Введение

Низкоуровневое программирование на языке C++ для Windows до сих пор сохраняет свою привлекательность и, несмотря на наличие современного инструментария, находит своих сторонников и фанатов. По нашему мнению — это обязательный этап при изучении курса программирования на C++, поскольку только на низком уровне можно добиться понимания того, как работает программа под управлением операционной системы Windows. Даже если мы в дальнейшем будем работать с современным фреймворком, это поможет нам строить более эффективный код.

Учебное пособие по сути является курсом программирования на C++ [1, 2] и рассчитано на студентов технических вузов, предполагается, что студенты уже изучили синтаксиса языка, стандартную библиотеку и умеют писать программы для консольного приложения в среде Visual Studio.

Материал излагается в соответствии со ставшей уже классической схемой, построенной такими авторами, как У. Мюррей [3], Ч. Петцолд [4, 5], Г. Шилд [6]. Несмотря на то что прошло уже достаточно времени, эти книги пользуются заслуженным вниманием.

Принципиальный вопрос, который возник при построении курса, — какой кодировки придерживаться? И среда программирования, и библиотека WinApi (*Windows Application Program Interface*) позволяют работать в кодировке как ANSI, так и Unicode. Еще в начале 2000 гг. Дж. Рихтер [7] призывал всех переходить на Unicode, и он, как показал опыт, был совершенно прав. Так что весь материал, за некоторыми исключениями, рассчитан на использование Unicode.

Материал лекций снабжен большим числом примеров, которые протестированы в среде Visual Studio 2019. Большей частью примеры составлены автором, однако использованы несколько наиболее удачных примеров из источников [4–8].

При изучении материала хотелось бы обратить внимание на электронные ресурсы для справки: https://metanit.com/cpp/tutorial/, https://riptutorial.com/ru/cplusplus/.

Лекция 1. Введение в оконный интерфейс

Все современные операционные системы являются многозадачными, и приложения, работающее под их управлением, должны быть построены с учетом этого. Даже консольное приложение выполняется в специальном потоке, который создается для него операционной системой. Разберемся теперь, как реализовано приложение, удовлетворяющее критерию многозадачности, т. е. возможности переключения контекста выполнения с одной задачи на другую.

Классическое Windows-приложение состоит как минимум из двух функций:

- WinMain() — точка входа, служит для инициализации приложения;
- WndProc() — оконная функция.

После загрузки приложения управление передается функции WinMain(), где выполняются подготовительная работа по запуску Windows-приложения:

- определяется класс окна;
- окно регистрируется в системе;
- окно создается и отображается;
- запускается цикл обработки сообщений.

Причем оконная функция не вызывается из головной функции WinMain(), вызов осуществляет операционная система для обработки очередного сообщения. Такие функции, вызов которых осуществляется через операционную систему, называют функциями обратного вызова.

Для того чтобы проще понять структуру приложения, напишем «вручную» минимальный код, который обеспечит выполнение этих условий.

Проект Windows-приложения

Для создания проекта выберем «Мастер классических приложений C++» и разместим, для простоты, проект и решение в одном каталоге. Тип приложения — «Классическое приложение (.exe)». Дополнительные параметры — «Пустой проект». Добавим к проекту «Файл C++ (.cpp)», здесь будет код листинга 1.1.

Листинг 1.1. Минимальный код Windows-приложения

```
#include < Windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int APIENTRY wWinMain(HINSTANCE hInstance,
   HINSTANCE hPrevInstance,
```

```
    LPWSTR    lpCmdLine,
    int       nCmdShow)
{
    MSG msg;
    WNDCLASSEXW wcex{ 0 };
    // Заполнить поля структуры
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.hInstance = hInstance;
    wcex.hIcon= LoadIcon(nullptr, IDI_APPLICATION);
    wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wcex.lpszClassName = L"MyWin";
    // Регистрировать класс окна
    RegisterClassExW(&wcex);
    // Создать окно и отобразить
    HWND hWnd = CreateWindowW(L"MyWin", L"Мое первое окно",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
 nullptr, nullptr, hInstance, nullptr);
    ShowWindow(hWnd, nCmdShow);
    // Цикл обработки сообщений
    while (GetMessage(&msg, nullptr, 0, 0))
    {
      DispatchMessage(&msg);
    }
    return 0;
}
// Оконная функция
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
  switch (message)
  {
     case WM_DESTROY: PostQuitMessage(0); break;
     default: return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
```

