А.Мешков, Ю.Тихомиров

VISUAL C++ И MFC. ПРОГРАММИРОВАНИЕ ДЛЯ WINDOWS NT И WINDOWS 95

Том 1

Книга посвящена объектно-ориентированному программированию для 32-разрядных систем Windows с использованием компилятора Visual C++ и библиотеки классов MFC версии 4.1.

Книга содержит исчерпывающую информацию о библиотеке классов MFC (Microsoft Foundation Classes) и методах создания приложений Windows, а также большое количество наглядных и подробно прокомментированных примеров приложений, полные тексты которых приведены на сопроводительной дискете. Первый том посвящен описанию возможностей системы Windows, структуре приложений Windows, использованию функций Win32 API, многозадачности и графическому выводу. Проведен обзор всех классов библиотеки MFC версии 4.1, рассмотрены принципы создания SDI и MDI приложений, а также все окна, элементы управления и панели элементов управления, составляющие неотъемлемую часть современного программного продукта. Приложения и предметный указатель содержат все классы и функции.

Для программистов, владеющих языками C и C++ и имеющих опыт программирования в 16-разрядной системе Windows

Содержание

Введение	2
ЧАСТЬ I. ОБЗОР WINDOWS 95 И WINDOWS NT	7
Глава 1. Основные черты	**9**
Требования к операционной системе 90-х	9
Различные версии Windows	10
Windows 95	12
Windows NT	13
Интерфейс прикладного программирования	15
Система, управляемая сообщениями	16
SDI и MDI	18
Глава 2. Пользовательский интерфейс	**19**
Окна, элементы управления, модальные и немодальные блоки диалога	19
Ресурсы приложения	25
Глава 3. Управление графическим выводом	**27**
Идеология графического вывода	27
Графические объекты	27
Аппаратно-независимый графический вывод	28
Контексты устройств	28
Графические режимы	29
Типы контекстов устройств	29
Экран	30

Принтер 31
Объект в памяти 31
Информационный контекст 32
Работа со шрифтами. Шрифты True Type 32
ЧАСТЬ II. ПРОГРАММИРОВАНИЕ ПОД MS WINDOWS 35
Глава 4. Интерфейс прикладного программирования Win32 API 37
Отличие 16- и 32-разрядных платформ 37
Функция WinMain 39
Новые возможности для создания программ 41
APIShell — пример приложения 43
Графический вывод 50
Сообщение WM_PAINT 51
Доступ к графическим объектам, их замена и удаление 56
Глава 5. Вытесняющая многозадачность 59
Процессы и потоки 63
Классы процессов и приоритеты потоков 66
Класс Idle 66
Класс Normal 66
Класс High 67
Класс Realtime 67
Относительные приоритеты потоков 67
Взаимодействие программ 70
Синхронизация потоков 71
Критические секции 71
Защищенный доступ к переменным 73
Ожидающие функции 74
Объекты синхронизации 75
API Sync — пример приложения 80
Глава 6. Структура приложения Windows 88
Стандартный цикл обработки сообщений 90
Оконная процедура 91
Основные типы оконных сообщений Windows 92
Аппаратные сообщения 93
Сообщения обслуживания окна 95
Сообщения об организации интерфейса пользователя 97
Сообщения о завершении работы приложения и системы 98
Частные сообщения 99
Информационные сообщения системных ресурсов 99
Сообщения об обмене данными 100
Внутрисистемные сообщения 101
APIWin — пример приложения 101
ЧАСТЬ III. БИБЛИОТЕКА КЛАССОВ MICROSOFT FOUNDATION CLASSES (MFC) 111

Глава 7. Обзор **113**
Иерархия классов MFC 114
CObject — основа всех классов 116
Сериализация 120
Классы, определяющие архитектуру приложения 126
Классы приложения и поддержки потоков 127
Классы маршрутизации команд 128
Классы документов 128
Классы шаблонов документов 129
Окна, блоки диалога и элементы управления 129
Классы фреймов окон 130
Классы представлений 130
Классы блоков диалога 132
Классы элементов управления 133
Классы панелей элементов управления 135
Классы для вывода на экран и печать 136
Классы контекста устройств 136
Классы графических объектов 136
Классы простых типов данных 137
Массивы, списки и ассоциативные списки 138
Классы шаблонов массивов, списков и ассоциативных списков 138
Готовые к использованию классы массивов 139
Готовые к использованию классы списков 139
Готовые к использованию классы ассоциативных списков 139
Классы для работы с файлами и базами данных 139
Классы файлов ввода/вывода 140
Классы для работы с DAO 140
Классы для работы с ODBC 141
Классы для работы в Internet и сетях 142
Классы ISAPI 142
Классы сокетов Windows 143
Классы для работы с OLE 143
Контейнерные классы OLE 143
Серверные классы OLE 144
Классы OLE для передачи данных и drag-and-drop 144
Классы автоматизации OLE 145
Классы, связанные с OLE 145
Классы для отладки и обработки исключений 146
Классы поддержки отладки 146
Классы обработки исключений 146
Глава 8. Макросы, глобальные функции и переменные **148**
Типы данных 148
Получение информации о приложении 149

Модель объекта времени выполнения 150
Диагностика объектов 151
Основные макросы 151
Основные глобальные переменные 152
Форматирование строк и окно сообщения 152
ЧАСТЬ IV. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ КЛАССОВ MFC 157
Глава 9. Основные составляющие приложения на базе библиотеки классов MFC 159
Соглашения об именах MFC 163
Включаемые файлы 163
Функция WinMain 164
Класс CWinApp 166
Минимальная программа для Windows 173
Регистрация класса окна 175
Создание окна 180
Глава 10. Обработка сообщений в библиотеке классов MFC 182
Цикл обработки сообщений MFC, 182
Категории сообщений 184
Карта сообщений 185
Компоненты карты сообщений 187
Класс CCmdTarget 191
Стандартный маршрут команды 192
Команды обновления и класс CCmdUI 194
Функции для работы с сообщениями 196
ЧАСТЬ V. СОЗДАНИЕ SDI И MDI ПРИЛОЖЕНИЙ НА БАЗЕ БИБЛИОТЕКИ MFC 199
Глава 11. Классы окон библиотеки MFC 201
Окна, определенные в системе Windows 202
Класс CWnd 207
Класс CFrameWnd 209
Создание окна и библиотека MFC 211
Создание главного окна SDI-приложения 224
Создание дочерних окон 230
Ограничение размеров окна 237
Функция GetSystemMetrics 238
Ограничение доступа к окну 240
Глава 12. Создание многодокументных приложений 241
Класс CMDIFrameWnd 242
Класс CMDIChildWnd 244
MDI — пример приложения 246
Обработка команд меню 262
UNICODE 264

Обработка сообщения WM_ PAINT 268
Глава 13. Панели элементов управления 272
Класс CControlBar 272
Класс CToolBar 277
Класс CToolBarCtrl 281
Класс CStatusBar 289
Класс CStatusBarCtrl 292
Класс CDialogBar 295
ControlBars — пример приложения 297
ЧАСТЬ VI. БЛОКИ ДИАЛОГА 311
Глава 14. Модальные и немодальные блоки диалога 313
Класс CDialog 316
Template — пример приложения 328
Блок диалога в качестве главного окна приложения 341
Обмен данными с блоком диалога 342
Создание блока диалога на основе шаблона в памяти 346
Глава 15. Стандартные блоки диалога 350
Класс CFileDialog 351
Класс CColorDialog 363
Класс CFontDialog 369
Класс CFindReplaceDialog 375
Класс CPrintDialog 379
Класс CPageSetupDialog 389
Заключение 396
ПРИЛОЖЕНИЯ 397
Приложение 1. Иерархия классов MFC 4.1 399
Приложение 2. Список функций класса CWnd 402
Атрибуты 402
Конструкторы и создание окон 402
Доступ к дереву окон 404
Общие функции работы с сообщениями 405
Обработка сообщений модального диалога 405
Функции работы с текстом окна 405
Функции меню (кроме дочерних окон) 406
Функции изменения размеров и расположения окна 406
Функции преобразования координат 407
Функции обновления и рисования 407
Функции использования таймера 409
Функции для работы со всплывающими подсказками 409
Функции состояния окна 409
Функции для работы с блоками диалога 410
Функции прокрутки 411
Функции доступа к окну 412

Функции обслуживания 413
Функции работы с буфером обмена (Clipboard) 413
Функции каретки 414
Функция взаимодействия 414
Функции работы с пиктограммами 414
Функции контекстно-зависимой справки 415
Функция поддержки диалога 415
Функции расположения и другие 415
Функции управления OLE 415
Функции обработки сообщений управления окнами 416
Функции обработки сообщений нерабочей области окна 418
Функции обработки системных сообщений 419
Функции обработки входных сообщений 421
Функции обработки инициализирующих сообщений меню 422
Функции обработки сообщений буфера обмена 422
Функции обработки управляющих сообщений 423
Функция обработки сообщений MDI 424
Функции обработки сообщений цикла меню 424
Функции обработки основных сообщений 424
Переопределяемые и другие вспомогательные функции 425
Приложение 3. Типы проектов и мастера 427
Типы проектов Visual C++ 427
Мастера (Wizards) 428
AppWizard 429
ClassWizard 435
Wizard Bar 438
Приложение 4. Настройки проекта 439
Настройка опций компилятора 439
Настройка опций компоновщика 442
Приложение 5. Значения параметра функции GetSystemMetrics 444
Предметный указатель 448

Предметный указатель

A
AfxGetApp 165
AfxGetInstanceHandle 167
AfxMesageBox 172
AfxRegisterClass 175
AfxRegistcrWndClass 175
AttachThreadInput 65
B
BeginPaint 51
C
caret см. каретка
CCdmUl
SetText 295
CClientDC 270
CCmdTargct
BeginWaitCursor 172, 191
EndWaitCursorl72, 191
OnCmdMsg 192
RestoreWaitCursor 172, 191
CCmdUl 195,277
ContinueRouting 196
Enable 196,277,290,295

m_nID 195
m_nIndex 195
m_pMenu 195
m_pOther 195
m_pSubMenu 195
SetCheck 196,267.277
SetRadio 196,277
SetText 196, 290
CColorDialog 350
CColorDialog 365
DoModal 365
GetColor 365
GetSavedCustomColors 365
m_cc 363
OnColorOK 365
SetCurrentColor 365
CCommandLineInfo 170
CCommonDialog
CCommonDialog 350
CControlBar 169,272
CalcFixedLayout 273
EnableDocking 275
GetBarStyle 273
GetCount 273
GetDockingFrame 275
IsFloating275
m_bAutoDclete 273
SetBarStyle 273
CCreateContex 225
CDataExchanae
Fail 345
m_bSaveAndValidate 345
m_pDlgWnd 345
PrepareCtrl 345
PrepareEditCtrl 345
CDC
DrawText 269
TextOut 270
CDialog 169,206
CDialog 316
Create 320
CreateIndirect 320
DoModal 316
EndDialog321
GetDenD 323
GotoDlgCtrl 323
InitModalIndirect 320
MapDialogRect 322
NextDlgCtrl 323
OnCancel 322
OnInitDialog 321
OnOK 322
OnSetFont 322
PrevDIgCtrl 323
SetDefID 323
CDialogBar 169,272,296
CDialogBar 296
Create 296
CFileDialog 350
CFileDialog 357
DoModal 357
GetFileExt 357
GctFileName 357
GetFileTitle 357
GetNextPathName 358
GetPathName 357
GetReadOnlyPref358
GetStartPosition 358
m_ofh 351
OnFileNameChange 358
OnFileNameOK 358
OnFolderChange 358
OnInitDone 358
OnLBSclChangedNotify 358
OnShareViolation 358
OnTypeChange 358
CFindRcplace
FindNext 378
CFindReplaceDialog 350
CFindReplaceDialog 378
Create 378
GetFindString 379
GetNotifier379
GetReplaceSiring 379
IsTerminating 379
m_fr 376

MatchCase 379
MatchWholeWord 379
ReplaceAll 379
ReplaceCurrent 379
SearchDown 379
CFontDialog 350
CFontDialog 373
DoModal 373
GetColor 373
GetCurrentFont 373
GetFaceName 373
GetSize 373
GctStyleName 373
GetWeight 373
Isltalic 373
IsStrikeOut 373
IsUnderLine 373
m_cf 369
CFormView 169
CFrameWnd 206
Create 210, 226
DockControlBar 275
EnableDocking 275
FloatControlBar 276
GetDockState211
LoadAccelTabie 210
LoadBarState210,275
LoadFrame210,225
m_bAutoMenuEnable 210
RecalcLayout 276
SaveBarState210,275
SetDockState211
ShowControlBar211,275
CHOOSECOLOR 363
CHOOSEFONT 369
CloseHandle 110
CMDIChildWnd 206
Create 245
GetMDIFrame 245
m_hMenuShared 245
MDIActivate 245
MDI Destroy 245
MDIMaximize 245
MDIRestore 245
CMDIFrameWnd 206
CreateClient 244
GetWindowMenuPopup 244
MDIActivate 243
MDICascade 244
MDIGetActive 243
MDlIconArrange 243
MDIMaximize 243
MDINext 243
MDIRestore 243
MDISetMenu 243
MDITile 244
CObject 116
COleResizerBar 272
CommandLineToArgvW 65
CPageSetupDialog 350
CPageSetupDialog 393
CreatePrinterDC 393
DoModal 393
GetDeviceName 394
GetDevMode 394
GetDriverName 394
GetMargins 394
GetPaperSize 394
GetPortName 394
OnDrawPage 394
PreDrawPagc 395
CPaintDC 268
CPoint 270
CPrintDialog 350, 388
CPrintDialog 388
CreatePrinterDC 388
DoModal 388
GetCopies 389
GetDefaults 388
GetDeviceName 388
GetDevMode 388
GetDriverName 388
GetFromPage 389
GetPortName 388
GetPrinterDC 388
GetToPage 389

m_pd 380, 389
PrintAll 389
PrintCollate 389
PrintRange 389
PrintSelection 389
CPropertyPage 169
CPropertySheet 169
CreateEvent 78
CreateFile 110
CreateMutex 76
CreateProcess 63
CreateRemoteThread 65
CreateSemaphore 79
CREATESTRUCT 227
CreateThread 64
CRect 269
CRITICAL_SECTION 72
CRuntimeClass 116
CSize 270
CStatusBar 272
Create 290
CStatusBar 290
GetPaneInfo291
GetPaneStyle291
GetPaneText 292
GetStatusBarCtrl 292
Setlndicators 291
SetPaneInfo 291
SetPaneStyle291
SetPaneText 292
CStatusBarCtrl 292
DrawItem 294
GetBorders 294
GetParts 293
GetRect 294
GetText 292
GetTextLenght 292
SetMinHeight 294
SetParts 293
SetSimple 294
SetText 293
CTollBarCtrl
DeleteButton 285
CTollBatCtrl
Customize 284
CToolBar 169, 272, 277
CommandToIndex 280
Create 278
CToolBar 277
GetButtonInfo 280
GetButtonStyle 280
GetButtonText281
GetItemID 280
GetItemRect 280
GetToolBarCtrl281
LoadBitmap 279
LoadToolBar 278
SetBitmap 279
SetButtonInfo281
SetButtons 279
SetButtonStyle 280
SetButtonText281
SetHeight 279
SetSizes 279
CToolBarCtrl 281
AddBitmap 284
AddButtons 285
AddString 284
AddStrings 284
AutoSize 285
GetBitmapFlags 284
GetButton 283
GetButtonCount 283
GetRows 284
InsertButton 285
RestoreState 285
SaveState 285
SetOwner 283
SetRows 284
CView 206
CWaitCursor 172
CWinApp 166, 168
DoMessageBox 172
DoWaitCursorl72
Enable3dControls 168
Enable3dControlsStatic 168

EnableShellOpen 169
ExitInstance 172
GetProfileBinary 171
GetProfileInt 170
GetProfileStringl70
HideApplication 172
InitInstance 171
LoadCursor 169
LoadIcon 170
LoadOEMCursor 169
LoadOEMIcon 170
LoadStandardCursor 169
LoadSlandardIcon 170
LoadStdProfileSettings 168
m_hInstancc 167
m_hPrevInstance 167
m_lpCmdLine 167
m_nCmdShow 167
m_pszAppName 167
m_pszExcNamc 167
m_psz.HelpFilePath 167
m_pszPronieName 167
m_pszRegistryKey 167
Onldle 172
ParseCommandLine 170
ProccssShellCommand 170
RegisterShellFileTypes 169
Run 172
SelectPrinter 393
SetDialogBkColoi-168
SetRegisiryKey 169
WriteProrileBinary 171
WriteProfileInt 170
WriteProt'lleString 170
CWinThread 166
m_pMainWnd 167,210
CWinTread
PumpMessage 183
CWnd 184,206
m_hWnd 207
ArrangeIconicWindows 208
Attach 207
BcginPaint 208, 268
BringWindowToTop 208, 261
CenierWindow 208
Create 207
CreateEx 207. 226
Default 196
Detach 207
DoDataExchange 344
EnableWindow 207, 240
EndPaint 208. 268
FindWindow 208, 260
FromHandle 207
FromHandlePermanent 207
GetActiveWnd 207
GetCapture 207
GetClienlRect 208, 269
GetCurrentMessage 196
GeiDC 208
GetDlgItem 323
GetExStyle 207. 236
GetFocus 207
GetFont 208
GctForegroundWindow 207, 261
Getlcon 208
GetParent 245
GetSafeWnd 207, 235
GetStyle 207. 236
GetWindowDC 208
GetWindowPlacement 208
GetWindowRect 208
GetWindowText 208
GetWindowTextLenght 208
Invalidate 208, 267
IsIconic 208, 260
IsWindowEnable 207, 240
IsZoomed 208
ModifyStyle 208, 236
ModifyStyleEx 236
MoveWindow 208, 233
PostMessage 197
PreCreateWindow 207
PreSubclassWindow 207
PreTranslateMessage 196
Print 208

PrinlClient 208
ReleaseDC 208
SendMessage 197
SendNotifyMessage 197
SetActiveWnd 207
SetCapture 207
SetFocus 207
SetFont 208
SetForegroundWindow 207, 261
Setlcon 208
SetWindowPlacement 208
SetWindowPos 208, 228
SetWindowText 208
ShowWindow 208
SubclassWindow 207
UpdateData 343
UpdateWindow 208
D
DeleteCriticalSection 72
desktop window Ñì. Îêíî ñèñòåìû ãëàâíîå
DEVMODE 385
DEVNAMES 387
DispatchMessage 90, 172, 196
DLGITEMTEMPLATE 318
DLGTEMPLATE317
E
EndPaint 51
EnterCriticalSection 72
EnumFontFamilies 56
EnumObject 56
ExcludeUpdateRgn 54
ExitProcess 64
ExitThread 64
F
FIFO 62
FINDREPLACE 376
FindWindow 40
FreeEnvironmentStrings 65
G
GDI Ñì. èíòåðôåéñ óñòðîéñòâà
ãðàôè÷åñêèé
GetClassLong 234
GetCommandLine 65
GetCurrentObject 56
GetCurrentProcess 65
GetCurrentProcessId 65
GetCurrentThread 65
GetCurrentThreadId 65
GetEnvironmentVariable 65
GetExitCodeProcess 65
GetExitCodeThread 65
GetFileSize 110
GetMessage 62, 90, 183, 197
GetObject 56
GetOpenFileName 110
GetProcess Version 65
GetSaveFileName 110
GetStartupInfo 65
GetSystemMetrics 238
GetUpdateRect 52
GetUpdateRgn 52
GetWindowLong 234
I
InitializeCriticalSection 72
InterlockedDecrement 73
InterlockedExchange 73
Interlockedlncrement 73
InvalidateRect 52
InvalidateRgn 52
IsIconic 41
L
LeaveCriticalSection 72
M
MDI Ñì. èíòåðôåéñ ìíîãîäîêóìåíòíûé
MINMAXINFO 238
MSG90
MsgWaitForMultipleObjects 74,75
O
OpcnEvent 78
OPENFILENAME 352
OpenGL43
OpenMutex 77
OpenProcess 65
OpenSemaphore 79
P
PAGESETUPDLG 390

PAINTSTRUCT 51
PeekMessage 60, 183, 197
PostMessagc 18
PreTranslateMcssage 172
PRINTDLG 380
process см.процесс
PulseEvent 78
R
ReadFile 110
RECI 269
RedrawWindow 55
RegisierWindowMessage 191, 197
ReleaseMutex 77
ReleaseSemaphore 80
ResetEvent 78
ResumeThread 65
RGB 168
S
SDI см.интерфейс однодокументный
SelectObject 57
SendMessage 18
SetClassLong 233
SetEndOfFile 110
SetEvent 78
SetForegroundWindow 41
SetPriorityClass 66
SetThreadPriority 67
SetWindowLong 234
ShowWindow 41
SuspendThread 65
T
TerminateProcess 64
TerminateThread 64
thread См. поток
TranslaleMessage 90, 172, 196
U
UpdateWindow 55
V
ValidateRect 52
ValidateRgn 52
W
WaitCommEvent 75
WaitForMultipleObjects 74
WaitForMultipleObjectsEx 74
WaitForSingleObjecl 74
WaitForSingleObjectEx 74
WaitForSingleObjectsEx 74
WinExec 65
WinMain39, 165
WM_CHAR 90
WM_ERASEBKGND 51
WVLNCPAINT 51
WM_PAINT 50
WM_QU1T 90
WM_TIMER 60
WNDCLASS 175
WNDCLASSEX 175
WriteFile 110
А
Активное ожидание 62
Б
Базовый уровень приоритета 69
В
Взаимоисключение 76
Д
Десинхронизация ввода 17
И
Интерфейс
  многодокументный 18
  однодокументный 18
  устройства графический 28
К
Каретка 51
Класс процессов
  High 67
  Idle 66
  Normal 66
  Realtime 67
М
Модуль 61
О
Окно системы главное 55
П
Перегрузка 63
Поток 61
Процесс 61

С
Семафор 79
Событие 77
Состояние
  блокировки 61
  выполнения 61
  готовности 61
Структура
  NMHDR 287
  TBNOTIFY 288

Т
Тупик 74

Идея написания этой книги возникла после первоначального знакомства с Visual C++, и конечно же, с библиотекой классов MFC — удобными и мощными средствами создания приложений для Windows. Со временем благоприятное впечатление об этих средствах только усилилось. Надеемся, что наша книга поможет вам избежать многих трудностей, с которыми так или иначе приходится сталкиваться при изучении тех возможностей, которые предоставляют Win32 API, Visual C++ и библиотека классов MFC

Введение

Windows... Пожалуй, сложно сейчас найти человека, имеющего хоть какое-то отношение к компьютерам, который бы не знал, что означает это слово. Как и у любого другого неординарного продукта, у Windows имеются и свои горячие сторонники, и не менее непримиримые критики. Но, хотим мы этого или нет, в настоящее время (и на ближайшее будущее) именно эта операционная система определяет требования к программным продуктам для персональных компьютеров. И если вы хотите привлечь к себе (или удержать) внимание на современном рынке программного обеспечения, то не можете оставить в стороне операционную систему Microsoft Windows.

Итак, программирование в Windows. Какой бы язык программирования вы не выбрали, очевидно, что базой является Windows API (Application Programming Interface — интерфейс прикладного программирования), и освоив[1] его, можно писать программы любой степени сложности, используя все возможности, предоставляемые операционной системой. Но... Сложность современных приложений поднялась на такой уровень, что разработка коммерческого программного обеспечения с использованием только Windows API и языка C уже не может удовлетворить программиста. Естественно, что технология программирования "не могла остаться в стороне", и в настоящее время предлагается два подхода к построению сложного программного обеспечения. С одной стороны, это использование систем визуального программирования. А с другой — применение библиотек классов, которые самостоятельно выполняют массу черновой работы, и при этом гораздо в большей степени, нежели система Windows сама по себе, ведут программиста, структурируя и облегчая процесс создания программного обеспечения. Не будем заниматься сравнением достоинств и недостатков — каждый вправе сам решить, что ему больше подходит. Скажем только, что основная проблема, с которой сталкивается программист, начиная работать в некоторой операционной системе, состоит в понимании фундаментальных принципов и моделей, включенных в ее архитектуру. И

[1] С каждой новой версией Windows увеличивается количество функций API. К настоящему моменту оно уже достигло той точки (в Win32 их больше 2000), когда даже опытные программисты не в состоянии их все запомнить.

только проникнувшись "образом мышления Windows", начинаешь понимать, что программирование под Windows не сложнее любого другого. А какой из подходов больше "прячет" эти принципы и модели, пусть каждый решает сам.

На наш взгляд, отдав предпочтение компилятору компании Microsoft Visual C++, вы получите в свое распоряжение очень удобную, мощную и гибкую систему разработки приложений для Windows. Сердцевиной Visual C++, во многом определяющей его мощь и гибкость, является библиотека классов MFC (Microsoft Foundation Classes — базовые классы Microsoft). Популярность этой библиотеки настолько высока, что не только такие известные фирмы как Watcom и Symantec приобрели на нее лицензии, но и практически единственная широко известная у нас фирма Borland, поставляющая компиляторы C++ с собственной библиотекой классов (OWL), включила в версию 5.0 возможность трансляции приложений, созданных с использованием библиотеки классов MFC.

Несмотря на то что сама библиотека классов сложна и многогранна, и на сегодняшний день существует более десяти ее версий, для начала работы с ней вполне достаточно владения языком программирования C++. Новичкам даже не потребуются знания о принципах программирования в Windows. Дело в том, что в компилятор Visual C++ включено специальное средство, позволяющее автоматизировать процесс создания приложения. Это средство — MFC AppWizard — создаст для вас работающее приложение. Конечно, это будет только шаблон, и наполнять его конкретным содержанием придется самостоятельно. Но первоначальное знакомство со сгенерированным кодом принесет новичкам значительную пользу.

Однако мы не ставили своей целью написать традиционный учебник по программированию в Windows — об этом написано (и пишется) достаточно много хороших (и не очень) книг. Основным объектом нашего внимания является библиотека классов MFC. И именно *детальному* знакомству с ней посвящена основная часть книги. Но, с другой стороны, классы библиотеки настолько тесно связаны с системой Windows, а многие непосредственно инкапсулируют ее объекты, что мы не могли не описать саму систему Windows, предоставляемые ею возможности и те механизмы, без понимания которых создание Windows-приложений попросту невозможно. Тем не менее, мы отдаем себе отчет в том, что эти разделы не могут заменить литературу, посвященную программированию в Windows с использованием функций API.

Должны сразу оговориться, что в одной книге невозможно осветить все аспекты программирования в Windows — это слишком обширная тема. И одной из проблем при выборе материала является чувство меры — очень трудно определить то место, где следует остановиться. Тем не менее, в предлагаемую книгу мы постарались включить достаточное количество информации, чтобы не только помочь начать писать собственные приложения для Windows, но и предоставить опытным программистам разнообраз-

ный материал не только по стандартным и уже не раз описанным в литературе (правда, на базе функций Windows API) возможностям, но и по тем "изюминкам", которые до сих пор нигде не описаны, и были найдены авторами как в процессе работы, так и при написании книги.

