периодом 1с
      */
      SetTimer(hWnd,1,1000,(TIMERPROC)NULL);
      return TRUE;
case WM_TIMER:
/*
Функция OutTimeDate запрашивает у системы текущие значе-
ния даты и времени, а затем организует их обработку в
главном окне приложения.
*/
      OutTimeDate(hWnd);
   break;
case WM_KEYDOWN:
/*Обрабатываем сообщение-нажатие клавиши.*/
      switch(wParam)
      {
           case VK_ESCAPE:
         /*
         Посылаем сообщение WM_CLOSE окну (hWnd), а после
         того, как оконная процедура обработает это сооб-
         щение, система передаст управление инструкции
         следующей за SendMessage.
         */
                SendMessage(hWnd,WM_CLOSE,0,0);
                break;
      }
      break;
case WM_COMMAND:
      switch(LOWORD(wParam)) //switch(wParam)
      {
      case ID_FILE_TEST:
           //…
           break;
      }
      break;
case WM_PAINT:
      /*Получаем контекст устройства*/
      hDc = BeginPaint(hWnd, &ps);
      /*Определяем размеры клиентской области окна*/
      GetClientRect(hWnd,&rect);
      /*Выбираем в контест созданный шрифт*/
      hOldFont=SelectFont(hDc,hFont);
      /*Получим метрики текста*/
      GetTextMetrics(hDc,&tm);
/*
Функция SetBkMode устанавливает текущий режим фона.
TRANSPARENT - в этом режиме вывода текста цвет фона
графического элемента игнорируется, т.е. символ
выводится на существующем фоне.
*/
```

```
     SetBkMode(hDc,TRANSPARENT);
/*
Функция SetTextColor устанавливает цвет текста для
контекста устройства, по умолчанию применяется черный
цвет. Цвет текста синий!
*/
     SetTextColor(hDc,RGB(0,0,128));
     DrawText(hDc,szCurrentTime,-1,&rect,
               DT_SINGLELINE|DT_CENTER|DT_VCENTER);
    /*Освобождаем контекст устройства*/

     EndPaint(hWnd, &ps);
     break;
case WM_CLOSE:
    /*
    Сообщение WM_CLOSE появляется при щелчке на кнопке
    закрытия окна - здесь предназначено для вывода преду-
    преждающего сообщения
    */
     if(MessageBox(hWnd,_T("Вы уверены?"),
                                _T("Bad boys"),
          MB_YESNO | MB_ICONQUESTION)==IDYES)
          {
     /*
     Функция DestroyWindow  разрушает указанное в ее па-
     раметре окно, т.е. она посылает окну сообщение
     WM_DESTROY. Затем вызывается функция
     PostQuitMessage, которя посылает сообщение WM_QUIT
     */
               DestroyWindow(hWnd);
          }
     break;
case WM_SIZE:
    /*
    Ширина width и высота height клиентской области окна
    в пикселях
    */
     width=LOWORD(lParam);
     height=HIWORD(lParam);
     //...
     break;
case WM_LBUTTONDOWN:
    /*Нажата клавиша Shift ?*/
    if(wParam & MK_SHIFT)
     {
          MessageBox(hWnd,_T("Нажата клавиша\nShift"),
                          _T("Сообщение!"),
                          MB_OK|MB_ICONEXCLAMATION);
     }
     break;
case WM_GETMINMAXINFO:
     lpmmi=(LPMINMAXINFO)lParam;
     lpmmi->ptMinTrackSize.x=xSize;
     lpmmi->ptMinTrackSize.y=ySize;
     lpmmi->ptMaxTrackSize.x=xSize;
     lpmmi->ptMaxTrackSize.y=ySize;
     break;
case WM_DESTROY:
/*
Функция DeleteObject удаляет логический объект.
```

```
      К удаляемым объектам относятся перья, растровые
      изображения, кисти, области, палитры и шрифты.
      */
           /*Удаляем созданный шрифт*/
           DeleteObject(hFont);
           /*Функция KillTimer удаляет таймер*/
           KillTimer(hWnd,1);
          /*PostQuitMessage() выполняет только одно действие -
          ставит в очередь сообщение WM_QUIT. Параметр у этой
          функции - код возврата, который помещается в wParam
          */

           PostQuitMessage(0);
           break;

      default:
          /*Обработка прочих сообщений по умолчанию*/

           return DefWindowProc(hWnd, message,wParam,
                                                 lParam);
      }
      return 0L;
}
/*
Фунуция получения текущего времени и преобразование
его в символы
*/
void OutTimeDate(HWND hWnd)
{
      LPTSTR szDay[]={_T("Вск."),_T("Пнд."),_T("Втр."),
                                 _T("Ср."),_T("Чтв."),
                                 _T("Птн."),_T("Суб.")
                      };

      LPTSTR szMonth[]={_T(""),_T("Янв."),_T("Февр."),
                                 _T("Март"),_T("Апр."),
                                 _T("Май"),_T("Июнь"),
                                 _T("Июль"),_T("Авг."),
                                 _T("Сент."),_T("Окт."),
                                 _T("Нояб."),_T("Дек.")
                             };
      TCHAR szT[20];
      SYSTEMTIME SystemTime;
      /*
      Функция GetLocalTime осуществляет выборку местного време-
      ни, на которое настроен компьютер, т.е. функция
      заполняет структуру типа SYSTEMTIME в числовом виде.
      */
      GetLocalTime(&SystemTime);
      /*День недели*/
      lstrcpy(szCurrentTime,
                             szDay[SystemTime.wDayOfWeek]);
      /*Разделяющий пробел*/
      lstrcat((LPTSTR)szCurrentTime,_T(" "));
      /*Месяц*/
      lstrcat((LPTSTR)szCurrentTime,
                             szMonth[SystemTime.wMonth]);
      /*Разделяющий пробел*/
      lstrcat((LPTSTR)szCurrentTime,_T(" "));
      /*Дату переводим в символы*/
```

```
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wDay);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющий пробел*/
        lstrcat((LPTSTR)szCurrentTime,_T(" "));
        /*Год переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wYear);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        lstrcat((LPTSTR)szCurrentTime,_T("---"));
        /**Часы переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wHour);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющее двоеточие*/
        lstrcat((LPTSTR)szCurrentTime,_T(":"));
        /*Минуты переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wMinute);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющее двоеточие*/
        lstrcat((LPTSTR)szCurrentTime,_T(":"));
        /*Сеуцнды переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wSecond);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Перерисовка окна*/
        InvalidateRect(hWnd,NULL,TRUE);
}
```

Результат работы программы представлен на рис. 2.

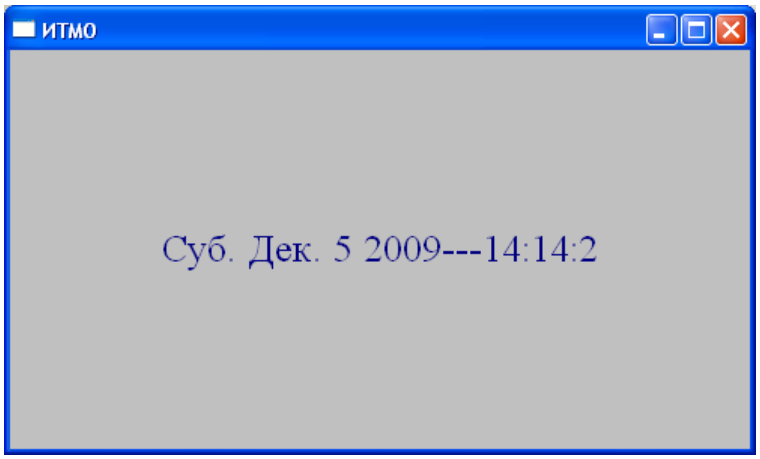


Рис. 2. Окно программы

2. МЕНЮ

2.1. Файл ресурсов

В файле ресурсов описываются форматы меню и диалоговых окон, растровые изображения, пиктограммы, курсоры, таблицы символьных строк, шрифты и т.д.

Файл ресурсов является текстовым файлом и имеет стандартное расширение .RC.

Файл ресурсов обрабатывается компилятором ресурсов, получая из него промежуточный файл с расширением .RES. Далее компоновщик компонует файлы .OBJ и .RES в единый загрузочный файл с расширением .EXE.

Элементы приложения, описанные в файле ресурсов сосредоточены в загрузочном файле в одном месте и хранятся в определенном формате. Это дает возможность специальным программам читать или даже модифицировать ресурсы непосредственно в загрузочном файле приложения.

2.2. Организация и виды меню

Меню является важнейшим элементом большинства Windows-приложений. Под строкой заголовка главного окна отображается полоса меню (menu bar), содержащая набор пунктов. Такое меню называется главным (main menu) или меню верхнего уровня, которое относится ко всему приложению.

Любое меню содержит пункты меню, которые обозначаются своими именами. Имя пункта может содержать выделенный подчеркиванием символ, который называется мнемоническим символом. Мнемонический символ определяет горячую клавишу (hotkey), служащую для выбора пункта меню при помощи клавиатуры.

Различают два типа пунктов меню:

1) пункт-команду – терминальный (конечный) пункт на иерархическом дереве меню, которому в программном коде соответствует уникальный целочисленный идентификатор;

2) пункт-подменю – заголовок вызываемого всплывающего меню (popup menu) следующего, более низкого уровня.

Реакция системы на выбор пункта зависит от типа пункта:

- выбран пункт-команда – система посылает приложению сообщение WM_COMMAND, которое содержит идентификатор этой команды, т.е. выбор этого пункта заставляет приложение выполнить некоторое действие;
- выбран пункт-подменю – система выводит на экран подменю, т.е. всплывающее меню (если пункт подменю содержит многоточие, то при выборе данного пункта выводится диалоговое окно).

Пункты меню могут быть разрешенными (enabled), запрещенными (disabled) или недоступными (grayed). Запрещенный и недоступный пункты меню по своему поведению одинаковы и различаются только внешним видом. Запрещенный пункт (disabled) выглядит так же, как и разрешенный (enabled), а недоступный (grayed) изображается серым

цветом. Следовательно, отмененный пункт меню для большей информативности должен быть определен как недоступный (grayed).

Для изменения статуса пунктов меню применяется функция EnableMenuItem(), ее прототип:

```
BOOL EnableMenuItem(
   HMENU hMenu,            //дескриптор меню
   UINT uIDEnableItem,     //идентификатор или позиция пункта
   UINT uEnable            //интерпретация второго параметра и
                           //выполняемое действие
);
```

Функция EnableMenuItem() может применяться для пунктов, как главного меню, так и подменю.

Пункт меню, для которого применяется функция, задается вторым параметром uIDEnableItem. Этому параметру передается либо идентификатор пункта меню в файле описания ресурсов, либо позиция пункта меню. Интерпретация этого параметра определяется третьим параметром – uEnable.

Параметр uEnable должен определяться комбинацией из двух флагов. Первый флаг может принимать одно из двух значений:

- MF_BUCOMMAND – второй параметр uIDEnableItem функции содержит идентификатор пункта меню;
- MF_BYPOSITION – второй параметр uIDEnableItem функции содержит позицию пункта меню, отсчитываемую от нуля.

Второй флаг может принимать одно из трех значений:

- MF_ENABLED – пункт разрешен;
- MF_DISABLED – пункт запрещен;
- MF_GRAYED – пункт недоступен.

Например: MF_BYCOMMAND | MF_GRAYED.

Если в результате применения функции EnableMenuItem() изменяется статус пункта главного меню, то следует обязательно применить функцию DrawMenuBar() для перерисовки изменившейся полосы меню. Прототип функции DrawMenuBar():

```
BOOL DrawMenuBar(HWND hwnd);
```

Пункты меню могут использоваться в роли флажков (check box). Флажок может быть установлен (символ «галочка») или сброшен. Переход из одного состояния в другое происходит при каждом выборе пункта меню. Флажки объединяются в группы и обеспечивают при этом выбор одной или нескольких опций одновременно.

Для управления выбором пунктов-флажков (ставит или снимает отметку на пункте меню) используется функция CheckMenuItem(), ее прототип:

```
DWORD ChecMenuItem(
HMENU hMenu,          //дескриптор меню
UINT uIDCheckItem,    //идентификатор или позиция пункта меню
UINT uCheck           //интерпретация второго параметра и
                      //выполняемое действие
);
```

Функция ChecMenuItem() может применяться только для пунктов подменю, а пункты главного меню не могут быть помечены. Функция возвращает предыдущее состояние пункта или –1.

Пункт подменю, для которого применяется функция, задается вторым параметром uIDCheckItem. Этому параметру передается либо идентификатор пункта меню в файле описания ресурсов, либо позиция пункта меню. Интерпретация этого параметра определяется третьим параметром – uCheck.

Параметр uCheck задается комбинацией двух флагов. Первый флаг может принимать одно из двух значений:

- MF_BUCOMMAND – второй параметр uIDCheckItem функции содержит идентификатор пункта меню;
- MF_BYPOSITION – второй параметр uIDCheckItem функции содержит позицию пункта меню, отсчитываемую от нуля.

Второй флаг может принимать одно из двух значений:

- MF_CHECKED – поместить отметку слева от пункта подменю;
- MF_UNCHECKED – снять отметку слева от пункта подменю.

Например: MF_BYCOMMAND|MF_CHECKED. Наиболее предпочтительно в комбинации всегда использовать флаг MF_BYCOMMAND.

Пункты меню могут использоваться в роли переключателей (radio button), которые, как и флажки, обычно используются в группе. Переключатели связываются только с взаимоисключающими опциями. Следовательно, в группе можно выбрать только один переключатель. При этом он отмечается точкой или закрашенным кружком.

Для управления выбором пунктов-переключателей (ставит отметку на выбранном пункте и снимает со всех остальных пунктов в указанной группе) используется функция CheckMenuRadioItem(), ее прототип:

```
BOOL CheckMenuRadioItem(
     HMENU hMenu,    //дескриптор меню
     UINT idFirst,   //идентификатор или позиция первого
                     //пункта в группе
     UINT idLast,    //идентификатор или позиция
                     //последнего пункта в группе
     UINT idCheck,   //идентификатор или позиция
                     //выбранного пункта в группе
     UINT uFlags     //интерпретация параметров idFirst,
                     //idLast и idCheck
);
```

Функция CheckMenuRadioItem() может применяться только для пунктов подменю, а *пункты главного меню не могут быть помечены*. Функция возвращает предыдущее состояние пункта или –1.

Пункт подменю, выбранный из группы пунктов, задается четвертым параметром idCheck. Параметрам idFirst, idLast и idCheck передается либо идентификатор пункта меню в файле описания ресурсов, либо позиция пункта меню. Интерпретация этих параметров определяется пятым параметром uFlags, который может принимать следующие значения:

- MF_BYCOMMAND – параметрам со второго по четвертый передаются идентификаторы соответствующих пунктов меню;
- MF_BYPOSITION – параметрам со второго по четвертый передаются позиции соответствующих пунктов меню.

Задавая пункты меню для группы переключателей, следует убедиться, что их идентификаторы в файле resource.h имеют сквозную нумерацию и упорядочены в соответствии с позициями этих пунктов на полосе меню.

Подменю может содержать пункт, который выполняется по умолчанию. Имя этого пункта выделяется жирным шрифтом. Если подменю содержит пункт по умолчанию, то при открытии подменю двойным щелчком мыши Windows автоматически выполнит соответствующую команду. Функцией SetMenuDefaultItem() может быть назначен любому пункту подменю так называемый атрибут «по умолчанию», ее прототип:

```
BOOL SetMenuDefaultItem(
  HMENU hMenu,
  UINT uItem,
  UINT fByPos
   );
```

Для получения дескриптора меню (hMenu) верхнего уровня применяется функция GetMenu(), а для получения дескриптора подменю – функция GetSubMenu(). Они имеют следующие прототипы:

```
HMENU GetMenu(HWND hWnd);
```

```
HMENU GetSubMenu(
    HWND hWnd, //дескриптор родительского меню
    int nPos   //позиция пункта-подменю в роди-
               //тельском меню
    );
```

Функция GetMenu() возвращает NULL, если окно hWnd не имеет меню.

Позиция пункта родительского меню nPos отсчитывается от нуля. Если пункт с позицией nPos не активизирует всплывающее меню, а является пунктом-командой, то функция возвращает значение NULL. Функцию GetMenu() нельзя применять для дочерних окон.

Меню можно создать следующими способами:

- на основе шаблона меню, определенного в файле ресурсов (в главном меню Visual Studio выполнить команды Project -> Add Resourse -> Menu , в результате будет открыто окно редактора меню);
- динамическим способом, т.е. непосредственно в приложение при помощи функций CreateMenu(), AppendMenu(), CreatePopupMenu() и SetMenu();
- альтернативным способом при помощи функции LoadMenuIndirect(), которая считывает определение меню в блок памяти и возвращает дескриптор созданного меню (подключение меню к окну – SetMenu()).

Прототип функции LoadMenuIndirect():

```
HMENU LoadMenuIndirect(CONST MENUTEMPLATE * lpMenuTemplate);
```

Чтобы меню, определенное в файле ресурсов, появилось в составе приложения, необходимо присвоить значение указателя на имя меню полю lpszMenuName структурной переменной типа WNDCLASSEX. В случае, когда имя меню определено как целочисленный идентификатор (IDR_MENU1), его необходимо преобразовать в указатель на строку ресурса, используя макрос MAKEINTRESOURCE, например:

```
wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU1);
```

В обоих случаях при создании окна с помощью функции CreateWindowEx() ее параметру hMenu следует передать значение NULL. Таким образом, присоединенное меню будет использоваться по умолчанию всеми окнами данного класса.

Для связывания с окном приложения другого меню, т.е. не меню, используемого по умолчанию, необходимо выполнить следующие действия:

1) загрузить при помощи функции LoadMenu() требуемое меню;

2) определить требуемое меню с помощью функции SetMenu(), передав ей в качестве второго параметра, дескриптор меню (hMenu), возвращенный функцией LoadMenu() (при этом новое меню заменяет старое меню, если оно уже было).

Функции LoadMenu() и SetMenu() имеют следующие прототипы:

```
HMENU LoadMenu(HINSTANCE hInstance,LPCTSTR lpMenuName);
BOLL SetMenu(HWND hWnd, HMENU hMenu);
```

На основании вышеизложенного создадим меню динамически, т.е. непосредственно в приложении (листинг 2.3). Необходимо иметь в виду строгую древовидную структуру меню, которая начинается с верхнего уровня. К меню верхнего уровня могут быть присоединены как конечные элементы меню, так и элементы, выбор которых приводит к появлению всплывающих (popup) меню. В свою очередь к всплывающим меню присоединяются элементы очередного уровня и т.д. Таким образом, алгоритм создания меню в приложении следующий:

1) выбрать подменю самого низкого уровня, которое содержит только конечные элементы меню, и создать их с помощью одной из функций CreateMenu() или CreatePopupMenu();

2) добавить функцией AppendMenu() в созданное пустое меню требуемые пункты меню;

3) создать меню следующего, более высокого уровня, и добавить в них пункты меню и меню, созданные на предыдущем шаге;

4) повторять пп. 1,2 и 3 до тех пор, пока не будут созданы все подменю;

5) создать главное меню программы функцией CreateMenu();

6) присоединить созданные подменю самого высокого уровня к главному меню программы функцией AppendMenu();

7) присоединить меню к окну функцией SetMenu();

8) прорисовать меню функцией DrawMenuBar(), так как строка меню не является частью клиентской области и, следовательно, не обновляется при вызове функции UpdateWindow().

Прототипы функций DrawMenuBar(), CreateMenu() и CreatePopupMenu() соответственно:

```
BOOL DrawMenuBar(HWND hWnd);
HMENU CreateMenu(VOID);
HMENU CreatePopupMenu(VOID);
```

Функция AppendMenu() добавляет новый элемент в конец меню. Всплывающими меню считаются все меню, кроме верхней строки.

```
BOOL AppendMenu(
   HMENU hMenu,       //дескриптор меню, к которому
                      //добавляется новый пункт
   UINT uFlags        //вид и правило поведения
                      //добавляемого пункта меню
   UINT idNewItem,    //идентификатор для нового
                      //пункта меню
   LPCTSTR lpszNewItem    //содержимое нового пункта
                          //меню (зависит от uFlags)
);
```

Значения параметра uFlags (полный список в MSDN):

- MF_POPUP – создается всплывающее меню;
- MFS_CHECKED – помещается отметка рядом с пунктом меню;
- MFS_DEFAULT – пункт меню, применяемый по умолчанию;
- MFS_GRAYED – пункт меню выделяется серым цветом и запрещается его выбор;
- MFS_HILITE – пункт меню высвечивается;
- MFT_STRING – пункт меню отображается с использованием текстовой строки.

Функция DestroyMenu() уничтожает меню, созданное с применением файла ресурса, или меню, созданное в теле программы с помощью функций CreateMenu() и CreatePopupMenu(). По завершении приложения будет уничтожено только одно меню, подключенное к окну приложения. Меню, не подключенные в данный момент к окну, останутся в памяти, поэтому функция DestroyMenu() должна обязательно применяться в приложениях с несколькими меню. Прототип функции DestroyMenu():

```
BOOL DestroyMenu(HMENU hMenu);
```

2.3. Приложение с главным окном и меню

```
/*
Файл resource.h
*/

//
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by 1111.rc
```

```
//
#define IDD_DIALOGBAR                   103
#define IDC_BUTTON1                     1002
#define IDC_STATIC_1                    1003