Как видно из представленного листинга, создание Windows-приложения требует бóльших накладных расходов, чем консольное. Вначале необходимо добавить файл включений Windows.h, где представлены прототипы основных функций и определений WinAPI (*Application Programming Interface*), работающих под управлением ОС Windows.

Соглашения WinAPI

Прежде чем рассматривать функцию wWinMain(), рассмотрим подробнее формализм WinAPI, отметим только префикс «w», который явно указывает, что функция работает с кодировкой Unicode.

Типы данных WinAPI

Библиотека WinAPI создавалась с расчетом на использование любым языком программирования, но, поскольку типы данных разных языков не всегда совпадают, пришлось вводить специальные WinAPI-типы, которые могут рассматриваться как псевдонимы существующих C++ типов и определены в файле включений так:

```
typedef unsigned char     BYTE;
typedef unsigned short    WORD;
typedef float             FLOAT;
```

Некоторые типы могут принимать разное значение, в зависимости от того, для какого процессора компилируется приложение, — так, для типа LRESULT, если это x86, будет long, если x64, __int64.

Отдельной группой стоят дескрипторы (описатели), имена этих типов начинаются с символа «H». Это сугубо Windows-типы, например HWND, HBRUSH, HINSTANCE. Формально тип дескриптора соответствует типу void*.

Дело в том, что, поскольку Windows является многозадачной системой, текущее приложение, выполняющееся в своем потоке, может не только потерять управление, но и быть выгружено из памяти в файл подкачки, если память потребуется другим стоящим в очереди на процессорное время потокам. Вновь получив квант процессорного времени, поток будет восстановлен из файла подкачки, но объекты будут размещены по ближайшим свободным адресам памяти. Так вот дескрипторы и позволяют определять текущий адрес объекта. По сути дескриптор объекта — это ссылка на запись в таблице дескрипторов, где хранится информация об объекте, т. е. текущий адрес или отметка о выгрузке в файл подкачки. Таким образом, используя косвенное обращение к объектам через дескрипторы, мы можем не беспокоиться о их реальном местоположении.

Соглашения по вызову

Перед именами функций стоит один из модификаторов WINAPI, CALLBACK или APIENTRY, все они являются псевдонимами __stdcall. Это соглашение по вызову функций WinAPI, означающее, что *аргументы функций помещаются в стек в обратном порядке — справа налево* и что после завершения работы вызываемая функция очистит стек.

Головная функция WinMain()

Первый вопрос, на который нужно ответить при создании Windows-приложения, — в какой кодировке будем работать? Еще в начале 2000-х годов Дж. Рихтер призывал отказаться от C-строк и переходить на Unicode. Сейчас это практически стало нормой, и тот проект, который будет затем построен

мастером-построителем, будет по умолчанию создан для Unicode. Мы также будем создавать проект в кодировке Unicode, хотя библиотека WinAPI и содержит дублирующие функции для C-строк с суффиксом «A» и для Unicode с суффиксом «W».

Функция имеет четыре параметра:

- HINSTANCE hInstance — дескриптор приложения;
- HINSTANCE hPrevInstance — дескриптор предыдущей копии приложения;
- LPWSTR lpCmdLine — командная строка;
- int nCmdShow — число параметров командной строки.

Параметр hInstance присваивается загрузчиком операционной системы, второй параметр hPrevInstance — рудиментный остаток от Windows 3.1 и сейчас не используется, lpCmdLine — указатель командной строки типа wchar_t*, устанавливается также загрузчиком, nCmdShow — режим отображения окна.