Мы использовали библиотеку MFC версии 4.1, предназначенной для 32-разрядных операционных систем Windows 95 и Windows NT,[1] и включенную в соответствующую версию Visual C++.

Материала по рассматриваемой теме настолько много, что для раскрытия большинства возможностей библиотеки MFC и ее места в программировании для Windows, мы разбили его на три тома. Следует учитывать также, что изложение материала, в силу специфики Windows, просто не может быть последовательным. Пришлось достаточно много времени посвятить поиску оптимальной, на наш взгляд, структуры изложения. Однако при прочтении вы не единожды столкнетесь со ссылками на другие разделы, разбросанные по всему тексту. За основу мы взяли три типа стандартных приложений Windows: приложения, поддерживающие SDI (Single Document Interface — однодокументный интерфейс) и MDI (Multiple Document Interface — многодокументный интерфейс), а также приложение на базе блока диалога.

В первый том, посвященный знакомству с основными классами библиотеки MFC и объектами Windows, мы включили следующие части:

- Первые две части (главы с 1 по 6) посвящены краткому введению в Windows, отличиям различных версий системы, 32-разрядному интерфейсу Win32 API, многозадачности (глава 5) и графическому выводу (глава 6). Главы содержат примеры приложений, написанных без использования библиотеки классов MFC, и знакомят читателя с общими требованиями и структурой приложений Windows.
- В части III приведен общий обзор классов библиотеки MFC, подробно описаны базовый класс библиотеки — *CObject* и процесс сериализации — мощный и универсальный механизм сохранения/восстановления данных на носителе (глава 7). В главе 8 представлены глобальные библиотечные функции и макросы, без которых не обходится ни одно приложение, написанное с использованием даже базовых возможностей библиотеки.
- Часть IV содержит описание требований к структуре и составляющим каждого приложения, созданного с использованием библиотеки классов MFC, а также рассматривается класс приложения *CWinApp* (глава 9). Глава 10 посвящена подробному изложению одного из основных механизмов Windows — работы с сообщениями.

---

[1] Каждая новая версия является расширением предыдущей, и материал, представленный в книге, в большинстве случаев справедлив и для других версий библиотеки. Однако, следует учитывать, что в более ранних версиях может не оказаться некоторых классов и возможностей. Все специфические особенности рассматриваемых версий сопровождаются в тексте пометками "только для Windows 95 и/или Windows NT".

Эти четыре части представляют собой некоторое теоретическое введение в Windows и библиотеку MFC, но при этом не забыты и практические аспекты программирования.

- Часть V посвящена основным объектам Windows — окнам и созданию на их основе однодокументных и многодокументных приложений. В главе 11 подробно рассмотрен базовый класс для всех типов и видов окон — класс *CWnd*, а также механизм создания однодокументных приложений. Глава 12 целиком посвящена описанию процесса создания многодокументных приложений. Глава 13 знакомит читателя с мощными средствами создания и использования панелей элементов управления, таких как панель инструментов и строка состояния.
- Часть VI полностью охватывает третий тип стандартных приложений Windows — приложения на базе блока диалога. В главе 14 рассматриваются основные принципы создания и работы с блоками диалога и класс *CDialog*, предназначенный для решения этих задач. Системные (или стандартные) блоки диалога описываются в главе 15. Их использование позволяет настраивать практически все параметры (открытие и сохранение файла, выбор шрифта и цвета, поиск и т. д.), которые могут понадобиться создателям приложений.

После прочтения первого тома книги вы сможете создавать практически любые приложения Windows, используя классы библиотеки MFC. Однако, для того чтобы сделать программирование не только удобным, но и чрезвычайно привлекательным, знакомства только с "основными" окнами недостаточно. Желательно еще уметь манипулировать многими другими элементами Windows, такими как GDI (Graphic Device Interface — интерфейс графических устройств) и его объекты, элементы управления и т. д. Все это вы найдете во втором томе:

- В части VII подробно описаны возможности графического вывода, предоставляемые классами библиотеки (основные положения представлены в главе 16). Глава 17 посвящена подробному описанию классов контекстов устройств (*CDC*, *CClientDC*, *CPaintDC* и т. д.), отвечающих в библиотеке за весь графический вывод, а глава 18 — основным объектам GDI, таким как карандаши, кисти, шрифты и т. п. Приводится большое количество примеров.
- Части VIII и IX знакомят с классами элементов управления, использование которых существенно упрощает разработку приложений. Сюда входит описание как стандартных (списки, кнопки и т. д.), так и дополнительных элементов управления.
- Часть X познакомит вас с мощным средством библиотеки MFC — архитектурой документ/представление (document/view). Она не имеет аналогов в самой системе Windows, но является настолько удобной, что мы настоятельно рекомендуем разобраться в ней. Общая структура и классы, входящие в нее, рассматриваются в главе 26. В главе 27 мы под-

робно рассматриваем два ключевых компонента этой архитектуры — классы *CView* и *CDocument*, их взаимодействие между собой, а также тесно с ними связанный класс *CSplitterWindow*. В главе 28 рассказывается о том, как организовать в приложении процессы печати и предварительного просмотра документа.

- В заключительной части XI описывается поддержка библиотекой классов MFC ресурсов Windows, таких как меню, акселераторы и т. д.

Заключительный, третий том книги предназначен, в основном, для достаточно опытных программистов, уже знакомых с основами, рассмотренными нами в предыдущих томах. Мы включили в него следующие темы:

- создание и работа с блокнотом свойств — классы *CPropertySheet* и *CPropertyPage*, а также завершение знакомства с дополнительными элементами управления — расширенный текстовый редактор (класс *CRichEdit*);
- многозадачность и создание многопотоковых приложений на основе классов библиотеки MFC, их взаимодействие и синхронизация, а также другие связанные с данной темой вопросы;
- механизм структурной обработки исключений и его поддержка в MFC — класс *CException* и производные от него;
- программирование трехмерной графики с использованием интересной библиотеки OpenGL;
- создание справочной системы и дистрибутива вашего приложения.

Прилагаемая к текущему тому дискета содержит полные исходные тексты и исполняемые файлы всех рассматриваемых примеров. Мы надеемся, что они будут полезны читателям.

В заключение хотелось бы сказать несколько слов в адрес тех людей, без которых было бы намного труднее закончить эту книгу. Прежде всего, это относится к нашим семьям, которые вынуждены были считаться с тем, что довольно долго мы все свободное время посвящали работе над книгой. Мы выражаем благодарность редактору Татьяне Темкиной — судя по тому как сложно читать, править и исправлять даже написанное самим собой, ее работа не менее трудна, чем написание собственного текста. Мы также благодарим всех сотрудников издательства "BHV — Санкт-Петербург", принимавших участие в подготовке рукописи к печати: Ольгу Сергиенко (компьютерная верстка), Зинаиду Дмитриеву (корректура), Николая Тверских (производство) и Елизавету Кароник (общее руководство).

Все пожелания и замечания по книге можно присылать по адресу:

195009, С.-Петербург, Бобруйская ул., 4
Издательство "BHV — Санкт-Петербург"

Часть I

Обзор Windows 95 и Windows NT

1. Основные черты
2. Пользовательский интерфейс
3. Управление графическим выводом

Глава 1

Основные черты

Сегодня уже трудно вспомнить то время, когда не было Windows, и, как бы программисты не относились к корпорации Microsoft и ее продуктам, приходится признать, что Windows — это настоящее, а может быть и будущее компьютерной индустрии.

Как известно, успех компьютера и операционной системы определяется количеством и разнообразием программного обеспечения, созданного для них. И к 1990 году, когда была достигнута критическая масса такого обеспечения для персональных компьютеров (не в последнюю очередь за счет программ самой фирмы Microsoft), как нельзя кстати оказалось появление новой версии Windows — 3.0. И именно с этого времени начинается победное шествие по миру операционной оболочки, а затем и полноценной системы Windows.

Об этом написано достаточно много и у нас и за рубежом, поэтому мы остановимся только на основных чертах, присущих двум последним, 32-разрядным операционным системам Windows 95 и Windows NT.

Чтобы оценить возможности и достоинства Windows, необходимо, прежде всего, перечислить требования к операционным системам, которые сформировались в результате создания и использования нескольких десятков операционных систем для компьютеров всех классов и поколений за последние 40 лет. Да, за это время были созданы именно десятки, если не сотни систем, и хотя сейчас лидерство по количеству проданных и установленных копий безусловно принадлежит Windows, кто сказал, что к лидеру должны предъявляться менее жесткие требования, чем к другим?

Требования к операционной системе 90-х

Коротко говоря, современная операционная система для персональных компьютеров, которые по своим возможностям сравнимы с рабочими станциями и даже мини-компьютерами прошлого десятилетия, должна реализовывать то, что было создано за последние десятилетия, поддерживать самые современные технологии и стандарты, как можно более полно и

эффективно использовать аппаратные средства, и при этом быть совместимой с накопленным прикладным программным обеспечением.

Итак, вот требования к операционной системе, наиболее важные на наш взгляд сегодня:

- 32-разрядный код системы и интерфейс программирования;
- совместимость с существующим прикладным программным обеспечением;
- интуитивно понятный графический интерфейс;
- максимальная поддержка пользователей при ее установке и использовании;
- мощные графические возможности;
- 32-разрядная поддержка файловой системы, которая должна обеспечивать защиту файлов и надежность их хранения;
- устойчивость к сбоям;
- защита ядра операционной системы;
- эффективная и прозрачная для приложений поддержка виртуальной памяти;
- эффективное управление программными ресурсами;
- разделение памяти между процессами, работающими в различных виртуальных адресных пространствах;
- вытесняющая многозадачность и эффективная приоритетная диспетчеризация процессов;
- возможность работы в режимах разделения времени и реального времени;
- средства разграничения доступа;
- поддержка различных сетей и возможности связи с глобальными информационными сетями и системами;
- поддержка периферийных устройств различных фирм, число которых постоянно растет;
- эффективное управление аппаратными ресурсами;
- поддержка стандарта самонастраивающихся устройств (Plug and Play);
- встроенная поддержка современных технологий обработки, хранения и передачи данных.

В следующем разделе мы рассмотрим, что же реализовано в Windows 95 и Windows NT.

Различные версии Windows

Уже можно смело сказать, что время Windows 3.x прошло, хотя в валютных обменных пунктах еще до сих пор используют даже программы для MS-DOS. Поэтому не будем останавливаться на особенностях этой системы. Отметим лишь то, что объединяет все версии Windows.

Изначально разработчики системы стремились обеспечить как можно более полную независимость программ от аппаратуры. С каждой новой версией прямой доступ к аппаратным компонентам применялся все реже и, наконец, стал использоваться только ядром операционной системы и драйверами. С самого начала это потребовало универсализации многих операций, например, вывода графической информации на экран, принтер и т. п., доступа к таким разным объектам, как файлы и коммуникационные порты (забегая вперед, стоит отметить, что по окончании работы закрытие всех объектов, работа с которыми происходила при помощи дескрипторов (handles), — файлов, коммуникационных портов, примитив синхронизации, каналов и т. д. —выполняется одной единственной функцией *CloseHandle*.

Сам графический интерфейс, большой набор разнообразных элементов управления, расширяющийся с каждой новой версией, обеспечивает унифицированный интерфейс приложений с пользователем, что существенно облегчает обучение работе с новым программным обеспечением. Корпорация Microsoft не только обеспечивает разработчиков все расширяющимися возможностями, но и выдвигает определенные требования к создаваемым ими приложениям. Мы наблюдаем редкий случай, когда монополизм приносит определенную пользу. Только программы, удовлетворяющие выдвинутым требованиям, могут претендовать на специальный знак Microsoft, наличие которого безусловно поможет продвижению продукта на рынке.

Практически с самого начала в составе Windows имелись средства, обеспечивающие обмен данными между приложениями, самое простое из них — буфер обмена (clipboard). Кроме этого фирмой Microsoft разработаны и другие механизмы обмена данными, которые постоянно ею развиваются: динамический обмен данными (Dynamic Data Exchange, DDE), связывание и внедрение объектов (Object Linking and Embedding, OLE). Жесткая конкуренция с другими фирмами в разработке аналогов OLE (возможно, более мощных) позволяет надеяться, что появление объявленного распределенного OLE не за горами.

Каждая следующая версия Windows создавалась так, чтобы она была совместима с более ранними, а также с MS-DOS (что удобно для пользователей). Программистам, как нам кажется, это создавало и определенные трудности — в новой версии недостатки зачастую не исчезали, а, наоборот, даже усугублялись.

Наконец, Windows обеспечивает более полное использование оперативной памяти и многозадачности (пусть и псевдо для Windows 3.x и ранее).[1]

---

[1] Апологетам OS/2 или UNIX скажем, что число приложений для этих операционных систем не идет ни в какое сравнение со всем валом программного обеспечения, написанного именно для Windows, а поддержка технической литературой, обучающими центрами, да и просто напором Microsoft отодвигает эти системы на второе и третье места.

Давайте посмотрим, что нового дают нам системы Windows 95 и Windows NT, какие возможности позволят создавать действительно жизнеспособные приложения.

Windows 95

Несмотря на то, что Windows 95 появилась после создания ОС Windows NT, эта система является продолжением семейства однопользовательской платформы Windows и следует сразу же за Windows 3.1. Хотя рассматриваемая система включает многое из того, что было наработано создателями Windows NT, и даже больше, имеет в своем составе возможности, присущие пока только ей, Windows 95 — это система для персонального компьютера, предназначенного для решения тех же задач, что и Windows 3.1. Далее перечислены основные отличия от предшествующих систем.

1. Использование части программного обеспечения MS-DOS для начальной загрузки компьютера и как шаблона "DOS-окружения" для тех DOS-программ, с которыми пользователь будет работать уже после загрузки Windows 95.

2. Использование возможностей, предоставляемых процессорами семейства Intel 386 и выше: поддержка 32-разрядной адресации, а также полное использование и управление виртуальной памятью (до сих пор полная поддержка виртуальной памяти была невозможной из-за малой мощности персональных компьютеров).

3. Вытесняющая многозадачность. Это значит, что именно операционная система управляет выделением времени центрального процессора каждому приложению. На самом деле все обстоит несколько сложнее. Каждое приложение в системе представлено процессом. *Процесс* — это приложение, находящееся в стадии выполнения. Сам процесс при этом — сущность пассивная, с ним не связаны инструкции, которые выполняет центральный процессор. А что же собственно выполняется, и для кого операционная система выделяет время центрального процессора? На стадии выполнения каждый процесс представлен в системе несколькими потоками (threads).[1] *Поток* — это отдельно выполняемая и управляемая часть программы. На наш взгляд, потоковая вытесняющая многозадачность — основное отличие от предыдущей версии Windows. Одно это позволяет существенно упростить решение многих задач программирования, расширить круг использования компьютера, ускорить выполнение некоторых операций и усилить надежность (независимость) программ.

---

[1] Этот термин дословно переводится как "нить", действительно, каждый процесс движется к своему завершению по одной или нескольким нитям управления, которые также могут разделяться (порождать новые), завершаться и т. д. Однако для лучшего понимания в русской технической литературе используется термин "поток".

4. Использование стольких очередей сообщений, сколько существует потоков в системе. Внедрение такого подхода, с одной стороны, исключает остановку многих запущенных приложений из-за одного из них, что происходило довольно часто, а с другой — никак не влияет на разработку программного обеспечения. (Кто бы мог подумать, что за некоторые улучшения не придется платить!)
5. Переход к 32-разрядной адресации существенно сказался на некоторых сообщениях, посылаемых оконным процедурам, прежде всего, на типах данных параметров сообщений и их формате. Однако проблемы могут возникнуть лишь при переносе "старого" кода в новую систему. Разумеется, с вновь создаваемыми программами сложностей быть не должно.
6. Пользовательский интерфейс Windows 95 — это то, что *кажется* наибольшим отличием от предыдущих версий. Он практически полностью изменен, а его реализация, настройка и управление приближены к тому, что Microsoft собирается воплотить в проекте Cairo. Речь идет об объектно-ориентированной операционной системе. Что касается программирования, то теперь существенно упрощены создание, настройка и управление такими обязательными элементами управления, как панели инструментов, подсказки, строки состояния, индикаторы, бегунки, окна просмотра деревьев, списков и т. д. Сама операционная система поддерживает мощное средство их редактирования с возможностью форматирования, задания шрифтов и вставки объектов, подготовленных различными приложениями.
7. Существенно переработано управление файловой системой, которая также немного изменилась. И то и другое проявляется в увеличении скорости обмена с накопителями и сетевыми дисками, а также в поддержке длинных имен файлов.
8. И, наконец, последнее. Стандартно система поддерживает OLE 2.1,[1] MAPI,[2] Windows Sockets,[3] взаимодействие с локальными и глобальными сетями.

Windows NT

Несмотря на то, что система Windows NT была создана раньше Windows 95, все что относится к последней, верно и для первой, за исключением MS-DOS, пользовательского интерфейса и особенностей интерфейса про-

---

[1] Раньше каждое приложение, использующее OLE, имело в инсталляционной поставке необходимые файлы. Это приводило к тому, что неаккуратно написанные программы при инсталляции могли заменить более свежую версию файла, используемую другими приложениями, своей устаревшей. Естественно, ни к чему хорошему это не приводило.

[2] MAPI (Messaging API) — интерфейс, т. е. набор функций для организации взаимодействия приложений по линиям связи.

[3] То же, что и MAPI, но для локальных сетей.

граммирования, о чем будет рассказано далее. Действительно, для полноценной 32-разрядной операционной системы, которой является Windows NT, MS-DOS не нужна в принципе. Что касается пользовательского интерфейса, то это относится к версиям системы до 3.51 включительно. Во время работы над книгой уже начала распространяться версия Windows NT 4.0, интерфейс которой схож с интерфейсом Windows 95. Основные отличия Windows NT от Windows 95 перечислены ниже.

1. Windows NT более надежна за счет усовершенствованного управления памятью, ресурсами, процессами и потоками. Как следствие, этой системе требуется и больше аппаратных ресурсов (оперативной памяти и мощности процессора).
2. Windows NT способна выполнять не только 32-разрядные приложения, написанные специально для нее, но и приложения, разработанные для других платформ: OS/2 1.x, POSIX, Presentation Manager 2.x, MS-DOS и 16-разрядных версий Windows.
3. Windows NT — переносимая операционная система, т. е. способная работать на компьютерах с процессорами, отличными от Intel x86. Многоуровневая организация системы позволяет большую часть модулей и программ перекомпилировать на соответствующем процессору компиляторе. Конечно, системно-зависимый уровень приходится написать заново, но это проще чем создать новую систему "с нуля". К лету 96-го года уже существовали версии системы для таких процессоров, как MIPS R4000, DEC Alpha и PowerPC. Переносимость операционной системы также позволяет создавать приложения, которые смогут работать на различных компьютерных платформах. К сожалению, и здесь не все хорошо, аппаратные различия приводят к тому, что реализацию ряда возможностей придется заменить при переходе на другую платформу.[1]
4. Windows NT — многопроцессорная операционная система, т. е. она способна выполняться на более чем одном центральном процессоре, выделять процессорное время действительно одновременно нескольким потокам. (А как возрастает производительность!) Конечно, эта возможность необходима далеко не всем, однако многопроцессорные серверы уже вполне доступны по цене, а, значит, уже возможно создание распределенных систем на базе однотипных компьютеров с одинаковой аппаратной и программной платформами. Использование Windows NT позволяет создавать не только чисто офисные системы, но и системы мониторинга и управления, системы реального времени.
5. Windows NT — многопользовательская операционная система с архитектурой клиент-сервер. Она распространяется в двух вариантах — Windows NT Server и Windows NT Workstation. Все это накладывает дополни-

[1] Этому вопросу посвящен большой раздел в документации к компилятору, да и в исходных текстах библиотек и примеров можно встретить отличную от других реализацию функций, например для компьютеров Apple Macintosh.

тельные требования как на администратора системы, так и на разработчиков программного обеспечения (которое, естественно, поддерживает многопользовательский режим и архитектуру клиент-сервер). Конечно, наличие больших возможностей имеет и "отрицательную" сторону — придется затратить гораздо больше времени на их изучение для программирования сложных систем, зато какой простор для фантазии откроется после (не говоря уже о том, что у грамотно написанной программы, не доставляющей хлопот ее пользователям, шансов на успех значительно больше!). В Windows NT можно защитить практически каждый объект, не только файлы и программы, но и процессы, потоки, каналы передачи и обмена данными, примитивы синхронизации и т. д. Соответствие систем на базе Windows NT классу защищенности С2 национального (американского) центра защиты информации позволяет использовать для работы с конфиденциальной и даже секретной информацией.

Интерфейс прикладного программирования

Интерфейс прикладного программирования — (Application Programming Interface, API) — это набор необходимых функций, при помощи которых любое приложение может взаимодействовать с операционной системой.

За последние 15 лет только в рамках Windows появилось три таких API (фактически их больше). Самый первый — Win16 — интерфейс 16-разрядных операционных систем Microsoft. Попытка увеличить разрядность существующей версии привела к появлению Win32s — 32-разрядного интерфейса для 16-разрядных систем (!). Естественно, с появлением 32-разрядных систем Windows NT и Windows 95 появился и 32-разрядный интерфейс — Win32. Самое интересное, что API обоих систем представляют собой пересекающиеся подмножества "полного" API, т. е. в одном API есть функции, не реализованные в другом, и наоборот. К сожалению, об этом можно и не узнать, т. к. нереализованные функции представлены в виде "заглушек".[1]

Итак, API — это связующее звено между приложениями и операционной системой. Масштаб Win32 API — впечатляет, он содержит около двух тысяч функций и несколько сотен сообщений, макросов и предопределенных констант. С одной стороны, программист избавлен от необходимости создавать части операционной системы в своей программе, как это обязательно происходит при создании приложений MS-DOS, с другой стороны широкие возможности серьезно усложняют процесс обучения, какие-то возможности, предоставляемые системой, оказываются не использованы. Вспомним, что из десятков тысяч слов русского языка средний человек

[1] Большинство функций возвращают код завершения, а интересующие значения передаются через указатели, заданные параметрами. Функция, реализованная в виде "заглушки", всегда возвращает код, соответствующий ошибке (NULL, FALSE, ..FAIL... и т. д.).

использует не более пяти тысяч. И в программировании примерно то же самое. Если добавить к функциям API еще немалое количество "обычных" функций, наличие которых является стандартным для компиляторов языка C и C++, то картина становится еще сложнее. Кстати говоря, в отличие от последних функции API во многом взаимозависимы, т. е. одну функцию невозможно использовать без другой, или параметры некоторых функций должны быть подготовлены другими функциями.

Win32 API, как и его предшественники, в основном состоит из трех компонентов: Kernel, User и GDI, которые обеспечивают интерфейс с базовой операционной системой, управлением окнами и приложениями, а так же поддержку графики. Таких функций порядка тысячи, значит, новые возможности обеспечиваются функциями, количество которых сравнимо с количеством всех функций API 16-разрядной Windows.

В Microsoft, видимо, очень популярны всякого рода макросы. Действительно, язык программирования C не поддерживает проверку типов передаваемых параметров, поэтому использование "функциональных" макросов не увеличивает общее количество ошибок на программу. Конечно, усложняется отладка, но это уже проблема программистов. Наверное, не стоило бы об этом говорить, но дело в том, что реально многие, особенно новые, возможности, предоставляемые API, являются макросами. Не будем голословными. Вот отрывок из оригинала, файла COMMCTRL.H:

```
#define ListView_Arrange(hwndLV, code) \
  (BOOL)SendMessage((hwndLV), LVM_ARRANGE, \
                    (WPARAM)(UINT)(code), 0L)
```

В документации сказано, что можно послать сообщение как с помощью этого макроса, так и напрямую, используя функцию *SendMessage*. Может быть, действительно, это и удобно, но надо во всяком случае представлять, где новые возможности, а где нет.

Основное внимание API мы уделим во второй части книги.

Система, управляемая сообщениями

Многие основные черты Windows обязаны своей спецификой тому факту, что она является системой, основанной на сообщениях. Если вы не разберетесь в механизме сообщений, то не поймете и Windows. Такой подход к программированию был впервые реализован в операционной системе для компьютеров Macintosh и заключается он в том, что реакция (и поведение) программы определяется внешними событиями.

В отличие от традиционного последовательного программирования, программирование для Windows является объектно-ориентированным. И неважно, что для этого могут быть использованы вовсе не объектно-ориентированные языки. (Объектно-ориентированные языки обеспечивают взаимодействие между объектами при помощи функций-сообщений). Система

Windows представляет собой набор объектов, и именно с ними приходится сталкиваться и при работе и при программировании. Самым важным таким объектом является окно. Окно первым появляется при запуске системы или любого приложения и последним исчезает при завершении работы. Windows сообщает приложению, точнее, окну приложения, о некотором событии при помощи *сообщения*, которое описывает некоторое изменение, произошедшее в окружающей приложение среде.[1] Кроме этого, механизм сообщений — единственный способ "общения" со своими окнами для самих приложений. Происходит это через операционную систему, которая перенаправляет сообщения нужному объекту (окну). Сообщения используются для настройки, изменения и диагностики состояния всех окон. Сообщение — это вход в приложение. Они представляют собой события, на которые, при необходимости, должно реагировать приложение. Каждое сообщение связывается с конкретным окном, с каждым из которых, в свою очередь, связана собственная *оконная процедура*. Все сообщения, источником которых служат аппаратные средства, система помещает в определенную структуру данных — *первичную входную очередь (*raw input queue*)*.

На сегодняшний день существует две реализации механизма очередей сообщений. В версиях Windows 3.x и более ранних система поддерживает лишь одну очередь сообщений, из которой приложения их и выбирают. В этих системах реализована так называемая невытесняющая многозадачность, при которой такой механизм сообщений вполне удовлетворителен. Проблемы возникают лишь при сбое в активном приложении, когда его невыбранные из очереди сообщения не дают возможность другим приложениям получить сообщения, относящиеся к ним. И хотя это время от времени случается, менять механизм, видимо, не имело смысла, прежде всего из-за увеличения "накладных расходов", которые легли бы непосильным бременем на компьютеры начала 90-х.

Напротив, в 32-разрядных операционных системах Windows 95 и Windows NT реализован метод поддержки отдельных очередей сообщений, который называется *десинхронизацией ввода* (input desynchronization), при котором новые сообщения лишь ненадолго попадают в первичную входную очередь, после чего перенаправляются в частную очередь конкретного потока 32-разрядного приложения. В самом начале своей работы 32-разрядное приложение обладает единственной очередью сообщений, которая относится к его первичному потоку. В случае, если процесс заводит еще один поток, система не создает еще одну очередь немедленно. Дополнительная очередь создается только после того, как второй поток данного приложения впервые направляет в систему относящийся к очереди запрос. Если потоку очередь сообщений не нужна, система не тратит ресурсы на ее создание.