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        101
#define _APS_NEXT_COMMAND_VALUE         40001
#define _APS_NEXT_CONTROL_VALUE         1004
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Если открыть файл описания ресурсов 1111.rc в текстовом режиме, то можно увидеть следующее определение шаблона диалогового окна:

```
//////////////
// Microsoft Visual C++ generated resource script.
//
#include "resource.h"
//...
////////////////////////////////////////////////////////
//
// Dialog
//
IDD_DIALOGBAR DIALOGEX 0, 0, 153, 62
STYLE DS_SETFONT | DS_MODALFRAME | WS_POPUP | WS_CAPTION |
WS_SYSMENU
CAPTION "СПб ГУ ИТМО"
FONT 10, "Lucida Console", 700, 0, 0xCC
BEGIN
    PUSHBUTTON      "Ok",IDOK,105,42,41,14,BS_CENTER
    CTEXT           "ЛИТМО\n КАФЕДРА ПКС\n2009",
                        IDC_STATIC,44,14,63,26,WS_TABSTOP
END
////////////////////////////////////////////////////////
/*
Файл 1111.cpp
*/
/*Операторы препроцессора*/
/*
#define UNICODE
#ifdef UNICODE
#define _UNICODE
#endif
*/
#define STRICT
#include <windows.h>
#include <windowsx.h>
/*
Файл tchar.h состоит из макросов, которые ссылаются на UNICODE
данные и функции, если определен макрос UNICODE, и на ANSI
данные и функции, если этот макрос не определен, кроме того он
полностью заменяет файл string.h
*/
#include <tchar.h>
```

```
#include "resource.h"
/*
Идентификаторы пунктов меню приведены для наглядности, лучше
всего их повестить в файл resource.h
 */
#define ID_FILE_TEST 40001
#define ID_FILE_EXIT 40002
#define ID_HELP_ABOUT 40003

/*Прототип оконной функции*/
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
/*Прототип функции модального диалога*/
INT_PTR CALLBACK AboutProc(HWND, UINT, WPARAM, LPARAM);
/*
Прототип функция обратного вызова обработки сообщений
от таймера
*/
VOID CALLBACK TimerProc(HWND,UINT,UINT_PTR,DWORD);
/*
Прототип функции получения текущего времени
и преобразование его в символы
*/
void OutTimeDate(HWND);
/*Дескрипторы всплывающих меню и дескриптор главного меню*/
HMENU hFileMenu,hHelpMenu,hMenu;
/*Будет создан логический шрифт*/
HFONT hFont,hOldFont;
/*Массив для формирования строки - текущие дата и время*/
TCHAR szCurrentTime[40];
int nTime=5;//Однократный интервал 5с
int WINAPI WinMain(HINSTANCE hInstance,
                                HINSTANCE hPrevInstance,
                                LPSTR lpCmdLine,int nCmdShow)
{

     /*Произвольный заголовок окна*/
     TCHAR szTitle[]=_TEXT("ИТМО");
     /*Произвольное имя класса*/
     TCHAR szWindowClass[]=_TEXT("QWERTY");

     /*
     Структурная переменная msg типа MSG для
     получения сообщений
     */
     MSG msg;
     /*
     Структурная переменная wcex типа WNDCLASSEX для задания
     характеристик окна
     */
     WNDCLASSEX wcex;
     HWND hWnd; //Дескриптор главного окна
     /*Проверяем, было ли это проложение запущено ранее*/
     if(hWnd=FindWindow(szWindowClass,NULL))
     {
     /*Проверяем, было ли это окно свернуто в пиктограмму*/
          if(IsIconic(hWnd))
               ShowWindow(hWnd,SW_RESTORE);
          /*Выдвигаем окно приложения на передний план*/
          SetForegroundWindow(hWnd);
          return FALSE;
```

```
}
/*Обнуление всех членов структуры wcex*/
memset(&wcex,0,sizeof(WNDCLASSEX));
/*Регистрируем класс главного окна*/
wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style  = CS_HREDRAW | CS_VREDRAW;
/*Определяем оконную процедуру для главного окна*/
wcex.lpfnWndProc     = (WNDPROC)WndProc;
//wcex.cbClsExtra    = 0;
//wcex.cbWndExtra    = 0;
wcex.hInstance = hInstance;//Дескриптор приложения
/*
Стандартная пиктограмма, которую можно загрузить функцией
LoadImage()
*/
wcex.hIcon = (HICON)LoadImage(hInstance,
                                IDI_APPLICATION,
                            IMAGE_ICON,32,32,0);
/*Стандартный курсор мыши*/
wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
/*
Кисть фона и ее цвет можно определить выражениями:
wcex.hbrBackground=(HBRUSH)(COLOR_APPWORKSPACE+1);
wcex.hbrBackground=(HBRUSH)GetStockObject(LTGRAY_BRUSH);
или с помощью  макроса  GetStockBrush(), в этом случае
необходимо подключить файл windowsx.h
*/
wcex.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
//wcex.lpszMenuName  = MAKEINTRESOURCE(IDC_MSG_1);
/*Имя класса главного окна*/
wcex.lpszClassName   = szWindowClass;
wcex.hIconSm = NULL;
if(!RegisterClassEx(&wcex))
          return FALSE;
/*Создаем главное окно и делаем его видимым*/
hWnd = CreateWindowEx(WS_EX_WINDOWEDGE,szWindowClass,
                    szTitle,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,0,CW_USEDEFAULT,0,
                    NULL, NULL, hInstance, NULL);

if (!hWnd)
     return FALSE;
/*
Исследуем адресное пространство приложения. Выведем
содержимое сегментных регистров команд, данных и
стека, а также смещение главной функции и строки
с именем класса
*/
TCHAR szAsm[80];
USHORT regCS,regDS,regES,regSS;
__asm{
     mov regCS,CS
     mov regDS,DS
     mov regES,ES
     mov regSS,SS
}
wsprintf((LPTSTR)szAsm,_T("CS=%X,DS=%X\nES=%X,SS=%X\n
                    WinMain=%X\nszWindowClass=%X"),
                    regCS,regDS,regES,regSS);
```

```
 MessageBox(NULL,(LPCTSTR)szAsm,_T("Регистры"),
                                    MB_ICONINFORMATION);
 /*
 Создаем  пустое всплывающее меню самого низкого
 уровня hFileMenu=CreatePopupMenu()и добавляем в
 него конечный элемент "Test"
*/
 AppendMenu((hFileMenu=CreatePopupMenu()),
                      MF_ENABLED | MFT_STRING,
                      ID_FILE_TEST,_TEXT("&Test"));
 /*
 Добавляем в созданное меню  конечный элемент "Exit"
 */
 AppendMenu(hFileMenu,MF_GRAYED | MFT_STRING,
                      ID_FILE_EXIT,_TEXT("&Exit"));
 /*
 Создаем  пустое всплывающее меню самого низкого
 уровня hHelpMenu=CreatePopupMenu()и добавляем в
 него конечный элемент "About"
 */
 AppendMenu((hHelpMenu=CreatePopupMenu()),
                      MF_ENABLED|MFT_STRING,
                      ID_HELP_ABOUT,_TEXT("&About"));
 /*
 Создаем меню верхнего уровня - главное меню
 hMenu=CreateMenu()и присоединяем созданное подменю"File"
 к главному меню"
 */
 AppendMenu((hMenu=CreateMenu()),
                 MF_ENABLED | MF_POPUP,
                 (UINT_PTR)hFileMenu, _TEXT("&File"));
 /*
 Присоединяем созданное подменю "Help" к главному
 Меню
 */
 AppendMenu(hMenu,MF_ENABLED|MF_POPUP,
                 (UINT_PTR)hHelpMenu,_TEXT("&Help"));
 /
 *Добавляем в конец главного меню конечный пункт
 "QUIT"
 */
 AppendMenu(hMenu,MF_GRAYED,(UINT_PTR)11,
                                      _TEXT("Quit"));
 /*Присоединяем созданное меню к окну приложения*/
 SetMenu(hWnd,hMenu);
 /*Делаем окно видимым на экране*/
 ShowWindow(hWnd, SW_SHOWNORMAL);
 /*
 Функция UpdateWindow() вызывает передачу
 сообщения WM_PAINT непосредственно оконной
 процедуре, а  функция InvalidateRect() вызывает
 постановку сообщения WM_PAINT в очередь приложения,
 а там оно обрабатывается с самым низким приоритетом
 */
 UpdateWindow(hWnd);
 /*Отображаем меню*/
 DrawMenuBar(hWnd);
 /*Цикл обработки сообщений*/
 while (GetMessage(&msg, NULL, 0, 0))
      {
```

```
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
          /*Код возврата*/
     return (int)msg.wParam;
}
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
     /*Горизонтальный размер главного окна*/
     int xSize=500;
     /*Верикальный размер главного окна*/
     int ySize=300;
     /* PAINTSTRUCT - структура с характеристиками рабочей
     области (заполняется функцией BeginPaint)
     */
     PAINTSTRUCT ps;
     /*
     TEXTMETRIC - структура для получения
     характеристик шрифта
     */
     TEXTMETRIC tm;
     /*
     LOGFONT - структура для для создания
     логических шрифтов
     */
     LOGFONT lf;
     /*
     RECT-структура,определящая прямоугольник
     */
     RECT rect;
     LPMINMAXINFO lpmmi;
     Static HFONT hFont,hOldFont;
     /*Дескриптор контекста устройства*/
     HDC hDc;
     /*
     Ширина width и высота height клиентской области
     окна в пикселях
     */

     UINT width, height;
     /*Имя для шрифта*/
     LPTSTR lpszFace=_TEXT("Times New Roman Cur");
     switch (message)
     {

     /*Переход по значению кода сообщения(msg)*/
      case WM_CREATE:
     /*Только здесь можно произвести модификацию
      Класса окна. Например,
      SetClassLong(hWnd,GCL_HBRBACKGROUND,
                (LONG)CreateSolidBrush(RGB(200,160,255));
     Значение дескриптора экземпляра приложения
     определяется,вызовом одной из следующих функций:
     hInst=GetModuleHandle(NULL);
     hInst=(HINSTANCE)GetClassLong(hWnd,GCL_HMODULE);
     */
          /*Обнуление всех членов структуры lf*/
          memset(&lf,0,sizeof(lf));
```

```
      /*Устанавливаем размер шрифта*/
      lf.lfHeight=30;
      /*Копируем в структуру имя шрифта*/
      lstrcpy(lf.lfFaceName,lpszFace);
      /*Создаем шрифт*/
      hFont=CreateFontIndirect(&lf);

/*Первый немедленный вывод текущего времени*/
      OutTimeDate(hWnd);
/*
Функция SetTimer создает системный таймер
с периодом 1с
*/
      SetTimer(hWnd,1,1000,(TIMERPROC)NULL);
      return TRUE;
case WM_TIMER:
/*
Функция OutTimeDate запрашивает у системы текущие
значения даты и времени, а затем организует
их обработку в главном окне приложения.
*/
      OutTimeDate(hWnd);
   break;
case WM_KEYDOWN:
/*Обрабатываем сообщение-нажатие клавиши.*/
      switch(wParam)
      {
           case VK_ESCAPE:
         /*
         Посылаем сообщение WM_CLOSE окну (hWnd), после
         того, как оконная процедура обработает это сооб-
         щение, система передаст управление инструкции
         следующей за SendMessage
         */
                SendMessage(hWnd,WM_CLOSE,0,0);
                break;
      }
      break;
case WM_COMMAND:
      switch(LOWORD(wParam)) //switch(wParam)
      {
      case ID_FILE_TEST:
        /*
        Изменяем статус пункта меню ID_FILE_EXIT
        */
            EnableMenuItem(hFileMenu,ID_FILE_EXIT,
                           MF_BYCOMMAND|MF_ENABLED);
        /*Ставим отметку (галочка) на пункте меню
        ID_FILE_TEST*/
            CheckMenuItem(hFileMenu,ID_FILE_TEST,
                           MF_BYCOMMAND|MF_CHECKED);
        /*Изменяем статус пункта главного меню "QUIT"*/
            EnableMenuItem(GetMenu(hWnd),
                 (UINT_PTR)11,MF_BYCOMMAND|MF_ENABLED);
        /*
        Так как изменился статус пункта главного
        меню, вызываем функцию DrawMenuBar для повторного
        отображения изменившейся полосы меню
        */
            DrawMenuBar(hWnd);
```

```
            /*Устанавливаем таймер на nTime секунд*/
            SetTimer(hWnd,2,nTime*1000,
                                (TIMERPROC)TimerProc);
            break;
      case ID_FILE_EXIT:
         /*
         Без запроса на закрытие окна - функция
         PostQuitMessage посылает сообщение WM_QUIT
         */
            PostQuitMessage(0);
         /*
         С запросом на закрытие, т.е. окно еще не разруше-
         но SendMessage(hWnd,WM_CLOSE,0,0);
         */
            break;
      case ID_HELP_ABOUT:
      /*
      Функция DialogBox создает и выводит на экран мо-
      дальное диалоговое окно по шаблону IDD_ABOUTBOX, и
      не возвращает управление в WndProc пока окно диало-
      ка не бедет закрыто.
      */
            DialogBox(GetModuleHandle()NULL,
                      MAKEINTRESOURCE(IDD_DIALOGBAR),
                      hWnd, (DLGPROC)AboutProc);
            break;
      case (UINT)11:
         /*
         Без запроса на закрытие окна - функция
         PostQuitMessage посылает сообщение WM_QUIT
         */
            PostQuitMessage(0);
            break;
      }
      break;
case WM_PAINT:
      /*Получаем контекст устройства*/
      hDc = BeginPaint(hWnd, &ps);
      /*Определяем размеры клиентской области окна*/
      GetClientRect(hWnd,&rect);
      /*Выбираем в контест созданный шрифт*/
      hOldFont=SelectFont(hDc,hFont);
      /*Получим метрики текста(если это необходимо)*/
      GetTextMetrics(hDc,&tm);
/*
Функция SetBkMode устанавливает текущий режим фона.
TRANSPARENT - в этом режиме вывода текста цвет фона гра-
фического элемента игнорируется, т.е. символ выводится на
существующем фоне
*/
      SetBkMode(hDc,TRANSPARENT);
/*
Функция SetTextColor устанавливает цвет текста для кон-
текста устройства, по умолчанию применяется черный цвет.
Цвет текста синий!
*/
      SetTextColor(hDc,RGB(0,0,128));
      DrawText(hDc,szCurrentTime,-1,&rect,
                  DT_SINGLELINE|DT_CENTER|DT_VCENTER);
     /*Освобождаем контекст устройства*/
```

```
     EndPaint(hWnd, &ps);
     break;
case WM_CLOSE:
    /*
    Сообщение WM_CLOSE появляется при щелчке на кнопке
    закрытия окна - здесь предназначено для вывода преду-
    преждающего сообщения
    */
     if(MessageBox(hWnd,_T("Вы уверены?"),
          _T("Предупреждение!"),
          MB_YESNO | MB_ICONQUESTION)==IDYES)
          {
     /*
     Функция DestroyWindow  разрушает указанное в ее па-
     раметре окно, т.е. она посылает окну сообщение
     WM_DESTROY. Затем вызывается функция
     PostQuitMessage, которая посылает сообщение WM_QUIT
     */
               DestroyWindow(hWnd);
          }
     break;
case WM_SIZE:
    /*
    Ширина width и высота height клиентской области окна
    в пикселях
    */
     width=LOWORD(lParam);
     height=HIWORD(lParam);
     //...
     break;
case WM_LBUTTONDOWN:
    /*Нажата клавиша Shift ?*/
    if(wParam & MK_SHIFT)
     {
          MessageBox(hWnd,_T("Нажата клавиша\nShift"),
                         _T("Bad boys"),
                         MB_OK|MB_ICONEXCLAMATION);
     }
     break;
case WM_GETMINMAXINFO:
     lpmmi=(LPMINMAXINFO)lParam;
     /*
     Минимальный и максимальный размеры
     окна совпадают
     */
     lpmmi->ptMinTrackSize.x=xSize;
     lpmmi->ptMinTrackSize.y=ySize;
     lpmmi->ptMaxTrackSize.x=xSize;
     lpmmi->ptMaxTrackSize.y=ySize;
     break;
case WM_DESTROY:
/*
Функция DeleteObject удаляет логический объект. К удаляе-
мым объектам относятся перья, растровые изображения, кис-
ти, области, палитры и шрифты
*/
     /*Удаляем созданный шрифт*/
     DeleteObject(hFont);
     /*Функция KillTimer удаляет таймер*/
```

```
            KillTimer(hWnd,1);
           /*
           PostQuitMessage() выполняет только одно действие -
           ставит в очередь сообщение WM_QUIT. Параметр у этой
           функции - код возврата, который помещается в wParam
           */

            PostQuitMessage(0);
            break;

      default:
          /*Обработка прочих сообщений по умолчанию*/
           return DefWindowProc(hWnd, message, wParam,
                                                  lParam);
      }
      return 0L;
}
/*Оконная функция диалогового окна*/
INT_PTR CALLBACK AboutProc(HWND hDlg, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
      switch (message)
      {
            case WM_INITDIALOG:
      /*
      Для обрабатываемых сообщений процедура всегда
      возвращает TRUE
      */
                  return (INT_PTR)TRUE;

            case WM_COMMAND:
                  if(LOWORD(wParam)==IDOK || LOWORD(wParam)==
                                                      IDCANCEL)
                  {
                        EndDialog(hDlg, LOWORD(wParam));
                        return (INT_PTR)TRUE;
                  }
                  break;
      }
      /*
      Для не обрабатываемых сообщений процедура всегда
      возвращает FALSE
      */
      return (INT_PTR)FALSE;
}
/*
Фунуция получения текущего времени и преобразование
его в символы
*/
void OutTimeDate(HWND hWnd)
{
      LPTSTR szDay[]={_T("Вск."),_T("Пнд."),_T("Втр."),
                                 _T("Ср."),_T("Чтв."),
                                 _T("Птн."),_T("Суб.")
                        };

      LPTSTR szMonth[]={_T(""),_T("Янв."),_T("Февр."),
                                 _T("Март"),_T("Апр."),
                                 _T("Май"),_T("Июнь"),
                                 _T("Июль"),_T("Авг."),
```

```
                                   _T("Сент."),_T("Окт."),
                                   _T("Нояб."),_T("Дек.")
                              };
     TCHAR szT[20];
     SYSTEMTIME SystemTime;
     /*
     Функция GetLocalTime осуществляет выборку местного време-
     ни, на которое настроен компьютер, т.е. функция
     заполняет структуру типа SYSTEMTIME в числовом виде.
     */
     GetLocalTime(&SystemTime);
     /*День недели*/
     lstrcpy(szCurrentTime,
                              szDay[SystemTime.wDayOfWeek]);
     /*Разделяющий пробел*/
     lstrcat((LPTSTR)szCurrentTime,_T(" "));
     /*Месяц*/
     lstrcat((LPTSTR)szCurrentTime,
                              szMonth[SystemTime.wMonth]);
     /*Разделяющий пробел*/
     lstrcat((LPTSTR)szCurrentTime,_T(" "));
     /*Дату переводим в символы*/
     wsprintf((LPTSTR)szT,_T("%d"),
                                   SystemTime.wDay);
     lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
     /*Разделяющий пробел*/
     lstrcat((LPTSTR)szCurrentTime,_T(" "));
     /*Год переводим в символы*/
     wsprintf((LPTSTR)szT,_T("%d"),
                                   SystemTime.wYear);
     lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
     lstrcat((LPTSTR)szCurrentTime,_T("---"));
     /*Часы переводим в символы*/
     wsprintf((LPTSTR)szT,_T("%d"),
                                   SystemTime.wHour);
     lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
     /*Разделяющее двоеточие*/
     lstrcat((LPTSTR)szCurrentTime,_T(":"));
     /*Минуты переводим в символы*/
     wsprintf((LPTSTR)szT,_T("%d"),
                                   SystemTime.wMinute);
     lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
     /*Разделяющее двоеточие*/
     lstrcat((LPTSTR)szCurrentTime,_T(":"));
     /*Секунды переводим в символы*/
     wsprintf((LPTSTR)szT,_T("%d"),
                                   SystemTime.wSecond);
     lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
     /*Перерисовка окна*/
     InvalidateRect(hWnd,NULL,TRUE);
}

/*Функция обратного вызова обработки сообщений от таймера*/
VOID CALLBACK TimerProc(HWND hWnd,UINT uMsg,
                              UINT_PTR idEvent,DWORD dwTime)
{
     TCHAR szTimer[100];
     KillTimer(hWnd,2);
     wsprintf(szTimer,
_T("С момента выбора\nпункта меню Test\nпрошло %d с!"),
```

```
                                                    nTime);
    MessageBox(NULL,(LPCTSTR)szTimer,
                    _T("Предупреждение"),MB_ICONHAND);
}
```

Результат работы программы приведен на рис. 3.

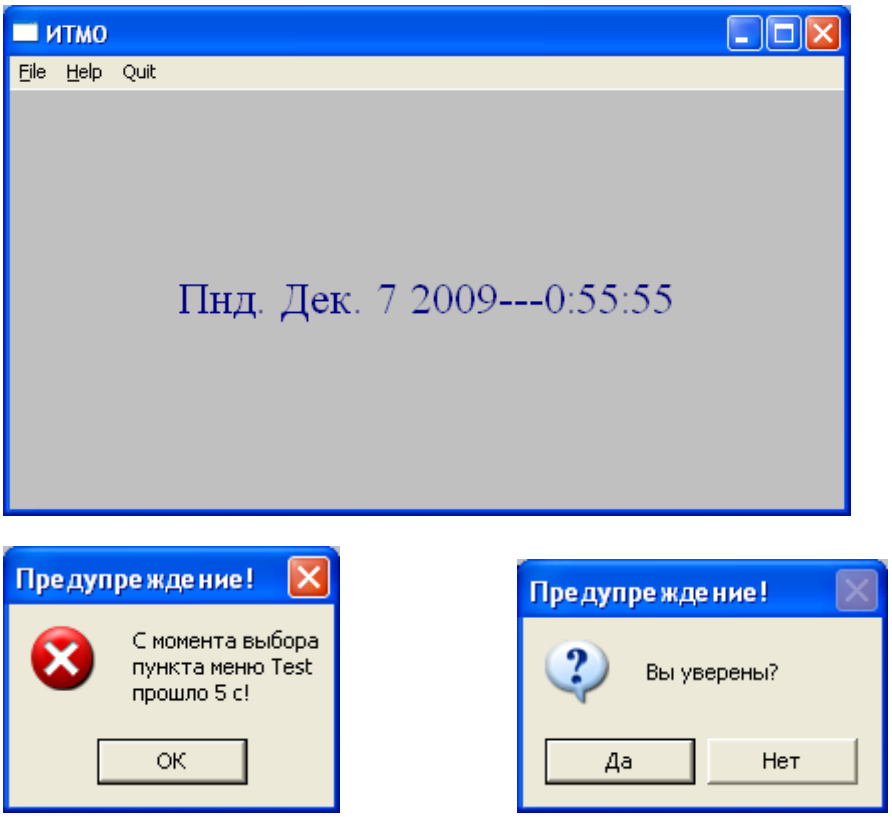


Рис. 3. Окно приложения с главным меню

3. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ ОБЩЕГО ПОЛЬЗОВАНИЯ

Рассмотрим элементы управления главного окна:

- `Toolbar` (панель инструментов) – состоит из кнопок быстрого доступа;
- `Statusbar` (строка состояния) – информационная строка, которая размещается в нижней части окна приложения.

