Федеральное агентство по образованию
Уральский государственный технический университет – УПИ
имени первого Президента России Б. Н. Ельцина

Электронный аналог печатного издания

ПРОГРАММИРОВАНИЕ ПОД WINDOWS В СРЕДЕ VISUAL C++

Методические указания по дисциплине
«Системное программирование»
для студентов специальности 230150 – Программное обеспечение
вычислительной техники и автоматизированных систем

Екатеринбург
УГТУ – УПИ
2010

УДК 004.2:004.9

Составитель О. Л. Чагаева

Научный редактор С. И. Тимошенко, канд. техн. наук

Программирование под Windows в среде Visual C++ : методические указания / сост. О. Л. Чагаева. – Екатеринбург : УГТУ – УПИ, 2010. – 42 с.

В работе рассматривается раздел современной технологии программирования – визуальное программирование под Windows. Приводятся элементы общей теории программирования под Windows, иерархии классов MFC и использование этой библиотеки при создании приложений.

Библиогр.: 3 назв. Табл. 2.

Подготовлены кафедрой
«Программные средства и системы»
факультета ускоренного обучения

Оглавление

Введение

Визуальное программирование — способ создания программы путём манипулирования графическими объектами вместо написания ее текста.

Языки визуального программирования могут быть дополнительно классифицированы в зависимости от типа и степени визуального выражения, на следующие типы:

- языки на основе объектов, когда визуальная среда программирования предоставляет графические или символьные элементы, которыми можно манипулировать интерактивным образом в соответствии с некоторыми правилами;
- языки на основе форм, когда программирование осуществляется помещением на специальные формы объектов и настройкой их свойств и поведения. Примеры: Delphi и C++ Builder фирмы Borland.
- языки схем, основанные на идее «фигур и линий», где фигуры (прямоугольники, овалы и т. п.) рассматриваются как субъекты и соединяются линиями (стрелками, дугами и др.), которые представляют собой отношения. Пример: UML.

Естественно-визуальные языки имеют неотъемлемое визуальное выражение, для которого нет очевидного текстового эквивалента (например, графический язык G в среде LabVIEW).

Визуально-преобразованные языки являются невизуальными языками с наложенным визуальным представлением. Примером является среда Visual C++ для языка C++.

C++ является языком программирования общего назначения. Естественная для него область применения — системное программирование, понимаемое в широком смысле этого слова. Кроме того, C++ успешно используется во многих областях приложения, далеко выходящих за указанные рамки. Реализации C++ теперь есть на всех машинах, начиная с самых скромных микрокомпьютеров и заканчивая самыми большими супер-ЭВМ, и практически для всех операционных систем.

1. Создание простейшего Windows-приложения с использованием WinAPI

1.1. Элементы Windows-приложения

Построение приложения Windows включает в себя выполнение следующих этапов:

1. Создание функции `WinMain(...)` и связанных с ней функций на языке C или C++.
2. Создание описаний меню и всех дополнительных ресурсов, помещение описаний в файл описания ресурсов.
3. Создание уникальных курсоров, пиктограмм и битовых образов.
4. Создание диалоговых окон.
5. Создание файла проекта.
6. Компиляция и компоновка всего кода.

1.2. Создание WINDOWS-приложения

Создадим пустой проект Windows-приложения с помощью мастера:

1. File→New→Project.
2. Project types: Win32 Templates: Win32 Project.
3. Ok.
4. Установить галочку Empty project.
5. Добавить в проект файл *.cpp.
6. Project→Properties. Вкладка Configuration Properties→General.
7. Значение поля Character Set устанавливаем Use Multi-Byte Character Set.

1.3. Структура программы

Все приложения Windows должны содержать два основных элемента: функцию `WinMain(...)` и функцию окна `WndProc`.

Функция `WinMain(...)` служит точкой входа в приложение. Эта функция отвечает за следующие действия:

1) регистрация типа класса окон приложения;
2) выполнение всех инициализирующих действий;
3) создание и инициализация цикла сообщений приложения;
4) завершение программы (обычно при получении сообщения `WM_QUIT`).

Функция `WndProc` отвечает за обработку сообщений Windows. Эта часть программы является наиболее содержательной с точки зрения выполнения поставленных перед программой задач. Если необходимо, чтобы программа обращала внимание на действия пользователя, то необходимо добавить ветки case для оператора switch в оконную процедуру `WndProc`. Например, если надо, чтобы приложение реагировало на щелчок левой кнопкой мыши, то добавляем ветку `case WM_LBUTTONDOWN`.

Заголовочный файл windows.h нужен для любой традиционной Windows-программы на C. Именно в нем содержатся разные определения констант (WM_DESTROY и т. д.).

1.4. Параметры функции WinMain

1. hInstance (тип HINSTANCE) — идентификатор текущего экземпляра приложения. Данное число однозначно определяет программу, работающую под управлением Windows.
2. Параметр hPrevInstance (тип HINSTANCE) указывал ранее (Windows 3.1) на предыдущий запущенный экземпляр приложения. В современных версиях Windows он равен NULL.
3. lpCmdLine — это указатель на строку, заканчивающуюся нулевым байтом. В этой строке содержатся аргументы командной строки приложения (как правило, содержит NULL).
4. nCmdShow — это параметр, который принимает значение одной из системных констант, определяющих способ изображения окна (например, SW_SHOWNORMAL, SW_SHOWMAXIMIZED или SW_SHOWMINIMIZED).

1.5. Регистрация класса окна

Каждое окно, которое создается в рамках приложения Windows, должно основываться на классе окна — шаблоне, в котором определены выбранные пользователем стили, шрифты, заголовки и т. д. Для всех определений класса окна используется стандартный тип структуры. Таким образом, сначала определяется структура WNDCLASS w, а затем поля структуры заполняются информацией о классе окна. У этой структуры много полей, но большинство из них можно определить нулем. В программе это делается строкой memset(&w,0,sizeof(WNDCLASS));

Рассмотрим следующий фрагмент программы:

```
w.style = CS_HREDRAW | CS_VREDRAW;
//значение указателя на функцию окна (WndProc), которая
//выполняет все задачи, связанные с окном
w.lpfnWndProc = WndProc;
//определяется экземпляр приложения, регистрирующий
//класс окна
w.hInstance = hInstance;
//определяется кисть, используемая для закраски фона
//окна.
w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
/*указатель на строку символов, заканчивающуюся на 0, которая определяет имя класса. Класс здесь – это не класс в смысле ООП (объектно-ориентированного программирования).
```

```
Термин один, но смысла два. Исторически классы окон воз-
никли раньше, чем классы ООП.*/
w.lpszClassName = "My Class";
//регистрация происходит при помощи вызова функции
RegisterClass()
RegisterClass(&w);
```

В этом фрагменте определяется стиль класса (все идентификаторы стилей начинаются с префикса `CS_`). В программе значения поля стиля задаются константами. `CS_HREDRAW` обеспечивает перерисовку содержимого клиентской области окна при изменении размера окна по горизонтали. `CS_VREDRAW` обеспечивает перерисовку содержимого клиентской области окна при изменении размера окна по вертикали.

1.6. Создание окна на основе класса окна

```
hwnd = CreateWindow("My Class","Окно пользователя",
WS_OVERLAPPEDWINDOW,500,300,500,380,NULL,NULL,hInstance,N
ULL);
```

Первый параметр функции служит для задания класса окна. Второй — это заголовок окна. Третий определяет стиль окна (обычное перекрывающее окно с заголовком, кнопкой вызова системного меню, кнопками минимизации и максимизации и рамкой). Следующие шесть параметров определяют положение окна на экране (по оси X и по оси Y), размеры окна по оси X и по оси Y, идентификатор родительского окна и идентификатор меню окна. Следующее поле (`hInstance`) содержит идентификатор экземпляра программы, далее следует информация об отсутствии дополнительных параметров (`NULL`). Если создание окна прошло успешно, то функция `CreateWindow(...)` возвращает идентификатор созданного окна, в противном случае — `NULL`. После того как окно создано, его надо показать и обновить. Для того чтобы вывести главное окно приложения на экран, необходимо вызвать функцию `Windows ShowWindow(...)`.

`ShowWindow(hwnd,nCmdShow)` выводит окно на экран. Параметр hwnd содержит идентификатор окна, созданного при вызове `CreateWindow(...)`. Второй параметр определяет, как окно выводится в первый момент.

Последний шаг при выводе окна на экран заключается в вызове функции Windows `UpdateWindow(hwnd)`, которая приводит к перерисовке клиентской области окна.

1.7. Создание цикла обработки сообщений

Теперь программа готова выполнять свою главную задачу — обрабатывать сообщения. Используется стандартный цикл C/C++ — цикл `while`:

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

Использование функции GetMessage

Вызов этой функции позволяет получить для обработки следующее сообщение из очереди сообщений приложения. Данная функция копирует сообщение в структуру сообщения, на которую указывает указатель `msg`, и передает его в основной блок программы. Если значение следующего параметра `NULL`, то будут поступать сообщения, относящиеся к любому окну приложения. Значения последних параметров (0,0) указывают функции, что не надо применять никаких фильтров сообщений. Фильтры могут применяться для распределения получаемых сообщений по категориям, например, нажатие на клавишу или перемещение мыши. После входа в цикл обработки сообщений выйти из него можно, получив только одно сообщение: `WM_QUIT`. Когда обрабатывается сообщение `WM_QUIT`, то возвращается значение «ложь» и цикл обработки сообщений завершается.

Использование функции TranslateMessage

Функция `TranslateMessage(...)` преобразует сообщения виртуальных клавиш в сообщения о символах.

Использование функции DispatchMessage

Функция `DispatchMessage(...)` используется для распределения текущего сообщения соответствующей функции окна.

1.8. Оконная функция WndProc

Вторая часть в любой программе под Windows — это оконная процедура.