Каждое окно в Windows имеет специальную функцию, которая вызывается для обслуживания сообщений. Операционная система может только послать

---

[1] Приложения, не имеющие окон, не получают и сообщения.

сообщение окну, а оно уже само решает, что делать с этим сообщением. Точнее, программист "учит" окно, как отвечать на получаемые сообщения. В течение этого процесса окно действует как автономный объект, живущий своей собственной жизнью.

Как уже говорилось ранее, Windows API имеет в своем составе сотни функций, но есть одна из них, которая сама по себе стоит их всех. Именно она используется для посылки сообщений объектам системы, естественно, если известны их идентификаторы. Речь идет о функции *SendMessage*. Основная ее задача, что видно и из названия, — послать сообщение некоторому объекту, возможно, с дополнительными данными, дождаться реакции этого объекта на сообщение, обычно в виде числа, которое и вернуть. Вроде бы все просто и ясно. Объектом может быть не только окно приложения, документа, блока диалога и т. д., но и любой элемент управления, например, кнопка. Сообщение, передаваемое с помощью функции *SendMessage*, практически немедленно обрабатывается оконной процедурой, т. к. последняя получает сообщение не из очереди, а напрямую.

Windows обеспечивает приложения и возможность послать сообщение некоторому объекту, используя очередь сообщений, посредством функции *PostMessage*. В этом случае приложение продолжает выполняться, не дожидаясь реакции на сообщение.

SDI и MDI

Изначально система Windows обеспечивает поддержку двух основных интерфейсов для программ, работающих с документами различного назначения.

Первый тип программ поддерживает однодокументный интерфейс (Single Document Interface, SDI). Это значит, что приложение позволяет работать только с одним открытым документом. Приложения, поддерживающие многодокументный интерфейс (Multiple Document Interface, MDI), одновременно могут открывать несколько документов, возможно, имеющих различные типы.

В SDI-приложениях главное окно является и окном документа. В MDI-приложениях окна, содержащие открытые документы, располагаются внутри главного окна приложения, при этом активизация того или иного документа может приводить к изменению элементов приложения, например, меню и панелей инструментов.

Стремление Microsoft к созданию операционной системы, ориентированной на работу не с файлами, а с документами, по мнению компании, приведет к вытеснению MDI-приложений SDI-приложениями.[1]

---

[1] Многие продукты самой фирмы Microsoft пока свидетельствуют об обратном.

Глава 2

Пользовательский интерфейс

Первое, с чем сталкивается пользователь, включив компьютер с установленной версией Windows — это графический пользовательский интерфейс. Нет никакой командной строки, Norton'а или чего-то подобного. Разработчики системы стремились не только к тому, чтобы система была удобна и интуитивно понятно бы было как с ней работать, но и чтобы приложения и системы третьих фирм по своему интерфейсу как можно меньше отличались от программ, разработанных самой фирмой Microsoft. Для этого операционная система поддерживает большое количество объектов, на основе которых можно реализовать большую часть того, что может потребоваться разработчикам.

Окна, элементы управления, модальные и немодальные блоки диалога

Окно (Window) — прямоугольная область экрана для организации обмена между пользователем и приложением. Окно совместно использует экран с другими окнами, в том числе и других приложений. Одновременно пользователь может осуществлять ввод только в одно окно и приложение, к которому это окно относится.

Блок диалога (Dialog Box) — временное окно, создаваемое для запроса ввода пользователя. Обычно блоки диалога используются для получения дополнительной информации для функционирования приложения. Блоки диалога содержат некоторое количество элементов управления, позволяющих вводить текст, устанавливать переключатели и т. д. Диалоги могут быть как модальные, так и не модальные. Модальные диалоги используются в случае, когда от пользователя требуется ввод некоторых данных или ответ на вопрос для продолжения работы. Другие элементы (окна) приложения недоступны. Немодальные диалоги могут использоваться для постоянного отображения на экране, например для быстрого изменения какого-либо параметра из возможных, с немедленной реакцией приложения на предпринятые изменения.

Элемент управления (Control) — дочернее окно, которое приложение использует для взаимодействия с другим окном, обеспечивая простейшие операции ввода/вывода. Чаще всего элементы управления используются в окнах диалога, хотя их возможно использовать и при работе с окнами других типов. Далее перечислены все предопределенные элементы управления.

Кнопка (BUTTON) — обычно уведомляет, что пользователь выбрал этот элемент. Различают несколько видов кнопок: обычная, срабатывающая по нажатию (PUSH BUTTON), переключатель (RADIO BUTTON), флажок (CHECK BOX) и группа (GROUP BOX).

Элемент редактирования (EDIT) — простейший редактор, позволяющий пользователю просматривать и редактировать текст.

Список (LIST BOX) — элемент отображения списка элементов, позволяет пользователю выбрать один или несколько его элементов.

Комбинированный список (COMBO BOX) — комбинация списка и однострочного элемента редактирования. Позволяет пользователю выбрать и редактировать элемент списка.

Полоса прокрутки (SCROLL BAR) — элемент управления, позволяющий пользователю выбрать направление и расстояние прокрутки информации в связанном с элементом окне.

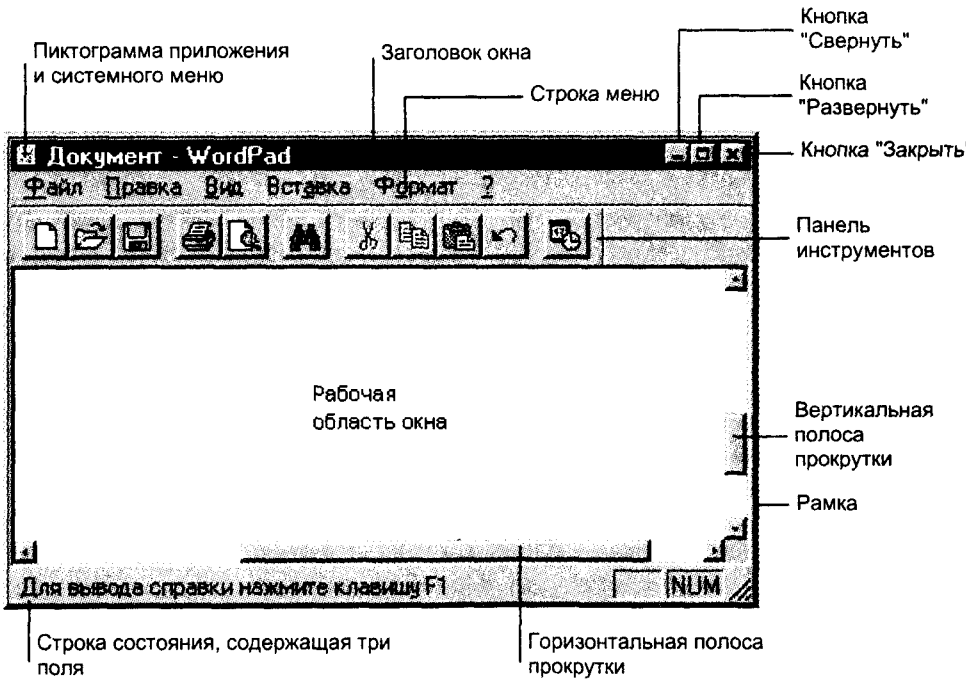


Рис. 1. Основные и дополнительные элементы окна

Статический элемент (STATIC) — элемент исключающий воздействие на него со стороны пользователя, обычно используется в качестве надписи для других элементов управления.

Операционная система предоставляет несколько путей для создания пользовательских элементов управления.

1. Для кнопок, списков и комбинированных списков можно использовать стиль *самоотображения* (owner-draw), который требует написание кода для самостоятельного прорисовывания элемента.
2. Наследование предопределенного элемента управления с заменой функции окна, в которой можно предусмотреть любые интересующие возможности, оставляя возможность использования части кода стандартной реализации.
3. Создание своего собственного класса окна.

Кроме *предопределенных элементов управления* (CONTROLS) и Windows 95, и Windows NT поддерживают большое количество так называемых *стандартных элементов управления* (COMMON CONTROLS). Поддержка осуществляется специальной динамически подгружаемой библиотекой. Как и остальные элементы управления, описываемые элементы представляют собой дочерние окна, которые приложение использует для взаимодействия с другими окнами, обеспечивая простейшие операции ввода/вывода. Отличие от обычных элементов управления состоит в большем их разнообразии и функциональности. Кроме того целый ряд элементов активно используются помимо блоков диалога и в окнах других типов. Далее перечислены все общие элементы управления.

Изменяемый список (DRAG LIST BOX) — специальный тип списка, позволяющий перетаскивать свои элементы, меняя порядок отображения.

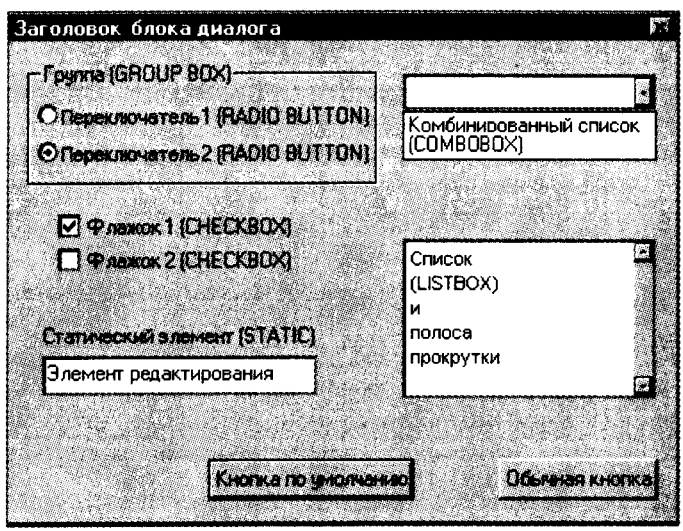


Рис. 2. Виды элементов управления в блоке диалога

Заголовок (HEADER) — дочернее окно обычно располагаемое над колонками текста или цифр. Содержит заголовок для каждого столбца и позволяет менять ширину столбцов.

Просмотр списка (LIST VIEW) — мощное средство отображения элементов списка. Позволяет отображать помимо текста связанные с элементами пиктограммы. Поддерживает четыре способа отображения. Три из них располагают элементы с пиктограммами по всему полю отображения, четвертый способ позволяет создать список, состоящий из набора столбцов, имеющих заголовки.

Строка состояния (STATUS LINE) — горизонтальное окно в нижней части родительского окна, служащее для отображения различной информации приложения, например, подсказки для выбранного пункта меню. Строка состояния может содержать несколько полей для отображения координат, времени, индикации нажатия специальных клавиш (<Ins>, <Scroll Lock>, <Num Lock>) и т. д.

Панель инструментов (TOOLBAR) — дочернее окно, состоящее из набора кнопок быстрого вызова тех или иных команд.

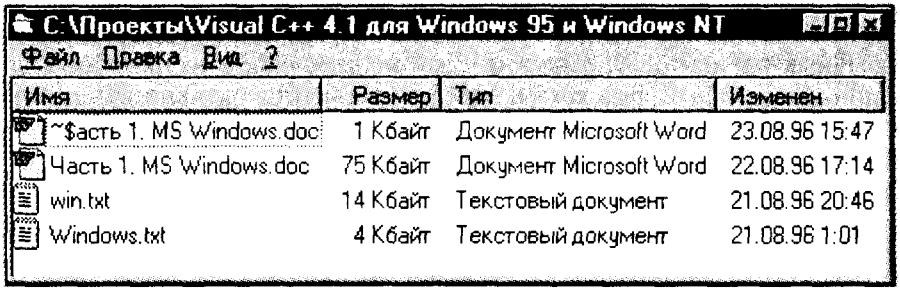


Рис. 3. Заголовок и просмотр списка

Подсказка (TOOLTIP) — небольшое всплывающее окно с однострочным текстом краткого описания некоторого элемента управления, например, кнопки.

Просмотр дерева (TREE-VIEW) — окно просмотра данных, организованных в виде дерева, например содержимого диска или директория. Каждый элемент дерева может, кроме текста, иметь и некоторое растровое (битовое) изображение. Узлы дерева можно раскрывать и закрывать, т. е. отображать и прятать элементы узла.

Вкладка (TAB) — средство, позволяющее создавать элементы, аналогичные закладкам, которые используются при чтении книги. Вкладки используются для смены страниц отображения в одной и той же области диалогового окна. Специальный тип вкладок позволяет отображать каждую вкладку аналогично кнопке, выбор которой приведет к выполнению команды вместо смены страницы.

Окно свойств (PROPERTY SHEET) — окно позволяющее просмотреть и редактировать свойства некоторого объекта. Достоинства этого элемента управления по сравнению с обычным диалоговым окном состоят в том, что большое количество параметров возможно сгруппировать, а доступ к каждой такой группе осуществлять, используя вкладки (TAB). В то же время на базе окна свойств можно строить так называемые мастера, где переход от одного этапа запроса пользователя к последующему или предыдущему осуществляется при помощи соответствующих кнопок. Создание каждого отдельного окна для группы свойств или группы запросов (для мастера) заключается в подготовке ресурса диалога.

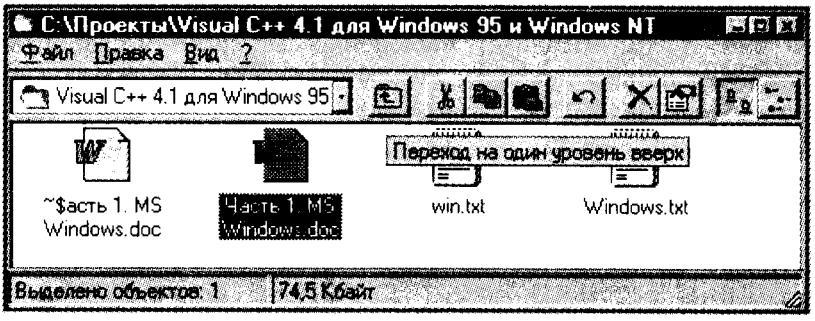


Рис. 4. Список в режиме больших пиктограмм, панель инструментов, строка состояния, подсказка

Клавиша вызова (HOT-KEY) — позволяет пользователю нажать комбинацию клавиш, которая в дальнейшем может быть использована для быстрого выполнения команды. Этот элемент отображает пользовательский выбор и проверяет его допустимость.

Элемент редактирования с форматированием (RICH EDIT) — окно редактирования. В отличие от предопределенного элемента редактирования (EDIT), редактор RTF-формата позволяет управлять шрифтами, цветом, размером, выравниванием абзацев. Кроме того, этот элемент управления поддерживает работу с внедренными объектами OLE.

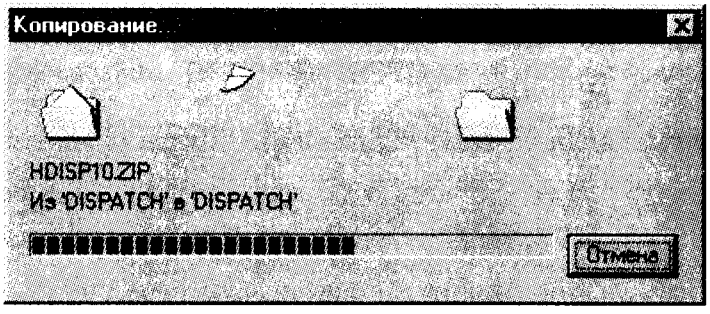


Рис. 5. Визуализация процесса копирования при помощи элемента просмотра видео-клипа и индикатора

Линейка с бегунком (TRACKBAR) — элемент, состоящий из бегунка и, возможно, градуированной линейки. Изменение положения бегунка производится при помощи клавиатуры, либо мышью. При этом ползунок посылает уведомляющие сообщения об изменении своего состояния.

Просмотр видео-клипов (ANIMATION) — этот элемент позволяет просматривать видео-клипы, состоящие из набора битовых изображений. Такие клипы должны быть записаны в формате Audio Video Interleaved (AVI). Этот элемент пригоден для отображения только клипов, не содержащих звуковое оформление.

Индикатор (PROGRESS BAR) — средство индикации некоторого продолжающегося процесса или операции, например копирования файлов. Реализуется в виде прямоугольника, постепенно заполняемого от начала до завершения операции.

Спин (UP-DOWN) — пара кнопок со стрелками для увеличения или уменьшения значения, связанного со счетчиком элемента управления, например элемента редактирования, содержащего численную информацию.

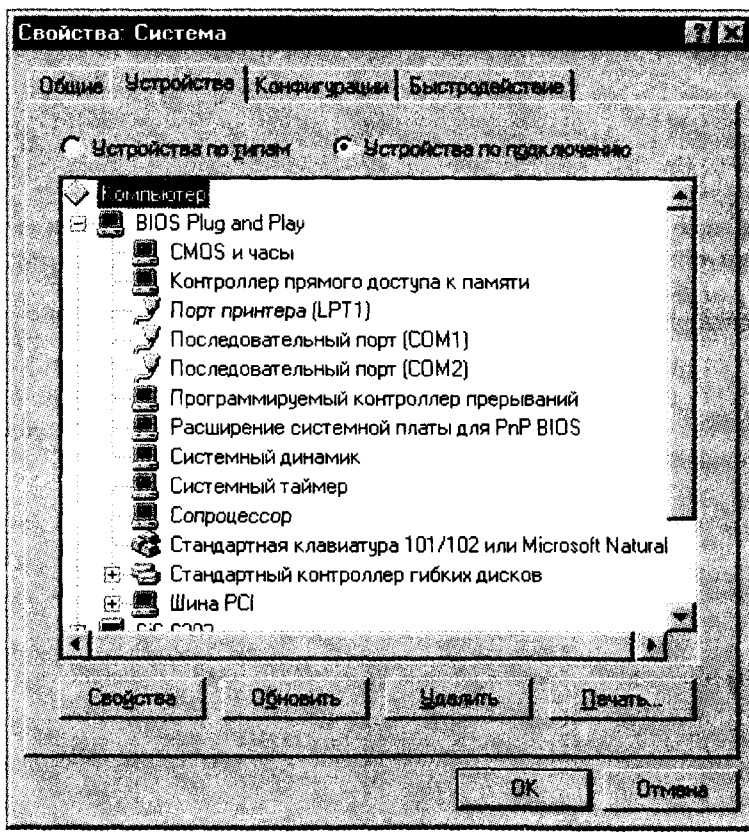


Рис. 6. Окно свойств со вкладками и просмотром дерева

Список рисунков (IMAGE LIST) — вспомогательный элемент, используемый, прежде всего, другими общими элементами управления. Представляет собой список совпадающих по размеру пиктограмм и битовых массивов, доступ к которым осуществляется по индексу. Используется с элементами "просмотр списка" и "просмотр дерева", для хранения пиктограмм вкладок и т. д.

Все вышеперечисленные общие элементы управления, наверное, каждому вполне знакомы, т. к. многие программы уже используют их даже в рамках Windows 3.x. Представьте себе сколько потребовалось труда программистам, чтобы реализовать их, не имея соответствующей поддержки операционной системы, которая теперь есть.

Ресурсы приложения

Каждое приложение обычно имеет целый ряд стандартных или пользовательских ресурсов. Ресурсами приложения называются массивы данных, которые разработчик может добавить в исполняемый модуль. Ниже перечислены стандартные ресурсы:

- акселераторы (accelerators) — структуры данных, содержащие списки горячих клавиш и команд, ассоциированных с ними;
- битовые массивы (bitmaps) — массивы точек для отображения на экране, в качестве, например, пунктов меню или элементов управления;
- курсоры (cursors) — битовые массивы, использующиеся в качестве растровых изображения курсоров мыши;
- шаблоны диалогов (dialogs) — структуры описывающие окно диалога, включая элементы управления, оконные стили, положение на экране и другие параметры. Используются для вывода диалоговых окон на экран;
- пиктограммы (icons) — битовые массивы, используются для визуального представления различных объектов в системе — приложений, документов, элементов списков некоторых типов и т. д.;
- шаблоны меню (menus) — описывают пункты меню, связанные с задаваемыми пользователем командами;
- строковые таблицы (string tables) — списки статических символьных массивов;
- шаблоны панелей инструментов (toolbars) — описывают элементы управления, состоящие из набора битовых массивов в качестве кнопок, каждая из которых связана с определенной командой;
- описания версий (version) — статические структуры специального вида, содержащие информацию о программе, включая имя приложения, номер версии, информацию об авторских правах и т. п. Используются операционной системой для вывода информации о программе.

Кроме стандартных, разработчик может создавать и любые пользовательские ресурсы, однако в этом случае он не сможет пользоваться стандартными функциями доступа к ним, т. к. системе неизвестна их семантика.

Использование термина *ресурсы* может показаться несколько неожиданным, тем не менее приложения работают с вышеперечисленными объектами действительно как с ресурсами, причем разделяемыми, потому что несколько копий приложения могут пользоваться только одной копией некоторого ресурса.

Дополнительное преимущество использования ресурсов приложения в том, что все они сосредоточены в файле (файлах), являющемся отдельной единицей компиляции, что чрезвычайно полезно, например, при наличии большого количества строковых сообщений, которые в случае их реализации в виде ресурсов, не разбросаны по различным файлам.

Особенно полезно использование ресурсов для создания шаблонов диалогов, шаблонов меню и акселераторов.

Доступ к ресурсам осуществляется по числовым или строковым идентификаторам.

Глава 3

Управление графическим выводом

Идеология графического вывода

Система Windows создавалась прежде всего как графическая оболочка. Соответственно главное отличие (по крайней мере на первый взгляд) от MS-DOS — это графическое представление информации. Естественно, что практически все приложения используют экран, чтобы отобразить данные, которыми они управляют. Даже если это не так, то все равно визуальные элементы приложений отображаются на экране самой операционной системой. Как уже упоминалось выше, Windows обеспечивает универсальность представления информации как на экране, так и на других устройствах вывода, например, на принтере. Причем для этого используются одни и те же примитивы отображения. Система сама распознает целевое устройство и активизирует соответствующий ему модуль. В связи с тем, что Windows многозадачная система, к приложениям предъявляется ряд требований, исключающих конфликты при использовании функций вывода. Однако это не значит, что Windows обеспечивает приложения только набором функций вывода на экран или печать — система управляет всем выводом.

Правильнее будет сказать, что приложения используют скорее окно как первичное устройство вывода, чем экран непосредственно. Каждое устройство вывода в Windows имеет набор текущих параметров, с использованием которых и происходит собственно вывод. Причем в каждый конкретный момент времени только одному приложению соответствует некоторое устройство вывода, что исключает одновременный доступ к последнему, изменение параметров вывода одним приложением перед началом процесса вывода другим.

Графические объекты

Windows обеспечивает приложения набором графических объектов, которые позволяют управлять выводом. Для этого применяются:

- битовые массивы (bitmaps) — прямоугольные массивы точек, формирующие растровые изображения;

- карандаши (pens) — используются для задания параметров рисования линий, таких как толщина, цвет и стиль (сплошная, прерывистая и т. п.);
- кисти (brushes) — используются для задания параметров заливки замкнутых контуров, таких как цвет и стиль;
- шрифты (fonts) — используются для задания параметров вывода текста, включая имя шрифта, размер символов и т. п.;
- регионы (regions) — области окна, которые могут быть ограничены прямоугольником, многоугольником, эллипсом или их комбинацией для выполнения операция заполнения, заливки, инверсии и т. п. Кроме того служат для определения местоположения курсора;
- логические палитры (logical palettes) — осуществляют интерфейс между приложением и цветным устройством вывода, таким как дисплей, содержат список цветов необходимых приложению;
- контуры (paths) — используются для целей заполнения или выделения контура различных фигур.

Аппаратно-независимый графический вывод

Одна из главных особенностей Win32 API — независимость графического вывода от устройства. Программное обеспечение, которое поддерживает независимость, содержится в двух динамически компонуемых библиотеках (DLL). Первая, GDI.DLL, обеспечивает графический интерфейс устройства (Graphics Device Interface, GDI), вторая является драйвером устройства, например, если приложение осуществляет вывод в область пользовательского окна на дисплее VGA, это библиотека — VGA.DLL, если на принтер Epson FX-80, это библиотека — EPSON9.DLL.

Перед операцией вывода на некоторое устройство приложение должно запросить GDI о загрузке соответствующего драйвера (обычно это происходит автоматически и не требует от программиста дополнительных усилий). Как только драйвер загружен, приложение может настроить ряд параметров вывода, таких как цвет линии и ее ширина, тип кисти и ее цвет, шрифт, область отсечения, и так далее. Windows обеспечивает хранение этих и других параметров в специальной структуре, называемой *контекстом устройства.*

Контексты устройств

Контекст устройства — структура, определяющая набор графических объектов и связанных с ними атрибутов и графических режимов, которые, собственно, и воздействуют на вывод. Графические объекты включают карандаши

для рисования линий, кисти для закрашивания и заполнения, битовые образы для копирования или прокрутки частей экрана, цветовые палитры для определения набора доступных цветов, области для отсечения и других операций, а также контуры для операций рисования и закрашивания.

Приложение не имеет прямого доступа к контексту устройства, как к большинству структур Win32 API — настройка параметров осуществляется посредством вызова соответствующих функций Win32 API.

Графические режимы

Windows поддерживает пять различных графических режимов, которые позволяют приложениям определять тип смешивания цветов, место и параметры вывода и т. д.

Режим настройки фона. Определяет, как происходит смешивание цветов фона текстовых объектов и растровых изображений и фона поля вывода (окна или экрана).

Режим отображения (рисования). Определяет, как происходит смешивание цвета карандашей (pens), кистей (brushes), текстовых объектов и растровых изображений с цветом фона поля вывода (окна или экрана).

Режим масштабирования. Определяет преобразования логических координат при графическом выводе в окна, на экран или принтер.

Режим заливки контуров. Определяет использование шаблонов кистей при заливке сложных контуров.

Режим сжатия. Определяет, каким образом происходит преобразование цветов растровых изображений при их увеличении (уменьшении).

Как и в случае графических объектов, контекст устройства инициализируется значениями по умолчанию и для графических режимов.

Типы контекстов устройств

Win32 API определяет четыре типа контекстов устройств:

- для экрана — обеспечивает операции рисования непосредственно на экране;
- для принтера — обеспечивает операции рисования на принтере или плоттере;
- для объектов в памяти — обеспечивает операции рисования без непосредственного вывода на экран;
- информационный — обеспечивает получение данных об устройстве.

Используя контекст устройства, приложение может осуществлять следующий набор операций:

- перечисление графических объектов;
- установка новых графических объектов;
- удаление графических объектов;
- сохранение графических объектов, их атрибутов и параметров графических режимов;
- восстановление графических объектов, их атрибутов и параметров графических режимов.

Кроме того, приложение может использовать контекст устройства для определения процесса графического вывода, прерывания длительных графических операций, начатых другим потоком многопоточного приложения, а также инициализировать принтер.