Большинство элементов управления общего пользования реализовано в виде окна соответствующего предопределенного класса и управляются специфичными для данного класса сообщениями. Элементы управления посылают уведомляющие сообщения родительскому окну, информируя его о происходящих событиях.

Чтобы использовать в приложении элементы управления общего пользования, необходимо вначале вызвать функцию

InitCommonControlsEx(), которая регистрирует оконные классы элементов управления. Описание функции находится в файле commctrl.h. Поэтому необходимо в начало исходного файла включить следующую директиву:

```
#include <commctrl.h>
```

Кроме того, следует указать компоновщику расположение библиотечного файла comctl32.lib. Функция имеет прототип:

```
BOOL InitCommonControlsEx(
        LPINITCOMMONCONTROLSEX lpInitCtrls);
```

Параметр lpInitCtrls – адрес структурной переменной типа INITCOMMONCONTROLSEX, которая содержит информацию о том, какие классы элементов управления должны быть зарегистрированы. Структура имеет вид:

```
typedef struct tagINITCOMMONCONTROLSEX {
     DWORD dwSize;  //размер структуры в байтах
     DWORD dwICC;   //флаги загрузки классов из DLL
} INITCOMMONCONTROLSEX, *LPINITCOMMONCONTROLSEX;
```

Второй параметр dwICC структуры может содержать один или несколько следующих флагов (полный список флагов в MSDN):

- ICC_BAR_CLASSES – toolbar, status bar, slider, tooltip;
- ICC_PROGRESS_CLASS – progress bar;
- ICC_TREEVIEW_CLASSES – tree view.

Создание органов управления Toolbar и Statusbar выполняется приложением при обработке функцией главного окна сообщения WM_CREATE.

3.1. Панель инструментов

Панель инструментов – это дочернее окно, расположенное под меню приложения, содержащее одну или несколько кнопок. Сами по себе кнопки не являются окнами. Они имеют одинаковые размеры и реализованы как графические объекты на поверхности окна панели инструментов.

Традиционно кнопки панели инструментов соответствуют некоторым пунктам меню приложения, т.е. индентификаторы кнопок совпадают с идентификаторами дублируемых пунктов меню. Кнопка, выбранная на панели инструментов, посылает сообщение WM_COMMAND родительскому окну.

Кнопка может содержать, кроме изображения, текстовую метку, которая может находиться правее или ниже картинки. Как правило, кнопки на панели инструментов содержат только растровые изображения, а их назначение поясняется с помощью всплывающих окон подсказок.

Кроме кнопок, панель инструментов может содержать и другие дочерние окна элементов управления, например комбинированный список (combo box). Встроенные элементы управления создаются с помощью функции CreateWindowEx().

Для добавления к приложению панели инструментов необходимо выполнить следующие действия:

1) определить ресурс растрового образа панели инструментов;

2) объявить и инициализировать массив структур типа TBBUTTON, содержащий информацию о кнопках панели инструментов;

3) вызвать функцию CreateToolBarEx() для создания и инициализации панели инструментов.

Для определения ресурса растрового образа панели инструментов необходимо в главном меню Visual Studio выполнить команду Project –> Add Resource –> Toolbar и нажать кнопку New. В результате будет открыто окно редактора панели инструментов. Для создаваемой кнопки следует нарисовать картинку и определить идентификатор кнопки. Если кнопка дублирует некоторый пункт меню, то идентификатор кнопки должен быть таким же, как у этого пункта.

Закончив проектирование панели инструментов, необходимо сохранить растровый образ в файле описания ресурсов(.rc) (будет также создан файл toolbar1.bmp). По умолчанию редактор присваивает растровому образу панели инструментов идентификатор IDR_TOOLBAR1. Если открыть файл(1112.rc) в текстовом режиме, то можно увидеть следующее определение панели инструментов:

```
//////////////////////////////////////////////////////
//
// Toolbar
//

IDR_TOOLBAR1 TOOLBAR  16, 15
BEGIN
    BUTTON      ID_BUTTON40001
    BUTTON      ID_BUTTON40002
    BUTTON      ID_BUTTON40003
END

//////////////////////////////////////////////////////
//
// Bitmap
//

IDR_TOOLBAR1            BITMAP                  "toolbar1.bmp"

//////////////////////////////////////////////////////
```

Для создания панели инструментов перед вызовом функции CreateToolBarEx() необходимо заполнить массив структур типа TBBUTTON (объявление массива – TBBUTTON tbb [Num];).

Структура TBBUTTON определена в файле commctrl.h следующим образом:

```
typedef struct _TBBUTTON{
 int iBitmap;  //индекс изображения кнопки, а для
               //разделителей –нуль
               //(нумерация начинается с нуля)
 int idCommand;     //идентификатор кнопки, который
```

```
                    //передается родительскому окну
                    //с сообщением WM_COMMAND
  BYTE fsState;     //флаг исходного состояния кнопки
  BYTE fsStyle;     //стиль кнопки
  DWORD dwData;     //дополнительные данные или 0L
  INT_PTR iString;  //индекс текстовой метки, которую
                //необходимо написать на поверхности кнопки
} TBBUTTON;
```

Поле fsState структуры должно содержать флаг исходного состояния кнопки:

- TBSTATE_CHECKED – кнопка (стиль TBSTYLE_CHECK), используется для кнопок с фиксацией;
- TBSTATE_PRESSED – кнопка любого стиля, изображается в нажатом состоянии;
- TBSTATE_ENABLED – кнопка доступна, т.е. находится в разблокированном состоянии (реагирует на действие мышью), но если этот флаг не установлен, значит, кнопка заблокирована и отображается серым цветом;
- TBSTATE_HIDDEN – скрытая кнопка (не отображается);
- TBSTATE_INDETERMINATE – отображается серым цветом (недоступна).
- Поле fsStyle структуры может принимать комбинацию следующих стилей:
- TBSTYLE_BUTTON – стандартная кнопка, не может находиться в нажатом состоянии;
- TBSTYLE_SEP – разделитель между группами кнопок, может использоваться для резервирования места для дочерних элементов управления;
- TBSTYLE_CHECK – кнопка с фиксацией, ведет себя как флажок (check box);
- TBSTYLE_GROUP – стандартная кнопка, остается нажатой до тех пор, пока не нажата другая кнопка из той же группы, т.е. является членом группы кнопок переключателей (radio button);
- TBSTYLE_CHECKGROUP – кнопка, объединяющая свойства стилей TBSTYLE_CHECK и TBSTYLE_GROUP.

Поле dwData служит для передачи дополнительных данных. В противном случае в него можно записать нулевое значение. Содержание этого поля можно получить, послав сообщение TB_GETBUTTON.

Поле iString содержит номер текстовой строки, которую необходимо написать на поверхности кнопки (текстовая метка). В этом случае необходимо создать список текстовых строк и отправить сообщение TB_ADDSTRING, передав вместе с ним адрес буфера с текстовыми строками (в параметре lParam). Строки в этом буфере должны заканчиваться двоичным нулем, а последняя строка – двумя двоичными нулями. Если текстовые строки не используются, в поле iString следует записать нулевое значение.

Функция CreateToolBarEx() создает и инициализирует панель инструментов (перед первым вызовом этой функции необходимо вызвать функцию InitCommonControlsEx()).

Прототип функции CreateToolBarEx():

```
HWND CreateToolBarEx(
    HWND hWnd,  //дескриптор родительского окна
    DWORD ws,   //стили панели инструментов
    UINT wID,   //идентификатор панели инструментов
    int nBitmaps,         //количество изображений кнопок
    HINSTANCE hInst,      //дескриптор экземпляра
                          //приложения
    UINT wBMID,      //идентификатор ресурса растрового
                     //образа
    LPCTBBUTTON lpButtons,     //адрес массива структур
                               //(TBBUTTON)
    int iNumButtons,           //количество кнопок
    int dxButton,              //ширина кнопок в пикселях
    int dyButtons,             //высота кнопок в пикселях
    int dxBitmap,    //ширина изображения кнопки в пикселях
    int dyBitmap,    //высота изображения кнопки в пикселях
    UINT uSructSize //размер структуры
    };
```

Параметр ws определяет стили окна панели инструментов. Окно панели инструментов всегда является дочерним по отношению к создавшему его окну. Поэтому параметр ws должен содержать флаг WS_CHILD. Для того чтобы окно панели инструментов было видимым и имело рамку, необходимо указать стили WS_VISIBLE и WS_BORDER, а для изменения внешнего вида панели инструментов – CCS ADJUSTABLE.

Стиль TBSTYLE_TOOLTIPS задает поддержку всплывающих подсказок для кнопок панели инструментов.

Параметр wID должен содержать идентификатор панели инструментов. Можно использовать любой целочисленный идентификатор при условии, что его значение не совпадает со значениями идентификаторов, определенных в файле ресурсов resource.h.

Если значения параметров dxButton, dyButton, dxBitmap и dyBitmap равны NULL, то используются соответственно следующие значения: 16,15,16,15.

Создание органов управления toolbar обычно выполняется приложением при обработке функцией главного окна сообщения WM_CREATE.

Размеры панели инструментов после её инициализации устанавливаются соответственно текущим размерам родительского окна. При изменении размера окна размер панели инструментов автоматически изменяться не будет. Для решения этой проблемы необходимо послать сообщение TB_AUTOSIZE панели инструментов сразу после ее создания и посылать каждый раз, когда изменяются размеры родительского окна. Это можно сделать при обработке сообщения WM_SIZE. Параметры wParam и lParam этого сообщения должны быть равны нулю, например:

```
SendMessage(hWndTb, TB_AUTOSIZE, 0, 0);
```

Необходимо отметить, что панель инструментов занимает часть клиентской области главного окно приложения. Поэтому необходимо корректировать размеры и начало координат клиентской области при появлении панели инструментов. Экранные координаты в пикселях (относительно левого верхнего угла экрана) окна панели инструментов можно получить с помощью функции

```
BOOL GetWindowRect(
     HWND hWnd,      //дескриптор дочернего окна панели
                     //инструментов
     LPRECT lpRect   //адрес структурной переменной типа RECT
   );
```