Рассмотрим функцию `WndProc`:

```
LONG WINAPI 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;
}
```

Основное назначение оконной функции — это обработка сообщений Windows. Каждое приложение получает много сообщений. Их источник может быть разным. Например, сообщения от пользователя или от самой Windows. Обработка этих сообщений происходит именно в оконной функции. Это означает, что для каждого сообщения необходимо написать свой обработчик. Если обработчика не будет, то приложение не будет обращать внимание на сообщение. У оконной функции четыре параметра. Первый из них, `hwnd` типа `HWND`, задает окно, которое будет обрабатывать сообщение. Второй, `UINT Message`, — это передаваемое сообщение. Два последних, `WPARAM wparam, LPARAM lparam`, задают дополнительные параметры для передаваемого сообщения. Они для каждого сообщения свои. Оконная процедура отправляет сообщение в `switch`, который в примере имеет только один case:

```
switch (Message)
{
  case WM_DESTROY:
  ...
```

То есть рассматриваемая программа пока обращает внимание только на сообщение `WM_DESTROY`. Это сообщение окно получает только при своем уничтожении. После принятия этого сообщения необходимо вызвать функцию `PostQuitMessage(...)`. В ответ на сообщение `WM_DESTROY` необходимо поместить в очередь сообщение `WM_QUIT`. Это и делает функция `PostQiutMessage(...)`, посылая это сообщение в очередь и говоря, что процесс должен быть завершен. Если мы хотим, чтобы наша программа реагировала еще на что-нибудь, то надо написать еще `case`.

Рассмотрим далее ветку `default`. В ней идет вызов функции `DefWindowProc(hwnd, Message, wparam, lparam)`. Основное предназначение этой функции – обработка сообщений Windows, которые не обрабатываются в нашей программе (то есть для которых нет своего `case`). При этом ничего не делается, но очередь из сообщений идет.

2. Библиотека Microsoft Foundation Classes (MFC). Иерархия классов

MFC — это библиотека классов, написанных на языке C++. MFC является оболочкой для Win32 API и содержит многоуровневую иерархию классов. Не все функции Win32 API включены в MFC. С другой стороны, эта библиотека классов охватывает большую часть функциональных возможностей Windows и предоставляет разработчику ряд дополнительных механизмов для проектирования и создания программных продуктов.

На вершине иерархии MFC находится единственный базовый класс CObject. Все остальные классы библиотеки MFC можно условно разбить на две группы: производные и не производные от него. Чаще всего, создание нового MFC-приложения поручается мастеру MFC Application Wizard. Мастер генерирует основной скелет приложения, который впоследствии заполняется нужным кодом, давая готовое приложение.

2.1. Основные классы MFC

Некоторые классы MFC порождаются непосредственно от `CObject`. Наиболее широко используемыми среди них являются `CCmdTarget`, `CFile`, `CDC`, `CGDIObject` и `CMenu`. Класс `CCmdTarget` предназначен для обработки сообщений. Класс `CFile` предназначен для работы с файлами. Класс `CDC` обеспечивает поддержку контекстов устройств. В этот класс включены практически все функции графики GDI. `CGDIObject` является базовым классом для различных GDI-объектов, таких как перья, кисти, шрифты и другие. Класс `CMenu` предназначен для работы меню.

Класс CCmdTarget

От класса `CCmdTarget` порождается очень важный класс `CWnd`. Он является базовым для создания всех типов окон, включая масштабируемые («обычные») и диалоговые, а также различные элементы управления. Наиболее широко используемым производным классом является `CFrameWnd`. В большинстве программ главное окно создается с помощью именно этого класса. От класса `CCmdTarget` через класс `CWinThread` порождается единственный из наиболее важных классов, обращение к которому в MFC-программах происходит напрямую — класс `CWinApp`. Это один из фундаментальных классов, поскольку предназначен для создания самого приложения. В каждой программе имеется один и только один объект этого класса. Как только он будет создан, приложение начнет выполняться.

Класс CWinApp

Класс `CWinApp` является базовым классом, на основе которого образуют обязательный объект — приложение Windows. Основными задачами объекта этого класса являются инициализация и создание главного окна, а затем опрос системных сообщений. Иерархия класса `CWinApp`: `CObject` →`CCmdTarget` →`CWinThread` →`CWinApp`

Класс CWnd

Класс `CFrameWnd` («окна-рамки») и производные от него классы определяют окна-рамки на мониторе. Элементы управления, создаваемые при проектировании интерфейса пользователя, принадлежат семейству классов элементов управления. Появляющиеся в процессе работы приложения диалоговые окна — это объекты классов, производных от `CDialog`. Классы `CView`, `CFrameWnd`, `CDialog` и все классы элементов управления наследуют свойства и поведение своего базового класса `CWnd` («окно»), определяющего, по существу, Windows-окно. Этот класс, в свою очередь, является наследником базового класса `CObject` («объект»). Как правило, структура приложения определяется архитектурой Document-View («документ-вид»). Это означает, что приложение состоит из одного или нескольких документов — объектов, классы которых являются производными от класса `CDocument` (класс «документ»). С каждым из документов связаны один или несколько видов – объектов классов, производных от `CView` (класс «вид») и определяющих методы обработки объектов класса документа.

2.2. Соглашение об именах MFC

В качестве префикса, обозначающего имя класса, библиотека MFC использует заглавную букву ***C*** (от слова class), за которой идет имя, характеризующее назначение класса. Например:

- CWinApp — класс, определяющий приложение;
- CWnd — базовый класс для всех оконных объектов;
- CDialog — класс диалога.

При определении имен функций-членов классов используется три варианта:

1. Имя объединяет глагол и существительное — DrawText (нарисовать текст).
2. Имя состоит только из существительного — DialogBox (блок диалога).
3. Для функций, предназначенных для преобразования одного типа в другой, используются такие имена, как XtoY (из X в Y).

Для членов классов библиотеки MFC используется следующий способ назначения имен: обязательный префикс m_ (от class member — «член класса»), за которым идет префикс, характеризующий тип данных, и завершается все заданием содержательного имени переменной. Например, m_pMainWnd — указатель на класс главного окна. Для переменных, которые не являются членами класса, `m_` не ставится.

2.3. Включаемые файлы

AFXWIN.H содержит описание основных классов библиотеки и сводит воедино все включаемые файлы, необходимые для работы MFC.

AFX.H содержит описания классов общего назначения, макросы, базовые типы данных MFC.

AFXRES.H подключает стандартные идентификаторы ресурсов.

2.4. Обработка сообщений в MFC

Операционная система Windows взаимодействует с приложением, посылая ему сообщения. Таким образом, обработка сообщений является ядром всех приложений. В традиционных приложениях Windows (написанных с использованием только API), каждое сообщение передается в качестве аргументов оконной функции. В оконной функции, с помощью оператора `switch`, определяется тип сообщения, извлекается информация и производятся нужные действия. Используя библиотеку MFC, все это можно сделать проще.

2.4.1. Карта сообщений

Для создания стандартного окна в приложении должен наследоваться класс от `CFrameWnd`. Он содержит конструктор и макрос `DECLARE_MESSAGE_MAP()`. Макрос декларирует карту сообщений, которая определяет, какая член-функция класса должна вызываться в ответ на сообщение Windows. Этот макрос применяется для любого окна, в котором обрабатываются сообщения. Он должен быть последним в декларировании класса, использующего карту сообщений. В конце программы помещается реализация карты сообщений:

```
BEGIN_MESSAGE_MAP(CMainWnd            /*класс окна*/,
                  CFrameWnd       /*класс-предок*/)
END_MESSAGE_MAP()
```

Первый макрос всегда имеет два параметра: первый — класс окна, второй — класс, от которого порожден класс окна. В данном примере карта сообщений пустая, то есть все сообщения обрабатывает MFC.

В библиотеке MFC все возможные сообщения разделены на три основные категории:

1) сообщения Windows;
2) извещения элементов управления;
3) командные сообщения (команды).

В первую категорию входят сообщения, имена которых начинаются с префикса `WM_`, за исключением `WM_COMMAND`. Во вторую категорию входят извещения (notification messages) от элементов управления и дочерних окон, направляемых родительским окнам. Третья категория охватывает все сообщения `WM_COMMAND`, называемых командами (командными сообщениями), от объектов интерфейса пользователя, который включает меню, кнопки панелей инструментов и акселераторы.

В MFC включен набор предопределенных функций – обработчиков сообщений, которые можно использовать в программе. Если программа содержит такую функцию, то она будет вызываться в ответ на поступившее, связанное с ней сообщение. Если в сообщении есть дополнительная информация, она пере-

дается в качестве аргументов функции. Для организации обработки сообщений нужно выполнить следующие действия:

- включить в карту сообщений программы команду соответствующего сообщения;
- включить прототип функции-обработчика в описание класса, ответственного за обработку данного сообщения;
- включить в программу реализацию функции-обработчика.

2.4.2. Включение макрокоманд в карту сообщений

Чтобы программа могла ответить на сообщение, в карту сообщений должна быть включена соответствующая макрокоманда. Названия макрокоманд соответствуют именам стандартных сообщений Windows, но дополнительно имеют префикс `ON_` и заканчиваются парой круглых скобок. Из этого правила есть исключение: сообщению `WM_COMMAND` соответствует макрокоманда `ON_COMMAND(...)`. Причина в том, что это сообщение обрабатывается особым образом. Чтобы включить макрокоманду в очередь сообщений, необходимо поместить ее между командами `BEGIN_MESSAGE_MAP(...)` и `END_MESSAGE_MAP()`. Например, если необходимо обработать в программе сообщение `WM_CHAR`, то очередь должна выглядеть так:

```
BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd)
ON_WM_CHAR()
END_MESSAGE_MAP()
```

В очереди может находиться более одной макрокоманды. Сообщение `WM_CHAR` генерируется при нажатии алфавитно-цифровой клавиши на клавиатуре.

Включение обработчиков сообщений в описание класса. Каждое сообщение, явно обрабатываемое в программе, должно быть связано с одним из обработчиков.