Экран

Win32 API обеспечивает три типа контекста устройства для экрана: контекст класса, общий и частный контексты. Контекст класса и частные контексты устройства используются в приложениях, которые выполняют многочисленные операции рисования. Например, программы автоматизированного проектирования, настольные издательские системы, т. е. такие приложения, которые самостоятельно и постоянно осуществляют вывод и, соответственно, время затрачиваемое на эти операции критично с точки зрения производительности приложения. Общие контексты устройства используются в приложениях, которые лишь время от времени выполняют операции рисования.

Приложение получает контекст устройства экрана, вызывая функции *BeginPaint* или *GetDC* и идентифицируя окно, в которое соответствующий вывод будет производиться. Причем тип контекста устройства зависит от того, как приложение зарегистрировало класс окна. Обычно, приложение получает контекст устройства экрана непосредственно перед выполнением операций вывода. После завершения вывода приложение должно обязательно освободить контекст устройства вызовом функций *EndPaint* или *ReleaseDC* соответственно.

Контекст класса

Контекст класса поддерживается только для совместимости с предыдущими версиями Windows. При создании Win32-приложения вместо контекстов класса следует использовать частные контексты.

Общий контекст

Общий контекст устройства — контекст устройства экрана, связанный с компонентом управления окна Win32 API и находящиеся в области администратора окна. Приложение получает общий контекст устройства, вызывая

функцию *GetDC*, *GetDCEx*, или *BeginPaint*. Windows инициализирует общие контексты устройства значениями по умолчанию, которые можно менять по мере необходимости при помощи функций, описанных в части II. Количество общих контекстов в системе ограничено и, следовательно, после завершения операции вывода необходимо сообщить системе об этом, вызвав функции *ReleaseDC* или *EndPaint*. В результате вызова одной из этих функций те изменения параметров контекста, которые были произведены приложением, будут отменены. Таким образом необходимо устанавливать параметры каждый раз.

Частный контекст

Частные контексты устройства отличаются от общих тем, что сохраняют любые изменения для заданных по умолчанию данных даже после вызова функции *ReleaseDC* или *EndPaint*. Частные контексты устройства не являются частью области администратора окна и, следовательно, лишь однократно инициализируются значениями по умолчанию. Администратор окна автоматически удаляет частный контекст устройства только после того, как последнее окно класса было разрушено.

Приложение создает частный контекст устройства, определяя стиль CS_OWNDC для класса окна в момент инициализации структуры WNDCLASS с последующей регистрацией класса окна (функция *RegisterClass*). Для получения частных контекстов окон, имеющих стиль CS_OWNDC, используются те же функции: *GetDC*, *GetDCEx* или *BeginPaint*.

Принтер

Win32 API обеспечивает один и тот же тип контекста устройства как для принтера, который может быть матричным, струйным или лазерным, так и для графопостроителя. Приложение создает контекст устройства принтера, вызывая функцию *CreateDC*. При этом задаются необходимые параметры, такие как имя драйвера принтера, имя принтера, файла или другого устройства вывода. После завершения печати приложение должно вызвать функцию *DeleteDC* для удаления созданного контекста, простое освобождение контекста при помощи функции *ReleaseDC* недостаточно и недопустимо!

Объект в памяти

Для улучшения характеристик вывода изображений, например исключения мигания, Win32 API предоставляет возможность приложениям полностью подготавливать изображение в памяти до его вывода на экран дисплея и только затем осуществлять вывод. Для этого предназначен контекст устройства памяти, который сохраняет растровые изображения для специфического устройства. Приложение создает контекст устройства памяти, вызывая

функцию *CreateCompatibleDC*. Результатом выполнения этой функции является растровое изображение (или растр), имеющее цветной формат, совместимый с форматом первоначального устройства (отсюда и название функции).

Первоначальное изображение в контексте устройства памяти имеет размер один на один пиксель. Прежде, чем приложение сможет начать работу с изображением, оно должно установить битовый массив с соответствующей шириной и высотой в контекст устройства, вызывая функцию *SelectObject*.

Информационный контекст

Win32 API поддерживает информационный контекст устройства, используемый, чтобы восстановить или получить заданные по умолчанию параметры устройства. Для создания информационного контекста приложение должно вызвать функцию *CreateIC*. Для получения информации об объектах, заданных по умолчанию для интересующего устройства, используются функции *GetCurrentObject* и *GetObject*. Использование информационного контекста устройства более эффективно, чем контекстов других типов, т. к. Win32 API работает с информационным контекстом на более низком уровне и не создает структуры, необходимые для их работы.

После завершения работы с информационным контекстом необходимо вызвать функцию *DeleteDC* для удаления созданного контекста.

Работа со шрифтами. Шрифты True Type

Операционная система Windows может работать со шрифтами следующих трех типов:

- растровые
- векторные
- шрифты TrueType

Отличие между видами шрифтов заключается в способе хранения параметров начертания символов в специальных шрифтовых файлах. В случае растровых шрифтов каждый символ хранится в виде растра (битового массива). Векторные шрифты хранят для каждого символа относительные координаты концов отрезков, из которых состоит соответствующий символ. Шрифты TrueType содержат информацию о линиях и командах изгиба, а также настроечную информацию (hints) для точного отображения символа, которая также используется при уменьшении или увеличении масштаба отображения.

Растровые шрифты имеют заданные размеры и, следовательно, качество их отображения зависит от устройства вывода. Напротив, векторные шрифты

независимы от устройства вывода, однако время, необходимое для их отображения больше, чем для растровых или шрифтов TrueType. Последние и обеспечивают приемлемую скорость вывода, и могут быть промасштабированы с сохранением изначального вида символов.

Windows обеспечивает разработчиков широким набором функций для использования шрифтового оформления своих приложений, начиная с того, что каждый контекст устройства имеет шрифт по умолчанию, и заканчивая предоставлением системного (встроенного) диалога для выбора шрифтов, который можно использовать в приложении для предоставления пользователям возможности выбора любого шрифта из имеющихся.

Часть II

Программирование под MS Windows

4. Интерфейс прикладного программирования Win32
5. Вытесняющая многозадачность
6. Структура приложения Windows

Глава 4

Интерфейс прикладного программирования Win32 API

Несмотря на то, что книга в целом посвящена программированию с использованием библиотеки классов MFC (Microsoft Foundation Classes), в этой главе мы опишем ряд приемов программирования, а также набор функций и понятий Win32 API, которые либо не поддерживаются библиотекой классов, либо ими в равной степени удобно пользоваться как при помощи MFC, так и с использованием Win32 API. Функции, реализующие создание процессов и потоков и их синхронизацию, также будут описаны в этой главе. Библиотека классов MFC тоже поддерживает эти возможности, подробности их реализации мы опишем во втором томе.

Отличие 16- и 32-разрядных платформ

Так как эту книгу мы пишем не для новичков в программировании вообще и для Windows в частности, обзор отличий будет кратким и довольно поверхностным, иначе пришлось бы написать еще несколько глав, посвященных, прежде всего, аппаратным особенностям всех компьютерных систем, на которых может работать Windows, и провести исторический экскурс, рассказывая о том, какими были компьютеры 15—20 лет назад.

Отличие двух платформ влияет на переносимость программ. И дело здесь не только и не столько в различии команд процессора и более широком их наборе, а, прежде всего, в размере элементарных типов данных, и, следовательно, в размере параметров функций. Для 16-разрядных систем 32-разрядные параметры использовались лишь в крайнем случае, а для 32-разрядной системы именно 32-разрядные параметры являются наиболее приемлемыми. Как ни странно, именно это служит причиной невозможности простой перекомпиляции старых программ. Ведь Win32 API практически полностью совместим по названиям функций со своим 16-разрядным предшественником. Возьмем, например, декларацию функции окна, без которой не обходится ни одно приложение.

```
/* так было раньше */
long far pascal WndProc(HWND, unsigned, WORD, long);
/* так сделано сейчас */
LRESULT APIENTRY WndProc(HWND, UINT, WPARAM, LPARAM);
```

Этот пример иллюстрирует гибкость новой реализации и независимость кода программы от разрядности API, так как не используются ключевые слова языка программирования, определяющие тип, а значит и размер параметров. Теперь, сколько бы разрядов не занимал второй параметр, 16, 32 или более, проблем переносимости быть не должно.

Но это отличие еще полбеды. Дело в том, что часто через разные части одного и того же параметра в функцию передаются значения. Формат таких параметров в Win32 API претерпел существенные изменения, которые должны сказаться на исходных текстах всех приложений, написанных для 16-разрядных Windows, если их нужно перекомпилировать под 32-разрядную среду. Это, прежде всего, касается данных, передаваемых при помощи сообщений. В следующем примере проиллюстрировано отличие интерпретации параметров.

```
/* Windows 3.x */
...
case WM_COMMAND:
 WORD nID = LOWORD(wParam);
 WORD nCode = HIWORD(wParam);
...
/* Windows 95 или Windows NT */
...
case WM_COMMAND:
 UINT nID = wParam;
 UINT nCode = HIWORD(lParam);
...
```

Теперь код команды занимает старшее слово параметра *lParam* оконной функции. Тут одной заменой наименований типов не обойдешься.

И последнее, что хотелось бы отметить в этом разделе, это появление большой группы функций, которые помещают запрашиваемое значение по передаваемому указателю, а не возвращают его. В качестве результата функции используется код возврата, прежде всего типа BOOL — успех/неуспех (TRUE/FALSE).

Это как всегда имеет и положительные и отрицательные стороны. К последним можно отнести увеличение объема кода за счет появления дополнительных переменных и строк. С другой стороны, отпадает необходимость проверки возвращаемого значения на допустимость, что вообще-то не всегда и возможно (исторически только нулевой указатель показывает неуспех вызова функции).

Перечисленные выше различия и нововведения больше связаны с переносом программ. Далее мы укажем на отличия, большинство из которых можно отнести к существенным улучшениям всей системы в целом:

1. Практически сняты ограничения на число ресурсов, доступных Win32-приложениям, прежде всего, дескрипторов (можно использовать до 32 767 дескрипторов окон вместо 200, как Windows 3.x).
2. Несмотря на то, что всегда рекомендуется освобождать захваченные при работе ресурсы, Windows 95 и, особенно, Windows NT после завершения приложения самостоятельно освобождают все захваченные приложением ресурсы. Достигается это за счет того, что каждый ресурс помечается идентификатором потока его владельца. Кроме того, разделяемые ресурсы имеют счетчики использования. По достижению нулевого значения счетчика ресурс также освобождается. Такое управление ресурсами практически снимает проблему постепенной деградации системы, как это было ранее.
3. Появились новые типы уведомляющих сообщений, что позволяет приложениям лучше адаптироваться к изменению конфигурации во время работы.
4. Стали использоваться ранее зарезервированные биты стилей окон, соответственно, появились новые стили, что упрощает программирование пользовательского интерфейса — операционная системы взяла еще часть работы на себя.

Функция WinMain

Все программы, написанные на языках C или C++, начинают свое выполнение с функции *main*. Фирме Microsoft понадобилось и здесь внести изменение в принятые стандарты, т. к. параметры стандартной функции *main* не подходят для приложений Windows. Теперь каждая Windows-программа на C или C++ начинает свое выполнение с вызова функции *WinMain*. Объявление этой функции в рамках Win32 API следующее:

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

hInstance — дескриптор текущего приложения; *hPrevInstance* — дескриптор предыдущей запущенной копии приложения; *lpCmdLine* — указатель на командную строку; *nCmdShow* — режим начального отображения главного окна приложения.

Вот пример простейшей программы для Windows:

```
/*************************************************************
Модуль APIMain
Copyright (c) 1997 Мешков А., Тихомиров Ю.
*************************************************************/

#include <windows.h>

int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
  return (FALSE);
}
```

Код программы несколько больше, чем просто вызов функции *main*, за счет подключаемого файла WINDOWS.H, который включает в себя еще пару десятков заголовочных файлов, состоящих из десятков тысяч строк, поэтому время компиляции даже такой программы может занять несколько минут.

От предыдущих версий это описание на первый взгляд отличается лишь использованием нового типа для возвращаемого значения. Реально все сложнее. Значение параметра *hPrevInstance* до сих пор позволяло определить наличие уже запущенных копий программы. В большинстве случаев это нужно для исключения повторного запуска одной и той же программы. В Win32 API этот параметр всегда равен NULL (и оставлен исключительно для совместимости). Связано это с тем, что каждое 32-разрядное приложение запускается в своем адресном пространстве, в котором, естественно, нет никаких его копий или других приложений.

В документации Microsoft предлагает несколько путей для определения наличия предыдущей копии. Один из них заключается в использовании функции *FindWindow*, а другой — в создании именованных каналов[1] или объектов синхронизации (их мы обсудим в разделе, посвященном синхронизации).

Функция *FindWindow* в случае обнаружения искомого окна возвращает его дескриптор. Один из параметров может быть равен NULL, в этом случае поиск будет производиться при помощи второго параметра.

```
HWND FindWindow(
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName
);
```

[1] Именованные каналы (named pipes) можно создавать лишь в рамках Windows NT.

lpClassName — указатель на строку, содержащую имя оконного класса или специального вида идентификатор этого класса; *lpWindowName* — указатель на строку, содержащую заголовок искомого окна

Одно из требований Microsoft к приложениям для Windows состоит в исключении повторного запуска. Действительно, большинству программ это не нужно. Например, для многооконных редакторов повторный запуск вместо открытия дополнительного окна приведет только к лишней трате системных ресурсов. Для программ мониторинга или управления вообще бессмысленно держать две копии приложения в памяти, т. к. такие программы обычно работают с неразделяемыми ресурсами. Хорошей практикой является не только завершение повторной копии приложения сразу после загрузки, но и активизация этого приложения, ведь, скорее всего, повторный запуск был произведен из-за того, что оно понадобилось пользователю. Активизация в общем случае совершается в два этапа. Во-первых, главное окно искомого приложения восстанавливается в размере, если оно было свернуто в значок, и, во-вторых, происходит собственно активизация — перевод окна и, соответственно, приложения на передний план. На этом обычно повторная копия приложения должна быть завершена. Для этого используются функции *IsIconic*, *ShowWindow*, *SetForegroundWindow*.

```
...
hwnd = FindWindow (szClassName, szWindowTitle);
if (hwnd)
{
 /* нашли копию приложения - восстановим его на экране и
    активизируем его */
  if (IsIconic(hwnd))
    ShowWindow(hwnd, SW_RESTORE);
  SetForegroundWindow (hwnd);
  /* данная копия должна быть завершена */
  return (FALSE);
}
...
```

Назначение *WinMain* гораздо шире, чем просто завершение. В следующей главе мы более подробно остановимся на этом вопросе.

Новые возможности для создания программ

Мы уже говорили о том, что и Windows 95, и Windows NT как и раньше базируются на модулях Kernel, User и GDI, которые поддерживают около половины функций Win32 API. Первый модуль обеспечивает интерфейс с базовой операционной системой, второй — управление окнами и приложениями, и третий — поддержку графики.

Win32 API в своем составе имеет и целый набор новшеств, поддерживаемых большим количеством новых системных функций. Перечислим их.

1. Функции поддержки OLE.
2. Функции поддержки новых общих элементов управления (COMMON CONTROLS).
3. Функции, реализующие вызов и настройку стандартных диалогов, некоторые из них будут описаны ниже.
4. Функции поддержки разархивирования файлов, которая необходима, например, для программ инсталляции нового программного обеспечения.
5. Функции поддержки динамического обмена данными DDE (Dynamic Data Exchange). Надо сказать, что этот механизм поддерживался в 16-разрядных Windows, но для лучшего понимания групп функций API мы упомянули о DDE отдельно. (Все возможности DDE поддерживаются и в OLE, но для ряда задач гораздо проще (особенно на стадии обучения) использовать именно DDE, а не его монстрообразное развитие).
6. Функции поддержки удаленных процедур RPC (Remote Procedure Call). Еще год-другой, и термин "персональный компьютер", видимо, будет забыт, распределенные вычисления — задача отнюдь не для персонального использования.
7. Функции поддержки интерфейса Windows Sockets, который создан для работы в сети, прежде всего для сетей, построенных на основе протокола TCP/IP и для приложений, не использующих RPC.
8. Функции сетевой поддержки (кроме двух предыдущих групп функций).
9. Коммуникационные функции MAPI (Messaging API). Эти функции предназначены для программ удаленного доступа и территориально распределенных сетей.
10. Функции поддержки работы и расширения возможностей графической оболочки (пока только для Windows 95 и Windows NT 4.0).
11. Функции поддержки Multimedia-приложений, служат для управления звуком и видеоизображениями.
12. Функции поддержки работы с принтерами и заданиями на печать.
13. Функции поддержки приложений, использующих рукописный ввод.
14. Функции поддержки и обеспечения безопасности, такие как аутентификация пользователей и контроль привилегий.
15. Функции поддержки приложений, использующих текстовый ввод/вывод, так называемых *консольных* приложений.
16. Функции поддержки таблицы символов Unicode, использующей 16 разрядов для хранения кода одного символа.[1]

[1] В Windows 95 кодировку Unicode поддерживают только подсистема OLE и файловая система.

17. Функции поддержки работы системы в качестве сетевого сервера.
18. Функции поддержки диспетчера управления различными службами, в качестве которых могут выступать различные по назначению подсистемы операционной системы, что позволяет еще более унифицировать управление ими.
19. Функции поддержки регистрации событий, происходящих в системе, например, ошибок, переконфигурации, регистрации пользователей, нарушений принятых правил безопасности и т. д.
20. Функции поддержки 3-х мерной графики (OpenGL).

APIShell — пример приложения

Следующий пример иллюстрирует целый класс приложений, основным окном которых является блок диалога (dialog box). Кроме того, в примере рассмотрена одна из возможностей управления графической оболочкой Windows 95 при помощи функции *Shell_NotifyIcon*.

```
/*************************************************************
Файл APIShell.c
Copyright (c) 1997 Мешков А., Тихомиров Ю.
*************************************************************/

#define _INC_OLE
#include <windows.h>
#undef _INC_OLE

#include <windowsx.h>

#include "resource.h"

/* определение номера сообщения */
#define WM_NOTIFYICON (WM_APP+100)
HINSTANCE g_hinst;
UINT g_uState;

/* идентификатор пиктограммы */
#define IDC_NOTIFY 1000

/*
  объявления используемых функций
*/

/* оконная функция окна диалога */
BOOL CALLBACK TestAppDlgProc(HWND hDlg, UINT uMsg,
                             WPARAM wParam, LPARAM lParam);
/* функция настройки уведомляющей области */
BOOL TrayMessage(HWND hDlg, DWORD dwMessage,
                 HICON hIcon, PSTR pszTip);
```

```
/* функция изменения состояния пиктограммы уведомления */
void StateChange(HWND hDlg);

/*
  реализация функций приложения
*/
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
/* сохранение дескриптора приложения, без проверки
   наличия предыдущих копий */
  g_hinst = hInstance;

/* создание и вывод диалога настройки -
   основного окна приложения */
  DialogBox(
    hInstance,
    MAKEINTRESOURCE(IDD_APISHELL),
    NULL,
    TestAppDlgProc);

/* основное окно закрыто - завершение приложения */
  return(FALSE);
}

/* функция настройки уведомляющей области */
BOOL CALLBACK TestAppDlgProc(HWND hDlg,
                             UINT uMsg,
                             WPARAM wParam,
                             LPARAM lParam)
{
  switch (uMsg)
  {
    case WM_INITDIALOG:
      /* инициализация окна диалога */
      TrayMessage(hDlg, NIM_ADD, NULL, NULL);
      StateChange(hDlg);
      break;

    case WM_DESTROY:
      /* разрушение окна диалога */
      TrayMessage(hDlg, NIM_DELETE, NULL, NULL);
      break;

    case WM_COMMAND:
      /* обработка сообщений от элементов окна диалога */
      switch (GET_WM_COMMAND_ID(wParam, lParam))
```

```
      {
        case IDCANCEL:
          EndDialog(hDlg, TRUE);
          break;

        case IDHIDE:
          ShowWindow(hDlg, SW_HIDE);
          break;
      }
      break;

    case WM_NOTIFYICON:
      /* обработка сообщений от пиктограммы
         области уведомления */
      switch (lParam)
      {
        case WM_LBUTTONDOWN:
          switch (wParam)
          {
            case IDC_NOTIFY:
              /* смена состояния с заменой пиктограммы */
              g_uState = !g_uState;
              StateChange(hDlg);
              break;

            default:
              break;
          }
          break;

        case WM_RBUTTONDOWN:
          /* активизация окна диалога */
          ShowWindow(hDlg, SW_SHOW);
          SetForegroundWindow(hDlg);
          break;

        default:
          break;
      }
      break;

    default:
      return(FALSE);
  }

  return(TRUE);
}

BOOL TrayMessage(HWND hDlg,
                 DWORD dwMessage,
                 HICON hIcon,
                 PSTR pszTip)
```

```
{
  BOOL retCode;
  NOTIFYICONDATA tnd;

  /* формирование структуры для работы с областью уведомления */
  tnd.cbSize = sizeof(NOTIFYICONDATA);
  tnd.hWnd = hDlg;
  tnd.uID = IDC_NOTIFY;

  tnd.uFlags = NIF_MESSAGE|NIF_ICON|NIF_TIP;
  tnd.uCallbackMessage = WM_NOTIFYICON;
  tnd.hIcon = hIcon;

  if (pszTip)
    lstrcpyn(tnd.szTip, pszTip, sizeof(tnd.szTip));
  else
    tnd.szTip[0] = '\0';

  /* собственно настройка области уведомления */
  retCode = Shell_NotifyIcon(dwMessage, &tnd);

  if (hIcon)
    DestroyIcon(hIcon);

  return retCode;
}
void StateChange(HWND hDlg)
{
  LPCSTR pszIDIcon;
  char szTip[64];

  /* загрузка нужного текста и пиктограммы
     в зависимости от текущего состояния */
  if (!LoadString(g_hinst, g_uState == 0 ?
      IDC_STATE1 : IDC_STATE2, szTip, sizeof(szTip)))
    *szTip = '\0';

  pszIDIcon = MAKEINTRESOURCE(
    g_uState ? IDI_STATE1 : IDI_STATE2);

  TrayMessage(hDlg, NIM_MODIFY,
    LoadImage(g_hinst, pszIDIcon, IMAGE_ICON, 16, 16, 0),
    szTip);
}
/*************************************************************
Файл APIShell.rc
Copyright (c) 1997 Мешков А., Тихомиров Ю.
*************************************************************/

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
```

```
#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include <windows.h>
//////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

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

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

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

1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
  "#include ""windows.h""\0"
END

#endif  // APSTUDIO_INVOKED

//////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon remains consistent on all systems.
IDI_STATE1        ICON  DISCARDABLE   "STATE1.ICO"
IDI_STATE2        ICON  DISCARDABLE   "STATE2.ICO"

//////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_APISHELL DIALOG DISCARDABLE 0, 0, 89, 26
```

```
STYLE DS_MODALFRAME | DS_CENTER | WS_MINIMIZEBOX |
  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "APIShell"
FONT 8, "MS Sans Serif"
BEGIN
  PUSHBUTTON    "&Спрятать",IDHIDE,19,6,50,14
END

////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
  IDC_STATE1        "Принтер занят"
  IDC_STATE2        "Принтер свободен"
END

#endif  // Russian resources

/*************************************************************
Файл resource.h
Copyright (c) 1997 Мешков А., Тихомиров Ю.
*************************************************************/

////////////////////////////////////////////////////////////////
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by APIShell.rc
//
#define IDD_APISHELL            103
#define IDI_STATE1              105
#define IDI_STATE2              106
#define IDC_STATE1              1009
#define IDC_STATE2              1010
#define IDHIDE                  1019

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE    111
#define _APS_NEXT_COMMAND_VALUE     106
#define _APS_NEXT_CONTROL_VALUE     1020
#define _APS_NEXT_SYMED_VALUE       101
#endif
#endif
```

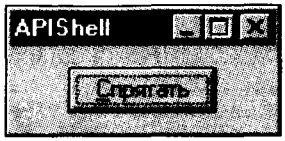


Рис. 7. Окно управления приложением APIShell



Рис. 8. Область уведомления до запуска приложения APIShell



Рис. 9. Область уведомления с пиктограммой принтера



Рис. 10. Область уведомления с измененной пиктограммой принтера

Первое, с чего начинается код приложения — включение необходимых для работы заголовочных файлов. Обратите внимание на макроопределение _INC_OLE. Таким нехитрым способом из процесса компиляции исключается большое количество строк кода, которые для данной программы абсолютно не нужны.[1]

```
#define _INC_OLE
#include <windows.h>
#undef _INC_OLE
#include <windowsx.h>
```

Создать диалоговое окно, как видите, очень просто: передаем идентификатор ресурса и указатель на функцию блока диалога. После завершения диалога выполнять больше нечего — завершаем функцию *WinMain*.

```
DialogBox(
   hInstance,
   MAKEINTRESOURCE(IDD_APISHELL),
   NULL,
   TestAppDlgProc);
```

В функции *TestAppDlgProc*, которая является функцией окна, необходимо явно обработать следующие сообщения:

- WM_INITDIALOG — в программе собственно блок диалога инициализируется по умолчанию, на основе того, что указано в шаблоне диалога.

[1] При создании приложений с использованием библиотеки классов MFC используется аналогичный, но более корректный метод исключения неиспользуемых заголовочных файлов.

Однако необходимо добавить в область уведомления пиктограмму принтера, передав системе также информацию об обработчике сообщений от той части области уведомления, где располагается пиктограмма.

- WM_DESTROY — разрушение или завершение диалога происходит как и в случае инициализации без нашего вмешательства, необходимо только удалить пиктограмму из области уведомления.
- WM_COMMAND — сообщения от элементов управления диалога (IDHIDE и IDCANCEL), которые позволяют пользователю, соответственно, скрывать блок диалога и завершать приложение.
- WM_NOTIFYICON — сообщение от пиктограммы области уведомления, анализ которого позволяет менять пиктограммы и активизировать блок диалога в зависимости от кнопки мыши, которую нажал пользователь, указав пиктограмму.

Графический вывод

В каких случаях приложению необходимо осуществлять вывод? Конечно же при изменении данных, чтобы пользователь мог это увидеть. Программы для того и пишутся, чтобы взаимодействовать с пользователями. В Windows, однако, вывод должен осуществляться еще в ряде случаев:

- в момент создания окна;
- при изменении размеров окна;
- при появлении окна или его части из-за другого окна;
- при разворачивании или сворачивании окна;
- при отображении данных из только что открытого файла;
- при прокрутке, изменении или выборе данных, находящихся в окне.

Во многих случаях Windows берет на себя инициирование процесса отображения, отмечая область перерисовки и посылая функции соответствующего окна сообщение WM_PAINT. Это сообщение обрабатывает само приложение, осуществляя процесс вывода данных в окно.

В ряде случаев приложение, изменив данные, может инициировать процесс вывода самостоятельно, используя то же сообщение. Однако может оказаться необходимым осуществлять вывод, продолжая выполнять некоторые действия. Характерный пример такого процесса — это чтение файла с одновременным выводом считанных данных в некоторое окно.