3.1.1. Приложение с главным меню и панелью инструментов

```
/*
Файл resource.h
*/

//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by 1112.rc
//
#define IDR_TOOLBAR1                    101
#define IDD_DIALOGBAR                   103
#define IDC_BUTTON1                     1002
#define IDC_STATIC_1                    1003
#define ID_BUTTON40001                  40001
#define ID_BUTTON40002                  40002
#define ID_BUTTON40003                  40003

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        103
#define _APS_NEXT_COMMAND_VALUE         40004
#define _APS_NEXT_CONTROL_VALUE         1004
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
/*Файл описания ресаурсов 1112.rc частично приведен на стр.
59*/
/*
Файл 1112.cpp
*/
/*#define UNICODE
#ifdef UNICODE
#define _UNICODE
#endif
*/
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
```

```
/*
Файл tchar.h состоит из макросов, которые ссылаются на UNICODE
данные и функции, если определен макрос UNICODE, и на ANSI
данные и функции, если этот макрос не определен, кроме того он
полностью заменяет файл string.h
*/
#include <tchar.h>
#include "resource.h"
/*
Идентификаторы пунктов меню приведены для наглядности, лучше
всего их повестить в файл resource.h
*/
#define ID_FILE_TEST 40001
#define ID_FILE_EXIT 40002
#define ID_HELP_ABOUT 40003
/*
Идентификаторы панели инструментов и строки состояния
*/
#define IDT_TOOLBAR 400
/*Прототип оконной функции*/
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
/*Прототип функции модального диалога*/
INT_PTR CALLBACK AboutProc(HWND, UINT, WPARAM, LPARAM);
/*
Прототип функция обратного вызова обработки сообщений
от таймера.
*/
VOID CALLBACK TimerProc(HWND,UINT,UINT_PTR,DWORD);
/*
Прототип функции получения текущего времени
и преоброзование его в символы.
*/
void OutTimeDate(HWND);
/*
Дескрипторы всплывающих меню и дескриптор главного меню
*/
HMENU hFileMenu, hHelpMenu, hMenu;
/*Будет создан логический шрифт*/
HFONT hFont, hOldFont;
/*
Дескриптор панели инструментов.
*/
HWND hWndTb;
/*Массив для формирования строки - текущие дата и время*/
TCHAR szCurrentTime[40];
/*Однократный интервал 5с*/
int nTime=5;
/*Массив структур типа TBBUTTON*/
TBBUTTON tBb[]=
{
     {0,ID_FILE_TEST,TBSTATE_ENABLED,TBSTYLE_BUTTON,0L,0},
     {1,ID_FILE_EXIT,TBSTATE_ENABLED,TBSTYLE_BUTTON,0L,0},
     {0,0,TBSTATE_ENABLED,TBSTYLE_SEP,0L,1},
     {2,ID_HELP_ABOUT,TBSTATE_ENABLED,TBSTYLE_BUTTON,0L,0}
};

int WINAPI WinMain(HINSTANCE hInstance,
     HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
```

```
 /*
 Произвольное имя класса
*/
 TCHAR szWindowClass[]=_TEXT("QWERTY");
/*
 Произвольный заголовок окна
*/
 TCHAR szTitle[]=_TEXT("ИТМО");
/*
Структурная переменная Msg типа MSG для получения сообще-
ний Windows
*/
 MSG msg;
/*
Структурная переменная wcex типа WNDCLASSEX для задания
характеристик окна
*/
 WNDCLASSEX wcex;
 /*Дескриптор главного окна*/
 HWND hWnd;
/*
Проверяем, было ли это приложение запущено ранее.
*/
 if(hWnd=FindWindow(szWindowClass,NULL))
 {
    /*
    Поверяем, было ли это окно свернуто в пиктограмму.
    */
      if(IsIconic(hWnd))
           ShowWindow(hWnd,SW_RESTORE);
    /*
    Выдвигаем окно приложения на передний план.
    */
      SetForegroundWindow(hWnd);
      return FALSE;
 }
 /*Обнуление всех членов структуры wcex*/
 memset(&wcex,0,sizeof(WNDCLASSEX));

 /*Регистрируем класс главного окна*/
 wcex.cbSize = sizeof(WNDCLASSEX);
 wcex.style     = CS_HREDRAW | CS_VREDRAW;
/*
Определяем оконную процедуру для главного окна
*/
 wcex.lpfnWndProc= (WNDPROC)WndProc;
 //wcex.cbClsExtra   = 0;
 //wcex.cbWndExtra   = 0;
 wcex.hInstance = hInstance; //Дескриптор приложения
 /*
 Стандартная пиктограмма, которую можно загрузить функцией
 LoadIcon(hInstance, MAKEINTRESOURCE(IDI_?_?))
 */
 wcex.hIcon = (HICON)LoadImage(hInstance,
                IDI_APPLICATION, IMAGE_ICON,32,32,0);
 /*Стандартный курсор мыши*/
 wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
 /*
 Кисть фона и ее цвет можно определить выражениями:
```

```
wcex.hbrBackground=(HBRUSH)(COLOR_APPWORKSPACE+1);
wcex.hbrBackground=(HBRUSH)GetStockObject(LTGRAY_BRUSH);
или с помощью  макроса  GetStockBrush(), в этом случае
необходимо подключить файл windowsx.h
*/
wcex.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
/*wcex.lpszMenuName = MAKEINTRESOURCE(IDC_MSG_1);*/
/*Имя класса главного окна*/
wcex.lpszClassName  = szWindowClass;
wcex.hIconSm        = NULL;
/* Маленькую пиктограмму можно загрузить
фукцией LoadImage()*/
if(!RegisterClassEx(&wcex))
     return FALSE;
/*
Инициализация библиотеки Common Control Library
*/
INITCOMMONCONTROLSEX iCc;
iCc.dwSize=sizeof(INITCOMMONCONTROLSEX);
iCc.dwICC=ICC_WIN95_CLASSES;
InitCommonControlsEx(&iCc);

/*Создаем главное окно и делаем его видимым*/
hWnd = CreateWindowEx(WS_EX_WINDOWEDGE,szWindowClass,
                    szTitle,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,0,CW_USEDEFAULT,0,
                    NULL, NULL, hInstance, NULL);
if (!hWnd)
     return FALSE;
/*
Исследуем адресное пространство приложения. Выведем
содержимое сегментных регистров команд, данных и
стека, а также смещение главной функции и строки
с именем класса
*/
TCHAR szAsm[80];
USHORT regCS,regDS,regES,regSS;
     __asm{
               mov regCS,CS
               mov regDS,DS
               mov regES,ES
               mov regSS,SS
          }
wsprintf((LPTSTR)szAsm,_T("CS=%X,DS=%X\nES=%X,SS=%X\n
                    WinMain=%X\nszWindowClass=%X"),
                    regCS,regDS,regES,regSS);
MessageBox(NULL,(LPCTSTR)szAsm,_T("Регистры"),
                              MB_ICONINFORMATION);

/*
Создаем  пустое всплывающее меню самого низкого
уровня hFileMenu=CreatePopupMenu()и добавляем в
него конечный элемент "Test"
*/
AppendMenu((hFileMenu=CreatePopupMenu()),
                    MF_ENABLED | MFT_STRING,
                    ID_FILE_TEST,_TEXT("&Test"));
/*
```

```
      Добавляем в созданное меню  конечный элемент "Exit"
      */
      AppendMenu(hFileMenu,MF_GRAYED | MFT_STRING,
                           ID_FILE_EXIT,_TEXT("&Exit"));
      /*
      Создаем  пустое всплывающее меню самого низкого
      уровня hHelpMenu=CreatePopupMenu()и добавляем в
      него конечный элемент "About"
      */
      AppendMenu((hHelpMenu=CreatePopupMenu()),
                           MF_ENABLED|MFT_STRING,
                           ID_HELP_ABOUT,_TEXT("&About"));
      /*
      Создаем меню верхнего уровня - главное меню
      hMenu=CreateMenu()и присоединяем созданное подменю"File"
      к главному меню"
      */
      AppendMenu((hMenu=CreateMenu()),
                     MF_ENABLED | MF_POPUP,
                     (UINT_PTR)hFileMenu, _TEXT("&File"));
      /*
      Присоединяем созданное подменю "Help" к главному
      Меню
      */
      AppendMenu(hMenu,MF_ENABLED|MF_POPUP,
                     (UINT_PTR)hHelpMenu,_TEXT("&Help"));
      /
      *Добавляем в конец главного меню конечный пункт
      "QUIT"
      */
      AppendMenu(hMenu,MF_GRAYED,(UINT_PTR)11,
                                          _TEXT("Quit"));
      /*Присоединяем созданное меню к окну приложения*/
      SetMenu(hWnd,hMenu);
      /*Делаем окно видимым на экране*/
      ShowWindow(hWnd, SW_SHOWNORMAL);
      /*
      Функция UpdateWindow() вызывает передачу
      сообщения WM_PAINT непосредственно оконной
      процедуре, а  функция InvalidateRect()вызывает
      постановку сообщения WM_PAINT в очередь приложения,
      а там оно обрабатывается с самым низким приоритетом
      */
      UpdateWindow(hWnd);
      /*Отображаем меню*/
      DrawMenuBar(hWnd);

       /*Цикл обработки сообщений*/

      while (GetMessage(&msg, NULL, 0, 0))
      {
           TranslateMessage(&msg);
           DispatchMessage(&msg);
      }
      /*Код возврата*/
      return (int)msg.wParam;
}
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                                 WPARAM wParam, LPARAM lParam)
```

```
{
     /*Горизонтальный размер главного окна*/
     int xSize=500;
     /*Верикальный размер главного окна*/
     int ySize=300;
     /* PAINTSTRUCT - структура с характеристиками рабочей
     области(структура заполняется функцией BeginPaint).
     */
     PAINTSTRUCT ps;
     /*
     TEXTMETRIC - структура для получения
     характеристик шрифта.
     */
     TEXTMETRIC tm;
     /*
     LOGFONT - структура для для создания
     логических шрифтов.
     */
     LOGFONT lf;
     Static HFONT hFont,hOldFont;
     /*
     RECT–структура, определяющая прямоугольник.
     */
     RECT rect, rcTb;
     HDC hDc;
     /*
     Высота ранели инструментов HeightTb */
     static UINT, HeightTb;
     /*
     Ширина width и высота height клиентской области окна
     в пикселях.
     */
     UINT width,height;
     int aWidth[PartN],SbWidth;

     LPTSTR lpszFace=_TEXT("Times New Roman Cyr");
     switch (message)
     {
     case WM_CREATE:
     /*
     Только здесь можно произвести модификацию класса
     окна. Например, SetClassLong(hWnd, GCL_HBRBACKGROUND,
               (LONG)CreateSolidBrush(RGB(200,160,255));
     Значение дескриптор экземпляра приложения (hInstance)
     определяется, вызовом одной из следующих функций:
     hInst = GetModuleHandle(NULL);
     hInst = (HINSTANCE)GetClassLong(hWnd,GCL_HMODULE);
     */
          /*Обнуление всех членов структуры lf*/
          memset(&lf,0,sizeof(lf));
          /*Устанавливаем размер шрифта*/
          lf.lfHeight=30;
          /*Копируем в структуру имя шрифта*/
          lstrcpy(lf.lfFaceName,lpszFace);
          /*Создаем шрифт*/
          hFont=CreateFontIndirect(&lf);
          /*Первый немедленный вывод текущего времени*/
          OutTimeDate(hWnd);
          /*
          Функция SetTimer создает системный таймер
```

```
      с периодом 1с
      */
      SetTimer(hWnd,1,1000,(TIMERPROC)NULL);
      /*Создаем панель инструментов Toolbar*/
      hWndTb=CreateToolbarEx(hWnd,
      WS_CHILD|WS_VISIBLE|WS_BORDER|TBSTYLE_TOOLTIPS,
                /*Идентификатор панели инструментов*/
                IDT_TOOLBAR,
                3,
                GetModuleHandle(NULL),
                /*Идентификатор ресурса растровог
                 образа
                */
                IDR_TOOLBAR1,
                (LPCTBBUTTON)tBb,
                4,
                0,0,
                0,0,
                sizeof(TBBUTTON));
                if(hWndTb==NULL)
                          return FALSE;
      /*
      Определяем размеры прямоугольника ToolBar
      в экранных координатах(пикселях).
      */
      GetWindowRect(hWndTb,&rcTb);
      /* Высота ToolBar в пикселях.*/
      HeightTb=rcTb.bottom-rcTb.top;
      return TRUE;
case WM_TIMER:
/*
Функция OutTimeDate запрашивает у системы текущие
значения даты и времени, а затем организует
их обработку в главном окне приложения.
*/
      OutTimeDate(hWnd);
      break;
case WM_KEYDOWN:
      switch(wParam)
      /*Обрабатываем сообщение-нажатие клавиши.*/
      switch(wParam)
      {
           case VK_ESCAPE:
         /*
         Посылаем сообщение WM_CLOSE окну (hWnd), после
         того, как оконная процедура обработает это сооб-
         щение, система передаст управление инструкции
         следующей за SendMessage
         */
                SendMessage(hWnd,WM_CLOSE,0,0);
                break;
      }
      break;
case WM_COMMAND:
      switch(LOWORD(wParam)) //switch(wParam)
      {
      case ID_FILE_TEST:
        /*
        Изменяем статус пункта меню ID_FILE_EXIT.
        */
```

```
            EnableMenuItem(hFileMenu,ID_FILE_EXIT,
                           MF_BYCOMMAND|MF_ENABLED);
          /*
          Ставим отметку(галочку) на пункте меню
          ID_FILE_TEST
          */
            CheckMenuItem(hFileMenu,ID_FILE_TEST,
                           MF_BYCOMMAND|MF_CHECKED);
          /*Изменяем статус пункта главного меню "QUIT"*/
            EnableMenuItem(GetMenu(hWnd),
                 (UINT_PTR)11,MF_BYCOMMAND|MF_ENABLED);
          /*
          Так как изменился статус пункта главного
          меню,вызываем функцию DrawMenuBar для повторного
          отображения изменившейся полосы меню
          */
            DrawMenuBar(hWnd);
            /*Устанавливаем таймер на nTime секунд*/
            SetTimer(hWnd,2,nTime*1000,
                                 (TIMERPROC)TimerProc);
            break;
        case ID_FILE_EXIT:
           /*
           Без запроса на закрытие окна - функция
           PostQuitMessage посылает сообщение WM_QUIT
           */
             PostQuitMessage(0);
           /*
           С запросом на закрытие, т.е. окно еще не разруше-
           но, функция SendMessage(hWnd,WM_CLOSE,0,0) посы-
           лает сообщение WM_CLOSE;
           */
             break;
        case ID_HELP_ABOUT:
        /*
        Функция DialogBox создает и выводит на экран мо-
        дальное диалоговое окно по шаблону IDD_ABOUTBOX, и
        не возвращает управление в WndProc пока окно диало-
        ка не будет закрыто
        */
            DialogBox(GetModuleHandle(NULL),
                      MAKEINTRESOURCE(IDD_DIALOGBAR),
                      hWnd, (DLGPROC)AboutProc);
            break;
        case (UINT)11:
           /*
           Без запроса на закрытие окна - функция
           PostQuitMessage посылает сообщение WM_QUIT
           */
             PostQuitMessage(0);
             break;
        }
        break;
    case WM_PAINT:
        /*Получаем контекст устройства*/
        hDc = BeginPaint(hWnd, &ps);
        /*Выбираем в контест созданный шрифт*/
        hOldFont=SelectFont(hDc,hFont);
        /*Получим метрики текста (при необходимости)*/
        GetTextMetrics(hDc,&tm);
```

```
     /*
     Определяем размеры клиентской области окна
     с учетом окна панели инструментов .
     */
     GetClientRect(hWnd,&rect);
     rect.top+=HeightTb;
/*
Функция SetBkMode устанавливает текущий режим фона.
TRANSPARENT - в этом режиме вывода текста цвет фона гра-
фического элемента игнорируется, т.е. символ выводится на
существующем фоне
*/
     SetBkMode(hDc,TRANSPARENT);
/*
Функция SetTextColor устанавливает цвет текста для кон-
текста устройства, по умолчанию применяется черный цвет.
Цвет текста синий!
*/
     SetTextColor(hDc,RGB(0,0,128));
     DrawText(hDc,szCurrentTime,-1,&rect,
               DT_SINGLELINE|DT_CENTER|DT_VCENTER);
     /*Освобождаем контекст устройства*/
     EndPaint(hWnd, &ps);
     break;

case WM_CLOSE:
     /*
    Сообщение WM_CLOSE появляется при щелчке на кнопке
    закрытия окна - здесь предназначено для вывода преду-
    преждающего сообщения
    */
     if(MessageBox(hWnd,_T("Вы уверены?"),
          _T("Предупреждение!"),
          MB_YESNO | MB_ICONQUESTION)==IDYES)
          {
     /*
     Функция DestroyWindow  разрушает указанное в ее па-
     раметре окно, т.е. она посылает окну сообщение
     WM_DESTROY. Затем вызывается функция
     PostQuitMessage, которая посылает сообщение WM_QUIT
     */
               DestroyWindow(hWnd);
          }
     break;

case WM_SIZE:
/*
Ширина width и высота height клиентской области окна
в пикселях
*/
     width=LOWORD(lParam);
     height=HIWORD(lParam);

     /*
     Изменяем размеры Toolbar в соответствии
     с новыми размерами окна,можно и так
     SendMessage(hWndTb,WM_SIZE,wParam,lParam);
     */
     SendMessage(hWndTb,TB_AUTOSIZE,0,0);
     break;
```

```
     case WM_LBUTTONDOWN:
          if(wParam & MK_SHIFT)
          {
               MessageBox(hWnd,_T("Нажата клавиша\nShift"),
                              _T("Уведомление!"),
                              MB_OK | MB_ICONEXCLAMATION);
          }
          break;
     case WM_GETMINMAXINFO:
          lpmmi=(LPMINMAXINFO)lParam;
          /*
          Минимальный и максимальный размеры
          окна совпадают
          */
          lpmmi->ptMinTrackSize.x=xSize;
          lpmmi->ptMinTrackSize.y=ySize;
          lpmmi->ptMaxTrackSize.x=xSize;
          lpmmi->ptMaxTrackSize.y=ySize;
          break;
     case WM_DESTROY:
     /*
     Функция DeleteObject удаляет логический объект. К удаляе-
     мым объектам относятся перья, растровые изображения, кис-
     ти, области, палитры и шрифты.
     */
          /*Удаляем созданный шрифт*/
          DeleteObject(hFont);
          /*Функция KillTimer удаляет таймер*/
          KillTimer(hWnd,1);
         /*
         PostQuitMessage() выполняет только одно действие -
         ставит в очередь сообщение WM_QUIT. Параметр у этой
         функции - код возврата, который помещается в wParam
         */
          PostQuitMessage(0);
          break;
     default:
         /*Обработка прочих сообщений по умолчанию*/
          return DefWindowProc(hWnd, message, wParam,lParam);
     }
     return 0L;
}
/*Оконная функция диалогового окна*/
INT_PTR CALLBACK AboutProc(HWND hDlg, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
     switch (message)
     {
          case WM_INITDIALOG:
     /*
     Для обрабатываемых сообщений процедура всегда
     возвращает TRUE.
     */
               return (INT_PTR)TRUE;
          case WM_COMMAND:
               if(LOWORD(wParam)==IDOK || LOWORD(wParam)==
                                                  IDCANCEL)
               {
                    EndDialog(hDlg, LOWORD(wParam));
                    return (INT_PTR)TRUE;
```

```
                }
                break;
        }
        /*
        Для не обрабатываемых сообщений процедура всегда
        возвращает FALSE.
        */
        return (INT_PTR)FALSE;
}
/*
Функция получения текущего времени и
преобразование его в символы
*/
void OutTimeDate(HWND hWnd)
{
        LPTSTR szDay[]={_T("Вск."),_T("Пнд."),_T("Втр."),
                                _T("Ср."),_T("Чтв."),
                                _T("Птн."),_T("Суб.")
                        };

        LPTSTR szMonth[]={_T(""),_T("Янв."),_T("Февр."),
                                _T("Март"),_T("Апр."),
                                _T("Май"),_T("Июнь"),
                                _T("Июль"),_T("Авг."),
                                _T("Сент."),_T("Окт."),
                                _T("Нояб."),_T("Дек.")
                            };
        TCHAR szT[20];
        SYSTEMTIME SystemTime;
        /*
        Функция GetLocalTime осуществляет выборку местного време-
        ни,на которое настроен компьютер, т.е. функция
        заполняет структуру типа SYSTEMTIME в числовом виде.
        */
        GetLocalTime(&SystemTime);
        /*День недели*/
        lstrcpy(szCurrentTime,
                            szDay[SystemTime.wDayOfWeek]);
        /*Разделяющий пробел*/
        lstrcat((LPTSTR)szCurrentTime,_T(" "));
        /*Месяц*/
        lstrcat((LPTSTR)szCurrentTime,
                            szMonth[SystemTime.wMonth]);
        /*Разделяющий пробел*/
        lstrcat((LPTSTR)szCurrentTime,_T(" "));
        /*Дату переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wDay);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющий пробел*/
        lstrcat((LPTSTR)szCurrentTime,_T(" "));
        /*Год переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wYear);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        lstrcat((LPTSTR)szCurrentTime,_T("---"));
        /*Часы переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wHour);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
```

```
      /*Разделяющее двоеточие*/
      lstrcat((LPTSTR)szCurrentTime,_T(":"));
      /*Минуты переводим в символы*/
      wsprintf((LPTSTR)szT,_T("%d"),
                                    SystemTime.wMinute);
      lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
      /*Разделяющее двоеточие*/
      lstrcat((LPTSTR)szCurrentTime,_T(":"));
      /*Секунды переводим в символы*/
      wsprintf((LPTSTR)szT,_T("%d"),
                                    SystemTime.wSecond);
      lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
      /*Перерисовка окна*/
      InvalidateRect(hWnd,NULL,TRUE);
}


/*Функция обратного вызова обработки сообщений от таймера*/
VOID CALLBACK TimerProc(HWND hWnd,UINT uMsg,
                        UINT_PTR idEvent,DWORD dwTime)
{
      TCHAR szTimer[100];
      KillTimer(hWnd,2);
      wsprintf(szTimer,
_T("С момента выбора\nпункта меню Test\nпрошло %d с!"),
                                                nTime);
      MessageBox(NULL,(LPCTSTR)szTimer,
                  _T("Предупреждение"),MB_ICONHAND);
}
```