Обработчик — это член-функция класса, вызываемая приложением в ответ на сообщение, связанное с ней с помощью карты сообщений.

Прототипы для обработчиков всех сообщений заранее заданы в MFC. Например, объявим класс с обработчиком сообщения `WM_PAINT`. Это сообщение посылается окну, когда оно должно перерисовать свою клиентскую область.

Пример:

```
class CMainWnd: public CFrameWnd
{
public:
  CMainWnd();
  afx_msg void OnPaint();
  DECLARE_MESSAGE_MAP()
}
```

Спецификатор `afx_msg` означает объявление обработчика сообщения. На данный момент он не используется и представляет собой пустой макрос. Но

в будущем возможны расширения. Поэтому использование спецификатора нужно считать обязательным. Для каждого обработчика должна быть описана его реализация. В ней могут производиться самые разные действия, которые требуются по логике работы программы.

2.4.3. Сообщение WM_PAINT

Операционная система Windows устроена таким образом, что за обновление содержимого окна отвечает программа. Например, если часть окна была перекрыта другим окном, а затем вновь открыта, или минимизированное окно было восстановлено, то окну посылается сообщение `WM_PAINT`. В ответ на него окно должно обновить свою клиентскую область. Прототип обработчика `WM_PAINT` следующий:

```
afx_msg void OnPaint();
```

Макрокоманда называется `ON_WM_PAINT()`.

Для примера создадим обработчик, который выводит строку "Использование OnPaint()" в клиентскую область по координатам x = 25, y = 25:

```
afx_msg void CMainWnd::OnPaint()
{
  CPaintDC paintDC(this);
  paintDC.TextOut(25,  25,  CString("Использование  On-
Paint()"));
}
```

В обработчике `WM_PAINT` нужно всегда пользоваться классом `CPaintDC`, который представляет собой класс клиентской области, но предназначен для использования именно с этим сообщением. Это обусловлено архитектурой самой Windows. Функция `TextOut(…)` предназначена для вывода текста в контекст устройства (в данном случае — в окно). При ее использовании, по умолчанию первые два параметра определяют координаты верхнего левого угла текстовой строки. По умолчанию координаты представляют собой реальные пиксели, ось x направлена слева направо, ось y — сверху вниз. Эта функция перегруженная, наиболее удобный для нас вариант — когда третий параметр имеет тип `CString`. Этот класс входит в MFC и является очень удобной заменой для строк, завершаемых нулем. Большинство реальных окон (за исключением диалогов) должны обрабатывать сообщение `WM_PAINT`. Более того, если Вы хотите написать корректную программу, то весь вывод в окно должен осуществляться только в обработчике `WM_PAINT`. В случае получения контекста из обработчика `WM_PAINT` с помощью класса `CPaintDC`, Windows гарантирует наличие свободного контекста.

3. Ресурсы

3.1. Понятие ресурсов

Структура EXE-файла для Windows такова, что в его конец могут быть записаны некоторые данные, совершенно не зависящие от самой программы. Они называются ресурсами. Ресурсы могут редактироваться совершенно раздельно, хотя находятся в том же файле, где код и данные. Обычно при загрузке программы на выполнение ресурсы не загружаются в память, это делается лишь по запросу от программы. В ресурсах программа может хранить любые данные какого угодно размера. Но есть стандартные типы ресурсов, такие как иконки, битовые образы, курсоры, диалоги, меню. Большинство диалоговых окон не создаются программным путем, а просто их шаблоны загружаются из ресурсов. Сами шаблоны, как и другие ресурсы, редактируются визуально с помощью специальных ресурсных редакторов. При сборке проекта приложения ресурсы добавляются к EXE-файлу уже после связывания. Для описания ресурсов существует специальный язык, а сами описания хранятся в текстовых файлах с расширением rc. Раньше программисты вручную писали сценарии ресурсов на языке ресурсов, сейчас используются визуальные редакторы. Перед добавлением к исполняемому файлу сценарии преобразуются в бинарный вид с помощью компилятора ресурсов, в результате получается файл с расширением res. Как правило, все эти шаги выполняются автоматически при работе из интегрированной среды Visual C++. Заметим, что «ручное» редактирование ресурсных сценариев требуется сейчас уже очень редко, лишь при определении нестандартных ресурсов в сложных проектах. Каждый ресурс имеет свой уникальный идентификатор. Это может быть либо строка, либо число (константа). Числа можно использовать всегда, а строки — не всегда. Редактор ресурсов из Visual C++ помещает имена констант в файл resource.h, который нужно включить в файлы программы. Стандартные идентификаторы хранятся в файле afxres.h, который обычно используется автоматически ресурсным редактором.

3.2. Меню

Меню, как правило, создаются визуально. В Visual C++ за это отвечает редактор ресурсов. Среда автоматически добавляет в проект сценарий ресурсов.

При создании меню для отдельных пунктов могут быть установлены опции:

- выделения серым цветом (в этом случае при выполнении программы пункт меню будет недоступен);
- вставки разделительной горизонтальной черты;
- перехода на новую строку (в этом случае пункты верхнего уровня будут начинаться с новой строки, а нижнего — в новом столбце через вертикальную черту).

Меню, как отдельному ресурсу, должен быть присвоен числовой или символьный идентификатор. При редактировании символьные идентификаторы заключаются в кавычки. Также каждому пункту меню должен быть присвоен

уникальный числовой идентификатор. Это позволит программе реагировать на выбор пункта в меню, при этом будет вызываться соответствующий обработчик. По принятому соглашению все идентификаторы пунктов меню начинаются с IDM_. В самих названиях пунктов можно указывать ключевые клавиши, поставив перед буквой символ &. В этом случае, если меню активно, пункт можно выбрать также и с клавиатуры.

3.2.1. Включение меню в окно приложения

Когда ресурс меню уже создан, его можно использовать в окне программы. Это можно сделать, указывая меню при создании окна: строковый идентификатор ресурса меню нужно указать в качестве последнего параметра в функции Create():

```
this->Create(0, "Приложение с меню", WS_OVERLAPPEDWINDOW,
rectDefautl, 0, "MYMENU");
```

В результате будет создано окно с меню. Но для того чтобы меню можно было использовать, необходимо создать обработчики сообщения WM_COMMAND для каждого пункта меню. Если для какого-то пункта нет обработчика, то MFC заблокирует этот пункт (он будет выделен серым цветом).

3.2.2. Сообщение WM_COMMAND

Это очень широко используемое сообщение. Так, оно посылается окну, когда пользователь выбирает пункт в меню. Идентификатор пункта меню передается как параметр сообщения. Идентификатор определяет, какой из обработчиков должен быть вызван. Для размещения обработчика этого сообщения используется следующая макрокоманда:

```
ON_COMMAND(Идентификатор, ИмяОбработчика);
```

Каждый обработчик для WM_COMMAND должен возвращать значение void. Обработчики не имеют параметров. Имя выбирается произвольно, обычно используется префикс On. Таким образом, можно написать обработчики для каждого пункта меню.

3.3. Акселераторы

Это специальный ресурс, не имеющий визуального представления. Он представляет собой таблицу из комбинаций клавиш и соответствующих им идентификаторов команд. Таблица может быть загружена для конкретного окна с помощью функции с прототипом:

```
BOOL CFrameWnd::LoadAccelTable(LPCSTR ResourceName);
```

После загрузки таблицы акселераторов нажатие заданных в ней комбинаций клавиш приводит к автоматической генерации сообщения WM_COMMAND с идентификатором, определенным в этой таблице для данной комбинации клавиш. Акселераторы легко создавать в среде Visual C++. Для каждого элемента таблицы нужно нажать желаемую клавишу или их комбинацию и указать числовой идентификатор. Если указать идентификаторы, которые уже использовались в меню, то мы получим клавиши быстрого доступа, дублирующие команды меню.

3.4. Окна сообщений

Это простейшие диалоговые окна, предопределенные в системе. Для создания окна сообщения используется функция с прототипом:

```
int CWnd::MessageBox(LPCSTR MessageText, LPCSTR WindowTi-
tle = 0, UINT MessageBoxType = MB_OK);
```

Параметр MessageText определяет само сообщение. Параметр WindowTitle — заголовок окна сообщения. Параметр MessageBoxType задает стиль окна, иконку, отображаемую слева от сообщения, и одну или несколько кнопок. Этот параметр задается комбинацией констант с помощью операции "|", начинающихся на префикс MB_. Все наборы кнопок заранее определены. Функция возвращает идентификатор нажатой кнопки: IDABORT, IDRETRY, IDIGNORE, IDCANCEL, IDNO, IDYES или IDOK. Функция MessageBox(...) выполняет все действия по созданию, отображению и удалению окна, а также обработку сообщений. Программист не должен об этом заботиться.

3.5. Иконки и курсоры

Иконки и курсоры являются ресурсами и обычно хранятся в области ресурсов исполняемого файла.

3.5.1. Создание иконки и курсора

Для создания иконки и курсора нужно использовать ресурсный редактор. Иконки сохраняются в файлах с расширением ico, а курсоры — в файлах с расширением cur. Все курсоры имеют размер 32 x 32. Обычно используются монохромные курсоры. Каждый курсор имеет так называемую горячую точку, по которой определяется положение курсора на экране. Она может располагаться в любом месте курсора. Для курсоров типа «указатель» это обычно вершина указателя. Иконки могут иметь размер 16 x 16, 32 x 32 и 48 x 48. Последний размер обычно не используется. Иконки могут иметь 16 или 256 цветов. При использовании редактора ресурсов среды Visual C++ 4.0 и выше иконки всех размеров можно хранить в одном файле, что обычно и делается. Для современных приложений обязательно наличие иконок размером как 16 x 16, так и 32 x 32.

3.5.2. Загрузка иконки и курсора из ресурсов

Для загрузки иконок и курсоров удобно использовать функции Win API. Перед рассмотрением этих функций следует рассмотреть понятие дескриптора.