В любом случае приложению для начала вывода необходимо получить дескриптор контекста устройства вывода. Обработчик сообщения WM_PAINT для этого должен использовать функцию *BeginPaint*, а при непосредственном выводе — функции *GetDC* или *GetDCEx*.

Сообщение WM_PAINT

Итак, в большинстве случаев вывод осуществляется обработчиком сообщения WM_PAINT. Windows посылает это сообщение окну только тогда, когда очередь сообщений пуста. Это достаточно разумно — изменения отображаются как бы блоками, сокращая тем самым время, затрачиваемое на вывод.

Для получения дескриптора контекста устройства приложение вызывает функцию *BeginPaint*, во время выполнения которой Windows настраивает область отсечения (обновления), т. е. определяет область, которую действительно необходимо отобразить. Попытки вывода вне области отсечения игнорируются. Кроме того, в это же время Windows может послать окну сообщения WM_NCPAINT и WM_ERASEBKGND. Первое используется для инициирования перерисовки неклиентских частей окна — рамки, заголовка, системного меню, полос прокрутки и т. д. Большинство программ не обрабатывают это сообщение, полагаясь на заданную по умолчанию функцию окна *DefWindowProc*, которая обрабатывает в том числе и это сообщение. Обработчик сообщения WM_ERASEBKGRND подготавливает фон области вывода, затирая все, что до этого было выведено. Если окно принадлежит классу, имеющему кисть фонового режима, функция *DefWindowProc* заполняет фон окна автоматически.

Функция *BeginPaint*, кроме вышеперечисленных действий, заполняет структуру *PAINTSTRUCT* информацией о размере части окна, подлежащей изменению. Эту информацию можно использовать для оптимизации операций вывода, самостоятельно исключить команды отображения вне области отсечения, а не полагаться на систему. Последнее, что делает функция *BeginPaint* —устанавливает область модификации равной NULL для исключения повторной генерации сообщения WM_PAINT, и скрывает каретку (caret) — маркер клавиатуры, защищая его от искажения при перерисовке.

По завершении операции вывода обработчик сообщения WM_PAINT должен освободить контекст устройства и восстановить каретку при помощи вызова функции *EndPaint*.

Приведем определения указанных функций и структуры *PAINTSTRUCT*.

Функция *BeginPaint* возвращает в случае успеха дескриптор контекста устройства для указанного окна, иначе NULL.

```
HDC BeginPaint(
HWND hWnd,
LPPAINTSTRUCT lpPaint);
```

hWnd — дескриптор окна, в которое будет произведен вывод; *lpPaint* — указатель на структуру *PAINSTRUCT*, заполняемую информацией для осуществления графического вывода.

Функция *EndPaint* всегда возвращает значение TRUE.

```
BOOL EndPaint(
PWHWND hWnd,
PPCONST PAINTSTRUCT *lpPaint);
```

hWnd — дескриптор окна вывода; *lpPaint* — указатель на структуру *PAINSTRUCT*, заполненную при выполнении функции *BeginPaint*.

```
typedef struct tagPAINTSTRUCT{
HDC hdc;
BOOL fErase;
RECT rcPaint;
BOOL fRestore;
BOOL fIncUpdate;
BYTE rgbReserved[32];
} PAINTSTRUCT;
```

hdc — дескриптор контекста устройства; *fErase* — определяет необходимость удаления содержимого области обновления перед выполнением функций отображения; *rcPaint* — определяет прямоугольную область обновления; *fRestore* — зарезервировано для внутреннего использования Windows; *fIncUpdate* — зарезервировано для внутреннего использования Windows; *rgbReserved* — зарезервировано для внутреннего использования Windows.

Функции определения и отмены области обновления

Приложение определяет часть окна, подлежащую обновлению, используя функции *InvalidateRect* или *InvalidateRgn*. Эти функции добавляют задаваемую прямоугольную область или регион окна путем объединения этой области с другими, заданными ранее. Результатом объединения является минимально возможная прямоугольная область, охватывающая все указанные области обновления. Функции *InvalidateRect* и *InvalidateRgn* не генерируют сообщения WM_PAINT. Вместо этого Windows накапливает изменения, давая возможность приложению обработать все другие сообщения, с тем чтобы в последствии модифицировать сразу все заданные области окна.

Напротив, функции *ValidateRect* и *ValidateRgn* отменяют модификацию задаваемых прямоугольных областей или регионов, например, если до обработки сообщения WM_PAINT приложение уже обновило часть окна. Для получения информации о текущей области обновления приложение может воспользоваться функциями *GetUpdateRect* и *GetUpdateRgn*.

Функция *InvalidateRect* возвращает в случае успеха значение TRUE, иначе FALSE.

```
BOOL InvalidateRect(
  HWND hWnd,
  CONST RECT *lpRect,
  BOOL bErase);
```

hWnd — дескриптор окна модификации; *lpRect* — адрес структуры, содержащей координаты добавляемой области модификации; *bErase* — флаг необходимости удаления содержимого области модификации.

Функция *InvalidateRgn* всегда возвращает значение TRUE.

```
BOOL InvalidateRgn(
  HWND hWnd,
  HRGN hRgn,
  BOOL bErase);
```

hWnd — дескриптор окна модификации; *hRgn* — дескриптор добавляемого региона модификации; *bErase* — флаг необходимости удаления содержимого региона модификации.

После вызова функции *BeginPaint* вызовы функций *ValidateRect* и *ValidateRgn* недопустимы, т. к. в этот момент область модификации пуста.

Функция *ValidateRect* возвращает в случае успеха значение TRUE, иначе FALSE.

```
BOOL ValidateRect(
  HWND hWnd,
  CONST RECT *lpRect);
```

hWnd — дескриптор окна модификации, если этот параметр равен NULL, Windows обновляет все окна, посылая сообщения WM_ERASEBKGND и WM_NCPAINT; *lpRect* — адрес структуры, содержащей координаты исключаемой области модификации, если этот параметр равен NULL, отменяется модификация всей клиентской области окна.

Функция *ValidateRgn* возвращает в случае успеха значение TRUE, иначе FALSE.

```
BOOL ValidateRgn(
  HWND hWnd,
  HRGN hRgn);
```

hWnd — дескриптор окна модификации; *hRgn* — дескриптор исключаемого региона модификации, если этот параметр равен NULL, отменяется модификация всей клиентской области окна.

Функция *GetUpdateRect* возвращает значение TRUE, если область модификации не пуста, иначе FALSE.

```
BOOL GetUpdateRect(
  HWND hWnd,
  LPRECT lpRect,
  BOOL bErase);
```

hWnd — дескриптор окна, для которого запрашивается текущая область модификации; *lpRect* — указатель на структуру, заполняемую информацией о координатах области модификации (установив значение этого параметра равным NULL, приложение быстро и эффективно может получить информацию о наличии или отсутствии области модификации); *bErase* — флаг необходимости удаления содержимого области модификации (если значение флага равно TRUE и область модификации не пуста, функция посылает указанному окну сообщение WM_ERASEBKGND для удаления содержимого области модификации).

Функция *GetUpdateRgn* возвращает значение, определяющее структуру запрашиваемого региона:

- COMPLEXREGION — регион состоит больше чем из одного прямоугольника;
- ERROR — при выполнении произошла ошибка;
- NULLREGION — регион пуст;
- SIMPLEREGION — регион представлен одним прямоугольником.

```
int GetUpdateRgn(
  HWND hWnd,
  HRGN hRgn,
  BOOL bErase);
```

hWnd — дескриптор окна, для которого запрашивается текущий регион модификации; *hRgn* — дескриптор региона, для заполнения информацией об области модификации; *bErase* — флаг необходимости удаления содержимого области модификации и рисования неклиентской области дочерних окон.

В некоторых случаях возникает необходимость предотвратить перерисовку всей области модификации при следующей реакции на сообщение WM_PAINT. Для этого используется функция *ExcludeUpdateRgn*, смысл возвращаемого значения которой такой же, как и для функции *GetUpdateRgn*.

```
int ExcludeUpdateRgn(
  HDC hDC,
  HWND hWnd);
```

hDC — дескриптор контекста устройства; *hWnd* — дескриптор окна, в котором предотвращается перерисовка.

Синхронный и асинхронный вывод

В большинстве случаев выполнение обработки сообщения WM_PAINT происходит асинхронно. Напомним, что это сообщение генерируется системой только тогда, когда других сообщений нет, т. е. очередь сообщений пуста и есть, что перерисовывать. Таким образом, имеет место некоторая, как правило случайная, задержка между моментом выполнения одной из функций *InvalidateRect* или *InvalidateRgn* и генерацией сообщения WM_PAINT.

Однако приложение имеет также возможность синхронного выполнения операций рисования (отображения). Типичное приложение сразу после создания своего главного окна немедленно отображает его на экране, сообщая тем самым пользователю об успешном начале функционирования. Некоторые элементы управления, например, кнопки, также должны быть немедленно перерисованы в случае изменения состояния, т. к. они имеют фокус ввода и не могут ждать, когда приложение освободится и предоставит

возможность перерисовки. Да вообще любому окну может понадобиться немедленное отображение изменений.

Для немедленного обновления содержимого окна в Win32 API предусмотрены функции *UpdateWindow* и *RedrawWindow*. Первая посылает сообщение WM_PAINT непосредственно окну в случае, если область модификации не пуста. Вторая также посылает сообщение WM_PAINT, но предоставляет программисту возможность управления процессом вывода, позволяя обновлять неклиентскую область окна, удалять содержимое окна до начала отображения, посылать сообщение WM_PAINT независимо от того, есть ли непустая область модификации. Эти функции посылают сообщение WM_PAINT непосредственно окну, независимо от наличия других сообщений в очереди приложения.

Функция *UpdateWindow* возвращает в случае успеха значение TRUE, иначе FALSE.

```
BOOL UpdateWindow(
  HWND hWnd);
```

hWnd — дескриптор окна, которое требуется обновить (перерисовать).

Функция *RedrawWindow* возвращает в случае успеха значение TRUE, иначе FALSE. Если эта функция вызывается для обновления главного окна системы (desktop window), последнее не получает сообщение WM_PAINT. Для перерисовки такого окна функция *RedrawWindow* должна вызываться с флагом RDW_ERASE для генерации сообщения WM_ERASEBKGND. Функция по-разному воздействует на окна в зависимости от наличия у них стиля WS_CLIPCHILDREN.

```
BOOL RedrawWindow(
  HWND hWnd,
  CONST RECT *lprcUpdate,
  HRGN hrgnUpdate,
  UINT flags);
```

hWnd — дескриптор перерисовываемого окна, параметр должен иметь значение NULL для главного окна системы; *lprcUpdate* — указатель на структуру, содержащую координаты перерисовываемой области окна, параметр игнорируется, если *hrgnUpdate* определяет регион модификации; *hrgnUpdate* — дескриптор, определяющий регион модификации (если оба параметра *hrgnUpdate* и *lprcUpdate* равны NULL, областью модификации считается вся клиентская часть окна); *flags* — определяет один или более флагов перерисовки; возможные флаги:

RDW_ERASE	задает генерацию сообщения WM_ERASEBKGND при наличии флага RDW_INVALIDATE
RDW_FRAME	задает генерацию сообщения WM_NCPAINT при наличии флага RDW_INVALIDATE
RDW_INTERNALPAINT	задает генерацию сообщения WM_PAINT независимо от наличия непустой области модификации

RDW_INVALIDATE	назначает область модификации через параметр *lprcUpdate* или *hrgnUpdate*
RDW_NOERASE	исключает генерацию сообщения WM_ERASEBKGND
RDW_NOFRAME	исключает генерацию сообщения WM_NCPAINT
RDW_NOINTERNALPAINT	исключает генерацию сообщения WM_PAINT
RDW_VALIDATE	исключает область модификации через параметр *lprcUpdate* или *hrgnUpdate*
RDW_ERASENOW	задает генерацию и обеспечивает получение окнами сообщений WM_NCPAINT и WM_ERASEBKGND до завершения функции; сообщение WM_PAINT обрабатывается как обычно
RDW_UPDATENOW	задает генерацию и обеспечивает получение окнами сообщений WM_NCPAINT, WM_ERASEBKGND и WM_PAINT до завершения функции
RDW_ALLCHILDREN	включает в процесс обновления все дочерние окна
RDW_NOCHILDREN	исключает из процесса обновления все дочерние окна

Перерисовка содержимого окна, требующая больших затрат времени на выполнение операций рисования, должна производиться асинхронно, чтобы предотвратить откладывание обработки других сообщений, и тем самым исключить блокирование работы приложения. Так же, приложение, которому часто требуется модификация небольших областей окна, при асинхронном выводе позволяет системе объединять все запросы на модификацию в одно сообщение WM_PAINT.

Доступ к графическим объектам, их замена и удаление

Каждый контекст устройства при создании получает набор графических объектов по умолчанию, за исключением битового массива и контуров. Приложение может получить информацию об объектах при помощи функций *GetCurrentObject* и *GetObject*.[1] Функция *GetCurrentObject* возвращает дескриптор, идентифицирующий карандаш, кисть, палитру, битовый массив или шрифт, а функция *GetObject* инициализирует структуру, содержащую атрибуты объекта. Для того чтобы изменить объекты и их параметры отрисовки, необходимо создать новые объекты с требуемыми параметрами и

[1] Ряд принтеров могут поддерживать набор резидентных карандашей, кистей и шрифтов. Для получения информации о таких объектах используются функции *EnumObject* и *EnumFontFamilies*.

заменить ими объекты, используемые по умолчанию, при помощи функции *SelectObject*.

Функция *GetCurrentObject* в случае успешного выполнения возвращает дескриптор запрашиваемого объекта, иначе NULL.

```
HGDIOBJ GetCurrentObject(
  HDC hdc,
  UINT uObjectType);
```

hdc — дескриптор контекста устройства; *uObjectType* — тип объекта (OBJ_PEN, OBJ_BRUSH, OBJ_PAL, OBJ_BITMAP или OBJ_FONT).

Функция *GetObject* в случае успешного выполнения возвращает количество байт, требуемое для буфера, который будет заполнен соответствующей информацией о запрашиваемом объекте, иначе функция вернет 0.

```
int GetObject(
  HGDIOBJ hgdiobj,
  int cbBuffer,
  LPVOID lpvObject);
```

hgdiobj — дескриптор интересующего объекта; *cbBuffer* — размерность буфера под данные в байтах; *lpvObject* — указатель на буфер.

Если параметр *lpvObject* равен NULL, функция только сообщит о размерности буфера.

Функция *SelectObject* назначает (выбирает) новый объект в контекст устройства, заменяя текущий объект того же типа и возвращая его дескриптор. Функция не работает с объектами-палитрами, в качестве графического объекта может использовать регионы, но в этом случае смысл возвращаемого значения другой, нежели для других графических объектов (подробнее см. документацию по Win32 API).

```
HGDIOBJ SelectObject(
  HDC hdc,
  HGDIOBJ hgdiobj);
```

hdc — дескриптор контекста устройства; *hgdiobj* — дескриптор устанавливаемого объекта.

Далее рассмотрим пример, в котором, в зависимости от текущего состояния кисти, может произойти ее замена.

```
HDC hdc;                          /* контекст устройства */
HBRUSH hbrushNew, hbrushOld;      /* дескрипторы кистей */
HBRUSH hbrush;                    /* дескриптор кисти */
LOGBRUSH lb;                      /* структура описания кисти */
/*
  получение дескриптора текущей кисти
*/
hbrush = GetCurrentObject(hdc, OBJ_BRUSH);
```

```
/*
  заполнение структуры описания кисти
*/
GetObject(hbrush, sizeof(LOGBRUSH), &lb);
/*
  если текущая кисть имеет стиль, отличный от BS_SOLID
  или ее цвет не черный -
  замена ее на черную кисть со стилем BS_SOLID
*/
if ((lb.lbStyle != BS_SOLID) ||
    (lb.lbColor != RGB(255, 255, 255))
{
  /*
    получение дескриптора нужной кисти
  */
  hbrushNew = GetStockObject(BLACK_BRUSH);
  hbrushOld = SelectObject(hdc, hbrushNew);
}
/*
  выполнение некоторых функций вывода, использующих кисть,
  например FillPath
*/

.........
/*
  по завершении рисования с использованием новой кисти
  приложение должно ее удалить (функция DeleteObject)
  предварительно восстановив кисть, использовавшуюся до
  этого. Т. к. переменная hbrushNew содержит дескриптор,
  полученный в результате выполнения функции GetStockObject,
  в принципе удаление делать не обязательно, однако если
  кисть создана такой функцией как CreateBrushIndirect,
  не удалив ее, приложение до своего завершения будет
  занимать уже недоступный ресурс
*/
if ((lb.lbStyle != BS_SOLID) ||
    (lb.lbColor != RGB(255, 255, 255))
{
  SelectObject(hdc, hbrushOld);
  DeleteObject(hbrushNew);
}
```

Глава 5

Вытесняющая многозадачность

Что бы там не говорили, но фундаментальное отличие 32-разрядной платформы Windows — это вытесняющая многозадачность. Надеемся с нами согласятся те, кто работал не только на персональных компьютерах, или хотя бы в общих чертах знакомы с теорией операционных систем и историей их развития.

То, что было реализовано в Windows до сих пор, является прямым следствием убогости аппаратной части. Действительно, на процессорах Intel 8086 и 8088 ни о какой аппаратной поддержки многозадачности речи не было. Причем главное здесь не отсутствие средств загрузки и сохранения контекста процесса, а отсутствие поддержки защиты памяти. Все, кто программировал для MS-DOS, знают, что к услугам программы предоставлено все — и устройства, и сама операционная система — читай, откуда хочешь, и пиши, куда захочешь. Конечно, при этом достигается необычайная гибкость, но любая ошибка (а они есть всегда) приводит к перезагрузке. Поэтому чисто программная реализация без аппаратной поддержки действительно проблематична. Кстати сказать, все версии Windows до 3.0 работали именно так.[1] Этим, в том числе, и объясняется сравнительно неширокое распространение системы до 90-го года.

Процессоры Intel 80286 и далее уже поддерживают как защиту памяти, так и многозадачность. Но сейчас становится понятным, почему на машинах с тактовой частотой по крайней мере до 40 МГц и малым количеством памяти реализация полноценной операционной системы была просто невозможной — решить задачу было бы быстрее на бумаге, чем на компьютере. Сейчас, когда уже купить машину на базе процессора 386 стало сложнее, чем Pentium, оперативная память подешевела почти в 20 раз, а быстродействие и объем жестких дисков выросли на 1—2 порядка при снижении цен в 5 раз, самое время вспомнить о давно реализованных системах и сделать операционную систему персонального компьютера достойной достижениям в области электроники и схемотехники.

---

[1] Версия Windows 3.0 также могла быть сконфигурирована для работы в режиме, при котором отсутствовала защита памяти.

Реализованный ранее механизм невытесняющей многозадачности, при котором каждое приложение само решало, когда освободить процессор, приводил к тому, что длительные операции либо останавливали всю систему, до окончания их выполнения, либо требовали от разработчиков ухищрений несоизмеримых с результатом. Чтобы программа вела себя корректно по отношению к остальным, т. е. давала бы и им возможность поработать, разработчикам приходится разбивать длительный по времени алгоритм на этапы. "Тактировать" программу можно двумя способами: либо при помощи таймера — "просить" систему время от времени посылать соответствующее сообщение программе (WM_TIMER), либо дать понять системе, что выполнение не закончено, но все равно передать ей управление при помощи функции *PeekMessage*.

В первом случае после завершения одного этапа до выполнения следующего управление будет передано системе, а реальное возобновление работы программы начнется с того же места, что и ранее, т. е. в оконной функции, при этом надо хранить информацию о том, как далеко продвинулась программа при выполнении операции, инициализировать заново или создавать глобальные переменные, требуемые для ее выполнения и т. д. Кроме того, независимо от мощности компьютера и загруженности системы в целом, время выполнения алгоритма всегда примерно одинаково. Иногда это может быть и не так страшно, но всегда ли очевидно, как разбить алгоритм на этапы?

Второй способ существенно проще — включите в текст программы вызовы функции *PeekMessage*. Как часто и где — решайте сами. Но если какая-то программа пошлет сообщение вашей, теперь они "вместе" будут выполнять этот самый длительный алгоритм. Ведь при таком подходе программа реально не обрабатывает никакие сообщения.

Если же программа вообще не вернет управления при таком механизме многозадачности, например, войдя в бесконечный цикл, то пострадает не только она, но и все запущенные приложения. Данные потеряны, открытые файлы, возможно, безвозвратно испорчены, да мало ли что может произойти!

Вытесняющая многозадачность решает эти проблемы. И приложения выполняются одновременно, и система гораздо более устойчива к отказам, т. к. ни одно приложение не может получить контроль над всеми системными ресурсами.

Мы уже упоминали, что поддержка центральным процессором многозадачности заключается в наличии средств сохранения и загрузки контекста процесса. Отсюда следует, что процессор подразумевает наличие практически не связанных участков кода для каждого процесса, в том числе и для ядра операционной системы. В 16-разрядной системе Windows все приложения и системный код представляют собой одну последовательную программу, где передача управления от приложения системе заключается в обязательном завершении функции приложения, которая в свою очередь была вызвана системой. Хотя это и несколько упрощенный взгляд, тем не менее

он хорош для оценки этой системы. Кстати, примерно так организованы многооконные редакторы, написанные для MS-DOS.

В Win32 появились две новые концепции, определяемые терминами *процесс* (process) и *поток* (thread). Кратко мы останавливались на этом в части I. Как уже говорилось, процесс — это экземпляр программы, загруженной на выполнение. С программой же связан термин *модуль*, который представляется файлом, записанным на носитель и содержащим все, что необходимо для ее выполнения — код, данные, ресурсы и т. д. Поток — это "путь", по которому проходит система, выполняя код процесса. При запуске приложения система создает и процесс и поток, т. к. каждый процесс имеет хотя бы один поток.

Распределение процессорного времени (вот где многозадачность!) происходит как раз между потоками, но не между процессами. Давайте посмотрим, в каких состояниях может находиться поток. Во-первых, он может выполняться, во-вторых, он может находиться в состоянии ожидания того момента, когда ему будет опять выделен процессор. Такое состояние называется состоянием готовности к выполнению.

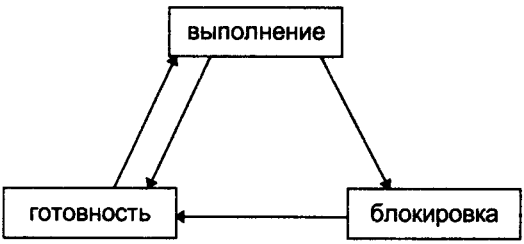


Рис. 11. Диаграмма состояний потока

Вы спросите, зачем про это даже упоминать — и так все ясно. Если бы все было так просто, серьезные программисты опять ждали бы с нетерпением новой версии Windows или стали работать в операционной системе UNIX. Каждый поток может быть еще и заблокированным. Наличие этого состояния существенно расширяет возможности Windows и круг задач, решение которых теперь доступно в рамках этой операционной системы. В состоянии блокировки поток находится тогда, когда он ждет возникновения какого-либо события, будь то завершение записи в файл или синхронизирующее событие от другого потока, но об этом немного позже. Для реализации такой модели многозадачности требуются очередь готовых к выполнению потоков, упорядоченная некоторым образом, и список заблокированных потоков. Вот полная диаграмма состояний потока.

Итак, на три состояния у нас имеется четыре перехода. Изначально поток попадает в очередь готовых к выполнению потоков. Через какое-то время операционная система выделяет в его распоряжение центральный процессор, где он может выполняться до истечения кванта времени.[1] В этом случае

---

[1] *Квантом* называется промежуток времени, выделяемый операционной системой для каждой единицы выполнения, в нашем случае — для потока. Величина кванта может быть фиксированной или настраиваемой. Слишком малое значение кванта приводит к увеличению накладных расходов на переключение, затрачиваемых операционной системой. Слишком большое значение приводит к значительным задержкам реакции на внешние события или сообщения.

система переведет поток в состояние готовности, поместив в очередь готовых на выполнение потоков. Если во время выполнения поток инициирует ожидание возникновения некоторого события, то операционная система заблокирует поток и освободит процессор для первого потока из очереди готовых. При возникновении ожидаемого события система переведет поток из состояния блокировки в состояние готовности. Кстати говоря, запрос на следующее сообщение (вызов функции *GetMessage*) как раз и блокирует поток в ожидании его прихода.

Наличие состояния блокировки существенно разгружает систему. Поясним этот тезис на примере. Процесс состоит из двух потоков, один из которых осуществляет чтение файла данных в память, а другой будет обрабатывать эти данные по окончании чтения. Предположим, что и тому и другому потоку доступна переменная, нулевое значение которой говорит о том, что данные еще не считаны, а единичное — что операция чтения завершена. При таком подходе код второго потока будет выглядеть примерно следующим образом:

```
. . .
while (чтение_не_закончено == 0);
. . .
```

Реализация такого подхода называется *активным ожиданием*. С помощью утилиты System Monitor (Системный монитор), входящей в состав Windows, можно увидеть, что процессор в момент выполнения этого цикла занят на 100%. Если же поток заблокирует себя в ожидании завершения процесса чтения, то он вообще не будет использовать центральный процессор, т. е. накладные расходы будут минимальны.

Важный момент в реализации многозадачности — организация очереди готовых к выполнению потоков. Простейший способ — так называемый принцип FIFO (первый пришедший обслуживается первым). Реализация именно этого принципа существенно ограничила бы возможности Windows. В Win32 многозадачность приоритетная, т. е. в очередь готовых потоки попадают в соответствии с приоритетами. При появлении потока с более высоким приоритетом, чем у выполняющегося, последний немедленно освобождает процессор и заменяется этим высокоприоритетным потоком.[1]

Что же будет с потоками, имеющими более низкий приоритет, они так и не получат процессор для работы? В принципе — да. Такая ситуация называет-

---

[1] Мы не раз упоминали о расширяющемся круге задач, которые можно теперь решать в рамках операционной системы Windows, благодаря именно ее приоритетной многозадачности. Теперь гораздо более надежными станут коммуникационные программы, исключающие пропуск данных от модемов, вообще все программы, работающие с внешними устройствами обмена данными. Теперь возможно создание приложений, которым требуется немедленное обслуживание внешних событий, т. е. систем реального времени. Это действительно шаг вперед!

ся *перегрузкой* (*starvation*). Вот тут как нельзя кстати механизм блокировки, который исключает перегрузку. Но все же, если создаваемому приложению требуются потоки с более высоким приоритетом, чем нормальный (об этом чуть ниже), следует очень тщательно отнестись к его написанию.

Процессы и потоки

В следующем разделе мы подробно опишем значение и использование приоритетов, а сейчас приведем те функции, при помощи которых можно работать с процессами и потоками.

Естественно, что и процессы и потоки не появляются из ниоткуда. Их необходимо создать с помощью функций Win32 API. Для создания процессов предусмотрена функция *CreateProcess*, для создания потоков — *CreateThread*. Для завершения процессов и потоков используются функции *ExitProcess* и *ExitThread* соответственно. Кроме того, для уничтожения процессов и потоков извне используются функции *TerminateProcess* и *TerminateThread*.