Результат работы программы приведен на рис. 4.

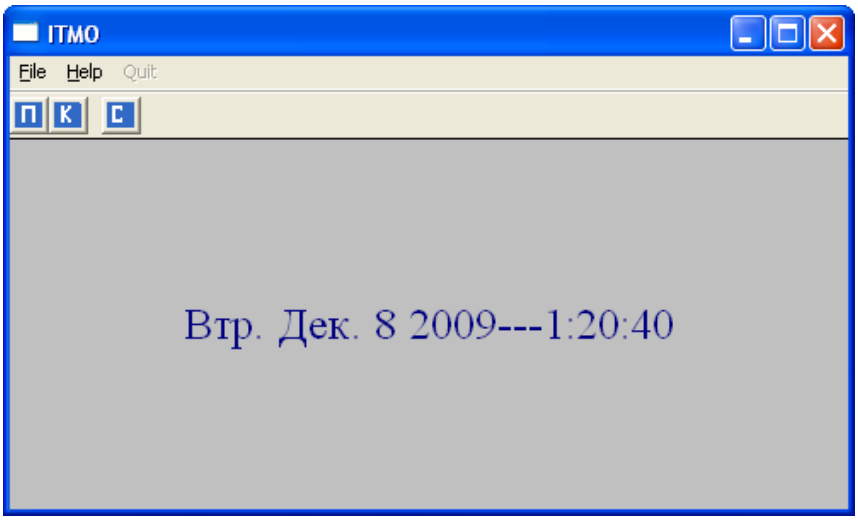


Рис. 4. Окно приложения с главным меню и панелью инструментов

3.2. Строка состояния

Строка состояния (`status bar`) – дочернее окно, расположенное в нижней части главного окна приложения и предназначенное для вывода информации о состоянии программы или о выполняемых операциях. Ширина строки равна ширине клиентской области родительского окна, а вы-

сота устанавливается на основе метрик шрифта, выбранного по умолчанию в контексте устройства элемента управления.

Для создания строки состояния служит функция CreateStatusWindow(), которая имеет следующий прототип:

```
HWND CreateStatusWindow(
   LONG style,       //стиль – должны быть включены флаги
                     //WS_CHILD | WS_VISIBLE
   LPCTSTR lpszText,//текст для первой области status bar
   HWND hWndParent, //идентификатор родительского окна
   UINT wID          //идентификатор status bar
);
```

Функция CreateStatusWindow() создает дочернее окно и выводит в окно строки состояния текст, который задан вторым параметром lpszText. Возвращаемое значение функции при успешном ее завершении – дескриптор строки состояния, а в противном случае – NULL.

Строка состояния может работать в двух режимах:

1) стандартном – строка состояния разбивается на несколько частей, для вывода в них текста или графической информации;

2) упрощенном – строка состояния реализована как единый элемент, в который можно выводить только текстовую информацию.

Переключение между режимами осуществляется при помощи сообщения SB_SIMPLE. Параметр lParam этого сообщения должен быть равен нулю, а параметр wParam может принимать следующие значения: TRUE – status bar переключается в упрощенный режим, FALSE – возвращается в стандартный режим. Например:

```
SendMessage(hwndSb,SB_SETTEXT,0,
```

Если строка состояния работает в стандартном режиме, то ее разделение на отдельные поля осуществляется посылкой сообщения SB_SETPARTS сразу после создания панели инструментов, например:

```
SendMessage(hwndSb, SB_SETPARTS, 3, (LPARAM)ptWidth);
```

Через параметр wParam передается количество полей, а через параметр lParam – адрес целочисленного массива, количество элементов которого равно количеству областей, на которые делится status bar. Элемент массива содержит значение ширины (в пикселях), отсчитываемой от левого края окна status bar до правой границы области, либо –1. В последнем случае правая граница области есть правая граница строки состояния.

В любом поле строки состояния можно поместить и любой другой элемент управления, например индикатор процесса (progress bar). В этом случае необходимо знать клиентские координаты поля, которые можно получить посылкой сообщения SB_GETRECT, например:

```
SendMessage(hwndSb, SB_GETRECT,2,(LPARAM)&rect);
```

Параметр wParam – это номер поля, отсчитываемый от нуля, а lParam – адрес структуры типа RECT, принимающей координаты поля.

Для записи текста в область строки состояния необходимо использовать сообщение SB_SETTEXT. Например:

```
SendMessage(hwndSb,SB_SETTEXT,0,
```

```
                    (LPARAM)"\tStatus bar Part 1 ");
```
или
```
SendMessage(hwndSb, SB_SETTEXT,1 | SBT_NOBORDERS,
                                                (LPARAM)"");
```

Параметр wParam содержит номер области, которая может быть скомбинирована при помощи логической операции ИЛИ с одной из констант, определяющих внешний вид области:

- SBT_NOBORDERS – поле рисуется без рамки;
- SBT_POPOUT – поле рисуется с выпуклой рамкой;
- SBT_OWNERDRAW – текст (или графическое изображение) рисуется родительским окном во время обработки сообщения WM_DRAWITEM.

Параметр lParam содержит указатель на строку текста с нулевым символом в конце, которая должна быть записана в область. Текстовая строка может быть пустой. Для выравнивания текста по центру или по правому краю необходимо включить в текстовую строку символ табуляции \t. Текст, расположенный после первого символа табуляции, выравнивается по центру, после второго– по правому краю области.

Изменить высоту строки можно, послав сообщение SB_SETMINHEIGHT, например:
```
SendMessage(hwndSb, SB_SETMINHEIGHT, minHeight, 0);
```

Параметр wParam (minHeight) – минимальная высота окна строки состояния в пикселях, а значение параметра lParam должно быть равно нулю.

При каждом изменении размеров родительского окна, т.е. при получении сообщения WM_SIZE , оконная процедура главного окна должна отправить строке состояния такое же сообщение (WM_SIZE), передав текущее значение параметров wParam и lParam, например:
```
SendMessage(hwndSb, WM_SIZE, wParam, lParam);
```

Необходимо отметить, что строка состояния занимает часть клиентской области главного окна приложения. Поэтому необходимо корректировать размеры и начало координат клиентской области при появлении строки состояния.

3.2.1. Приложение с главным меню, панелью инструментов и строкой состояния

Модифицируем диалоговое окно, т.е. изменим шрифт в статическом элементе этого окна. Модификацию произведем в функции диалогового окна при обработки сообщения WM_INITDIALOG.

Диалоговые окна бывают модальные(modal) и немодальные (modeless). Функция DialogBox создает и выводит на экран модальное диалоговое окно по определенному шаблону.

Для определения шаблона диалогового окна в файле ресурсов

необходимо в главном меню выполнить следующую команду Project->Add Resource->Dialog->New. В результате будет открыто окно редактора диалоговых окон.

Проектирование шаблона диалогового окна начинается с установки его свойств. Закончив проектирование шаблона его необходимо сохранить в файле описания ресурсов 1113.rc. Редактор присваивает шаблону идентификатор IDD_DIALOGBAR.