Для типов данных WinAPI используется венгерская нотация, так, тип LPWSTR можно расшифровать следующим образом — «длинный» (L) указатель (P) широких (W) строк (STR).

Примечание

В ОС Windows 3.1 были «короткие» 16-битные указатели, которые адресовали до 64K, и «длинные» 32-битные указатели с адресацией до 4GB. Начиная с Windows NT, все указатели «длинные».

Рассмотрим тело функции. Опишем вначале структуру MSG, там хранится информация о сообщениях Windows. Эта структура нам понадобится для организации цикла обработки сообщений.

Структура WNDCLASSEXW необходима для регистрации класса окна, мы заполним лишь необходимые поля, оставив остальные со значением 0 по умолчанию. Большинство структур WinAPI имеют размер, зависящий от выбранной кодировки, поэтому в поле cbSize нужно указать текущий размер.

Поле style определяет стиль создаваемого окна, в нашем случае складывается из двух значений CS_HREDRAW | CS_VREDRAW — перерисовка окна при изменении его ширины или высоты.

Указатель на оконную функцию WndProc присваиваем полю lpfnWndProc.

Следующее поле получает hInstance — дескриптор, который нашему приложению присвоил системный загрузчик.

Поле hIcon определяет дескриптор иконки (пиктограммы), отображающейся в левом верхнем углу заголовка окна, загружается при помощи функции LoadIcon(), первый параметр которой nullptr указывает, что иконка берется из ресурса Windows, второй параметр IDI_APPLICATION — идентификатор стандартной иконки.

Дескриптор курсора hCursor определяется аналогичным образом функцией LoadCursor().

Цвет окна определяется кистью, дескриптор которой hbrBackground можно определить относительно системного цвета — (HBRUSH)(COLOR_WINDOW+1).

Здесь требуется явное приведение типа.

Последнее поле, которое обязательно должно быть определено, — это lpszClassName, здесь указывается имя класса окна, с которым оно будет зарегистрировано в системе функцией RegisterClassExW().

При регистрации окна будет создан так называемый «объект ядра», где будет храниться информация об окне. Причем класс окна принадлежит операционной системе и доступ к нему возможен только по имени.

Создается окно функцией CreateWindowW(), где первый параметр L"MyWin" — имя класса окна, второй параметр L"Мое первое окно" — строка заголовка. Суффикс «W» в имени функции явно указывает, что функция принимает строки в кодировке Unicode, поэтому их нужно описывать с оператором «L».

Параметр WS_OVERLAPPEDWINDOW описывает стиль окна (префикс WS означает Windows Style) — это синтетический стиль, который складывается из следующих стилей: WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX. То есть окно является перекрывающимся, с заголовком, системным меню, граница окна — тонкая рамка, имеются кнопки минимизации и максимизации.

Следующие четыре параметра определяют X,Y-координаты левого верхнего угла окна относительно родительского окна (в нашем случае родителем является «Рабочий стол»), а также ширина и высота окна в пикселях. Использование константы CW_USEDEFAULT приводит к тому, что система сама определяет оптимальное расположение и размеры окна.

Далее следует параметр hWndParent — это дескриптор родительского окна (у нас nullptr), hMenu — дескриптор главного меню (у нас нет меню, поэтому nullptr), hInstance — дескриптор приложения, lpParam — дополнительный параметр (у нас nullptr).

Окно создано, теперь его можно отобразить функцией ShowWindow(), где значение второго параметра nCmdShow определяет режим отображения, например:

SW_SHOW — отображает окно в текущем размере;

SW_HIDE — скрывает окно;

SW_MINIMIZE — сворачивает окно;

SW_MAXIMIZE — разворачивает окно на весь экран.