Функции *CreateProcess* и *CreateThread* имеют много параметров. Вместо того чтобы дублировать документацию, где они подробно описаны (как это делается в большинстве книг), мы приведем лишь определения этих функций, а их использование продемонстрируем в соответствующем примере (приложение APISync).

Функция *CreateProcess* создает новый процесс и его главный поток. Создание заключается в запуске исполняемого модуля, заданного именем соответствующего файла. Функция возвращает значение TRUE в случае успеха и FALSE в противном случае.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPCTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation);
```

lpApplicationName — имя исполняемого модуля; *lpCommandLine* — указатель на командную строку; *lpProcessAttributes* — указатель на структуру, содержащую атрибуты защиты процесса; *lpThreadAttributes* — указатель на структуру, содержащую атрибуты защиты главного потока; *bInheritHandles* — флаг наследования некоторых объектов (например, открытых файлов); *dwCreationFlags* — флаги

создания процесса; *lpEnvironment* — указатель на новый блок строк окружения; *lpCurrentDirectory* — указатель на имя текущего каталога; *lpStartupInfo* — указатель на структуру *STARTUPINFO*; *lpProcessInformation* — указатель на структуру *PROCESS_INFORMATION*.

Функция *CreateThread* создает новый поток в адресном пространстве вызывающего процесса. Функция возвращает дескриптор потока или NULL в случае невозможности его создания.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  DWORD dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId);
```

lpThreadAttributes — указатель на структуру, содержащую атрибуты защиты потока; *dwStackSize* — начальный размер стека потока в байтах; *lpStartAddress* — указатель на функцию потока; *lpParameter* — аргумент функции потока; *dwCreationFlags* — флаги создания потока; *lpThreadId* — указатель на переменную, которой будет присвоен идентификатор потока.

Функция *ExitProcess* завершает процесс и все его потоки.

```
VOID ExitProcess(UINT uExitCode);
```

uExitCode — код завершения процесса.

Функция *ExitThread* завершает поток.

```
VOID ExitThread(DWORD dwExitCode);
```

dwExitCode — код завершения потока.

Функция *TerminateProcess* уничтожает процесс и все его потоки. Возвращает значение TRUE в случае успеха и FALSE в противном случае.

```
BOOL TerminateProcess(
  HANDLE hProcess,
  UINT uExitCode);
```

hProcess — дескриптор процесса; *uExitCode* — код завершения процесса.

Функция *TerminateThread* уничтожает поток. Возвращает значение TRUE в случае успеха и FALSE в противном случае.

```
BOOL TerminateThread(
  HANDLE hThread,
  DWORD dwExitCode);
```

hThread — дескриптор потока; *uExitCode* — код завершения потока.

Кроме вышеперечисленных функций, могут пригодиться еще некоторые функции Win32 API, которые приведены здесь с краткими комментариями.

AttachThreadInput	назначает (или наоборот отменяет) вывод одного потока на вход другого.
CommandLineToArgvW	производит разбор командной строки, заданной в кодировке Unicode (только для Windows NT).
CreateRemoteThread	создает поток в адресном пространстве другого процесса (только для Windows NT).
FreeEnvironmentStrings	освобождает блок, занятый строками окружения.
GetCommandLine	возвращает указатель на командную строку текущего процесса.
GetCurrentProcess	возвращает псевдодескриптор текущего процесса, который не может наследоваться дочерними процессами.
GetCurrentProcessId	возвращает идентификатор текущего процесса.
GetCurrentThread	возвращает псевдодескриптор текущего потока, который не может наследоваться дочерними процессами.
GetCurrentThreadId	возвращает идентификатор текущего потока.
GetEnvironmentStrings	возвращает указатель на блок, содержащий строки окружения.
GetEnvironmentVariable и *SetEnvironmentVariable*	соответственно возвращает и назначает переменные окружения, задаваемые именем и значением в виде символьной строки с нуль-символом в конце.
GetExitCodeProcess	возвращает код завершения указанного процесса.
GetExitCodeThread	возвращает код завершения указанного потока.
GetProcessVersion	возвращает версию Windows, для которой предназначен указанный процесс.
GetStartupInfo	создает копию структуры *STARTUPINFO* текущего процесса.
OpenProcess	возвращает дескриптор процесса по его идентификатору.
ResumeThread	уменьшает на единицу значение счетчика вызовов функции SuspendThread, при достижении нулевого значения поток возобновляется.
SuspendThread	приостанавливает поток, если он еще не приостановлен, и увеличивает значение счетчика вызовов этой функции на единицу.
WinExec	запускает указанное приложение (для совместимости с предыдущими версиями Windows). Для 32-разрядных приложений следует использовать функцию CreateProcess.

Классы процессов и приоритеты потоков

Система поддерживает 32 приоритета (от 0 до 31), чем больше номер приоритета, тем выше приоритет. Интересно, что Windows назначает приоритеты не только потокам, чего и следовало бы ожидать, но и процессам, для которых правильнее будет говорить о *классах*. Класс процесса определяет базовый приоритет для всех создаваемых потоков. Таких классов четыре: *Idle*, *Normal*, *High*, *Realtime*. Для установки класса процесса можно использовать, например, функцию *SetPriorityClass*. Вот ее определение:

```
BOOL SetPriorityClass(
   HANDLE hProcess,
   DWORD dwPriorityClass);
```

hProcess — дескриптор процесса; *dwPriorityClass* — класс процесса.

В табл. 1 перечислены классы и соответствующие им значения второго параметра функции *SetPriorityClass*.

Таблица 1. *Классы процессов*

Класс	Базовый уровень приоритета	Значение параметра
Idle	4	IDLE_PRIORITY_CLASS
Normal	7–9	NORMAL_PRIORITY_CLASS
High	13	HIGH_PRIORITY_CLASS
Realtime	24	REALTIME_PRIORITY_CLASS

Класс Idle

По умолчанию потоки процесса класса *Idle* создаются с приоритетом 4. Такие потоки хороши для задач, которым процессор требуется время от времени, например для мониторинга системы, фоновой печати или для каких-нибудь несрочных действий.

Класс Normal

Все пользовательские процессы по умолчанию относятся именно к этому классу, если только порождающий процесс не относится к классу *Idle*. В этом случае новый процесс также получает класс приоритета *Idle*.

В табл. 1 базовый уровень приоритета для класса *Normal* указан в промежутке от 7 до 9. Конкретное значение зависит от ряда причин, указанных ниже. Для простоты будем считать, что базовый уровень процессов этого класса равен 8.

Процессы этого класса самые многочисленные, и система относится к ним несколько иначе, чем к процессам всех других классов. В соответствии с терминологией, принятой в теории операционных систем, такие процессы возможно назвать *интерактивными*, т. е. их работа практически постоянно требует действий пользователя, например, в графических или текстовых редакторах. Процессы класса *Normal* делятся на процессы *переднего плана* (foreground) и *фоновые* (background). И Windows 95, и Windows NT различают эти два типа процессов: в момент перевода процесса на передний план приоритеты всех его потоков увеличиваются на 1 в Windows 95 или на 2 в Windows NT. Сделано это для того, чтобы активный в данный момент процесс быстрее реагировал на действия пользователя. Это один из примеров динамического изменения приоритетов.

Класс High

Создавать процессы, относящиеся к классу *High* и базовый уровень приоритета которых равен 13, следует с большой осторожностью. Обычно это процессы, которые большую часть времени ожидают какого-либо события. В Windows NT таким процессом является Task Manager, ожидающий нажатия клавиш <Ctrl>+<Esc>, чтобы активизироваться и позволить, например, исключение зависшего процесса из списка выполняющихся. Естественно, что класс процесса программы Task Manager и определяется ее назначением.

Класс Realtime

Базовый уровень приоритета процессов класса *Realtime* равен 24. Если процессы предыдущего класса рекомендуется создавать только при большой необходимости, то процессы класса *Realtime* вообще практически никогда не стоит создавать, кроме двух случаев: во-первых, процессами этого класса могут быть лишь программы, напрямую общающиеся с аппаратурой, во-вторых, если приложение выполняет операцию, которую ни в коем случае нельзя прерывать. Что касается последнего, то таких случаев не так уж и много.

Относительные приоритеты потоков

По умолчанию создаваемый поток получает фактический приоритет в соответствии с классом своего процесса. После создания потока его приоритет может изменять не только операционная система, но и приложение. В тоже время можно назначать только относительный приоритет потока, задаваемый в соответствии с классом процесса. Для установки относительного приоритета потока можно использовать, например функцию *SetThreadPriority*.

```
BOOL SetThreadPriority(
   HANDLE hThread,
   int nPriority);
```

hThread — дескриптор потока; *nPriority* — относительный приоритет потока.

В табл. 2 перечислены возможные значения параметра *nPriority* функции *SetThreadPriority*.

Таблица 2. *Относительные приоритеты потоков*

Относительный приоритет потока	Значение параметра	Описание
Idle (простаивающий)	THREAD_PRIORITY_IDLE	Приоритет потока равен 1 для классов Idle, Normal и High и 16 для процессов Realtime
Lowest (самый низкий)	THREAD_PRIORITY_LOWEST	Приоритет потока на 2 меньше базового уровня приоритета класса процесса
Below normal (ниже нормального)	THREAD_PRIORITY_BELOW_NORMAL	Приоритет потока на 1 меньше базового уровня приоритета класса процесса
Normal (нормальный)	THREAD_PRIORITY_NORMAL	Приоритет потока равен базовому уровню приоритета класса процесса
Above normal (выше нормального)	THREAD_PRIORITY_ABOVE_NORMAL	Приоритет потока на 1 больше базового уровня приоритета класса процесса
Highest (наивысший)	THREAD_PRIORITY_HIGHEST	Приоритет потока на 2 больше базового уровня приоритета класса процесса
Time critical (критичный по времени)	THREAD_PRIORITY_TIME_CRITICAL	Приоритет потока равен 15 для классов Idle, Normal и High и 31 для процессов Realtime

Примечания. 1) Самый низкий (нулевой) приоритет присваивается только одному системному потоку, задача которого заключается в обнулении страниц файла подкачки. Ни один другой поток не может иметь такой приоритет. 2) Потоки процесса, относящегося к классу Realtime, не могут иметь приоритет менее 16, а остальные — более 15.

Ранее уже упоминалось о том, что система может динамически менять приоритеты некоторых потоков. Для уменьшения времени реакции приложения, например, на действия пользователя, операционная система может повысить текущий приоритет потока. Такой приоритет называется *динамическим приоритетом потока* (thread's dynamic priority) и может принимать значения не больше 15. После нажатия клавиши приоритет потока на определенное время возрастает, чтобы приложение как можно быстрее обработало соответствующее сообщение. Затем постепенно приоритет потока вернется к своему первоначальному уровню — *базовому уровню приоритета потока* (thread's base priority level).

Приоритеты потоков процессов класса *Realtime* динамически никогда не меняются. Это позволяет разделять все процессы (и потоки) на обычные — с приоритетами от 0 до 15 включительно, и реального времени — с приоритетами от 16 до 31 включительно.

Давайте посмотрим, как система определяет фактический приоритет потока, комбинируя класс приоритета процесса с относительным приоритетом потока.

***Таблица 3**. Фактический приоритет потока*

Относительный приоритет потока	Класс процесса					
	Idle	**Normal в фоновом режиме**	**Normal в режиме (+1)**	**Normal в режиме (+2)**	**High**	**Realtime**
Time critical (критичный по времени)	15	15	15	15	15	31
Highest (наивысший)	6	9	10	11	15	26
Above normal (выше нормального)	5	8	9	10	14	25
Normal (нормальный)	4	7	8	9	13	24
Below normal (ниже нормального)	3	6	7	8	12	23
Lowest (самый низкий)	2	5	6	7	11	22
Idle (простаивающий)	1	1	1	1	1	16

Система поддерживает два вида потоков: *интерфейсные потоки*, например, главный поток процесса, и *рабочие потоки*, реализуемые единственной функцией, завершение которой приводит и к завершению потока.

Каждый поток процесса может прекратить свое существование в трех случаях:

1. Поток самостоятельно завершается.
2. Поток процесса может быть принудительно завершен любым другим потоком этого же процесса. При определенных условиях вообще любой поток может вмешаться в работу всех потоков в системе. Необходимо лишь знать "имя" и, для Windows NT, иметь достаточно прав на это.
3. По завершении процесса все потоки, связанные с ним, будут удалены, т. е. завершены операционной системой, поэтому можно не беспокоиться, что через некоторое время работы система деградирует настолько, что придется перезагружать компьютер.

В заключение следует сказать, что большинство создаваемых потоком объектов считается собственностью процесса, которому принадлежит данный поток. Все блоки памяти и объекты GDI, созданные потоком принадлежат процессу. Большая же часть объектов модуля User, кроме пиктограмм, курсоров и оконных классов, принадлежит потоку. Принадлежность объекта процессу в целом делает его доступным всем потокам процесса. Многозадачность системы в свою очередь дает возможность одновременного обращения к такому "разделяемому" объекту несколькими потоками. (Даже в однопроцессорной системе, где допустимо лишь последовательное переключение между потоками, возможна ситуация, когда один процесс частично изменяет, например, блок памяти, затем происходит переключение на другой поток, который читает этот блок памяти целиком, где части данных не соответствуют друг другу). Далее мы коснемся вопроса синхронизации действий потоков.

Взаимодействие программ

Любая многозадачная система будет ущербна, если она позволяет только одновременное выполнение нескольких программ. Конечно и это удобно, но именно взаимодействие открывает и перед разработчиками, и перед пользователями поистине широчайшие возможности использования компьютера в различных областях. Многопотоковые системы, которыми являются Windows 95 и Windows NT, кроме того, просто немыслимы без возможности взаимодействия отдельных потоков и в рамках одного и того же процесса.

Для взаимодействия могут быть использованы уже имеющиеся средства, такие как механизм сообщений, DDE и OLE. Однако назначение последних двух механизмов прежде всего в организации обмена данными между приложениями, а организация взаимодействия на их основе скорее вторич-

на, да и к тому же реализована на очень высоком уровне. Кроме того, оба механизма, особенно OLE, требуют существенных затрат ресурсов и довольно сложны в использовании, что может быть неприемлемо для ряда задач.

Далее мы рассмотрим то, что принципиально отличает Win32 API от своего 16-разрядного предшественника как раз в области организации и поддержки взаимодействия программ.

Синхронизация потоков

Так как собственно выполняются не процессы, а потоки, есть смысл говорить лишь о взаимодействии и синхронизации именно потоков, принадлежащих как одному процессу, так и разным. Только в одном случае можно упомянуть о взаимодействии именно процессов – когда один процесс ожидает завершения другого. При этом дескриптор процесса является объектом синхронизации, о чем будет подробно рассказано ниже.

В этом разделе мы кратко остановимся на механизмах, позволяющих синхронизировать действия потоков (принадлежащих как одному, так и различным процессам).

Система Windows в настоящее время поддерживает широкий набор средств, обеспечивающих синхронизацию. В любой многозадачной системе встает проблема синхронизации, если система позволяет одновременный доступ к объектам. Вся теория синхронизации была разработана в 60-х—начале 70-х годов[1] Деккером (Dekker), Дейкстра (Dijkstra) и др.

Механизм синхронизации, реализованный в Windows, достаточно прост и гибок в использовании, требует минимальных затрат ресурсов вычислительной системы и совершенно необходим в работе! Механизм синхронизации включает в себя критические секции, защищенный доступ к переменным, набор ожидающих (блокирующих) функций, объекты синхронизации, асинхронный ввод/вывод.

Критические секции

Критической секцией называется участок кода, в котором поток получает доступ к разделяемому ресурсу (обычно некоторая переменная), доступному другим потокам. Критическую секцию имеет смысл создавать, если она будет использоваться не менее двух раз — тогда, когда доступ к одному и тому же ресурсу одновременно требуется разным потокам.

Объект "критическая секция" обеспечивает синхронизацию так же, как и объект "взаимоисключение" (см. ниже), однако он может использоваться

[1] Подробнее см. Дейтел "Введение в операционные системы", т. 1.

лишь в рамках одного процесса. Использование критических секций обеспечивает немного более быстрый и более эффективный механизм синхронизации. Как и взаимоисключением, объектом "критическая секция" одновременно владеет только один поток, что обеспечивает защиту разделяемого ресурса.

Win32 API поддерживает специальный тип данных для критических секций — CRITICAL_SECTION, две функции для их создания и удаления и две функции для их использования.

```
CRITICAL_SECTION section;
```

Функция *InitializeCriticalSection* используется для инициализации объекта "критическая секция".

```
VOID InitializeCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

lpCriticalSection — адрес переменной типа CRITICAL_SECTION.

Функция *DeleteCriticalSection* используется для освобождения объекта "критическая секция".

```
VOID DeleteCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

lpCriticalSection — адрес переменной типа CRITICAL_SECTION.

Функция *EnterCriticalSection* используется для запроса на вход в критическую секцию.

```
VOID EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

lpCriticalSection — адрес переменной типа CRITICAL_SECTION.

Функция *LeaveCriticalSection* используется для уведомления о выходе из критической секции.

```
VOID LeaveCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection);
```

lpCriticalSection — адрес переменной типа CRITICAL_SECTION.

Для использования критической секции достаточно создать переменную типа CRITICAL_SECTION, а перед использованием проинициализировать ее с помощью функции *InitializeCriticalSection*. Для запроса входа в критическую секцию поток должен вызывать функцию *EnterCriticalSection*, а по завершении — функцию *LeaveCriticalSection*. Если другой процесс находится в своем критическом участке кода, то вызов функции *EnterCriticalSection* вызовет блокировку потока до тех пор, пока другой процесс не покинет свой критический участок.

Важно знать, что повторные запросы на вход в критическую секцию не вызовут самоблокировку потока и достаточно однократного вызова *LeaveCriticalSection* для выхода из критической секции. С другой стороны вход в критическую секцию окажет влияние только на потоки, также запросившие вход в свою критическую секцию, остальные потоки продолжают нормально выполняться.

Любой поток может, используя вызов функции *DeleteCriticalSection*, освободить соответствующий системный ресурс, захваченный при инициализации объекта, после чего объект "критическая секция" уже не может быть использован для синхронизации.

Защищенный доступ к переменным

Win32 API предоставляет три функции для защищенного доступа к 32-разрядным переменным. Функции *InterlockedIncrement* и *InterlockedDecrement* соответственно увеличивают и уменьшают значение переменной. Функция *InterlockedExchange* устанавливает защищенную переменную в заданное значение. Эти функции чрезвычайно полезны для работы, т. к. скрытым от программиста путем обеспечивают тот же эффект, что и использование критических секций.

Функция *InterlockedIncrement* обеспечивает увеличение защищенной переменной и возвращает ноль, если новое значение переменной равно нулю, или возвращает число с тем же знаком, что и у переменной, при этом сами значения могут не совпадать.

```
LONG InterlockedIncrement(LPLONG lpAddend);
```

lpAddend — адрес защищенной переменной.

Функция *InterlockedDecrement* обеспечивает увеличение защищенной переменной и возвращает ноль, если новое значение переменной равно нулю, или возвращает число с тем же знаком, что и у переменной, при этом сами значения могут не совпадать.

```
LONG InterlockedDecrement(LPLONG lpAddend);
```

lpAddend — адрес защищенной переменной.

Функция *InterlockedExchange* устанавливает новое значение защищенной переменной и возвращает ее предыдущее значение.

```
LONG InterlockedExchange(
  LPLONG Target,
  LONG Value);
```

Target — адрес защищенной переменной; *Value* — новое значение для защищенной переменной.

Ожидающие функции

Win32 API имеет целый ряд функций, позволяющих потоку блокировать себя в ожидании некоторых событий. Их общее название — *ожидающие функции* (имена всех таких функций начинаются с префикса *Wait*). Они переводят поток в состояние блокировки до того, как либо произойдет некоторое событие, либо истечет заданный промежуток времени (таймаут). Событиями выступают один или несколько синхронизирующих объектов.

В момент вызова одной из ожидающих функций проверяется состояние указанных синхронизирующих объектов. Таких состояний два: "установлен" (signaled) и, соответственно, "сброшен" (nonsignaled). Если ни один из объектов не установлен и таймаут не истек, поток переводится в состояние блокировки, при этом накладные расходы операционной системы, в данном случае — расходы процессорного времени на ожидание возникновения события (событий) для разблокировки потока, минимальны.

Ожидающих функций пять: *WaitForSingleObject*, *WaitForSingleObjectEx*, *WaitForMultipleObjects*, *WaitForMultipleObjectsEx* и *MsgWaitForMultipleObjects*. В большинстве случаев, если соответствующая функция завершается в связи с тем, что объект был установлен, он опять переводится в состояние "сброшен":

1. Счетчик семафора уменьшается на единицу, и если он станет равным нулю, семафор будет в состоянии "сброшен".
2. Объекты "взаимоисключение", автоматически сбрасываемые события и уведомления об изменении файлов переводятся в состояние "сброшен".
3. Состояние остальных объектов синхронизации не меняется.

Функции *WaitForSingleObject*, *WaitForSingleObjectEx* предназначены для ожидания установки одного синхронизирующего объекта или окончания таймаута.

Функции *WaitForMultipleObjects*, *WaitForMultipleObjectsEx* и *MsgWaitForMultipleObjects* предназначены для ожидания установки одного или сразу всех синхронизирующих объектов, заданных массивом дескрипторов, или окончания таймаута. Если любая из этих функций ожидает установки сразу всех синхронизирующих объектов, то состояние указанных объектов не сбрасывается до тех пор, пока все объекты не будут установлены. Такая реализация снимает проблему возникновения *тупиковой ситуации* (deadlock), когда несколько потоков ожидают друг от друга освобождения ресурсов.

Функции *WaitForSingleObjectsEx* и *WaitForMultipleObjectsEx* отличаются от других, т. к. они могут осуществлять возобновление потока при завершении асинхронного ввода/вывода, подробно на котором в данном томе мы останавливаться не будем.[1]

---

[1] Об асинхронном вводе/выводе см. книгу Джеффри Рихтера "Windows для профессионалов".

Функция *MsgWaitForMultipleObjects* в основном аналогична функции *WaitForMultipleObjects*, за исключением того, что поток может возобновить выполнение при появлении в его очереди сообщения заданного типа, например, от клавиатуры или мыши.

В заключение нельзя не сказать о функции *WaitCommEvent*, предназначенной для осуществления обмена данными через последовательные порты. Данная функция по сути схожа с вышеперечисленными, за исключением того, что в качестве объекта синхронизации выступает сигнал от последовательного порта.

Объекты синхронизации

Итак, механизм синхронизации описан. Теперь необходимо сказать от том, какие же объекты предоставляет операционная система для синхронизации. Вот они.

1. *Процессы* как синхронизирующие объекты используются для сигнализации о их завершении.
2. *Потоки* как синхронизирующие объекты используются для сигнализации о их завершении.
3. *Файлы*, *коммуникационные устройства* и *каналы обмена и передачи данных* могут выступать в качестве синхронизирующих объектов в режиме асинхронного ввода/вывода.
4. *Уведомления об изменении файлов* используются для обновления списков директориев, окон редакторов и т. д.
5. *Консольный ввод* используется для уведомления о наличии данных во входном буфере консоли.
6. *Взаимоисключения* (mutexes) позволяют координировать взаимное исключение доступа к разделяемому ресурсу.
7. *События* (events) используются для уведомления одного или нескольких потоков о наступлении некоторого события.
8. *Семафоры* (semaphores) поддерживают счетчики, значения которых меняются от нуля до задаваемых максимальных значений.

Из списка видно, что набор объектов синхронизации достаточно широк и разнообразен. Важность использования первых пяти объектов поясним на примере. Ожидание одним потоком завершения другого при помощи объекта "событие" слишком громоздко, к тому же установка соответствующего события перед завершением потока еще не означает, что поток завершился. Гораздо проще ожидать завершение потока, используя его дескриптор. В равной степени это относится и к процессам.

Далее подробно рассмотрим объекты синхронизации, назначение которых только в синхронизации и состоит:

1. Взаимоисключения
2. События
3. Семафоры

Взаимоисключения, события и семафоры могут быть как безымянными, так и поименованными. Первые служат для синхронизации потоков одного процесса, вторые — для синхронизации потоков различных процессов. В качестве имени используется строка символов. Доступ к поименованному объекту синхронизации из другого процесса осуществляется только по имени, другого способа (например, по дескриптору) не существует.

Взаимоисключения

Объект "взаимоисключение" позволяет координировать взаимное исключение доступа к разделяемому ресурсу. Объект имеет два состояния — "установлен" и "сброшен".

Объект "взаимоисключение" установлен в момент времени, когда обеспечение доступа к разделяемому ресурсу не требуется, говорят, что такой объект *не принадлежит* ни одному потоку, и, наоборот, объект сброшен, когда один из потоков уже осуществляет доступ к нему или *владеет* объектом. Таким образом доступ разрешается, когда другие потоки уже освободили разделяемый ресурс. Как упоминалось выше, разрешение на доступ (т. е. завершение одной из ожидающих функций) одновременно сбрасывает состояние объекта, приостанавливая таким образом все последующие запросы. По завершении работы с разделяемым ресурсом следует освободить объект "взаимоисключение", т. е. опять перевести его в состояние "установлен".

Для объявления переменной, именующей объект в приложении, используется тип HANDLE, т. к. необходимо лишь место для хранения дескриптора создаваемого впоследствии объекта. Для работы с объектом "взаимоисключение" Win32 API предоставляет функции: для создания — *CreateMutex*, для доступа к поименованному объекту — *OpenMutex* и для сигнализации об окончании процесса взаимоисключения — *ReleaseMutex*.

Для того чтобы объявить существующий объект "взаимоисключение" принадлежащим потоку, следует вызвать одну из ожидающих функций. По завершении работы с описываемым объектом достаточно освободить дескриптор, вызвав функцию *CloseHandle*.

Поток, которому принадлежит в данный момент объект "взаимоисключение", может и далее использовать его в ожидающих функциях, не вызывая блокировку самого себя.

Функция *CreateMutex* создает безымянный или поименованный объект "взаимоисключение" и, в случае успеха, возвращает его дескриптор, иначе значение NULL.

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName);
```

lpMutexAttributes — указатель на атрибуты защиты; *bInitialOwner* — начальное состояние объекта; *lpName* — указатель на строку символов, содержащую имя объекта, NULL — для безымянных объектов.

Функция *OpenMutex* возвращает дескриптор существующего объекта "взаимоисключение" и, в случае успеха, возвращает его дескриптор, иначе значение NULL.

```
HANDLE OpenMutex(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

dwDesiredAccess — флаг доступа; *bInheritHandle* — флаг наследования; *lpName* — указатель на строку символов, содержащую имя объекта.

Функция *ReleaseMutex* переводит объект "взаимоисключение" в состояние "установлен", сигнализируя о том, что доступ к разделяемому ресурсу завершен. При возникновении ошибки возвращает значение FALSE, иначе TRUE.