Дескриптор — это 32-разрядное беззнаковое целое значение, которое идентифицирует объект в Windows.

При традиционном SDK-программировании дескрипторы используются очень широко. Так, свои дескрипторы имеют иконки, курсоры и само приложение. Для дескрипторов каждого объекта существует свой тип, например, `HICON`, `HCURSOR`, `HINSTANCE`. Будем предполагать, что в декларировании класса основного окна (речь идет о классе C++) объявлены переменные `m_hIconSmall`, `m_hIconBig`, `m_hCursor` — соответственно дескрипторы иконки размером 16 x 16, иконки размером 32 x 32 и курсора. Тогда для загрузки иконок и курсора нужно выполнить следующий код:

```
// Получить дескриптор модуля (приложения)
HINSTANCE hInst = AfxGetInstanceHandle();
// Загрузить иконку 16x16
m_hIconSmall = (HICON) ::LoadImage( hInst,
MAKEINTRESOURCE (IDI_ICON), IMAGE_ICON, 16, 16,
LR_DEFAULTCOLOR;
// Загрузить иконку 32x32
m_hIconBig = (HICON) ::LoadImage( hInst,
MAKEINTRESOURCE(IDI_ICON), IMAGE_ICON, 32, 32,
LR_DEFAULTCOLOR;
// Загрузить курсор
m_hCursor = AfxGetApp()->LoadCursor(IDC_CURSOR);
```

Сначала мы получаем дескриптор модуля с помощью глобальной функции MFC `AfxGetInstanceHandle()`. Затем мы загружаем маленькую и большую иконки с помощью функции Win API `LoadImage(…)` и получаем их дескрипторы. Эта функция позволяет загружать изображения из ресурсов и из файлов. И наконец, мы загружаем курсор уже с помощью функции MFC `LoadCursor(…)`, члена класса `CWinApp`. Функция `AfxGetApp()` возвращает адрес объекта приложения.

3.5.3. Изменение иконки и курсора окна

После того как мы получили дескрипторы иконок и курсора, надо изменить класс окна. Класс окна надо изменять для установки курсора, а для установки иконок использовать функции MFC. Для этого используется API-функция `SetClassLong(…)`, которая изменяет атрибут класса окна (здесь опять имеется в виду структура данных). Первый параметр этой функции — дескриптор окна. Дескриптор окна хранится в члене класса MFC `CWnd` под названием `m_hWnd`. Если главным окном приложения является диа-

лог, то для изменения иконки и курсора потребуется следующий код (он будет работоспособен и для обычных окон):

```
// Устанавливаем курсор для диалогового окна.
SetClassLong(m_hWnd, GCL_HCURSOR, (long) m_hCursor);
// Делаем то же самое для всех элементов управления
for(int i = 0; i < 0xDFFF; ++i)
{
  CWnd *pCtrl = this->GetDlgItem(i);
  if(pCtrl != 0)
    SetClassLong(pCtrl->m_hWnd,    GCL_HCURSOR,    (long)
m_hCursor);
}
// Устанавливаем иконки. Здесь можно использовать MFC.
this->SetIcon(m_hIconBig, TRUE);
this->SetIcon(m_hIconSmall, FALSE);
```

Сначала мы модифицируем курсор в оконном классе самого окна. Затем то же самое проделываем со всеми элементами управления, которые есть или могут быть в диалоговом окне (если их нет в окне, то ничего страшного не произойдет). Мы перебираем все возможные идентификаторы, и, если элемент присутствует, изменяем его оконный класс. И наконец, мы изменяем большую и маленькую иконки.

3.5.4. Стандартные иконки и курсоры

Часто в программах необходимо использовать курсоры и иконки, уже предопределенные в Windows. Для этого можно использовать или функцию API `LoadImage(…)`, или функции MFC `LoadStandardIcon(…)` и `LoadStandardCursor(…)`. Работа со стандартными курсорами и иконками почти ничем не отличается от работы с пользовательскими курсорами и иконками. Для стандартных иконок есть предопределенные идентификаторы с префиксом `IDI_`, а для стандартных курсоров — с префиксом `IDC_`. Например, стандартный курсор в виде песочных часов имеет идентификатор `IDC_WAIT`.

4. Элементы управления и диалоги

4.1. Диалоги. Элементы управления

Диалог (диалоговое окно) представляет собой специальный вид окон, которые предназначены для взаимодействия с пользователем. Обычно они используются для изменения настроек приложения и ввода информации. Например, практически все окна настроек приложения Microsoft Word являются диалогами. Есть два типа диалогов: модальные и немодальные. Наиболее распространены первые. Если диалог модальный, то при активизации диалога основное окно приложения становится пассивным и перестает реагировать на действия пользователя до тех пор, пока он не закроет диалог. Если диалог немодальный, то он существует независимо от других окон, и основное окно также может быть активизировано.

4.2. Взаимодействие между диалогом и пользователем

Взаимодействие между диалогом и пользователем осуществляется с помощью элементов управления. Это особый тип окон для ввода или вывода. Элемент управления принадлежит окну-владельцу, в данном случае — диалогу. Все версии Windows поддерживают некоторый набор стандартных элементов управления, к которым относятся кнопки, контрольные переключатели, селекторные кнопки, списки, поля ввода, комбинированные списки, полосы прокрутки и статические элементы.

Рассмотрим кратко каждый из них:

- **Обыкновенная кнопка (push button)** — это кнопка, которую пользователь «нажимает» мышью или клавишей Enter, переместив предварительно на нее фокус ввода.
- **Контрольный переключатель (check box, флажок)** может быть либо выбранным, либо нет. Если в диалоге есть несколько контрольных переключателей, то могут быть выбраны одновременно несколько из них.
- **Селекторная кнопка (radio button)** С — это почти то же, что и контрольный переключатель. Отличие состоит в том, что при наличии нескольких кнопок в группе может быть выбрана только одна.
- **Список (list box)** содержит набор строк, из которого можно выбрать одну или несколько. Широко используется при отображении имен файлов.
- **Поле ввода (edit box)** — это элемент, позволяющий ввести строку текста.
- **Комбинированный список (combo box)** представляет собой список со строкой ввода.
- **Статический элемент (static control)** предназначен для вывода текста или графики, но не для ввода.

Элементы управления способны как генерировать сообщения в ответ на действия пользователя, так и получать их от приложения. В последнем случае сообщения являются фактически командами, на которые элемент управления должен отреагировать.

4.3. Классы MFC для элементов управления

В MFC содержатся классы для всех стандартных элементов управления. Эти классы описывают сами элементы, а также содержат функции для работы с ними. Их называют классами управления (табл. 1). Они порождаются от класса `CWnd`. Таким образом, все они обладают характеристиками окна.

Таблица 1

Основные классы управления

Класс	Элемент управления
CButton	Кнопки, селекторные кнопки и контрольные переключатели
CEdit	Поля ввода
CListBox	Списки
CComboBox	Комбинированные списки
CScrollBar	Полосы прокрутки
CStatic	Статические элементы

В MFC допускается непосредственное обращение к элементам управления, но на практике это происходит очень редко. Удобнее пользоваться соответствующими классами. Наиболее часто элементы управления используются с диалоговыми окнами, хотя можно создавать и отдельные элементы, расположенные в главном окне.

4.4. Диалоги как ресурсы

Диалоги не создаются программно. При необходимости из ресурсов загружаются описания диалогов, и Windows по этому описанию формирует окно и размещает на нем все элементы управления. Диалоги редактируются визуально из ресурсного редактора. Диалог вместе со всеми элементами управления представляет собой один ресурс со своим идентификатором. Кроме того, каждый элемент управления имеет свой идентификатор, который может быть только числовым. Обычно идентификаторы имеют префикс в соответствии с названием данного элемента управления, хотя при желании можно использовать любые идентификаторы.

В MFC все диалоги являются экземплярами либо класса `CDialog`, либо порожденных от него классов. Лишь самые простые диалоги используют непосредственно класс `CDialog`. В общем же случае необходимо определять собственный класс. Класс CDialog имеет конструкторы со следующими прототипами:

```
CDialog::CDialog(LPCSTR ResourceName, CWnd *Owner = 0);
CDialog::CDialog(UINT ResourceID, CWnd *Owner = 0);
CDialog::CDialog();
```

Параметр `ResourceName` или `ResourceID` определяет идентификатор диалога в ресурсах, строковый или числовой. Параметр `Owner` — это указатель на окно-собственник, если он равен 0, то собственником будет главное окно

приложения. Последняя форма конструктора предназначена для создания немодальных диалогов.

4.4.1. Обработка сообщений от диалогов

Все диалоги являются разновидностью окон, поэтому для них используется такой же механизм сообщений, как и для главного окна. Для каждого диалога организуется собственная очередь сообщений, так же точно, как и для главного окна. Когда элемент управления диалога активизируется, диалогу посылается сообщение `WM_COMMAND`. С этим сообщением передается идентификатор элемента управления. Для обработки сообщений в карту сообщений диалога нужно поместить макрос `ON_COMMAND()`. Многие элементы управления генерируют также идентификационный код, который позволяет определить, какое действие было произведено с элементом управления. Во многих случаях по этому коду выбирается тот или иной обработчик.

4.4.2. Вызов модального диалога

После того как объект класса диалога создан, необходимо вызвать член-функцию `DoModal()`. Результатом вызова будет модальное отображение диалога. Прототип функции следующий:

```
virtual int CDialog::DoModal();
```

Функция возвращает код завершения, генерируемый диалогом при закрытии, или -1, если окно не может быть отображено. Если при отображении диалога произошла ошибка, возвращается `IDABORT`. Функция не завершается, пока диалог не будет закрыт.