После чего нужно запустить цикл обработки сообщений от операционной системы, которые функция GetMessage() извлекает из очереди сообщений. Первым параметром стоит ссылка на структуру MSG, получающую информацию сообщения:

```
struct MSG {
 HWND   hwnd;          // дескриптор окна, которому послано сообщение
 UINT   message;       // код сообщения
 WPARAM wParam;        // w-параметр
```

```
 LPARAM lParam;     // l-параметр
 DWORD  time;       // время, когда сообщение было послано
 POINT    pt;       // положение курсора в экранных координатах
};
```

Второй параметр — дескриптор окна-источника сообщения; если nullptr, обрабатываются сообщения от всех окон приложения. Два оставшихся параметра определяют диапазон обрабатываемых сообщений; если заданы нулевые значения, фильтрация диапазона не производится и обрабатываются все доступные сообщения.

Всю работу в цикле выполнит функция DispatchMessage(), которая отправляет сообщение в оконную функцию. Вот и все, запущен цикл обработки сообщений, который ожидает сообщений от операционной системы и обеспечивает вызов оконной функции, передавая ей информацию о полученном сообщении.

Оконная функция

Оконная функция вызывается операционной системой и должна выполнить действия, определенные полученным сообщением. Имеет 4 параметра: hWnd — дескриптор окна, пославшего сообщение, message — код сообщения, wParam, lParam — параметры сообщения. Обычно оконная функция представляет собой большой переключатель switch, где каждая строка альтернативы case обеспечивает обработку соответствующего сообщения. Поскольку сообщений очень много (больше 1000), то предусмотрена функция DefWindowProc(), которая обработает остальные сообщения по умолчанию. При нормальном завершении работы оконная функция возвращает 0.

В нашем примере никакой функциональности для окна не запланировано, мы обрабатываем лишь одно сообщение о закрытии окна. При нажатии кнопки закрытия генерируется сообщение WM_DESTROY (его код 2), здесь мы вызываем функцию PostQuitMessage() с аргументом 0. Эта функция отправляет сообщение WM_QUIT с wParam = 0 в очередь сообщений приложения и завершает работу.

Теперь функция GetMessage() в цикле обработки сообщения, получив сообщение WM_QUIT, возвращает false, и цикл завершится, что приведет к завершению работы приложения.

Стандартный каркас классического Windows-приложения

Для создания приложения с оконным интерфейсом достаточно выбрать проект «Классическое приложение Windows» и мастер построитель создаст каркас, на который и будет надстраиваться содержательная часть нашего приложения. Этот каркас будет несколько отличаться от того минимального кода, который мы построили в листинге 1.1.

- Все файлы включений собраны в файле «framework.h».
- Создан файл ресурсов WindowsProject1.rc и Resource.h, где размещаются текстовые строки, шаблон меню, ссылки на внешние файлы и т. д. Для

получения данных из ресурса используется специальный набор функций типа: LoadStringW(), LoadIcon(), LoadCursor().

- Добавлена еще одна оконная функция About(), которая выводит информацию «О программе».

Теперь более подробно посмотрим головную функцию.

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                      _In_opt_HINSTANCE hPrevInstance,
                      _In_LPWSTR lpCmdLine,
                      _In_int nCmdShow)
```

Макросы **_In_** и **_In_opt_** в заголовке функции информируют компилятор, что это входной параметр, который не может быть изменен.

Макросы

```
UNREFERENCED_PARAMETER(hPrevInstance);
UNREFERENCED_PARAMETER(lpCmdLine);
```

указывают компилятору, что параметры hPrevInstance и lpCmdLine не используются в программе и не нужно создавать предупреждающего сообщения.

Строки, которые именуют класс окна и его заголовок размещены в файле ресурса и считываются оттуда:

```
LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
LoadStringW(hInstance, IDC_WINDOWSPROJECT1, szWindowClass,
                       MAX_LOADSTRING);
```

Первый параметр hInstance указывает дескриптор приложения, из ресурса которого считывается строка; если это nullptr, строка будет искаться в ресурсах Windows. Второй параметр IDS_APP_TITLE — идентификатор строки, определенный в файле Resource.h. Дальше идет указатель на строку, куда помещаются результат и его максимальный размер.

Регистрация класса окна и его создание выделено отдельными функциями, причем производится проверка, создано ли окно? Если окно по какой-либо причине не может быть создано, функция CreateWindowW() возвращает нулевой дескриптор.

Отметим еще использование функции UpdateWindow() после отображения окна. Дело в том, что отображение окна — это задача достаточно низкого приоритета для Windows и иногда окно просто не успевает прорисоваться. Функция UpdateWindow() отправит сообщение WM_PAINT непосредственно в оконную процедуру указанного окна, минуя очередь сообщений. Если область обновления пуста, сообщение не отправляется, проще говоря, функция обеспечит перерисовку окна, если оно не успело обновиться.

Далее при помощи функции LoadAccelerators() извлекается таблица клавиш-акселераторов, которые определены в ресурсах приложения. Эти «быстрые клавиши» сопоставляются соответствующим пунктам меню и приводят к таким же действиям.

Примечание

Функции чтения из ресурса используют макрос MAKEINTRESOURCE, который позволяет интерпретировать идентификатор объекта как указатель типа LPWSTR.

Проверка на действие «быстрой клавиши» осуществляется в цикле обработки сообщений, где появляется строка с обращением к функции TranslateAccelerator(). Эта функция преобразует сообщение от «быстрой клавиши» в сообщение WM_COMMAND и отправляет его в оконную функцию, возвращая true, в этом случае оконная функция повторно не вызывается.