Файлы resource.h и 1113.rc аналогичны файлам приложения листинг 3.1.1.

```
/*Файл 1113.cpp*/

/*#define UNICODE
#ifdef UNICODE
#define _UNICODE
#endif
*/
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
/*
Файл tchar.h состоит из макросов, которые ссылаются на UNICODE
данные и функции, если определен макрос UNICODE, и на ANSI
данные и функции, если этот макрос не определен, кроме того он
полностью заменяет файл string.h
*/
#include <tchar.h>
#include "resource.h"
/*
Идентификаторы пунктов меню приведены для наглядности, лучше
всего их повестить в файл resource.h
*/
#define ID_FILE_TEST 40001
#define ID_FILE_EXIT 40002
#define ID_HELP_ABOUT 40003
/*
Идентификаторы панели инструментов и строки состояния
*/
#define IDT_TOOLBAR 400
#define IDS_STATUSBAR 401
#define PartN 3
/*Прототип оконной функции*/
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
/*Прототип функции модального диалога*/
INT_PTR CALLBACK AboutProc(HWND, UINT, WPARAM, LPARAM);
/*
Прототип функция обратного вызова обработки сообщений
от таймера
*/
VOID CALLBACK TimerProc(HWND,UINT,UINT_PTR,DWORD);
/*
Прототип функции получения текущего времени
и преобразование его в символы
*/
void OutTimeDate(HWND);
/*
```

```
Дескрипторы всплывающих меню и дескриптор главного меню
*/
HMENU hFileMenu,hHelpMenu,hMenu;
/*
Дескрипторы панели инструментов и строки состояния
*/
HWND hWndSb,hWndTb;
/*Массив для формирования строки - текущие дата и время*/
TCHAR szCurrentTime[40];
/*Однократный интервал 5с*/
int nTime=5;
/*Массив структур типа TBBUTTON*/
TBBUTTON tBb[]=
{
     {0,ID_FILE_TEST,TBSTATE_ENABLED,TBSTYLE_BUTTON,0L,0},
     {1,ID_FILE_EXIT,TBSTATE_ENABLED,TBSTYLE_BUTTON,0L,0},
     {0,0,TBSTATE_ENABLED,TBSTYLE_SEP,0L,1},
     {2,ID_HELP_ABOUT,TBSTATE_ENABLED,TBSTYLE_BUTTON,0L,0}
};

int WINAPI WinMain(HINSTANCE hInstance,
     HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{

    /* Произвольное имя класса */
     TCHAR szWindowClass[]=_TEXT("QWERTY");
    /* Произвольный заголовок окна */
     TCHAR szTitle[]=_TEXT("ИТМО");
    /*
    Структурная переменная Msg типа MSG для получения сообще-
    ний Windows
    */
     MSG msg;
    /*
    Структурная переменная wcex типа WNDCLASSEX для задания
    характеристик окна
    */
     WNDCLASSEX wcex;
     HWND hWnd; //Дескриптор главного окна
    /*
    Проверяем, было ли это приложение запущено ранее.
    */
     if(hWnd=FindWindow(szWindowClass,NULL))
     {
        /*
        Поверяем, было ли это окно свернуто в пиктограмму.
        */
          if(IsIconic(hWnd))
               ShowWindow(hWnd,SW_RESTORE);
        /*
        Выдвигаем окно приложения на передний план
        */
          SetForegroundWindow(hWnd);
          return FALSE;
     }
     /*Обнуление всех членов структуры wcex*/
     memset(&wcex,0,sizeof(WNDCLASSEX));
     /*Регистрируем класс главного окна*/
```

```
 wcex.cbSize = sizeof(WNDCLASSEX);
 wcex.style     = CS_HREDRAW | CS_VREDRAW;
/*
Определяем оконную процедуру для главного окна
*/
 wcex.lpfnWndProc= (WNDPROC)WndProc;
 //wcex.cbClsExtra   = 0;
 //wcex.cbWndExtra   = 0;
 wcex.hInstance = hInstance; //Дескриптор приложения
 /*
 Стандартная пиктограмма, которую можно загрузить
 Функцией  LoadImage().
 */
 wcex.hIcon = (HICON)LoadImage(hInstance,
                IDI_APPLICATION, IMAGE_ICON,32,32,0);
 /*Стандартный курсор мыши*/
 wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
 /*
 Кисть фона и ее цвет можно определить выражениями:
 wcex.hbrBackground=(HBRUSH)(COLOR_APPWORKSPACE+1);
 wcex.hbrBackground=(HBRUSH)GetStockObject(LTGRAY_BRUSH);
 или с помощью  макроса  GetStockBrush(), в этом случае
 необходимо подключить файл windowsx.h
 */
 wcex.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
 //wcex.lpszMenuName = MAKEINTRESOURCE(IDC_MSG_1);
 /*Имя класса главного окна*/
 wcex.lpszClassName  = szWindowClass;
 wcex.hIconSm        = NULL;
 /* или так LoadIcon(wcex.hInstance,
                 MAKEINTRESOURCE(IDI_SMALL));*/
 if(!RegisterClassEx(&wcex))
      return FALSE;
 /*
 Инициализация библиотеки Common Control Library
 */
 INITCOMMONCONTROLSEX iCc;
 iCc.dwSize=sizeof(INITCOMMONCONTROLSEX);
 iCc.dwICC=ICC_WIN95_CLASSES;
 InitCommonControlsEx(&iCc);
 /*Создаем главное окно и делаем его видимым*/
 hWnd = CreateWindowEx(WS_EX_WINDOWEDGE,szWindowClass,
                    szTitle,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,0,CW_USEDEFAULT,0,
                    NULL, NULL, hInstance, NULL);
 if (!hWnd)
      return FALSE;
 /*
 Исследуем адресное пространство приложения. Выведем
 содержимое сегментных регистров команд, данных и
 стека, а также смещение главной функции и строки
 с именем класса
 */
 TCHAR szAsm[80];
 USHORT regCS,regDS,regES,regSS;
      __asm{
                mov regCS,CS
                mov regDS,DS
```

```
                mov regES,ES
                mov regSS,SS
            }
wsprintf((LPTSTR)szAsm,_T("CS=%X,DS=%X\nES=%X,SS=%X\n
                    WinMain=%X\nszWindowClass=%X"),
                    regCS,regDS,regES,regSS);
MessageBox(NULL,(LPCTSTR)szAsm,_T("Регистры"),
                                MB_ICONINFORMATION);

/*
Создаем  пустое всплывающее меню самого низкого
уровня hFileMenu=CreatePopupMenu()и добавляем в
него конечный элемент "Test"
*/    AppendMenu((hFileMenu=CreatePopupMenu()),
                MF_ENABLED | MFT_STRING,
                ID_FILE_TEST, _TEXT("&Test"));
/*
Добавляем в созданное меню  конечный элемент "Exit"
*/
AppendMenu(hFileMenu,MF_GRAYED | MFT_STRING,
                ID_FILE_EXIT,_TEXT("&Exit"));
/*
Создаем  пустое всплывающее меню самого низкого
уровня hHelpMenu=CreatePopupMenu()и добавляем в
него конечный элемент "About"
*/
AppendMenu((hHelpMenu=CreatePopupMenu()),
                MF_ENABLED|MFT_STRING,
                ID_HELP_ABOUT,_TEXT("&About"));
/*
Создаем меню верхнего уровня - главное меню
hMenu=CreateMenu()и присоединяем созданное
Подменю "File" к главному меню"
*/
AppendMenu((hMenu=CreateMenu()),
                MF_ENABLED | MF_POPUP,
                (UINT_PTR)hFileMenu, _TEXT("&File"));
/*
Присоединяем созданное подменю "Help" к главному
Меню
*/
AppendMenu(hMenu,MF_ENABLED|MF_POPUP,
                (UINT_PTR)hHelpMenu,_TEXT("&Help"));
/
*Добавляем в конец главного меню конечный пункт
"QUIT"
*/
AppendMenu(hMenu,MF_GRAYED,(UINT_PTR)11,
                                        _TEXT("Quit"));
/*Присоединяем созданное меню к окну приложения*/
SetMenu(hWnd,hMenu);
/*Делаем окно видимым на экране*/
ShowWindow(hWnd, SW_SHOWNORMAL);
/*
Функция UpdateWindow() вызывает передачу
сообщения WM_PAINT непосредственно оконной
процедуре, а  функция InvalidateRect()вызывает
постановку сообщения WM_PAINT в очередь приложения,
а там оно обрабатывается с самым низким приоритетом
*/
```

```
      UpdateWindow(hWnd);
      /*Отображаем меню*/
      DrawMenuBar(hWnd);
      /*Цикл обработки сообщений*/
      while (GetMessage(&msg, NULL, 0, 0))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      /*Код возврата*/
      return (int)msg.wParam;
}
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                                    WPARAM wParam, LPARAM lParam)
{
      int xSize=500;
      int ySize=300;
      /* PAINTSTRUCT - структура с характеристиками рабочей
      области (заполняется функцией BeginPaint)
      */
      PAINTSTRUCT ps;
      /*
      TEXTMETRIC - структура для получения
      характеристик шрифта.
      */
      TEXTMETRIC tm;
      /*
      LOGFONT - структура для для создания
      логических шрифтов.
      */
      LOGFONT lf;
      /*Будет создан логический шрифт*/
      Static HFONT hFont,hOldFont;
      /*
      RECT-структура,определяющая прямоугольник.
      */
      RECT rect,rcSb,rcTb;
      HDC hDc;
      /*
      Высота панели инструментов HeightTb и вывсота строки
      состояния HeightSb в пикселях.
      */
      static UINT HeightSb,HeightTb;
      /*
      Ширина width и высота height клиентской области окна
      в пикселях.
      */
      UINT width,height;
      /*Массив ширин для StatusBar*/
      int aWidth[PartN],;
      int SbWidth;
      LPTSTR lpszFace=_TEXT("Times New Roman Cyr");
      switch (message)
      {
      case WM_CREATE:
      /*
      Только здесь можно произвести модификацию класса
      окна. Например, SetClassLong(hWnd, GCL_HBRBACKGROUND,
                  (LONG)CreateSolidBrush(RGB(200,160,255));
```

```
Значение дескриптор экземпляра приложения (hInstance)
определяется, вызовом одной из следующих функций:
hInst = GetModuleHandle(NULL);
hInst = (HINSTANCE)GetClassLong(hWnd,GCL_HMODULE);
*/
     /*Обнуление всех членов структуры lf*/
     memset(&lf,0,sizeof(lf));
     /*Устанавливаем размер шрифта*/
     lf.lfHeight=30;
     /*Копируем в структуру имя шрифта*/
     lstrcpy(lf.lfFaceName,lpszFace);
     /*Создаем шрифт*/
     hFont=CreateFontIndirect(&lf);
     /*Первый немедленный вывод текущего времени*/
     OutTimeDate(hWnd);
     /*
     Функция SetTimer создает системный таймер
     с периодом 1с
     */
     SetTimer(hWnd,1,1000,(TIMERPROC)NULL);
     /*Создаем панель инструментов ToolBar*/
     hWndTb=CreateToolbarEx(hWnd,
     WS_CHILD|WS_VISIBLE|WS_BORDER|TBSTYLE_TOOLTIPS,
               IDT_TOOLBAR,
               3,
               GetModuleHandle(NULL),
               IDR_TOOLBAR1,
               (LPCTBBUTTON)tBb,
               4,
               0,0,
               0,0,
               sizeof(TBBUTTON));
     if(hWndTb==NULL)
          return FALSE;
     /*
     Определяем размеры прямоугольника ToolBar
     в экранных координатах(пикселях).
     */
     GetWindowRect(hWndTb,&rcTb);
     /* Высота ToolBar в пикселях.*/
     HeightTb=rcTb.bottom-rcTb.top;
     /*Создаем Statusbar*/
     hWndSb=CreateStatusWindow(WS_CHILD|WS_VISIBLE,
                    (LPTSTR)" ",
                    hWnd,
                    IDS_STATUSBAR);
     if(hWndSb==NULL)
               return FALSE;
     /*
     Определяем размеры прямоугольника StatusBar
     в экранных координатах(пикселях).
     */
     GetWindowRect(hWndSb,&rcSb);
   /* Высота StatusBar в пикселях */
     HeightSb=rcSb.bottom-rcSb.top;
     return TRUE;
case WM_TIMER:
/*
Функция OutTimeDate запрашивает у системы текущие
значения даты и времени, а затем организует
```

```
их обработку в главном окне приложения
*/
     OutTimeDate(hWnd);
     break;
case WM_KEYDOWN:
     switch(wParam)
     /*Обрабатываем сообщение-нажатие клавиши*/
     switch(wParam)
     {
          case VK_ESCAPE:
        /*
        Посылаем сообщение WM_CLOSE окну (hWnd), после
        того, как оконная процедура обработает это сооб-
        щение, система передаст управление инструкции
        следующей за SendMessage
        */
                SendMessage(hWnd,WM_CLOSE,0,0);
                break;
     }
     break;
case WM_COMMAND:
     switch(LOWORD(wParam)) //switch(wParam)
     {
     case ID_FILE_TEST:
       /*
       Изменяем статус пункта меню ID_FILE_EXIT
       */
          EnableMenuItem(hFileMenu,ID_FILE_EXIT,
                         MF_BYCOMMAND|MF_ENABLED);
       /*Ставим отметку на пункте меню ID_FILE_TEST*/
          CheckMenuItem(hFileMenu,ID_FILE_TEST,
                         MF_BYCOMMAND|MF_CHECKED);
       /*Изменяем статус пункта главного меню "QUIT"*/
          EnableMenuItem(GetMenu(hWnd),
                (UINT_PTR)11,MF_BYCOMMAND|MF_ENABLED);
       /*
       Так как изменился статус пункта главного
       меню, вызываем функцию DrawMenuBar для повторного
       отображения изменившейся полосы меню
       */
          DrawMenuBar(hWnd);
          /*Устанавливаем таймер на nTime секунд*/
          SetTimer(hWnd,2,nTime*1000,
                                (TIMERPROC)TimerProc);
          break;
     case ID_FILE_EXIT:
        /*
        Без запроса на закрытие окна - функция
        PostQuitMessage посылает сообщение WM_QUIT
        */
          PostQuitMessage(0);
        /*
        С запросом на закрытие, т.е. окно еще не разруше-
        но SendMessage(hWnd,WM_CLOSE,0,0);
        */
          break;
     case ID_HELP_ABOUT:
     /*
     Функция DialogBox создает и выводит на экран мо-
     дальное диалоговое окно по шаблону IDD_ABOUTBOX, и
```

```
      не возвращает управление в WndProc пока окно диало-
      ка не бедет закрыто
      */
           DialogBox(GetModuleHandle(NULL),
                     MAKEINTRESOURCE(IDD_DIALOGBAR),
                     hWnd, (DLGPROC)AboutProc);
           break;
      case (UINT)11:
         /*
         Без запроса на закрытие окна - функция
         PostQuitMessage посылает сообщение WM_QUIT
         */
           PostQuitMessage(0);
           break;
      }
      break;
case WM_PAINT:
      /*Получаем контекст устройства*/
      hDc = BeginPaint(hWnd, &ps);
      /*Выбираем в контест созданный шрифт*/
      hOldFont=SelectFont(hDc,hFont);
      /*Получим метрики текста при необходимости*/
      GetTextMetrics(hDc,&tm);
      /*
      Определяем размеры клиентской области окна
      с учетом окна панели инструментов и окна
      строки состояния
      */
      GetClientRect(hWnd,&rect);
      rect.top+=HeightTb;
      rect.bottom-=HeightSb;
/*
Функция SetBkMode устанавливает текущий режим фона.
TRANSPARENT - в этом режиме вывода текста цвет фона гра-
фического элемента игнорируется, т.е. символ выводится на
существующем фоне
*/
      SetBkMode(hDc,TRANSPARENT);
/*
Функция SetTextColor устанавливает цвет текста для кон-
текста устройства, по умолчанию применяется черный цвет.
Цвет текста синий!
*/
      SetTextColor(hDc,RGB(0,0,128));
      DrawText(hDc,szCurrentTime,-1,&rect,
                DT_SINGLELINE|DT_CENTER|DT_VCENTER);
      /*Освобождаем контекст устройства*/
      EndPaint(hWnd, &ps);
      break;
case WM_CLOSE:
      /*
     Сообщение WM_CLOSE появляется при щелчке на кнопке
     закрытия окна - здесь предназначено для вывода преду-
     преждающего сообщения
     */
      if(MessageBox(hWnd,_T("Вы уверены?"),
           _T("Предупреждение!"),
           MB_YESNO | MB_ICONQUESTION)==IDYES)
           {
      /*
```

```
     Функция DestroyWindow  разрушает указанное в ее па-
     раметре окно, т.е. она посылает окну сообщение
     WM_DESTROY. Затем вызывается функция
     PostQuitMessage, которая посылает сообщение WM_QUIT
     */
               DestroyWindow(hWnd);
          }
     break;

case WM_SIZE:
/*
Ширина width и высота height клиентской области окна в
пикселях.
*/
     width=LOWORD(lParam);
     height=HIWORD(lParam);

     /*
     Изменяем размеры Toolbar в соответствии
     с новыми размерами окна,можно и так
     SendMessage(hWndTb,WM_SIZE,wParam,lParam);
     */
     SendMessage(hWndTb,TB_AUTOSIZE,0,0);
     /*
     Изменяем размеры Statusbar в соответствии
     с новыми размерами окна.
     */
     SendMessage(hWndSb,WM_SIZE,wParam,lParam);
     /*Рассчитывает размеры областей Statusbar*/
     SbWidth=width/PartN;;
     aWidth[0]=SbWidth;
     aWidth[1]=SbWidth*2;
     aWidth[2]=-1;
     /*Устанавливаем новые размеры областей Statusbar*/
     SendMessage(hWndSb,
               SB_SETPARTS,PartN,(LPARAM)aWidth);
     /*Инициализируем строкой 0-ю область Statusbar*/
     SendMessage(hWndSb,SB_SETTEXT,0,
                         (LPARAM)_T("\tСПб ГУ ИТМО"));
     /*
     Инициализируем строкой 1-ю область Statusbar,
     поле рисуется без рамки
     */
     SendMessage(hWndSb,SB_SETTEXT,1 | SBT_NOBORDERS,
                         (LPARAM)_T("\tКафедра ПКС"));
     /*
     Клиентские координаты  2 поля можно определить так:
     SendMessage(hWndSb,SB_GETRECT,2,(LPARAM)&reSb);
     */
     break;
case WM_LBUTTONDOWN:
     if(wParam & MK_SHIFT)
     {
          MessageBox(hWnd,_T("Нажата клавиша\nShift"),
                         _T("Уведомление!"),
                         MB_OK | MB_ICONEXCLAMATION);
     }
     break;
case WM_GETMINMAXINFO:
     lpmmi=(LPMINMAXINFO)lParam;
```

```
            /*
            Минимальный и максимальный размеры
            окна совпадают
            */
            lpmmi->ptMinTrackSize.x=xSize;
            lpmmi->ptMinTrackSize.y=ySize;
            lpmmi->ptMaxTrackSize.x=xSize;
            lpmmi->ptMaxTrackSize.y=ySize;
            break;
      case WM_DESTROY:
      /*
      Функция DeleteObject удаляет логический объект. К удаляе-
      мым объектам относятся перья, растровые изображения, кис-
      ти, области, палитры и шрифты.
      */
            /*Удаляем созданный шрифт*/
            DeleteObject(hFont);
            /*Функция KillTimer удаляет таймер*/
            KillTimer(hWnd,1);
           /*
           PostQuitMessage() выполняет только одно действие -
           ставит в очередь сообщение WM_QUIT. Параметр у этой
           функции - код возврата, который помещается в wParam
           */

            PostQuitMessage(0);
            break;

      default:
          /*Обработка прочих сообщений по умолчанию*/
            return DefWindowProc(hWnd, message,wParam,lParam);
      }
      return 0L;
}
/*Оконная функция диалогового окна*/
INT_PTR CALLBACK AboutProc(HWND hDlg, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
      static HWND hStatic;
      /*Объявление экземпляра структуры LOGFONT с
      одновременной инициализацией ее полей нулями*/
      static LOGFONT lf1;
      static HFONT hFont3;
      /*Имя  шрифта для элемента управления IDC_STATIC_1
      в диалоговом окне */
      LPTSTR lpszFace1=_TEXT("Times New Roman Cyr");

      switch (message)
      {
            case WM_INITDIALOG:
                  /*Дескриптор элемента управления в диалого-
                  вом окне (hDlg) получаем по его идентифика-
                  тору (hDlg),вызывая функцию GetDlgItem*/
                  hStatic=GetDlgItem(hDlg,IDC_STATIC_1);
                  /*Устанавливаем размер шрифта*/
                  lf1.lfHeight=23;
                  /*Копируем в структуру имя шрифта*/
                  lstrcpy(lf1.lfFaceName,lpszFace1);
                  /*Создаем шрифт*/
                  hFont3=CreateFontIndirect(&lf1);
```

```
                /*Модифицируем шрифт элемента hStatic*/
                SendMessage(hStatic,WM_SETFONT,
                                (WPARAM)hFont3,TRUE);
                /*
                Для обрабатываемых сообщений процедура всегда
                возвращает TRUE
                */
                return (INT_PTR)TRUE;

           case WM_COMMAND:
                if(LOWORD(wParam)==IDOK || LOWORD(wParam)==
                                                    IDCANCEL)
                {
                     DeleteObject(hFont3);
                     EndDialog(hDlg, LOWORD(wParam));
                     return (INT_PTR)TRUE;
                }
                break;
     }
     /*
     Для не обрабатываемых сообщений процедура всегда
     возвращает FALSE.
     */
     return (INT_PTR)FALSE;
}
/
*Функция получения текущего времени и преобразование
его в символы
*/
void OutTimeDate(HWND hWnd)
{
     LPTSTR szDay[]={_T("Вск."),_T("Пнд."),_T("Втр."),
                              _T("Ср."),_T("Чтв."),
                              _T("Птн."),_T("Суб.")
                    };

     LPTSTR szMonth[]={_T(""),_T("Янв."),_T("Февр."),
                              _T("Март"),_T("Апр."),
                              _T("Май"),_T("Июнь"),
                              _T("Июль"),_T("Авг."),
                              _T("Сент."),_T("Окт."),
                              _T("Нояб."),_T("Дек.")
                         };
     TCHAR szT[20];
     SYSTEMTIME SystemTime;
     /*
     Функция GetLocalTime осуществляет выборку местного време-
     ни,на которое настроен компьютер, т.е. функция
     заполняет структуру типа SYSTEMTIME в числовом виде.
     */
     GetLocalTime(&SystemTime);
     /*День недели*/
     lstrcpy(szCurrentTime,
                         szDay[SystemTime.wDayOfWeek]);
     /*Разделяющий пробел*/
     lstrcat((LPTSTR)szCurrentTime,_T(" "));
     /*Месяц*/
     lstrcat((LPTSTR)szCurrentTime,
                         szMonth[SystemTime.wMonth]);
     /*Разделяющий пробел*/
```

```
        lstrcat((LPTSTR)szCurrentTime,_T(" "));
        /*Дату переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wDay);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющий пробел*/
        lstrcat((LPTSTR)szCurrentTime,_T(" "));
        /*Год переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wYear);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        lstrcat((LPTSTR)szCurrentTime,_T("---"));
        /*Часы переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wHour);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющее двоеточие*/
        lstrcat((LPTSTR)szCurrentTime,_T(":"));
        /*Минуты переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wMinute);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Разделяющее двоеточие*/
        lstrcat((LPTSTR)szCurrentTime,_T(":"));
        /*Секунды переводим в символы*/
        wsprintf((LPTSTR)szT,_T("%d"),
                                        SystemTime.wSecond);
        lstrcat((LPTSTR)szCurrentTime,(LPTSTR)szT);
        /*Перерисовка окна*/
        InvalidateRect(hWnd,NULL,TRUE);
}

/*Функция обратного вызова обработки сообщений от таймера*/
VOID CALLBACK TimerProc(HWND hWnd,UINT uMsg,
                            UINT_PTR idEvent,DWORD dwTime)
{
        TCHAR szTimer[100];
        KillTimer(hWnd,2);
        wsprintf(szTimer,
_T("С момента выбора\nпункта меню Test\nпрошло %d с!"),
                                                nTime);
        MessageBox(NULL,(LPCTSTR)szTimer,
                        _T("Предупреждение"),MB_ICONHAND);
}
```