4.4.3. Закрытие модального диалога

По умолчанию диалог закрывается при получении сообщения с идентификатором либо `IDOK`, либо `IDCANCEL`. Они предопределены и обычно связаны с кнопками подтверждения и отмены. Класс `CDialog` содержит встроенные обработчики для этих двух случаев, `OnOK()` и `OnCancel()`. Их не нужно включать в очередь сообщений диалога. Но их можно переопределить, что дает возможность программисту управлять закрытием диалога. Для программного закрытия диалога необходимо вызвать член-функцию с прототипом:

```
void CDialog::EndDialog(int RetCode);
```

Параметр определяет значение, которое вернет функция `DoModal()`. Обычно возвращаются значения `IDOK` или `IDCANCEL`, другие значения используются редко.

4.4.4. Инициализация диалога

Часто на практике возникает ситуация, когда различные переменные и элементы управления, связанные с диалогом, должны быть инициализированы до того, как диалог будет отображен. Чтобы позволить диалогу выполнить подобные действия, Windows автоматически посылает ему сообщение `WM_INITDIALOG` в момент создания. При получении такого сообщения MFC автоматически вызывает метод `OnInitDialog()`, который является стандартным обработчиком, определенным в классе `CDialog`. Эта функция переопределяется в программе, если необходимо выполнение инициализации. Прототип функции:

```
virtual BOOL CDialog::OnInitDialog();
```

Функция вызывается до того, как диалог будет отображен. Она должна возвращать `TRUE`, чтобы Windows могла передать фокус ввода на первый элемент управления в окне (то есть сделать его активным). Первым действием в переопределенной функции должен быть вызов функции `CDialog::OnInitDialog()`.

4.4.5. Немодальные диалоги

Немодальные диалоги получают сообщения параллельно с основным окном приложения. То есть как минимум два окна будут одновременно активными. Поэтому работа с немодальными диалогами требует больше усилий — должны быть выполнены дополнительные операции. Для создания немодального диалога необходимо создать «пустой» объект диалога, то есть не связанный с шаблоном из ресурсов. Привязка к ресурсам осуществляется через функцию `Create(…)`. Рассмотрим этот процесс подробнее. Для создания объекта немодального диалога необходимо использовать конструктор `CDialog::CDialog()` без параметров. Он объявлен как protected-член класса. Это означает, что он может быть вызван только изнутри члена-функции порожденного класса. Это сделано для того, чтобы программист обязательно определял свой порожденный класс для немодального диалога, а в нем — дополнительные операции для немодального диалога. Когда экземпляр создан, он привязывается к ресурсам с помощью функций:

```
BOOL CDialog::Create(LPCSTR ResourceName, CWnd *Owner =
0);
BOOL CDialog::Create(UINT ResourceId, CWnd *Owner = 0);
```

Первый параметр определяет идентификатор диалога в ресурсах. Второй параметр определяет окно-собственник для диалога. Необходимо помнить о том, что объект немодального диалога должен существовать в течение всего времени использования диалога. Функция `Create(…)` отображает окно и после этого немедленно завершает свою работу. А объект окна должен существо-

вать. В отличие от модальных окон, немодальные не становятся автоматически видимыми при вызове. Чтобы диалог сразу был видимым, необходимо в ресурсном редакторе установить опцию Visible. Также можно использовать функцию `ShowWindow(…)`. Для закрытия немодального диалога необходимо использовать функцию DestroyWindow(). Это означает, что функции `OnCancel()` и/или `OnOK()` должны быть переопределены.

4.4.6. Использование диалога в качестве главного окна

Часто бывает удобным использование диалога в качестве главного окна. Реализовать этот случай достаточно просто.

Во-первых, необходимо создать диалог в ресурсах.

Во-вторых, породить класс главного окна приложения от `CDialog`. Перед конструктором класса главного окна необходимо вызвать конструктор класса `CDialog`, и в нем привязать объект к ресурсам, например:

```
CMainFrame::CMainFrame():CDialog(IDD_MYDIALOG)
{
//... здесь тело конструктора
}
```

В-третьих, в функции `CApp::InitInstance()` должен присутствовать следующий код:

```
// Создаем объект диалогового окна
CMainFrame dlgWnd;
// Сообщаем MFC адрес окна
m_pMainWnd = &dlgWnd;
// Отображаем модальный диалог
dlgWnd.DoModal();
// Возвратим FALSE, чтобы MFC не пыталась инициировать
// очередь сообщений главного окна.
return FALSE;
```

Мы отображаем модальный диалог. Так как при завершении функции `DoModal()` нам уже не нужна очередь сообщений, мы «обманываем» MFC, делая вид, что инициализация прошла неудачно.

5. Элементы управления Windows

5.1. Списки

Список является одним из наиболее распространенных элементов управления. В MFC работа со списком осуществляется через класс `CListBox`. Списки являются элементами управления, требующими двустороннего взаимодействия между ними и программой. То есть список может как посылать, так и принимать сообщения. Например, сообщения посылаются списку при его инициализации. Сюда входит передача набора строк, которые будут отображены в окне списка (по умолчанию список создается пустым). Когда список инициализирован, он посылает сообщения о действиях, произведенных с ним пользователем.

5.1.1. Прием идентификационных кодов списка

Список может генерировать сообщения различных типов. Например, сообщения посылаются при двойном щелчке мышью на элементе списка, при потере списком фокуса ввода и при выборе другого элемента из списка. Каждое такое событие описывается идентификационным кодом. Этот код является частью сообщения `WM_COMMAND`. Некоторые другие элементы также используют идентификационные коды. Рассмотрим код `LBN_DBLCLK`. Он посылается, когда пользователь выполняет двойной щелчок на элементе списка. При определении списка в ресурсах должна быть установлена опция Notify, чтобы он мог генерировать это сообщение. Когда выбор произведен, необходимо запросить список, чтобы узнать о том, какой элемент выбран. Для обработки сообщения `LBN_DBLCLK` необходимо поместить его обработчик в карту сообщений. Но это будет не макрос `ON_COMMAND()`. Вместо этого используются специальные макрокоманды. Для нашего сообщения это будет `ON_LBN_DBLCLK()`. Она имеет такой вид:

```
ON_LBN_DBLCLK (ИдентификаторСписка, ИмяОбработчика)
```

Многие сообщения обрабатываются подобным образом. Названия всех макросов для таких сообщений начинаются с префикса `ON_LBN_`.

5.1.2. Передача сообщений списку

В традиционных Windows-программах сообщения посылаются элементам управления с помощью API-функций, например `SendDlgItemMessage()`. Но в программах на MFC для этих целей применяются соответствующие функции – члены класса. Эти функции автоматически посылают необходимое сообщение элементу управления. В этом заключается преимущество использования MFC по сравнению с традиционным методом программирования. Списку может быть послано несколько разных сообщений. Для каждого сообщения класс

CListBox содержит отдельный член – функцию класса. Например, рассмотрим следующие функции:

```
int CListBox::AddString(LPCSTR StringToAdd);
int CListBox::GetCurSel() const;
int CListBox::GetText(int Index, LPCSTR StringVariable);
```

Функция AddString(…) вставляет указанную строку в список. По умолчанию она вставляется в конец списка, при этом начало списка имеет индекс 0. Функция GetCurSel() возвращает индекс текущего выделенного элемента. Если ни один элемент не выбран, то функция возвращает LB_ERR. Функция GetText(…) получает строку, связанную с указанным индексом. Строка копируется в символьный массив по адресу StringVariable.

5.1.3. Получение указателя на список

Функции CListBox работают с объектами CListBox. Поэтому необходимо получить указатель на объект списка, что делается с помощью функции GetDlgItem(), являющейся членом класса CWnd:

```
CWnd *CWnd::GetDlgItem(int ItemIdentifier) const;
```

Функция возвращает указатель на объект, чей идентификатор передан как параметр. Если такой объект не существует, то возвращается 0. Значение, возвращенное функцией, должно быть приведено к типу указателя на конкретный класс управления. Например, в нашем случае это тип CListBox*.

5.1.4. Инициализация списка

По умолчанию список создается пустым, поэтому он должен инициализироваться каждый раз, когда отображается диалог. Для этого необходимо переопределить функцию OnInitDialog(), в которой в список добавлялись бы строки. Если при добавлении элементов в список их число превысит то, которое помещается в окне списка, то в этом окне автоматически появится вертикальная полоса прокрутки.

5.2. Поле ввода

На практике поля ввода используются очень широко, так как дают возможность ввести строку по своему усмотрению. Поля ввода принимают многие сообщения и сами могут генерировать несколько типов сообщений. Но обычно отвечать на большинство из них нет необходимости, так как поля ввода самостоятельно выполняют большинство функций редактирования. Для этого не требуется взаимодействия с программой. Необходимо только решить, когда затребовать содержимое поля ввода. Для получения текущего содержимого поля ввода, состоящего из одной строки, используется функция GetWindowText(). Ее прототип:

```
int CWnd::GetWindowText(LPSTR StringVariable, int Max-
StringLen) const;
```

В результате выполнения функции содержимое поля ввода будет скопировано в строку по адресу StringVariable. Эта функция позволяет получить текст, связанный с любым окном или элементом управления. Применительно к обычному окну функция получает заголовок окна. В момент создания поле ввода является пустым. Для инициализации его содержимым используется еще одна функция – член класса CWnd — SetWindowText(). Она отображает строку в элементе управления, который вызвал эту функцию. Ее прототип:

```
void CWnd::SetWindowText(LPCSTR String);
```

5.3. Контрольные переключатели

Контрольный переключатель — это элемент управления, предназначенный для установки или снятия определенной опции. Визуально он состоит из маленького прямоугольного поля, в котором может стоять метка выбора. Кроме этого, с переключателем связано текстовое поле с описанием предоставляемой переключателем опции. Если в переключателе стоит метка выбора, то говорится, что он выбран (установлен). Контрольные переключатели в MFC описываются с помощью класса CButton (так как контрольный переключатель — разновидность кнопки). Контрольные переключатели могут быть автоматическими и программными. Автоматический переключатель сам меняет свое состояние при щелчке мышью. Программный же этого не делает, а подразумевается, что сообщение о щелчке будет обработано в программе, и она изменит состояние переключателя. На практике почти всегда используются автоматические переключатели.