```
while (GetMessage(&msg, nullptr, 0, 0))
{
   if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
}
```

Если же сообщение не принадлежит «быстрой клавише», обработка в цикле осуществляется обычным порядком. Однако здесь мы видим новую функцию TranslateMessage(), которая требуется для преобразования сообщения от алфавитно-цифровой клавиатуры в символьное сообщение WM_CHAR.

Примечание

Сообщения от клавиатуры направляется верхнему в Z-порядке окну, которое находится в фокусе ввода.

Возвращаемое значение головной функции будет wParam. Как мы уже отмечали, это значение передается через параметр функции PostQuitMessage(), которая вызывается в оконной функции для завершения работы.

Как уже отмечалось выше, всю работу осуществляет оконная функция WndProc(), где в переключателе switch создан код для действий при выборе пункта меню, когда генерируется сообщение WM_COMMAND. Идентификатор выбранного пункта меню передается в младшем слове wParam, где LOWORD — макрос, извлекающий младшие 16 битов из 32-разрядного параметра. Более подробно о построении меню мы поговорим ниже.

Сообщение WM_PAINT генерируется для перерисовки окна. В ОС Windows принято соглашение, которое возлагает ответственность за перерисовку (восстановление) окна на само приложение, поэтому код сообщения WM_PAINT должен отображать текущее состояние окна. Когда может потребоваться перерисовка окна? Это могут быть операции «распахивания» окна на весь экран, изменение размера или восстановление окна после перекрытия его другим окном и т. п. Приложение также может инициировать перерисовку окно, для этого его требуется объявить «недействительным» вызовом функции InvalidateRect() или InvalidateRgn(), что приведет к генерации сообщения WM_PAINT для данного окна.

Windows GDI

Приложения для Windows не имеют прямого доступа к графическому оборудованию. Интерфейс графического устройства (Graphics Device Interface, GDI) является посредником для взаимодействия приложения с драйвером устройства отображения. Функции GDI, размещенные в библиотеке динамической компоновки Gdi32.dll, отвечают за отрисовку линий, кривых и растровых изображений, отображение шрифтов.

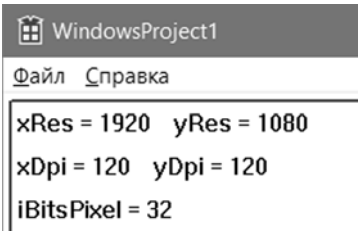

Рис. 1.1. Система координат

Библиотека GDI опирается на декартову систему координат с началом в левом верхнем углу клиентской области окна, по умолчанию ось x направлена вправо, ось y — вниз. Разрешающая способность (dots per inch, DPI), а также размер экрана в пикселях (экранных точках) зависят от используемой аппаратуры и могут быть определены при помощи функции GetDeviceCaps(). Так, на рис. 1.1 можно посмотреть некоторые характеристики нашего компьютера. Итак, у нас экран 1920×1080 точек при разрешении 120 точек на дюйм, 32-битный цвет.

Контекст устройства

Все функции GDI взаимодействуют с контекстом устройства (Device Context, DC) — это внутренняя структура, которая служит для управления устройством вывода. Приложение направляет вывод в контекст устройства, а операционная система уже переправляет его на конкретное устройство. Такой подход позволяет использовать одинаковый код для вывода на различные устройства, например дисплей, принтер и т. д.

В набор характеристик контекста устройства входят следующие объекты: перо для рисования линий, кисть для заливки фигур, шрифт, регион для отсечения области вывода.

Различают следующие типы контекста: Display DC, Printer DC, Memory DC, Metafile DC и Information DC. Пока будем работать с контекстом дисплея, который используется для отображения в окне. По умолчанию создается *общий контекст устройства*, это критически важный ресурс операционной системы, и, пока приложение его не освободит, никто больше не может осуществлять вывод на дисплей. Можно создать *контекст устройства для класса окна*, который хранится в отдельной области памяти и используется всеми окнами, созданными на базе этого класса. При регистрации такого класса его стиль должен быть CS_CLASSDC. Этот контекст освобождать не требуется. Стилем CS_OWNDC создается *личный контекст* для каждого окна этого класса. Еще один стиль CS_PARENTDC создает *родительский контекст*, который позволяет дочернему окну наследовать свойства от родителя. Эти контексты также не нужно освобождать до завершения работы с окном, однако памяти они «съедают» немало, поэтому чаще всего используется все-таки общий контекст.

Для начала работы необходимо получить контекст. Для сообщения WM_PAINT контекст нужно получить вызовом функции BeginPaint(), а после отрисовки окна освободить его функцией EndPaint(). Дело в том, что только эти функции могут удалить сообщение WM_PAINT из очереди сообщений после его обработки, иначе бы происходила циклическая перерисовка одного и того же окна. Причем, поскольку сообщение WM_PAINT имеет достаточно низкий приоритет, возможна ситуация, когда несколько сообщений WM_PAINT последовательно выстроились в очереди для обработки, в этом случае сообщения «складываются» и обрабатываются за один проход. Если точнее, то складываются области вывода.

Теперь рассмотрим более подробно функцию BeginPaint(hWnd, &ps).

Первый параметр hWnd — это дескриптор окна, контекст которого мы получаем. Второй параметр &ps — ссылка на структуру:

```
struct PAINTSTRUCT {
   HDC  hdc;              //дескриптор полученного контекста устройства
   BOOL fErase;           //признак стирания фона клиентской области
   RECT rcPaint;          //недействительный прямоугольник
   BOOL fRestore;         //зарезервировано
   BOOL fIncUpdate;       //для
   BYTE rgbReserved[32];  //Windows
};
```

Сообщение о перерисовке окна WM_PAINT мы можем создать, объявив недействительным окно функцией

InvalidateRect(hWnd, &rect, bErase);

Второй параметр &rect — указатель недействительного прямоугольника, который и определяет область перерисовки. NULL — перерисовывается все окно. Если последний параметр TRUE, фон окна перерисовывается, иначе нет.

Поскольку перерисовка окна достаточно трудоемкая работа, по возможности нужно ограничивать «недействительную» область.

Контекст устройства может быть получен и при обработке других сообщений функцией GetDC(), и мы также можем осуществлять вывод в окно, однако, если после этого будет вызвано сообщение WM_PAINT, мы потеряем все, что мы нарисовали вне этого сообщения. Контекст окна должен быть освобожден, но уже функцией ReleaseDC().

Режимы отображения

При рисовании мы имеем две битовые последовательности — контекст устройства (вообще-то это область памяти, заполненная битовым образом исходного изображения, как минимум фоновым цветом), а также рисуемое изображение. Над этими битовыми полями можно проделать битовую операцию для получения результирующего изображения.

Устанавливается режим вызовом функции SetROP2(hdc, op).

Всего предусмотрено 16 операций, определенных символическими именами. По умолчанию принята операция R2_COPYPEN — копирование цвета пера. Иногда полезно использовать операции R2_BLACK или R2_WHITE, а также R2_XORPEN («Исключающее ИЛИ»). Если вспомнить, что повторное применение этой операции восстанавливает код, то нетрудно найти ей применение для восстановления изображения.

Посмотреть установленный режим можно функцией GetROP2(hdc).

Приведем в заключение в листинге 1.2 код, который позволил получить информацию о графической системе, показанную на рис. 1.1. Объяснение пока оставим на будущее, отметим только, что приводить мы будем только оконную функцию, убрав из нее все лишнее, головную функцию, созданную мастером построителем, мы, без особой необходимости, не трогаем.

Листинг 1.2. Информация о графическом режиме

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
                         LPARAM lParam)
{
  wchar_t str[40];
  switch (message)
  {
  case WM_PAINT:
    {
      PAINTSTRUCT ps;
      HDC hdc = BeginPaint(hWnd, &ps);
      HPEN pen = CreatePen(PS_SOLID, 2, RGB(0, 0, 255));
      SelectObject(hdc, pen);
      MoveToEx(hdc, 5, 5, nullptr);
      LineTo(hdc,5,100);
      MoveToEx(hdc, 5, 5, nullptr);
      LineTo(hdc,250,5);
      int yRes = GetDeviceCaps(hdc, VERTRES);
      int xRes = GetDeviceCaps(hdc, HORZRES);
      int logX = GetDeviceCaps(hdc, LOGPIXELSX);
      int logY = GetDeviceCaps(hdc, LOGPIXELSY);
      wsprintf(str, L"xRes = %d    yRes = %d", xRes, yRes);
      TextOut(hdc, 10, 10, str, wcslen(str));
      wsprintf(str, L"xDpi = %d    yDpi = %d", logX, logY);
      TextOut(hdc, 10, 40, str, wcslen(str));
      int iBitsPixel = GetDeviceCaps(hdc, BITSPIXEL);
      wsprintf(str, L"iBitsPixel = %d", iBitsPixel);
      TextOut(hdc, 10, 70, str, wcslen(str));
      int rop2 = GetROP2(hdc);
      wsprintf(str, L"rop = %d", rop2);
      TextOut(hdc, 10, 100, str, wcslen(str));
      EndPaint(hWnd, &ps);
```

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

Контрольные вопросы

1. Какие операции выполняет функция WinMain() при загрузке приложения?
2. Как работает цикл обработки сообщений?
3. Как происходит вызов оконной функции?
4. Что такое дескриптор?
5. Что означает модификатор функции __stdcall?
6. Кому принадлежит «класс окна» после регистрации в системе?
7. Как создается окно?
8. Какие действия выполняет функция DefWindowProc()?
9. Как завершить работу приложения?
10. Как работают «быстрые клавиши»?
11. Зачем нужна функция TranslateMessage()?
12. Как вывести код возврата приложения?
13. Как можно инициировать перерисовку окна?
14. Что означают термины: GDI, «контекст устройства»?
15. Как работает обработчик сообщения WM_PAINT?
16. Можно ли «рисовать» в окне при обработке других сообщений?
17. Что такое «недействительный прямоугольник»?
18. Как установить «Режим отображения» в окне?