```
BOOL ReleaseMutex(HANDLE hMutex);
```

hMutex — дескриптор объекта "взаимоисключения".

События

Событие (event) применяется для уведомления одного или нескольких ожидающих (блокированных) потоков.

Существует два типа событий. Сброс событий первого типа осуществляется вручную функцией *ResetEvent*. События этого типа используются для уведомления нескольких потоков. События второго типа — *автоматически сбрасываемые события* — сбрасываются ожидающими функциями.

Для объявления переменной, именующей объект, в приложении используется тип HANDLE, т. к. необходимо лишь место для хранения дескриптора создаваемого впоследствии объекта. Для работы с объектом "событие" Win32 API предоставляет функцию для создания — *CreateEvent*, функцию для доступа к поименованному объекту — *OpenEvent*, две функции для установки события — *SetEvent* и *PulseEvent* и функцию для сброса события — *ResetEvent*. Для ожидания возникновения события используются ожидающие функции. По завершении работы с описываемым объектом достаточно освободить дескриптор, вызвав функцию *CloseHandle*.

Особо следует сказать о функции *PulseEvent*. Эта функция устанавливает событие и затем переводит его в сброшенное состояние после возобновле-

ния заблокированных потоков. Для объектов с ручным сбросом все ожидающие потоки разблокируются. Для автоматически сбрасываемых — разблокируется только один поток, даже если в состоянии ожидания или блокировки находятся несколько потоков. Если ожидающих потоков нет, функция *PulseEvent* сбрасывает событие и завершается.

Функция *CreateEvent* создает безымянный или поименованный объект "событие" и, в случае успеха, возвращает его дескриптор, иначе значение NULL.

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName);
```

lpEventAttributes — указатель на атрибуты защиты; *bManualReset* — тип объекта; *bInitialState* — начальное состояние объекта; *lpName* — указатель на строку символов, содержащую имя объекта, NULL — для безымянных объектов.

Функция *OpenEvent* возвращает дескриптор существующего объекта "событие" и, в случае успеха, возвращает его дескриптор, иначе значение NULL.

```
HANDLE OpenEvent(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

dwDesiredAccess — флаг доступа; *bInheritHandle* — флаг наследования; *lpName* — указатель на строку символов, содержащую имя объекта.

Функция *SetEvent* устанавливает объект "событие", заданный дескриптором. При возникновении ошибки возвращает значение FALSE, иначе TRUE.

```
BOOL SetEvent(HANDLE hEvent);
```

hEvent — дескриптор события.

Функция *PulseEvent* устанавливает и сбрасывает объект "событие", заданный дескриптором. При возникновении ошибки возвращает значение FALSE, иначе TRUE.

```
BOOL PulseEvent(HANDLE hEvent);
```

hEvent — дескриптор события.

Функция *ResetEvent* сбрасывает объект "событие", заданный дескриптором. Применяется для сбрасываемых вручную событий. При возникновении ошибки возвращает значение FALSE, иначе TRUE.

```
BOOL ResetEvent(HANDLE hEvent);
```

hEvent — дескриптор события.

Семафоры

Семафор (semaphore) — это объект синхронизации, реализованный в виде счетчика, принимающего значения от нулевого до задаваемого максимального. Семафоры обычно применяются для ограничения количества потоков, одновременно обращающихся за набором разделяемых ресурсов. Например, программа поддерживает пять ресурсов одного типа, которые кратковременно могут быть выделены потокам. Как только все пять ресурсов будут выделены, следующий поток, запрашивающий ресурс данного типа будет приостановлен до момента высвобождения хотя бы одного ресурса.

В установленном состоянии семафор находится в том случае, если значение его счетчика больше нуля, в противном случае он "сброшен". Ожидающие функции, использующие семафор в качестве объекта синхронизации, уменьшают значение счетчика. Поток может последовательно запрашивать семафор, используя ожидающую функцию. Но использование функции *WaitForMultipleObjects* (или подобной), у которой в качестве массива объектов задан массив дескрипторов одного и того же семафора, уменьшит значение счетчика последнего только на единицу.

Win32 API поддерживает следующие функции для использования семафоров: *CreateSemaphore* — создает семафор с заданными начальным значением счетчика и максимальным значением, которое он может получить; *OpenSemaphore* — осуществляет доступ к поименованному семафору; *ReleaseSemaphore* — увеличивает значение счетчика на заданную величину (значение счетчика не может быть меньше нуля и больше максимального значения); по завершении работы с описываемым объектом достаточно освободить дескриптор, вызвав функцию *CloseHandle*.

Функция *CreateSemaphore* создает безымянный или поименованный семафор и, в случае успеха, возвращает его дескриптор, иначе NULL.

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName);
```

lpEventAttributes — указатель на атрибуты защиты; *lInitialCount* — начальное значение счетчика; *lMaximumCount* — максимальное значение счетчика; *lpName* — указатель на строку символов, содержащую имя объекта, NULL — для безымянных объектов.

Функция *OpenSemaphore* возвращает дескриптор существующего семафора в случае успеха, иначе значение NULL.

```
HANDLE OpenSemaphore(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

dwDesiredAccess — флаг доступа; *bInheritHandle* — флаг наследования; *lpName* — указатель на строку символов, содержащую имя объекта.

Функция *ReleaseSemaphore* увеличивает значение счетчика, заданного дескриптором, на заданную величину и передает предыдущее значение счетчика. Возвращает значение FALSE при возникновении ошибки или возможном превышении счетчиком максимального значения (при этом значение счетчика остается без изменений), иначе TRUE.

```
BOOL ReleaseSemaphore(
  HANDLE hEvent,
  LONG lReleaseCount,
  LPLONG lpPreviousCount);
```

hEvent — дескриптор события; *lReleaseCount* — число для увеличения счетчика, должно быть больше нуля; *lpPreviousCount* — адрес переменной, куда будет записано предыдущее значение счетчика, или NULL, если в получении значения нет необходимости.

APISync — пример приложения

Здесь приведена программа, в которой мы постарались отразить значительную часть описанных в главе функций и механизмов, связанных с процессами и потоками, не делая пример уж очень искусственным. Хотим отметить, что предлагаемая программа написана без использования средств автоматической генерации программного текста, таких как AppWizard или ClassWizard.

Основная цель примера — проиллюстрировать приемы обнаружения запущенной ранее копии приложения, запуск нового процесса и нескольких потоков, а также показать различные варианты синхронизации процессов и потоков. Задача последних — захватить определенное число ресурсов.

Подробные комментарии в тексте программы делают излишним описание алгоритма ее работы.

```
/*************************************************************
Модуль APISync
Copyright (c) 1997 Мешков А., Тихомиров Ю.
*************************************************************/

#include <windows.h>          /* описание функций Win32 API */

#define NUMBER_OF_THREADS   10
#define NUMBER_OF_RESOURCES 10

/* массив для хранения имени модуля */
TCHAR szModuleName[_MAX_PATH];

/* массивы для текстовых сообщений */
TCHAR szMessage[64 * NUMBER_OF_THREADS];
TCHAR szMessageCode[64];
```

```
/* дескрипторы событий и семафоров */
HANDLE hEmpty, hQuit, hResource;

/* объект "критическая секция" */
CRITICAL_SECTION section;

/* массив дескрипторов потоков */
HANDLE hThread[NUMBER_OF_THREADS];

/* массив для хранения количества полученных ресурсов */
DWORD dwCount[NUMBER_OF_THREADS];

/* количество ресурсов */
DWORD dwResources = NUMBER_OF_RESOURCES;

/* описание потоковой функции */
DWORD ThreadFunc(LPVOID lParam);

/* главная функция приложения */
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
  int i;
  DWORD dwWait, dwExit, dwThreadId;
  PROCESS_INFORMATION pi;
  STARTUPINFO si;

  /* проверяем не была ли уже запущена копия приложения,
     будем использовать этот объект и для инициирования
     завершения потоков */
  hQuit = CreateEvent(
    NULL,         /* используем атрибуты защиты по умолчанию */
    TRUE,              /* создаем события с ручным сбросом */
    FALSE,              /* начальное состояние - "сброшен" */
    "Abracadabra"                          /* уникальное имя */
    );

  /* если объект "событие" уже создан ранее -
     запущена другая копия приложения -
     завершаем выполнение с кодом 1 */
  if (GetLastError() == ERROR_ALREADY_EXISTS)
    return (1);

  /*
    запускаем копию процесса
  */

  /* заполняем структуру STARTUPINFO */
  GetStartupInfo(&si);
```

```
/* получаем имя исполняемого модуля */
GetModuleFileName(NULL, szModuleName, _MAX_PATH);

/* создаем копию процесса */
CreateProcess(
  NULL,                    /* имя модуля приложения передадим
                                  во втором параметре */
  szModuleName,                  /* командная строка процесса */
  NULL,  /* используем атрибуты защиты процесса по умолчанию */
  NULL,    /* используем атрибуты защиты потока по умолчанию */
  FALSE,                          /* дескриптор ненаследуемый */
  CREATE_DEFAULT_ERROR_MODE, /* стандартная обработка ошибок */
  NULL,                      /* переменные окружения не нужны */
  NULL,                      /* текущий каталог не используем */
  &si,   /* передаем копию собственной структуры STARTUPINFO */
  &pi   /* получим дескриптор и другую информацию о процессе */
  );

/*
  дочерний процесс должен завершиться немедленно,
  т. к. уже загружена его копия
*/

/* ожидаем завершения дочернего процесса */
dwWait = WaitForSingleObject(pi.hProcess, 5000);

/*
  проводим диагностику результата
  завершения дочернего процесса
*/
if (dwWait == WAIT_OBJECT_0)
{
  GetExitCodeProcess(pi.hProcess, &dwExit);
  wsprintf(szMessage,
    "Копия процесса завершена.\nКод завершения: %d",
    dwExit);
  MessageBox(NULL, szMessage, "APISync", MB_OK);
}
else if (dwWait == WAIT_TIMEOUT)
  MessageBox(NULL, "дочерний процесс не завершился"
    "по истечении 5 секунд", "APISync", MB_OK);
else
            /* этот код никогда выполняется */

/*
  объект используется для сигнализации
  о том, что ресурсы исчерпаны
*/
hEmpty = CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
/* создаем семафор для синхронизации доступа к ресурсам */
hResource = CreateSemaphore(
  NULL,         /* используем атрибуты защиты по умолчанию */
  NUMBER_OF_RESOURCES,      /* начальное значение счетчика */
  NUMBER_OF_RESOURCES,  /* максимальное значение счетчика */
  NULL                        /* создаем объект безымянным */
  );

/* инициализируем критическую секцию для защиты переменной,
   содержащей количество свободных ресурсов */
InitializeCriticalSection(&section);
/* создаем все потоки */
for (i = 0; i < NUMBER_OF_THREADS; i++)
  hThread[i] = CreateThread(
    NULL,       /* используем атрибуты защиты по умолчанию */
    0,                            /* начальный размер стека,
                                  как и у создающего потока */
    (LPTHREAD_START_ROUTINE)ThreadFunc,       /* указатель
                                          на функцию потока */
    &dwCount[i],          /* адрес переменной для хранения
                            количества выделенных ресурсов */
    (i % 3) ? 0 : CREATE_SUSPENDED,       /* часть потоков
                                    создаем, но не запускаем */
    &dwThreadId     /* по завершении получим идентификатор
                                  потока, хотя в дальнейшем
                                          не используем его */
    );

/*
  ждем момента, когда все ресурсы будут
  исчерпаны или истекут 5 секунд
*/
WaitForSingleObject(hEmpty, 5000);

/* инициируем окончание потоков */
PulseEvent(hQuit);

/* ждем завершения потоков, но не более 5 секунд */
WaitForMultipleObjects(
  NUMBER_OF_THREADS, /* количество синхронизирующих объектов */
  hThread,                              /* массив дескрипторов
                                     синхронизирующих объектов */
  TRUE,           /* ждем установки всех указанных объектов, */
  5000                                /* но не более 5 секунд */
  );

/*
  формируем протокольное сообщение:
    - заголовок
```

```
      - номер потока, код возврата и значение переменной,
        хранящей количество выделенных потоку ресурсов
        (должны быть равны - печатаем для проверки)
      - количество свободных ресурсов
 */
 lstrcpy(szMessage, "Протокол завершения потоков.\n\n");

  for (i = 0; i < NUMBER_OF_THREADS; i++)
  {
    /* получаем код возврата, содержащий
      количество выделенных потоку ресурсов */
    GetExitCodeThread(hThread[i], &dwExit);

    /* часть потоков могла не завершиться:
      одни потоки находились в приостановленном состоянии,
      другие в момент вызова функции PulseEvent
      еще не вызвали в очередной раз ожидающую
      функцию WaitForMultipleObjects - удаляем их */
    if (dwExit == STILL_ACTIVE)
    {
      /* принудительно удаляем поток */
      TerminateThread(hThread[i], 0);

      if ((i % 3) == 0) /* такие потоки были приостановлены */
        wsprintf(szMessageCode, "\tПоток %d - удален\n", i);
      else
        wsprintf(szMessageCode,
          "\tПоток %d - удален (ресурсов %d)\n",
          i, dwCount[i]);
    }
    else
      wsprintf(szMessageCode,
        "\tПоток %d - ресурсов %d (%d)\n",
        i, dwExit, dwCount[i]);
    lstrcat(szMessage, szMessageCode);
  }

  wsprintf(szMessageCode,
    "\nСвободных ресурсов %d\n", dwResources);

  lstrcat(szMessage, szMessageCode);
  MessageBox(NULL, szMessage, "Win32® API", MB_OK);

  /* удаляем синхронизирующие объекты */
  DeleteCriticalSection(&section);

  CloseHandle(hResource);
  CloseHandle(hQuit);

  /* нормальное завершение процесса */
  return (0);
}
```

```
/*
    функция потока ожидает освобождения ресурса или
    сигнала о завершении, если потоку уже выделено пять и
    более ресурсов - освобождаем все
*/
DWORD ThreadFunc(LPVOID lParam)
{
  DWORD dwWait;

  /* преобразуем параметр в указатель на переменную
    для хранения количества выделенных потоку ресурсов */
  LPDWORD lpdwCount = (LPDWORD)lParam;

  /*
    формируем массив синхронизирующих объектов:
      - событие для сигнализации о
          необходимости завершения
      - дескриптор семафора, синхронизирующего
          выделение ресурсов
  */
  HANDLE hHandle[] = { hQuit, hResource };
  *lpdwCount = 0;

  for (;;)
  {
    dwWait = WaitForMultipleObjects(
      2,            /* количество синхронизирующих объектов */
      hHandle,                        /* массив дескрипторов
                                      синхронизирующих объектов */
      FALSE,            /* ждем установки хотя бы одного из
                                          указанных объектов */
      INFINITE                /* ограничения по времени нет */
  );

    if (dwWait == WAIT_OBJECT_0)       /* hQuit - установлен */
      break;
    else                                  /* получили ресурс */
    {
      if ((*lpdwCount + 1) >= 5)      /* освобождаем ресурсы */
      {
        /* изменяем переменную, содержащую
           количество свободных ресурсов */
        EnterCriticalSection(&section);

        /* увеличиваем значение счетчика семафора */
        ReleaseSemaphore(hResource, *lpdwCount + 1, NULL);

        /* увеличиваем значение счетчика свободных ресурсов */
        dwResources += *lpdwCount;
```

```
        LeaveCriticalSection(&section);

        *lpdwCount = 0;
      }
      else                        /* закрепляем ресурс за собой */
      {
        *lpdwCount += 1;

        /* изменяем переменную, содержащую
           количество свободных ресурсов */
        EnterCriticalSection(&section);

        if (--dwResources == 0)
          SetEvent(hEmpty);              /* сигнализируем о
                                    том, что ресурсы исчерпаны */

        LeaveCriticalSection(&section);
      }
    }
  }
  /*
    завершаем поток и возвращаем
    количество захваченных ресурсов
  */
  return *lpdwCount;
}
```

После запуска второй копии приложения ожидается его немедленное завершение с кодом 1. Так действительно и происходит.

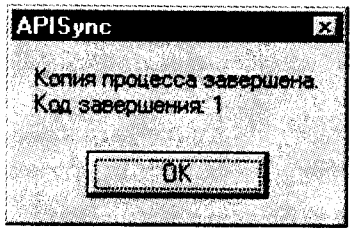


Рис. 12. Окно сообщения

Потоки с номерами 0, 3, 6 и 9 создаются с флагом CREATE_SUSPENDED и, соответственно, не выполняются. Естественно, что они не могут самостоятельно завершиться и захватить ни один ресурс. Потоки с номерами 1, 4, 7 и 8 в момент выполнения функции *PulseEvent*, видимо, были приостановлены в ожидании входа в свои критические участки кода, когда

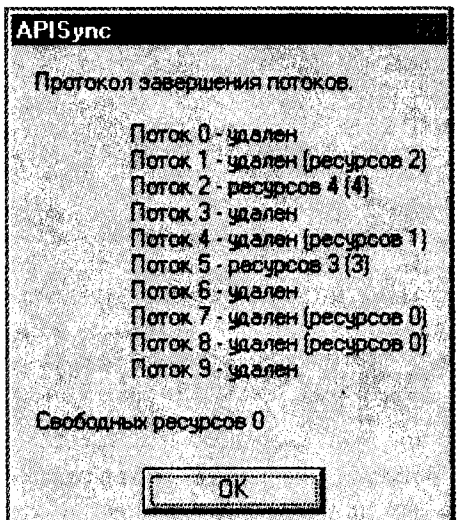


Рис. 13. Результат работы программы

и были принудительно уничтожены. Однако до этого 1-й и 4-й потоки все-таки получили часть ресурсов (то же может касаться 7-го и 8-го потоков, которые могли успеть освободить захваченные ресурсы).

Глава 6

Структура приложения Windows

Любое приложение Windows, имеющее интерфейс с пользователем (за исключением консольных приложений), состоит из двух основных частей: функции *WinMain* и так называемой функции окна. В главе 4 мы рассмотрели пример APIShell, в котором назначение функции *WinMain* заключалось в создании окна диалога. Здесь мы рассмотрим действия, которые должна выполнять функция *WinMain* приложения, поддерживающего однодокументный (SDI) или многодокументный (MDI) интерфейс. Вот стандартная, хотя и немного усеченная реализация такой функции.

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

  /* хотя параметр hPrevInstance в Win32 всегда равен NULL,
     продолжаем проверять его значение для совместимости */
  if (!hPrevInstance)
  {
    /* инициализируем приложение -
       подготавливаем данные класса окна и регистрируем его */
    if (!InitApplication(hInstance))
      return (FALSE);
  }

  /* завершаем создание копии приложения -
     создаем главное окно приложения */
  if (!InitInstance(hInstance, nCmdShow))
    return (FALSE);
```

```
  /* Цикл обработки сообщений */
  while (GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return (msg.wParam);
}
```

Функция состоит из трех частей: регистрации класса окна, создания главного окна приложения и цикла обработки сообщений. Используемые функции *InitApplication* и *InitInstance* не системные, они должны быть написаны разработчиком программы.

```
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;

  ...

  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)WndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, IDI_APLICATION);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wc.lpszMenuName  = NULL;
  wc.lpszClassName = szAppName;

  return RegisterClass(&wc);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;

  hWnd = CreateWindow(
    szAppName, szTitle, WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInstance, NULL);
  if (!hWnd)
    return (FALSE);

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);

  return (TRUE);
}
```

Стандартный цикл обработки сообщений

Стандартный цикл обработки сообщений имеет следующий вид:

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

Данный цикл работает в течение всего времени выполнения программы. Каждая итерация представляет извлечение одного сообщения из очереди сообщений текущего потока. За это отвечает функция *GetMessage*, второй аргумент которой (NULL) говорит о том, что следует обрабатывать все сообщения. Когда сообщение выбирается из очереди, оно помещается в специальную структуру *MSG*, которая выглядит следующим образом:

```
typedef struct tagMSG
{
  HWND   hwnd
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG;
```

hwnd — содержит дескриптор (логический номер) окна, оконная процедура которого получает сообщение; *message* — определяет номер сообщения; *wParam* и *lParam* — определяют дополнительную информацию о сообщении. Конкретное содержание этих параметров зависит от типа сообщения; *time* и *pt* — описывают состояние системы в момент постановки сообщения в очередь: *time* хранит время, а *pt* — позицию указателя мыши в этот момент.

После того как сообщение взято из очереди сообщений, оно передается в функцию *TranslateMessage*, которая вызывает драйвер клавиатуры Windows для преобразования виртуальных кодов клавиш в ASCII-значения, которые ставятся в очередь программных событий в виде сообщения WM_CHAR. Это позволяет программе отличить, например, "A" от "a" без анализа состояния клавиши регистра.

Последняя функция цикла — *DispatchMessage* — берет данные о сообщении из структуры *msg* и передает их в соответствующую оконную процедуру для обработки. После того как сообщение передано, снова вызывается функция *GetMessage*, чтобы взять из очереди следующее сообщение, если таковое имеется. За исключением WM_QUIT, каждое сообщение заставляет

GetMessage возвратить значение TRUE. Принимая WM_QUIT, программа выходит из цикла обработки сообщений и завершает работу.

Важно понимать, что этот, или подобный ему, цикл без конца повторяется в течение всей "жизни" приложения. Он составляет суть любого приложения Windows.

Теперь пришло время поговорить о "получателе" сообщений.

Оконная процедура

Для того чтобы программа могла взаимодействовать "с внешним миром", необходимо это каким-либо образом обеспечить. В любом приложении Windows роль получателя сообщений выполняет оконная процедура, которой передаются сообщения, поступающие из очереди приложения, и где они обрабатываются индивидуально:

```
LRESULT CALLBACK WindowProc(
  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;
}
```

В простейшем случае приложение явно отвечает только на сообщение WM_DESTROY, а все остальные передаются в *DefWindowProc* — функцию, управляющую поведением окна по умолчанию. Она "умеет" обрабатывать почти все виды поведения, ассоциированные с конкретным типом окна. Например, она "знает", как обрабатывать действия мыши, изменяющие форму и местоположение окна. Однако *DefWindowProc* не занимается сообщением WM_DESTROY, и поэтому вы должны сами обработать его. Строки

```
case WM_DESTROY:
  PostQuitMessage(0);
  break;
```

представляют собой не что иное, как простейший *обработчик сообщения*,[1] которые широко используются как при написании традиционных С-программ, так и программ, базирующихся на какой-либо библиотеке классов, будь то OWL[2] или MFC.

Теперь, когда вы познакомились с главной артерией, поддерживающей жизнь приложения Windows, мы хотим более подробно остановиться на типах сообщений, действующих в системе Windows.

Основные типы оконных сообщений Windows

В настоящее время в системе Windows имеется более 900 стандартных сообщений. К счастью, нет необходимости знакомиться со всеми, поскольку большинство из них используются очень редко. Например, большое количество сообщений характерно для одного типа окна. Другие сообщения играют особую роль в совместном использовании данных или при реализации стандарта многодокументного интерфейса (MDI). Третьи проходят по нашему циклу сообщений, но никогда не используются. Это внутренние сообщения, которые Windows создает для своих собственных целей.

Все сообщения имеют имена, которые определены в WINUSER.H следующим образом:

```
#define WM_COMMAND 0x0111
```

или

```
#define WM_PAINT   0x000F
```

Префикс WM_ говорит о том, что речь идет об "оконном сообщении" (Window Message). Эти символические имена обозначают числа. Кроме WM_ широко используются и другие префиксы: EM_ (Edit Message) — для сообщений элемента управления EDIT, BM_ (Button Message) — для сообщений элемента управления BUTTON, LM_ (Listbox Message) — для сообщений элемента управления LIST BOX и т. д. Помимо этого вы можете создавать и собственные сообщения. Однако в данной главе мы рассмотрим только оконные сообщения.

Чтобы как-то систематизировать такое огромное количество сообщений, мы разделили их на 8 категорий:

---

[1] На самом деле, обработчиком сообщения называют специальную функцию, которая отвечает за обработку одного единственного сообщения. При появлении файла WINDOWSX.H, содержащего макрос HANDLE_MSG, который осуществляет синтаксический анализ параметров, ассоциированных с любым сообщением, и передает их в отдельную процедуру, задача создания обработчиков сообщений значительно упростилась. Однако более подробное обсуждение этого вопроса выходит за рамки нашей книги.

[2] Библиотека классов компилятора Borland C++.

1. *Аппаратные* — входные данные от мыши, клавиатуры и таймера;
2. *Обслуживание окна* — уведомление, требование действия, запрос;
3. *Организация интерфейса пользователя* — меню, указатель мыши, полосы прокрутки, блоки диалога, MDI;
4. *Завершение работы* — закрытие приложения или системы;
5. *Частные* — элементы управления: кнопка, элемент редактирования и т. д.;
6. *Информация о системных ресурсах* — изменение цвета, шрифтов, буферизация печати, режимы работы устройства;
7. *Обмен данными* — буфер обмена и динамический обмен данными;
8. *Внутрисистемные* — недокументированные сообщения.

Аппаратные сообщения

Оконная процедура получает сообщения, инициируемые тремя частями аппаратного обеспечения: клавиатурой, мышью и системным таймером. Все они генерируют аппаратные прерывания. Поскольку диспетчеризация Windows не действует в режиме прерываний, аппаратные события должны быть буферизованы. Это обеспечивает обработку событий в порядке их поступления. Например, когда вы нажимаете клавишу <K>, прерывание уведомляет систему о том, что приготовлены данные от клавиатуры. Драйвер клавиатуры отыскивает эти данные и создает элемент в первичной очереди, откуда система их передает в очередь приложения, точнее, конкретного потока. Когда наступает очередь сообщения, содержащего информацию о клавише <K>, оно направляется в соответствующую оконную процедуру. Сообщения от мыши и системного таймера обрабатываются аналогично. Далее представлены основные сообщения, которые формируются как реакция на сообщения от аппаратного обеспечения.

Сообщения от мыши в клиентской области окна

WM_LBUTTONDBLCLK	Двойное нажатие левой кнопки
WM_LBUTTONDOWN	Нажатие левой кнопки
WM_LBUTTONUP	Отпускание левой кнопки
WM_MBUTTONDBLCLK	Двойное нажатие средней кнопки
WM_MBUTTONDOWN	Нажатие средней кнопки
WM_MBUTTONUP	Отпускание средней кнопки
WM_MOUSEMOVE	Перемещение мыши
WM_RBUTTONDBLCLK	Двойное нажатие правой кнопки
WM_RBUTTONDOWN	Нажатие правой кнопки
WM_RBUTTONUP	Отпускание правой кнопки

Сообщение WM_MOUSEMOVE заслуживает особого упоминания, т. к. обрабатывается особым образом, чтобы избежать переполнения очереди аппаратных событий. Дело в том, что при быстром перемещении мыши могут произойти сотни прерываний. Для того чтобы избежать переполнения очереди, система Windows хранит только одно сообщение о перемещении мыши в каждый момент времени. Когда приходит новое сообщение Windows проверяет, есть ли уже подобное. Если оно есть, то в нем обновляется информация о положении. Новое сообщение помещается в очередь сообщений только в том случае, если сообщения о перемещении мыши еще не было.