5.3.1. Сообщения контрольного переключателя

Каждый раз, когда пользователь щелкает мышью на контрольном переключателе (или нажимает клавишу Spacebar, когда фокус ввода находится на переключателе), диалогу посылается сообщение WM_COMMAND с идентификационным кодом BN_CLICKED. Это сообщение обрабатывается с помощью макроса ON_BN_CLICKED(). При работе с автоматическими переключателями отвечать на это сообщение нет необходимости. Но при работе с программными переключателями, чтобы изменять их состояние, необходимо отвечать на это сообщение. Для этого необходимо поместить макрос в карту сообщений и написать обработчик.

5.3.2. Установка и чтение состояния контрольного переключателя

Чтобы установить контрольный переключатель в заданное состояние, необходиимо использовать функцию SetCheck(…) с прототипом:

```
void CButton::SetCheck(int Status);
```

Параметр определяет требуемое состояние: если он равен 1, то переключатель устанавливается, если 0 — сбрасывается. По умолчанию, при первом вызове диалога переключатель будет сброшен. Автоматический переключатель также может быть установлен в требуемое состояние этой функцией. Текущее состояние переключателя можно определить с помощью функции `GetCheck()`:

```
int CButton::GetCheck() const;
```

Функция возвращает 1, если переключатель установлен, и 0 в противном случае.

5.3.3. Инициализация контрольных переключателей

При вызове диалога переключатели сброшены. Но обычно они должны устанавливаться в предыдущее состояние при каждом вызове диалога. Таким образом, переключатели необходимо инициализировать. Для этого необходимо переопределить функцию `OnInitDialog()`, и в ней использовать функкцию `SetCheck()` для установки начальных состояний.

5.3.4. Статические элементы управления

Статическим называется элемент, который не принимает и не генерирует сообщений. Формально этим термином называют то, что просто отображается в диалоговом окне, например, текстовая строка или рамка, предназначенная для визуального объединения нескольких элементов управления, или рисунок. Если элементу присвоен идентификатор `IDC_STATIC (-1)`, то он не будет принимать и генерировать сообщения. Но в общем случае статические элементы управления могут это делать. Для этого элементу нужно присвоить другой идентификатор, и тогда элемент уже не будет статическим. Это часто используется. Например, можно поменять текст в текстовой строке с помощью функции `SetWindowText()`, чтобы отобразить некоторую информацию.

5.4. Селекторные кнопки

Использование селекторных кнопок очень похоже на использование контрольных переключателей, но их работа организована таким образом, что из группы кнопок может быть установлена только одна. При установке другой кнопки предыдущая установка сбрасывается. Селекторные кнопки бывают программные и автоматические; но так как управлять радиокнопками сложно, то сейчас почти всегда используются автоматические. Радиокнопки объединяются в группы. В одном диалоге может быть несколько групп. Для первой радиокнопки каждой группы в редакторе ресурсов нужно установить опцию Group, а для других радиокнопок группы она должна быть сброшена. Радиокнопки нумеруются в порядке значений их идентификаторов (то есть в порядке их создания в редакторе ресурсов). Если в диалоге все радиокнопки образуют одну

группу, то опцию Group можно не устанавливать. Селекторные кнопки управляются с помощью класса CButton. Также как для контрольных переключателей, состояние селекторных кнопок можно изменять с помощью функции SetCheck() и читать с помощью функции GetCheck(). При создании диалога все селекторные кнопки сброшены. Таким образом, в функции OnInitDialog() необходимо установить начальное состояние программно. Хотя из программы можно установить сразу несколько селекторных кнопок или сбросить все, хороший стиль программирования под Windows предполагает, что всегда будет установлена одна и только одна селекторная кнопка.

5.5. Полосы прокрутки

В Windows есть два типа полос прокрутки. Элементы первого типа являются частью окна (включая диалоговое окно), поэтому их называют полосами прокрутки окна. Элементы второго типа существуют независимо и называются независимыми полосами прокрутки. Элементы первого типа описываются классом CWnd, а второго — CScrollBar.

5.5.1. Создание стандартных полос прокрутки

Если требуется, чтобы окно содержало стандартные полосы прокрутки, они должны быть явно заданы. Применительно к главному окну это означает, что при вызове функции Create() в качестве параметров стиля должны быть указаны опции WS_VSCROLL и WS_HSCROLL. В случае диалогового окна достаточно установить соответствующие опции диалога в ресурсном редакторе. Если все это сделано, то полосы прокрутки будут отображаться в окне автоматически.

Для включения в диалог независимой полосы прокрутки используется ресурсный редактор. Можно создавать горизонтальные и вертикальные полосы прокрутки. Также можно установить требуемые длину и ширину полосы прокрутки. Полоса прокрутки, так же как и любой другой элемент управления, должна иметь свой уникальный идентификатор.

5.5.2. Обработка сообщений полосы прокрутки

Так как полоса прокрутки пришла из 16-разрядной Windows 3.1, то управлять ею довольно сложно. Полоса прокрутки сама ничего не делает. Даже для того, чтобы она «прокручивалась» на экране, необходим дополнительный программный код. Полосы прокрутки при выполнении над ними действий посылают сообщения WM_VSCROLL и WM_HSCROLL при активизации соответственно вертикальной или горизонтальной полосы прокрутки. Эти сообщения обрабатываются функциями со следующими прототипами:

```
    afx_msg void CWnd::OnVScroll(UINT SBCode, int Pos,
CScrollBar *SB);
    afx_msg void CWnd::OnHScroll(UINT SBCode, int Pos,
CScrollBar *SB);
```

Следует отметить, что при наличии нескольких горизонтальных или вертикальных полос прокрутки для всех них будет вызываться один и тот же обработчик. Первый параметр, SBCode, содержит код выполненного над полосой прокрутки действия. Если работа ведется с вертикальной полосой прокрутки, то при каждом изменении положения ползунка на одну позицию вверх посылается код SB_LINEUP. При изменении позиции на одну вниз посылается код SB_LINEDOWN. Аналогично при постраничном перемещении генерируются коды SB_PAGEUP и SB_PAGEDOWN. Если работа ведется с горизонтальной полосой прокрутки, то при каждом передвижении ползунка на одну позицию влево посылается код SB_LINELEFT. При изменении его положения на одну позицию вправо посылается код SB_LINERIGHT. При постраничном перемещении генерируются сообщения SB_PAGELEFT и SB_PAGERIGHT. Для обоих типов полос прокрутки при перемещении ползунка на новую позицию посылается код SB_THUMBPOSITION. Если при этом кнопка мыши удерживается нажатой, то дополнительно генерируется сообщение с кодом SB_THUMBTRACK. Это позволяет отслеживать перемещения ползунка, прежде чем мышь будет отпущена. Параметр Pos указывает текущую позицию ползунка. Если сообщение сгенерировано стандартной полосой прокрутки, то параметр SB будет равен 0. Если же оно было сгенерировано независимой полосой прокрутки, то этот параметр будет содержать указатель на объект. Это предоставляет весьма неуклюжий способ различать, какая конкретно независимая полоса прокрутки сгенерировала сообщение. Для этого нужно использовать функцию CWnd:: GetDlgCtrlID(), которая возвращает идентификатор элемента управления. Такое неудобство связано с тем, что MFC повторяет внутреннее устройство Windows, а не является библиотекой сверхвысокого уровня для быстрой разработки приложений.

5.5.3. Управление полосой прокрутки

Ранее для установки различных параметров полосы прокрутки использовались отдельные функции, которые были в Windows 3.1. С появлением Windows 95 появилась возможность управления полосами прокрутки с помощью одной функции `SetScrollInfo()`. Эта функция позволяет сделать полосу прокрутки пропорциональной (в этом случае чем меньше диапазон полосы прокрутки, тем длиннее будет ее ползунок). Функция `GetScrollInfo()` предназначена для чтения параметров полосы прокрутки. В отличие от старых функций эти функции работают с 32-разрядными данными. Для стандартных полос прокрутки используется функция:

```
BOOL CWnd::SetScrollInfo(int Which, LPSCROLLINFO pSI,
BOOL Redraw = TRUE);
```

Значение `Which` указывает, с горизонтальной или вертикальной полосой ведется работа. Параметр `pSI` указывает на структуру, содержащую информацию для полосы прокрутки. Последний параметр задает необходимость перери-

совки полосы прокрутки. Обычно используется значение по умолчанию. Для независимых полос прокрутки используется функция:

```
BOOL CScrollBar::SetScrollInfo(LPSCROLLINFO pSI, BOOL
Redraw = TRUE);
```

Оба параметра имеют такой же смысл. Для чтения параметров стандартных полос прокрутки используется функция:

```
BOOL CWnd::GetScrollInfo(int Which, LPSCROLLINFO pSI,
UINT Mask = SIF_ALL);
```

Информация, получаемая от полосы прокрутки, записывается в структуру по адресу `pSI`. Значение параметра `Mask` определяет, какая информация записывается в структуру. По умолчанию заполняются все поля. Для независимых полос прокрутки вариант функции таков:

```
BOOL CScrollBar::SetScrollInfo(LPSCROLLINFO pSI, UINT
Mask = SIF_ALL);
```

Значение параметров — аналогичное предыдущему случаю. Во всех вариантах функций используется следующая структура типа `SCROLLINFO`:

```
typedef struct tagSCROLLINFO
 {
     UINT cbSize;     // размер самой структуры
     UINT fMask;      // маска для параметров
     int nMin;        // нижняя граница диапазона
     int nMax;        // верхняя граница диапазона
     UINT nPage;      // размер страницы
     int nPos;        // позиция ползунка
     int nTrackPos;   //   позиция   ползунка   во   время
                      //перемещения
} SCROLLINFO;
```

Поле `fMask` определяет, какое из полей структуры содержит требуемую информацию. Используются константы с префиксом `SIF_`, которые можно объединять операцией "| ". Поле `nPos` содержит статическую позицию ползунка. Поле `nPage` содержит размер страницы для пропорциональных полос прокрутки. Для получения обычной пропорциональной полосы прокрутки в этом поле нужно задать значение 1. Поля `nMin` и `nMax` содержат нижнюю и верхнюю границы диапазона полосы прокрутки. Поле `nTracksPos` содержит позицию ползунка при его перемещении, это значение не может быть установлено.