Результат работы программы представлен на рис. 5.

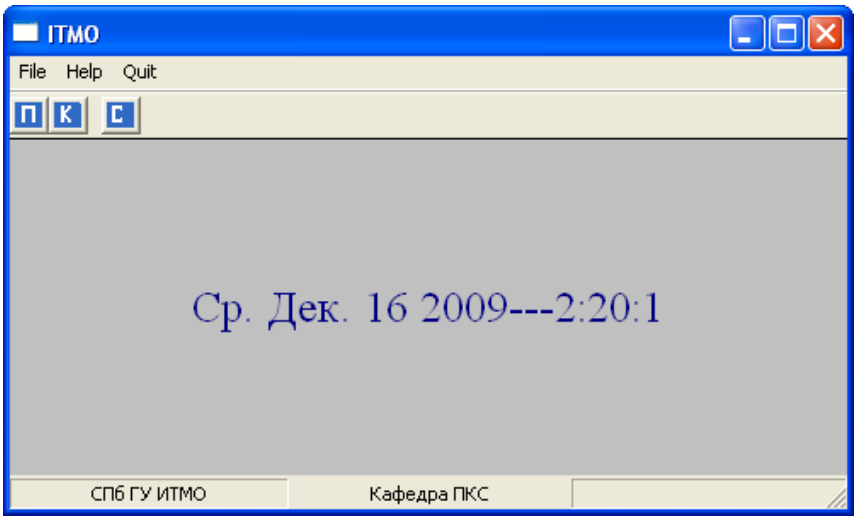


Рис. 5. Окно приложения с главным меню, панелью инструментов и строкой состояния

ПРИЛОЖЕНИЕ

Типы данных Win32

Типы данных Windows определены посредством директив `#define` и `typedef` в заголовочных файлах Win32, например:

```
#define WINAPI    __stdcall,
#define CALLBACK  __stdcall,
#define APIENTRY    WINAPI,
typedef  UINT    WPARAM,
typedef  LONG   LPARAM,
typedef  LONG   LRESULT,
typedef  WORD  ATOM.
```

В Win32 все указатели являются ближними, хотя и занимают четыре байта (32-разрядное целое число без знака).

Наиболее часто встречающиеся типы данных:

`BOOL` – `int` (32 бита);
`BYTE` – `unsigned char` (8 бит);
`CHAR` – `char` (8 бит);
`COLORREF` – `unsigneg long` (32 бита);
`DWORD` – `unsigned long` (32 бита);
`DOUBLE` – вещественное число (`double` – 64 бита)
`FLOAT` – вещественное число (`float` – 32 бита)
`HANDLE` – дескриптор объекта (указатель на переменную любого типа – 32 бита);
`HBITMAP` – дескриптор растрового изображения (указатель – 32 бита);
`HBRUSH` – дескриптор кисти (указатель – 32 бита);
`HCURSOR` – дескриптор курсора (указатель – 32 бита);
`HDC` – дескриптор контекста устройства (указатель – 32 бита);
`HICON` – дескриптор пиктограммы (указатель – 32 бита);
`HFONT` – дескриптор шрифта (указатель – 32 бита);
`HINSTANCE` – дескриптор экземпляра приложения (указатель – 32 бита);
`HMENU` – дескриптор меню (указатель – 32 бита);
`HPEN` – дескриптор пера (указатель – 32 бита);
`HWND` – дескриптор окна (указатель – 32 бита);
`WORD` – `unsigned short` (16 бит);
`LONG` – `signed long` (32 бита);
`INT` – `signed long` (32 бита);
`LPARAM` – 32-разрядное целое число без знака (описатель четвертого параметра оконной функции);
`WPARAM` – 32-разрядное целое число без знака (описатель третьего параметра оконной функции);
`LRESULT` – 32-разрядный описатель возвращаемых значений типа long;
`LPCSTR` – указатель на константную `C`-строку;
`LPCWSTR` – указатель на константную `Unicode`-строку;
`LPSTR` – указатель на `C`-строку;
`LPCTSTR` –константный указатель на строку символов;

SHORT – signed short (16 бит);
USHORT – unsigned short (16 бит);
UCHAR – unsigned char (8 бит)
UINT – unsigned long (32 бита);

```
#If defined(_WIN64)
    typedef _int64 INT_PTR
#else
    typedef  int   INT_PTR
#endif
```

Венгерская нотация

Суть венгерской нотации заключается в том, что имя переменной или функции предваряется одной или несколькими строчными буквами – префиксом, говорящим о типе этой переменной. Само имя может состоять как из прописных, так и из строчных букв, но первая буква имени всегда прописная. Имена, состоящие только из префиксов, принято использовать для временных или вспомогательных переменных.

Имя функции может образовываться объединением глагола и существительного, например: CreateWindow(), DrawText() или LoadIcon(), но может состоять и только из существительных – DialogBox().

Префиксы для имен переменных, соответствующих указанным типам данных, имеют следующие обозначения:
a – массив;
b – тип данных BOOL;
by – BYTE;
ch – char;
dw – DWORD;
fn – функция;
h – дескриптор;
i – INT;
l – LONG;
lp – дальний указатель;
lpsz – дальний указатель на строку, заканчивающуюся нулевым байтом;
n – unsigned short;
p – указатель;
psz – указатель на строку, заканчивающуюся нулевым байтом;
pv – указатель на тип void;
sz – строка, которая заканчивается нулевым байтом;
u – UINT;
v – void;
w – WORD;
pt – точка с координатами x и y (два 32-битных целых);
rgb – длинное целое, содержащее цветовую комбинацию RGB .

Библиографический список

1. *Биллинг, В.А*., Мусикаев И.Х. Visual C++ 4. Книга для программистов. М.: Изд. отдел «Русская редакция» Too «Channel Trading Ltd.», 1996. 352 с.

2. *Саймон, Р*. Windows 2000 API. Энциклопедия программиста: пер. с англ. СПб.: ООО «ДиаСофтЮП», 2002. 1088 с.

3. *Фролов, Л.В*., Фролов Г. В. Операционная система WINDOWS 95. Для программиста. М.: ДИАЛОГ-МИФИ, 1996. 288 с. (Библиотека системного программиста; т. 22).

4. *Пирогов, В. Ю.* Ассемблер для Windows. 2-е изд., перераб. и доп. СПб.: БХВ–Петербург, 2003. 656 с.

5. *Щупак, Ю. А*. Win32 API. Эффективная разработка приложений. СПб.: Питер, 2007. 572 с.

6. *Юров, В. И*. Assembler: учебник для вузов. 2-е изд. СПб.: Питер, 2004. 637 с.

О Г Л А В Л Е Н И Е

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007–2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация этой программы создала основу формирования программы дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

КАФЕДРА ПРОЕКТИРОВАНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

1945–1966 РЛПУ (кафедра радиолокационных приборов и устройств). Решением Советского правительства в августе 1945 г. в ЛИТМО был открыт факультет электроприборостроения. Приказом по институту от 17 сентября 1945 г. на этом факультете была организована кафедра радиолокационных приборов и устройств, которая стала готовить инженеров, специализирующихся в новых направлениях радиоэлектронной техники, таких как радиолокация, радиоуправление, теленаведение и др. Организатором и первым заведующим кафедрой был д.т.н., профессор С. И. Зилитинкевич (до 1951 г.). Выпускникам кафедры присваивалась квалификация инженер-радиомеханик, а с 1956 г. – радиоинженер (специальность 0705).

В разные годы кафедрой заведовали доцент Б.С. Мишин, доцент И.П. Захаров, доцент А.Н. Иванов.

196–1970 КиПРЭА (кафедра конструирования и производства радиоэлектронной аппаратуры). Каждый учебный план специальности 0705 коренным образом отличался от предыдущих планов радиотехнической специальности своей четко выраженной конструкторско-технологической направленностью. Оканчивающим институт по этой специальности при

сваивалась квалификация инженер-конструктор-технолог РЭА. Заведовал кафедрой доцент А.Н. Иванов.

1970–1988 КиПЭВА (кафедра конструирования и производства электронной вычислительной аппаратуры). Бурное развитие электронной вычислительной техники и внедрение ее во все отрасли народного хозяйства потребовали от отечественной радиоэлектронной промышленности решения новых ответственных задач. Кафедра стала готовить инженеров по специальности 0648. Подготовка проводилась по двум направлениям – автоматизация конструирования ЭВА и технология микроэлектронных устройств ЭВА. Заведовали кафедрой: д.т.н., проф. В.В. Новиков (до 1976 г.), затем проф. Г.А. Петухов.

1988–1997 МАП (кафедра микроэлектроники и автоматизации проектирования). Кафедра выпускала инженеров-конструкторов-технологов по микроэлектронике и автоматизации проектирования вычислительных средств (специальность 2205). Выпускники этой кафедры имеют хорошую технологическую подготовку и успешно работают как в производстве полупроводниковых интегральных микросхем, так и при их проектировании, используя современные методы автоматизации проектирования. Инженеры специальности 2205 требуются микроэлектронной промышленности и предприятиям-разработчикам вычислительных систем. Кафедрой с 1988 г. по 1992 г. руководил проф. С.А. Арустамов, затем снова проф. Г.А. Петухов.

С **1997 ПКС** (кафедра проектирования компьютерных систем). Кафедра выпускает инженеров по специальности 210202 «Проектирование и технология электронно-вычислительных средств». Область профессиональной деятельности выпускников включает в себя проектирование, конструирование и технологию электронных средств, отвечающих целям их функционирования, требованиям надежности, дизайна и условиям эксплуатации. Кроме того, кафедра готовит специалистов по защите информации, специальность 090104 «Комплексная защита объектов информатизации». Объектами профессиональной деятельности специалиста по защите информации являются методы, средства и системы обеспечения защиты информации на объектах информатизации.

С 1996 г. кафедрой заведует д.т.н., профессор Ю.А. Гатчин.

За время своего существования кафедра выпустила 4264 инженеров. На кафедре защищено 62 кандидатских и 7 докторских диссертаций.

Вячеслав Алексеевич Безруков
Win32 API.
Программирование.

Учебное пособие

В авторской редакции В.А. Безруков
Дизайн В.А. Безруков
Верстка В.А. Безруков
Редакционно-издательский отдел Санкт-Петербургского государственного университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати 11.12.09
Заказ № 2174
Тираж 100 экз.
Отпечатано на ризографе

Редакционно-издательский отдел
Санкт-Петербургского государственного университета информационных технологий, механики и оптики
197101, Санкт-Петербург, Кронверкский пр., 49