Задание для самостоятельной работы

Создать «Классическое приложение Windows» и исследовать его. Просмотрите структуру файла ресурсов в среде Visual Studio, затем откройте его в Блокноте и посмотрите там. Однако вносить изменения в файл ресурсов внешними средствами крайне не рекомендуется.

Лекция 2. Графические построения

Windows GDI, который используется для графических построений, представляет собой набор графических объектов, таких как перо и кисть, графические примитивы — точка, линия, прямоугольник и т. п., а также несколько сотен функций и связанных с ними типов данных, макросов и структур.

Графические объекты

Объекты GDI имеют некие общие правила применения, они могут создаваться и уничтожаться в любой момент времени, могут сохраняться и между обработкой сообщений. Все созданные объекты должны быть уничтожены до завершения приложения, поскольку они создаются в локальной памяти библиотеки GDI и не уничтожаются автоматически как объекты глобальной памяти.

Создаются объекты функциями, начинающиеся с префикса Create, и возвращают дескриптор объекта, например CreatePen() или CreateSolidBrush(). После чего объект должен быть выбран в контекст устройства функцией Select Object (). Уничтожается объект функцией DeleteObject(), но только после того, как будет отсоединен от контекста.

Перо

По умолчанию в контексте дисплея имеется сплошное черное перо толщиной 1 пиксель, однако мы можем создавать другие перья и переключаться на них по мере надобности. В каждый момент времени может быть активно только одно перо. Этим пером рисуются линии, прямоугольники и другие фигуры. Функция создания пера имеет три параметра:

CreatePen(iStyle, cWidth, color);

Стиль линии для данного пера определяется константой iStyle:

PS_SOLID	сплошная линия (——)
PS_DASH	штриховая линия (– – –)
PS_DOT	точечная линия (. . .)
PS_DASHDOT	штрихпунктирная линия (–.–)
PS_DASHDOTDOT	штрихпунктирная линия (–..–)
PS_NULL	прозрачная линия.

Примечание

Штриховые линии создаются только при толщине пера 1 пиксель.

Второй параметр задает толщину пера в логических единицах (по умолчанию в пикселях); если 0, перо имеет толщину 1 пиксель.

Третий параметр типа COLORREF (соответствует unsigned long) задает цвет пера. Проще всего для задания цвета воспользоваться макросом RGB(r, g, b), где числами от 0 до 255 задаются интенсивности соответственно красного, зеленого и синего цветов. Макрос формирует 32-битное число, где младший байт — r, следующие — g и b, старший байт не используется.

Возникает вопрос — когда создавать перо? Если перо нам необходимо длительное время, то разумно создать его при создании окна при обработке сообщения WM_CREATE, которое будет сгенерировано Windows еще до отображения окна, а дескриптор пера объявить в статической области памяти, например:

```
static HPEN hpenRed;
switch (message)
{
case WM_CREATE:
      hpenRed = CreatePen(PS_SOLID, 4, RGB(255, 0, 0));
    break;
case WM_PAINT:
{
      PAINTSTRUCT ps;
      HDC hdc = BeginPaint(hWnd, &ps);
      SelectObject(hdc, hpenRed);
      . . .
      EndPaint(hWnd, &ps);
}
   break;
. . .
}
```

Здесь мы при создании окна создали сплошное красное перо толщиной 4 пикселя, а использовать его будем при обработке сообщения WM_PAINT. Перо сохранит свое значение, поскольку объявлено в статической области памяти с модификатором static.

Примечание

В ресурсах Windows имеется набор готовых перьев. Дескриптор системного пера можно получить функцией GetStockObject(). В зависимости от значения параметра мы можем получить черное, белое и пустое перо: BLACK_PEN, WHITE_PEN, DC_PEN, NULL_PEN. Для использования пера нужно установить его в контексте окна функцией SelectObject(), удалять системное перо по завершении работы не требуется.

Например, hpen = (HPEN)GetStockObject(BLACK_PEN);

Здесь необходима операция явного приведения типа, поскольку функция может возвращать различные объекты: перья, кисти и т. п.