6. Графический вывод

6.1. Классические функции графического устройства

При выводе на экран графической информации (линии, текста, изображения и т. п.) программа обращается к функциям GDI (graphic device interface) интерфейса графического устройства. Эти функции поддерживаются каркасом MFC и для удобства разработчика объединены в классы. Основным классом для работы с графикой является класс `CDC` и производные от него `CPaintDC`, `CClientDC`, `CWindowDC`, которые отличаются от базового класса только конструкторами и деструкторами. Исключением является класс `CMetaFileDC`. Класс `CDC` инкапсулирует понятие контекста устройства DC (device context).

Контекст устройства DC(device context) — структура данных, которая определяет набор графических объектов и методов для графического вывода. Контекст устройства является посредником между операционной системой Windows и устройством вывода, тем самым обеспечивается аппаратная независимость программы. Весь процесс отображения графики осуществляется с помощью этого класса.

6.1.1. Особенности производных классов контекста устройства

Объекты CMetaFileDC обеспечивают доступ к метафайлам Windows. Вызовы функций – членов класса `CMetaFileDC` записываются в связанном с соответствующим объектом файле. Для построения изображения требуется воспроизвести последовательность команд, записанных в метафайле.

Классы CClientDC и CWindowDC отличаются друг от друга лишь тем, что представляют для рисования различные области окна. `CClientDC` представляет клиентскую часть (часть окна без рамки, заголовка, меню, панели управления и строки состояния). `CWindowDC` — полнооконный контекст устройства, позволяющий рисовать в произвольной области окна программы.

Объекты класса CPaintDC используются только в обработчике сообщения `WM_PAINT`, генерируемого в ответ на вызов функций `UpdateWindow` или `RedrawWindow` и необходимы, если требуется переопределить функцию `OnPaint()` для конкретного дисплея. По умолчанию обработчик `OnPaint()` вызывает `OnDraw(CDC*)` с уже настроенным нужным образом контекстом. Конструктор `CPaintDC` определен так, что выполняет все действия, необходимые для инициализации данного дисплея.

6.1.2. Создание и уничтожение объектов CDC

Управление созданием и удалением объектов `CDC` является важной частью каждой программы. При неправильной работе с контекстами теряется память до завершения работы программы. Рассмотрим два варианта корректной работы с объектами `CDC`.

1. Создать объект в стеке, тогда он будет уничтожен автоматически:

```
void CMyView::SomeFunction(...)
{
 CRect cr;
 CClientDC dc(this);
 dc.GetClipBox(cr);
}
```

2. Получить указатель на объект с помощью `CWnd::GetDC()`, при этом перед выходом из функции вызвать `ReleaseDC()`:

```
void CMyView::SomeFunction(...)
{
  CRect rect;
  CDC* pDC = GetDC();
  pDC->GetClipBox(rect);
  ReleaseDC(pDC);
}
```

Нельзя удалять `CDC`-объект, указатель на который передается функции `OnDraw`. За его удаление отвечает каркас MFC.

6.1.3. Состояние контекста устройства. Объекты GDI

Состояние контекста устройства определяется связанными с ним графическими объектами. Свойства контекста устройства назначаются с помощью методов класса CDC. GDI-объекты загружаются в контекст устройства вызовом перегруженной функции `SelectObject`. В любой текущий момент с контекстом может быть связан только один объект каждого типа. Все объекты GDI представлены в MFC с помощью классов. `CGdiObject` — базовый абстрактный класс для GDI-объектов, которые являются экземплярами классов наследников.

1. **CBitmap** — класс, инкапсулирующий растровые изображения (битовые массивы). Растровые изображения используются для отображения картинок и создания кистей.
2. **CBrush** — кисть. Точечный шаблон, использующийся для закрашивания областей окна.
3. **CFont** — шрифт. Полный набор символов алфавита определенного вида и размера. Шрифты хранятся на диске как ресурсы.
4. **CPallete** — палитра. Таблица преобразования цветов, позволяет приложению полностью задействовать цветовые возможности устройства, не вызывая конфликта с другими приложениями, работающими с этим же устройством.
5. **CPen** — перо. Инструмент для рисования линий и границ фигур.
6. **CRgn** — регион. Область окна, определяемая прямоугольником, эллипсом или всевозможными их комбинациями.

При создании GDI-объектов вызываются конструкторы соответствующих классов. Но для некоторых этого недостаточно. Например, создание объектов типа `CFont` или `CRgn` требует вызова `CreateFont(…)` или `CreatePolygonRgn(…)`. Прежде чем удалять GDI-объект, его требуется вначале «отсоединить» от контекста устройства. Память, выделенная под GDI-объекты, принадлежит процессу и освобождается при его завершении. Такие объекты, как растровые изображения, занимают значительный объем памяти, за их своевременным удалением необходимо следить.

Пример работы с GDI-объектом:

```
void CMyView::OnDraw(CDC* pDC)
{
 CPen myPen(PS_SOLID, 2, RGB(255,0,0));
 CPen *oldPen = pDC->SelectObject(&myPen);
 //Присоединение нового пера и сохранение старого
 //-------------------------------------
      //Рисование....
 //-------------------------------------

 //Возвращение контекста устройства в прежнее
 //состояние
 pDC->SelectObject(oldPen);
 }
 //созданное в стеке перо будет удалено при выходе
 //из функции
```

При уничтожении контекста устройства все связанные с ним GDI-объекты отсоединяются. Если известно, что контекст устройства будет уничтожен раньше, чем удалены выбранные в него объекты, отсоединять эти объекты не надо. При работе с контекстом дисплея в начале каждой функции обработчика сообщений создается новый контекст. Набор выбранных объектов, а также режим преобразования координат и другие параметры теряются. Поэтому необходимо настраивать его каждый раз заново. Для настройки преобразования координат используется функция `OnPrepareDC`, но собственными GDI-объектами необходимо управлять самим.

6.1.4. Основные методы класса CDC контекста устройства

Из всех методов класса CDC можно выделить две основные группы: 1) функции создания и настройки контекста устройства, 2) функции рисования. Рассмотрим их подробнее.

BOOL CreateDC(LPCTSTR lpszDriverName, LPCTSTR lpszDeviceName, LPCTSTR lpszOutput, const void* lpInitData)

Это основная функция для инициализации контекста устройства. Первый параметр — указатель на строку с именем драйвера устройства. Второй пара-

метр — указатель на строку с именем устройства. Необходим, если драйвер поддерживает несколько устройств. Третий параметр — указатель на строку с именем файла или порта, куда будет осуществляться вывод. Четвертый содержит особые параметры для настройки данного устройства. Функция возвращает true или false в зависимости от успеха или неудачи. Эта функция редко используется, обычно каркас MFC сам создает необходимый контекст.

BOOL CreateCompatibleDC(CDC* pDC)

Создает в памяти объект контекста устройства, указатель на который передается в качестве параметра, совместимый с данным.

SelectObject(…)

Это основная функция для связи с контекстом устройства GDI-объекта. Рассмотрим подробнее ее прототип. `CPen* SelectObject (CPen* pPen)` связывает перо, указатель на которое передан в качестве параметра, с контекстом устройства. Возвращает указатель на перо, которое находилось в контексте устройства, до вызова функции. При неудаче возвращает NULL. Далее приведены перегруженные варианты данной функции.

```
CBrush* SelectObject(CBrush* pBrush)
virtual CFont* SeiectObject (CFont* pFont)
CBitmap* SelectObject (CBitmap* pBitmap)
int SelectObject (CRgn* pRgn)
```

В зависимости от типа параметра с контестом устройства связывается соответствующий GDI-объект.

virtual CGdiObject* SelectStockObject (int nlndex)

Связывает с контекстом один из стандартных объектов, идентификатор которого передается в качестве параметра. Также как и `SelectObject(…)`, в случае успеха возвращает указатель на объект, который был связан с контекстом до ее вызова, в случае неудачи — `NULL`.

CPen* GetCurrentPen( ) const

Возвращает указатель на выбранное в контекст перо. Далее приведены примеры аналогичных функций.

```
CPalette* GetCurrentPalette( ) const
CBrush* GetCurrentBrush( ) const
```

Функции с префиксом Get возвращают определенные параметры контекста устройства.

int SetBkMode (int nBkMode)

Устанавливает режим закрашивания фона. В качестве параметра передается идентификатор, определяющий тип закрашивания. Возвращает идентификатор предыдущего типа закрашивания.

6.1.5. Функции для преобразования системы координат

Будут далее рассмотрены подробнее. Ниже приведены их прототипы.

```
virtual int SetMapMode(int nMapMode)
virtual CPoint SetViewportOrg(CPoint point)
CPoint SetWindowOrg(CPoint point)
virtual CSize SetViewportExt(CSize size)
virtual CSize SetWindowExt(CSize size)
void DPtoLP(LPPOINT lpPoints, int nCount = 1) const
void LPtoDP(LPPOINT lpPoints, int nCount = 1) const
```

COLORREF SetPixel(int X, int Y, COLORREF Color)

Закрашивает пиксель области с координатами (X,Y), переданными в качестве первых двух аргументов цветом `Color` (третий аргумент функции). Возвращает цвет, который имел пиксель до вызова данной функции.

BOOL Rectangle(int upX, int upY, int lowX, int lowY)

Функция, которая рисует прямоугольник загруженным в контекст пером и заполняет его загруженной в контекст кистью. В качестве аргументов передаются координаты противоположных углов.

BOOL RoundRect(int upX, int upY, int lowX, int lowY, int curveX, int curveY)

Рисует и заполняет текущей кистью контекста прямоугольник со скругленными углами. Параметры `curveX` и `curveY` задают ширину и высоту эллипса, определяющего дугу для скругленных углов.

BOOL Ellipse(int upX, int upY, int lowX, int lowY)

Рисует и заполняет текущей кистью контекста эллипс, вписанный в прямоугольник, координаты углов которого передаются в функцию в качестве параметров.

CPoint MoveTo(CPoint point)

Перемещает фокус в точку point. Возвращает предыдущие координаты фокуса.

BOOL LineTo(POINT point)

Проводит линию пером, загруженным в контекст устройства, из фокуса в точку, переданную функции в качестве параметра.

6.2. Битовые образы

Битовые образы — очень важная часть Windows. При хранении битовых образов в отдельном файле обычно используется расширение BMP (это единственный растровый формат, который напрямую поддерживается Windows). Битовые образы могут храниться и в ресурсах. Битовые образы используются чаще, чем все остальные ресурсы. Это объясняется наличием для них чрезвычайно мощной поддержки. В Windows многие вещи, которые можно легко нарисовать программно, отображаются с помощью готовых битовых об-

разов. Например, кнопки в нажатом и отпущенном состоянии, каркасы для целых окон. Так как компьютеры теперь имеют большие жесткие диски, то выбор между программным рисованием объекта и готовой картинкой часто однозначно решается в пользу последней.

6.2.1. Создание битовых образов

В MFC битовые образы описываются классом CBitmap. Для их создания можно либо использовать ресурсный редактор, либо импортировать в ресурсы готовые файлы BMP, созданные при помощи графических пакетов. Битовый образ является таким же ресурсом, как иконка или диалог. Необходимо помнить, что область ресурсов с битовыми образами в EXE-файле может занимать большой размер. Но это не критично, так как ресурсы автоматически не загружаются в память.

6.2.2. Вывод битового образа на экран

Когда битовый образ помещен в ресурсы, его можно выводить на экран. Сначала необходимо создать объект типа `CBitmap` и с помощью функции LoadBitmap() загрузить в него битовый образ из ресурсов. Прототип функции:

```
BOOL CBitmap::LoadBitmap(LPCSTR ResourceName);
```

Параметр определяет строковый идентификатор ресурса. Битовый образ, после его загрузки, необходимо вывести в клиентскую область окна. Для этого обработчик `WM_PAINT` должен содержать приблизительно такой код (предполагается, что битовый образ загружен в объект `backgroundBitmap`):

```
CPaintDC clientDC(this);
CDC memDC; // Контекст памяти
// Создать совместимый контекст памяти
memDC.CreateCompatibleDC(&clientDC);
// Выбрать битовый образ в контекст устройства
memDC.SelectObject(&backgroundBitmap);
// Получить характеристики битового образа в структуру
//BITMAP
BITMAP bmp;
backgroundBitmap.GetBitmap(&bmp);
// Скопировать битовый образ из контекста памяти в
//контекст клиентской области
clientDC.BitBlt(0, 0, bmp.bmWidth, bmp.bmHeight, &memDC,
0, 0, SRCCOPY);
```

Сначала объявляются два контекста устройства. Первый связан с текущим окном. Второй не инициализирован и предназначен для области памяти, в которой будет храниться изображение. Затем, с помощью функции `CreateCompatibleDC(…)`, этот контекст объявляется совместимым с контекстом окна. Функция имеет прототип:

```
virtual BOOL CDC::CreateCompatibleDC(CDC *pDC);
```

Область памяти используется для вывода изображения на экран. Перед выводом на экран изображение должно быть выбрано в контекст устройства, связанный с областью памяти, с помощью функции `SelectObject(…)`. Мы используем ее вариант с прототипом:

```
CBitmap *CDC::SelectObject(CBitmap *pBmp);
```

Параметр `pBmp` — это указатель на объект битового образа. Для вывода изображения на экран используется функция `BitBlt(…)`, которая копирует изображение из исходного контекста устройства в контекст, связанный с вызывающим функцию объектом. Прототип функции такой:

```
BOOL CDC::BitBlt(int x, int y, int Width, int Height,
CDC *pSourceDC, int SourceX, int SourceY, DWORD RasterOp-
Code);
```

Первые два параметра задают координаты начальной точки изображения. Размеры изображения задают следующие два параметра. Параметр `pSourceDC` является указателем на исходный контекст устройства. Координаты `SourceX` и `SourceY` задают левый верхний угол изображения и обычно равны 0. Последний параметр задает код операции, которая будет проделана при передаче изображения из одного контекста в другой. Мы будем использовать только значение `SRCCOPY`, в этом случае изображение просто копируется. Существует также много других констант. Следует отметить, что указанным методом нельзя корректно выводить битовые образы более чем с 16 цветами в видеорежимах с 256 цветами. В режимах же HiColor и TrueColor без всяких проблем этим методом выводятся любые битовые образы. Так как на всех современных компьютерах используются, по крайней мере, режимы HiColor, мы не будем рассматривать ограничения худших режимов и манипуляции с палитрой. В библиотеке MFC, распространяющейся вместе с MS Visual Studio 2005 Professional, встроен класс `CImage`, который упрощает работу с битовыми образами. Принципы работы с ним рассмотрены далее.

6.3. Настройка системы координат

6.3.1. Стандартные функции каркаса MFC для настройки систем координат

В MFC встроены функции для настройки аппаратной и логической систем координат. Также предусмотрены функции перехода от одной к другой. Задача программиста состоит в том, чтобы определить, когда и какую систему координат использовать.

Основные правила при работе с системами координат:

- Все параметры, передаваемые в методы CDC, — это логические координаты.
- Все параметры, передаваемые в методы CWnd, — это аппаратные координаты.

- Значения, сохраняемые длительное время, должны использовать логические координаты.

Рассмотрим функции для работы с системами координат.

Функция `virtual int SetMapMode(int nMapMode)` устанавливает направления осей и определяет логические единицы. Возможные значения параметра `nMapMode` приведены в табл. 2.

Таблица 2

Значения параметра `nMapMode` функции `SetMapMode`

MM_TEXT	Одна логическая единица равна одному пикселю, ось x направлена вправо, ось y — вниз. Режим задан по умолчанию.
MM_HIENGLISH	Одна логическая единица равна 0.001 дюйма, ось x направлена вправо, ось y — вверх.
MM_HIMETRIC	Одна логическая единица равна 0.01 миллиметра, ось x направлена вправо, ось y — вверх.
MM_LOENGLISH	Одна логическая единица равна 0.01 дюйма, ось x направлена вправо, ось y — вверх.
MM_LOMETRIC	Одна логическая единица равна 0.1 миллиметра, ось x направлена вправо, ось y — вверх.
MM_ANISOTROPIC	Режим позволяет настраивать (с помощью функций SetWindowExt и SetViewportExt) размерность (отдельно для каждой из осей), их направления и начало отсчета
MM_ISOTROPIC	Режим позволяет настраивать (с помощью функций SeWindowExt и SetViewportExt) размерность осей, их направления и начало отсчета, единица оси x равна единице оси y.
MM_TWIPS	Одна логическая единица — твипс (twips) — равна 1/20 пункта (point) или 1/1440 дюйма, ось x направлена вправо, ось y — вверх.

Функции для перемещения центров систем координат: `virtual CPoint SetViewportOrg(CPoint point)` и `CPoint SetWindowOrg(CPoint point)` . Первая смещает центр аппаратных координат, а вторая — логических, в точку, переданную в качестве параметра. Обе функции возвращают координаты предыдущего центра.

Функции `virtual CSize SetViewportExt(CSize size)` и `virtual CSize SetWindowExt(CSize size)` используются для задания единиц измерения. Первая функция устанавливает единицы измерения аппаратной системы координат, вторая — логической.

Для перехода от аппаратных координат к логическим используется функция void `DPtoLP(LPPOINT lpPoints, int nCount = 1) const`, а для перехода от логических к аппаратным — `void LPtoDP(LPPOINT lpPoints, int nCount = 1) const`. Аргументами обоих функций являются указатель на массив с точками, которые нужно преобразовать, и размерность этого массива.

Эти функции универсальны, хорошо подходят для использования в задачах, где не требуются сложные преобразования системы координат. Для осуществления сложных многоуровневых преобразований рекомендуется вводить собственные функции преобразования системы координат.

Библиографический список

1. Давыдов В. Visual C++. Разработка Windows-приложений с помощью MFC и API-функций [Текст] / В. Давыдов. – Санкт-Петербург: BHV, 2008. – 567 с.

2. Довбуш Г., Хомоненко А. Visual C++ на примерах [Текст] / Г. Довбуш, А. Хомоненко. – Санкт-Петербург: BHV, 2007. – 528 с.

3. Рихтер Д., Назар К. Windows via C/C++. Программирование на языке Visual C++ [Текст] / Д. Рихтер, К. Назар. – Санкт-Петербург: Питер, 2009. – 896 с.

Учебное издание

ПРОГРАММИРОВАНИЕ ПОД WINDOWS В СРЕДЕ VISUAL C++

Составитель *Чагаева Ольга Леонидовна*

Редактор *Е. В. Рябая*

Подписано в печать 12.01.2010. Формат 60x84 1/16.
Бумага 80 г/м^2. Цифровая печать. Усл. п. л. 2,56.
Уч.-изд. л. 2,8. Тираж 100. Заказ

Редакционно-издательский отдел УГТУ – УПИ
620002, г. Екатеринбург, ул. Мира, 19
rio@fat.ustu.ru

Отпечатано в отделении полиграфии ИВТОБ
620002, Екатеринбург, ул. Мира, 19, ауд. И-120
Тел. (343) 375-41-43
opivtob@mail.ustu.ru