Сообщения от мыши в неклиентской области окна

WM_NCLBUTTONDBLCLK	Двойное нажатие левой кнопки
WM_NCLBUTTONDOWN	Нажатие левой кнопки
WM_NCLBUTTONUP	Отпускание левой кнопки
WM_NCMBUTTONDBLCLK	Двойное нажатие средней кнопки
WM_NCMBUTTONDOWN	Нажатие средней кнопки
WM_NCMBUTTONUP	Отпускание средней кнопки
WM_NCMOUSEMOVE	Перемещение мыши
WM_NCRBUTTONDBLCLK	Двойное нажатие правой кнопки
WM_NCRBUTTONDOWN	Нажатие правой кнопки
WM_NCRBUTTONUP	Отпускание правой кнопки

Сообщения от клавиатуры делятся на два основных типа: обычные и системные. Обычно системные сообщения от клавиатуры (начинающиеся на WM_SYS...) игнорируются, т. к. оконная процедура, устанавливаемая по умолчанию, превращает их в соответствующую системную команду. Подробно все сообщения, поступающие от клавиатуры, мы обсудим во втором томе, когда речь пойдет о получении входных данных от пользователя.

Сообщения от клавиатуры

WM_CHAR	Ввод символа
WM_DEADCHAR	Диакритический знак (умляут, ударение и т. д.)
WM_KEYDOWN	Была нажата клавиша
WM_KEYUP	Была отпущена клавиша
WM_SYSCHAR	Ввод специального символа
WM_SYSDEADCHAR	Специальный диакритический знак
WM_SYSKEYDOWN	Была нажата специальная клавиша
WM_SYSKEYUP	Была отпущена специальная клавиша

Сообщение от таймера

WM_TIMER	Истекло заданное время

Сообщение от таймера помещается в очередь так же, как и сообщения от мыши и клавиатуры (т. е. таймеры в системе Windows не являются точными!).

Сообщения обслуживания окна

Здесь собраны наиболее часто используемые сообщения, и вам следует хорошенько разобраться с каждым из них. К сожалению, имена не окажут вам большой помощи. Начнем с того, что сообщения этой группы можно разбить на три части: уведомление, требование действия и запрос.

Уведомление

WM_ACTIVATE	Окно активно
WM_ACTIVATEAPP	Приложение активно
WM_CREATE	Окно было создано
WM_DESTROY	Окно было уничтожено
WM_ENABLE	Был разрешен ввод в окно
WM_KILLFOCUS	Окно потеряло фокус ввода от клавиатуры
WM_MOUSEACTIVATE	Окно станет активным после нажатия кнопки мыши
WM_MOVE	Окно было перемещено
WM_SETFOCUS	Окно получило фокус ввода от клавиатуры
WM_SIZE	Изменились размеры окна

Требование действия

WM_CLOSE	Закрыть (уничтожить) окно
WM_ERASEBKGND	Удалить фон
WM_ICONERASEBKGND	Удалить фон окна, показываемого в виде пиктограммы
WM_NCACTIVATE	Изменить заголовок окна для показа активного состояния
WM_NCCREATE	Создать неклиентскую область окна
WM_NCDESTROY	Уничтожить неклиентскую область окна
WM_NCPAINT	Перерисовать неклиентскую область окна
WM_PAINT	Перерисовать клиентскую область окна
WM_PAINTICON	Перерисовать клиентскую область окна, показываемого в виде пиктограммы
WM_SETREDRAW	Запретить перерисовку окна
WM_SETTEXT	Изменить текст окна
WM_SHOWWINDOW	Изменить режим отображения окна

Запросы

WM_GETMINMAXINFO	Запрос информации о минимальном/максимальном размере окна
WM_GETTEXT	Запрос текста заголовка окна, а для окон, не имеющих заголовка — текста, размещенного в окне, например, для кнопок
WM_GETTEXTLENGTH	Запрос длины текста окна
WM_NCCALCSIZE	Запрос размеров клиентской области
WM_QUERYNEWPALETTE	Запрос существования новой палитры
WM_QUERYOPEN	Запрос возможности открытия окна, отображаемого в виде пиктограммы

Сообщение-уведомление говорит оконной процедуре о том, что состояние окна было изменено. При этом оконная процедура вовсе не должна выполнять действие, обозначенное именем сообщения. Например, сообщение WM_MOVE не является запросом на некоторое перемещение. Наоборот, оно является "сообщением после действия", и напоминает, что окно уже было перемещено. Обратите внимание, что из сообщения не ясно, каким образом произошло перемещение. Это могло быть результатом действий пользователя, равно как и другой программы. В любом случае, сообщения-уведомления представляют собой одностороннюю связь с оконной процедурой.

Сообщение-требование действия говорит само за себя, т. к. отсутствие необходимого действия вызывает ошибку в интерфейсе пользователя. Например, сообщение WM_PAINT посылается оконной процедуре, если окно необходимо перерисовать (обновить) из-за изменения данных, связанных с окном, или в результате активизации и перевода окна на передний план. Если оконная процедура не перерисует окно, то его "видимое" содержимое не будет актуальным. Обычно устанавливаемая по умолчанию оконная процедура обеспечивает некоторые необходимые минимальные действия, однако зачастую следует перехватывать и обрабатывать такого рода сообщения самому, как в случае с сообщением WM_PAINT. При этом нередко нужно воспроизвести все действия обработчика, устанавливаемого по умолчанию.

Сообщения-запросы требуют ответа. Они используются для двусторонней связи между Windows и вашей программой. Подобно сообщениям-требованиям действия, оконная процедура, устанавливаемая по умолчанию, даст приемлемый ответ в большинстве случаев. Их следует перехватывать, если вы хотите изменить ответ по умолчанию. Например, сообщение WM_QUERYOPEN вы получите, когда пользователь попытается открыть минимизированное окно. Если вы хотите, чтобы программа выполнялась только тогда, когда окно отображается в виде пиктограммы, то нужно просто вернуть FALSE вместо TRUE, что имеет место по умолчанию.

Сообщения об организации интерфейса пользователя

Эта группа содержит сообщения для других объектов пользовательского интерфейса, включающих меню приложения, указатель мыши, полосы прокрутки, блоки диалога и их элементы управления, а также сообщения, используемые для поддержки многодокументного интерфейса (MDI).

Сообщения меню

WM_COMMAND	Был выбран пункт меню
WM_INITMENU	Уведомление о возможности изменить меню перед его активизацией
WM_INITMENUPOPUP	Уведомление о возможности изменить пункты всплывающего меню перед его отображением на экране
WM_MENUCHAR	Для выбора меню использована мнемоническая клавиша
WM_MENUSELECT	Пользователь просматривает меню

Системная команда

WM_SYSCOMMAND	Была выбрана системная команда

Сообщения указателя мыши

WM_NCHITTEST	Передача в окно текущего положения указателя мыши
WM_SETCURSOR	Уведомление окну при перемещении мыши с возможностью изменения курсора в зависимости от его положения

Сообщения полосы прокрутки

WM_HSCROLL	Была нажата горизонтальная полоса прокрутки
WM_VSCROLL	Была нажата вертикальная полоса прокрутки

Сообщения блока диалога и его элементов управления

WM_COMMAND	Управляющая связь с блоком диалога
WM_COMPAREITEM	Посылается родителю с запросом на сравнение двух элементов для сортировки
WM_CTLCOLOR	Управляющий запрос на установку цветов
WM_DELETEITEM	Уведомление, направляемое элементам управления LIST BOX и COMBO BOX о том, что элемент удален
WM_DRAWITEM	Запрос родителю пользовательского элемента управления или меню на рисование

WM_GETDLGCODE	Запрос оконной процедуре о том, какие типы сообщений будут ею в последствии обрабатываться
WM_GETFONT	Запрос дескриптора основного шрифта отображения
WM_INITDIALOG	Запрос на инициализацию диалогового окна
WM_MEASUREITEM	Запрос родителю элемента управления на обеспечение его размеров при прорисовке
WM_SETFONT	Требование к элементу управления: использовать данный шрифт

Сообщения многодокументного интерфейса (MDI)

WM_CHILDACTIVATE	Объявляет порождающему окну об активизации дочернего окна
WM_MDIACTIVATE	Объявляет дочернему окну MDI, что оно получает или теряет активизацию
WM_MDICASCADE	Требование расположить дочерние окна MDI в каскадном (ступенчатом) порядке
WM_MDICREATE	Требование окну клиента MDI создать дочернее окно
WM_MDIGETACTIVE	Запрос окна клиента MDI о текущем активном дочернем окне
WM_MDIICONARRANGE	Требование упорядочить расположение дочерних окон MDI, отображаемых в виде пиктограмм
WM_MDIMAXIMIZE	Требование развернуть дочерние окна MDI
WM_MDINEXT	Требование активизации следующего дочернего окна MDI
WM_MDIRESTORE	Требование восстановления дочернего окна MDI в его предыдущее состояние
WM_MDISETMENU	Настраивает меню для текущего дочернего окна MDI
WM_MDITITLE	Требование расположить дочерние окна MDI в мозаичном порядке (без перекрытий)

Все эти сообщения будут нами подробно рассмотрены при обсуждении соответствующих тем.

Сообщения о завершении работы приложения и системы

Эта группа самая малочисленная, но все ее сообщения очень важны, т. к. они используются для управления завершением приложений.

WM_QUIT	Требование на завершение приложения
WM_QUERYENDSESSION	Запрос готовности приложения к завершению работы всей системы
WM_ENDSESSION	Уведомление о результатах запроса на завершение работы системы

Частные сообщения

Очень просто определить собственные сообщения и использовать их для связи между различными окнами, которые вы создаете. При определении частного сообщения необходимо использовать область значений, начинающуюся с WM_USER, которая определена в WINUSER.H следующим образом:

```
#define WM_USER 0x0400
```

Например, если мы хотим работать с окнами, которые показывают числа, то могли бы контролировать появление чисел различных систем счисления с помощью следующих частных сообщений (PM_ — префикс частного сообщения, от Private Message):

```
#define PM_DECIMAL WM_USER + 0
#define PM_BINARY  WM_USER + 1
#define PM_HEX     WM_USER + 2
#define PM_OCTAL   WM_USER + 3
```

Кроме того, есть несколько частных системных сообщений:

WM_CANCELMODE	Требование системы отменить режим, например, захват мыши
WM_ENTERIDLE	Уведомление о том, что система находится в режиме ожидания

Информационные сообщения системных ресурсов

Имеется всего восемь сообщений с уведомлением об изменении некоторых системных ресурсов. Они посылаются окну верхнего уровня каждой программы при изменении системного ресурса. Например, когда добавляются или удаляются шрифты в системе, распространяется сообщение WM_FONTCHANGE. А когда пользователь изменяет системные цвета или системное время, отправляются сообщения WM_SYSCOLORCHANGE или WM_TIMECHANGE соответственно. Типичной реакцией на часть такого рода сообщений является немедленная запись произведенных изменений, т. е. сохранение выполненной работы.

WM_COMPACTING	Недостаточно системной памяти
WM_DEVMODECHANGE	Изменилась настройка принтера
WM_FONTCHANGE	Изменились установленные в системе шрифты
WM_PALETTECHANGED	Палитра аппаратных цветов изменена
WM_SPOOLERSTATUS	Задание было удалено из очереди буфера печати
WM_SYSCOLORCHANGE	Изменены один или более системных цветов
WM_TIMECHANGE	Изменилось системное время
WM_WININICHANGE	Изменился файл инициализации win.ini

Не всякое изменение представляет интерес для каждой программы. Например, программа, использующая таймер, вероятно, будет проверять новое значение времени при получении сообщения WM_TIMECHANGE. Но если она не работает со шрифтами, то, скорей всего, сообщение WM_FONTCHANGE будет проигнорировано. Поскольку большинство программ использует системные цвета, то они будут реагировать на сообщение WM_SYSCOLORCHANGE, организуя перерисовывание новыми цветами.

Сообщения об обмене данными

В Windows есть сообщения, которые посылаются при совместном использовании данных для поддержки механизмов работы с буфером обмена (clipboard) и динамического обмена данными (DDE), которые определены в файлах WINUSER.H и DDE.H соответственно. И хотя полное обсуждение этого вопроса выходит за рамки книги, мы приведем список сообщений этой группы.

Сообщения буфера обмена

WM_ASKCBFORMATNAME	Запрос имени формата содержимого буфера обмена
WM_CNANGECBCHAIN	Уведомление об изменении цепочки просмотра
WM_DESTRIYCLIPBOARD	Содержимое буфера обмена уничтожено
WM_DRAWCLIPBOARD	Содержимое буфера обмена изменено
WM_HSCROLLCLIPBOARD	Горизонтальная прокрутка содержимого буфера обмена
WM_PAINTCLIPBOARD	Запрос рисования содержимого буфера обмена
WM_RENDERALLFORMATS	Запрос данных для всех ранее объявленных форматов содержимого буфера обмена
WM_RENDERFORMAT	Запрос данных для ранее объявленного формата содержимого буфера обмена
WM_SIZECLIPBOARD	Изменен размер содержимого буфера обмена
WM_VSCROLLCLIPBOARD	Вертикальная прокрутка содержимого буфера обмена

Сообщения DDE

WM_DDE_ACK	Подтверждение приема
WM_DDE_ADVISE	Запрос от пользователя DDE об установлении постоянного канала передачи данных
WM_DDE_DATA	Уведомление о направлении элемента данных от DDE-сервера клиенту DDE
WM_DDE_EXECUTE	Указание DDE-серверу выполнить серию команд
WM_DDE_INITIATE	Вход в DDE-сервер

WM_DDE_POKE	Запрос пользователя серверу об обновлении определенного элемента данных
WM_DDE_REQUEST	Однократный запрос пользователя на информацию
WM_DDE_TERMINATE	Выход из DDE-сервера
WM_DDE_UNADVISE	Закрытие постоянного канала передачи данных, инициализированный сообщением WM_DDE_ADVISE

Внутрисистемные сообщения

Помимо перечисленных, имеется большая группа сообщений, не описанных ни в какой документации. Это внутренние системные сообщения, которые Windows использует в своих собственных целях. Эти сообщения передаются оконной процедуре, и если она их не обработает, то направляются обработчику сообщений, устанавливаемому по умолчанию.

В принципе ничто не мешает вам использовать эти сообщения в собственных целях. Но будьте осторожны! В будущих версиях Windows может измениться способ использования этих сообщений, а некоторые могут быть вообще удалены. Исходя из опыта, недокументированные возможности интересно изучать, но опасно использовать в программных продуктах, предназначенных для широкого распространения.

Если вы внимательно ознакомились с представленными сообщениями, то наверняка обратили внимание, что не рассмотрено ни одно сообщение элементов управления (как стандартных, так и общих). Все эти сообщения обязательно будут описаны нами при рассмотрении соответствующих элементов управления, но уже с точки зрения их использования в рамках библиотеки классов MFC.

APIWin — пример приложения

Приложение, исходные тексты которого приведены ниже, является ничем иным, как простейшим текстовым редактором. Это SDI-приложение, т. е. приложение обеспечивает работу только с одним документом. Мы приводим текст только одного файла APIWIN.C. Файл описания идентификаторов RESOURCE.H содержит идентификаторы команд меню, а файл ресурсов — ссылки на пиктограммы, шаблон меню и таблицу акселераторов. При желании все это можно написать самостоятельно, основываясь, прежде всего, на содержимом функции окна приложения *WndProc*.

При ознакомлении с текстом обратите внимание на отличие операции регистрации окна в зависимости от системы, в которой создается исполняемый модуль приложения, и от системы, в рамках которой выполняется приложение.

```
/*****************************************************************
Модуль APIWin.c
Copyright (c) 1997 Мешков А., Тихомиров Ю.
*****************************************************************/

/* имя приложения */
#define APPNAME "APIWIN"

#include <windows.h>          /* описание функций Win32 API */
#include "resource.h"         /* определение идентификаторов */

/* макрос для определения версии системы */
#if defined (WIN32)
  #define IS_WIN32 TRUE
#else
  #define IS_WIN32 FALSE
#endif
#define IS_NT IS_WIN32 && (BOOL)(GetVersion() < 0x80000000)
#define IS_WIN32S IS_WIN32 && (BOOL)(!(IS_NT) && \
                  LOBYTE(LOWORD(GetVersion()))<4)
#define IS_WIN95 (BOOL)(!(IS_NT) && !(IS_WIN32S)) && IS_WIN32

/* глобальные переменные */
HINSTANCE hInst;                  /* идентификатор приложения */
HWND hwndEdit;               /* идентификатор окна редактора */
char szAppName[] = APPNAME;               /* имя приложения */

/* заголовок главного окна */
char szTitle[] = APPNAME": Приложение-пример";

CHAR szFileName[_MAX_PATH];
CHAR szFileTitle[_MAX_PATH];

/* определение используемых функций */
ATOM APIWinRegisterClass(CONST WNDCLASS *);
BOOL InitApplication(HINSTANCE);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

BOOL LoadFile(HANDLE hwnd, BOOL bFileNew);
BOOL SaveFile(HANDLE hwnd, BOOL bSaveAs);

/*
   главная функция приложения - инициализирует приложение и
   выполняет цикл обработки сообщений
*/
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
```

```
{
  MSG msg;
  HANDLE hAccelTable;

  if (!hPrevInstance)
  {
    /* инициализируем приложение -
       подготавливаем данные класса окна и регистрируем его */
    if (!InitApplication(hInstance))
      return (FALSE);
  }

  /* завершаем создание копии приложения -
     создаем главное окно приложения */
  if (!InitInstance(hInstance, nCmdShow))
    return (FALSE);

  hAccelTable = LoadAccelerators(hInstance, szAppName);

  /* Цикл обработки сообщений */
  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator (msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (msg.wParam);
}

/*
   функция регистрации оконного класса (для Windows 95)
*/
ATOM APIWinRegisterClass(CONST WNDCLASS *lpwc)
{
  HANDLE  hMod;
  FARPROC proc;
  WNDCLASSEX wcex;

  hMod = GetModuleHandle("USER32");
  if (hMod != NULL)
  {
#if defined (UNICODE)
    proc = GetProcAddress(hMod, "RegisterClassExW");
#else
    proc = GetProcAddress(hMod, "RegisterClassExA");
#endif

    if (proc != NULL)
```

```
    {
      wcex.style         = lpwc->style;
      wcex.lpfnWndProc   = lpwc->lpfnWndProc;
      wcex.cbClsExtra    = lpwc->cbClsExtra;
      wcex.cbWndExtra    = lpwc->cbWndExtra;
      wcex.hInstance     = lpwc->hInstance;
      wcex.hIcon         = lpwc->hIcon;
      wcex.hCursor       = lpwc->hCursor;
      wcex.hbrBackground = lpwc->hbrBackground;
      wcex.lpszMenuName  = lpwc->lpszMenuName;
      wcex.lpszClassName = lpwc->lpszClassName;

      /* добавляем элемент, нужный для Windows 95 */
      wcex.cbSize = sizeof(WNDCLASSEX);
      wcex.hIconSm = LoadIcon(wcex.hInstance, "APIWIN_SMALL");

      return (*proc)(&wcex);
    }
  }

  return (RegisterClass(lpwc));
}

/*
   функция инициализации данных окна и его регистрации
*/
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;
  HWND hwnd;

  /* в Win32 API параметр hPrevInstance всегда равен NULL,
     поэтому для нахождения ранее запущенного приложения
     используем функцию поиска окна (если не хотим
     повторного запуска приложения) */
  hwnd = FindWindow (szAppName, NULL);
  if (hwnd)
  {
    /* нашли копию приложения - восстановим его на экране и
       активизируем его */
    if (IsIconic(hwnd))
      ShowWindow(hwnd, SW_RESTORE);

    SetForegroundWindow (hwnd);

    /* данная копия должна быть завершена */
    return (FALSE);
  }
```

```
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)WndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, szAppName);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wc.lpszMenuName  = szAppName;
  wc.lpszClassName = szAppName;

  if (IS_WIN95)
    return APIWinRegisterClass(&wc);
  else
    return RegisterClass(&wc);
}

/*
   функция создания главного окна
*/
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;

  hInst = hInstance;
  hWnd = CreateWindow(
    szAppName, szTitle, WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInstance, NULL);

  if (!hWnd)
    return (FALSE);

  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return (TRUE);
}

/*
   функция главного окна
   обрабатывает следующие сообщения:
     WM_CREATE - создание окна
     WM_SIZE - изменение размеров окна
     WM_COMMAND - обработка пользовательских команд
     WM_DESTROY - команда на завершение приложения
*/
LRESULT CALLBACK WndProc(
  HWND hWnd,
  UINT message,
```

```
  WPARAM wParam,
  LPARAM lParam)
{
  int wmId, wmEvent;
  RECT rc;

  switch (message)
  {
    case WM_CREATE:
      /* создаем окно редактора */
      hwndEdit = CreateWindow(
        "edit", NULL,
        WS_CHILD|WS_HSCROLL|WS_MAXIMIZE|WS_VISIBLE|WS_VSCROLL|
        ES_AUTOHSCROLL|ES_AUTOVSCROLL|ES_MULTILINE,
        0, 0, 0, 0,
        hWnd, GetMenu(hWnd),
        hInst, NULL);

      SetFocus (hwndEdit);
      break;
    case WM_SIZE:

       /* при изменении размеров главного окна
          изменяем и окно редактора */
       GetClientRect(hWnd, &rc);
       MoveWindow(
         hwndEdit,
         rc.left, rc.top,
         rc.right-rc.left,
         rc.bottom-rc.top, TRUE);
       break;
    case WM_COMMAND:
      wmId = LOWORD(wParam);
      wmEvent = HIWORD(wParam);

      /* анализ команды меню */
      switch (wmId)
      {
        case IDM_EXIT:
          DestroyWindow (hWnd);
          break;

        case IDM_NEW:
          LoadFile(hWnd, TRUE);
          break;
        case IDM_OPEN:
          LoadFile(hWnd, FALSE);
          break;
        case IDM_SAVE:
          SaveFile(hWnd, FALSE);
          break;
```

```
        case IDM_SAVEAS:
          SaveFile(hWnd, TRUE);
          break;

        /* команды которые пока не обрабатываем */
        default:
          return (DefWindowProc(
            hWnd, message, wParam, lParam));
      }
      break;

    case WM_DESTROY:
      PostQuitMessage(0);
      break;

    default:
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (0);
}

/*
   функция для загрузки содержимого файла или
   очистки редактора
*/
BOOL LoadFile(HANDLE hwnd, BOOL bFileNew)
{
  OPENFILENAME ofn;
  DWORD dwFileSize;
  HGLOBAL hBuffer;
  LPSTR lpBuffer;
  HANDLE hFile;

  if (bFileNew)
  {
    szFileName[0] = '\0';
    SendMessage(hwndEdit, WM_SETTEXT, 0, (LPARAM)"");
  }
  else
  {
    memset(&ofn, 0, sizeof(ofn));
    ofn.lStructSize       = sizeof(OPENFILENAME);
    ofn.hwndOwner         = GetActiveWindow();
    ofn.lpstrFilter       = "Все файлы(*.*)\0*.*\0";
    ofn.lpstrCustomFilter = NULL;
    ofn.nMaxCustFilter    = 0;
    ofn.nFilterIndex      = 0;
    ofn.lpstrFile         = szFileName;
    ofn.nMaxFile          = _MAX_PATH;
```

```
    ofn.lpstrInitialDir    = NULL;
    ofn.lpstrFileTitle     = szFileTitle;
    ofn.nMaxFileTitle      = _MAX_PATH;
    ofn.lpstrTitle         = "Открытие файла";
    ofn.lpstrDefExt        = NULL;
    ofn.Flags              = OFN_FILEMUSTEXIST |
                             OFN_HIDEREADONLY;

    /* используем стандартный диалог */
    GetOpenFileName((LPOPENFILENAME)&ofn);

    /* открываем файл по имени */
    hFile = CreateFile(
      szFileName, GENERIC_READ, 0, NULL,
      OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    /* определяем длину файла */
    dwFileSize = GetFileSize(hFile, NULL);

    /* создаем буфер для чтения из файла */
    hBuffer = GlobalAlloc(GHND, dwFileSize+1);
    lpBuffer = (LPSTR)GlobalLock(hBuffer);

    /* читаем содержимое файла */
    ReadFile(hFile, lpBuffer, dwFileSize, &dwFileSize, NULL);
    lpBuffer[dwFileSize] = '\0';

    /* заполняем окно редактора */
    SendMessage(hwndEdit, WM_SETTEXT, 0, (LPARAM)lpBuffer);

    /* освобождаем буфер */
    GlobalUnlock(lpBuffer);
    GlobalFree(hBuffer);

    /* закрываем файл */
    CloseHandle(hFile);
  }
  return TRUE;
}
/*
   функция для сохранения содержимого редактора
   в файле
*/
BOOL SaveFile(HANDLE hwnd, BOOL bSaveAs)
{
  OPENFILENAME ofn;
  DWORD dwFileSize;
  HGLOBAL hBuffer;
  LPSTR lpBuffer;
  HANDLE hFile;
```

```
  if (bSaveAs)
  {
    memset(&ofn, 0, sizeof(ofn));

    ofn.lStructSize       = sizeof(OPENFILENAME);
    ofn.hwndOwner         = hwnd;
    ofn.lpstrFilter       = "Все файлы(*.*)\0*.*\0";
    ofn.lpstrCustomFilter = NULL;
    ofn.nMaxCustFilter    = 0;
    ofn.nFilterIndex      = 0;
    ofn.lpstrFile         = szFileName;
    ofn.nMaxFile          = _MAX_PATH;
    ofn.lpstrInitialDir   = NULL;
    ofn.lpstrFileTitle    = szFileTitle;
    ofn.nMaxFileTitle     = _MAX_PATH;
    ofn.Flags             = OFN_OVERWRITEPROMPT |
                            OFN_HIDEREADONLY;

    /* Используем стандартный диалог */
    GetSaveFileName((LPOPENFILENAME)&ofn);
  }

  /* открываем или создаем, если не существует,
     файл по имени */
  hFile = CreateFile(
    szFileName, GENERIC_WRITE, 0, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

  /* удаляем содержимое файла */
  SetEndOfFile(hFile);

  /* определяем размер текста в редакторе */
  dwFileSize = SendMessage(hwndEdit, WM_GETTEXTLENGTH, 0, 0);

  /* создаем буфер для чтения из буфера редактора */
  hBuffer = GlobalAlloc(GHND, dwFileSize+1);
  lpBuffer = (LPSTR)GlobalLock(hBuffer);

  /* читаем содержимое буфера */
  SendMessage(hwndEdit, WM_GETTEXT,
    dwFileSize+1, (LPARAM)lpBuffer);

  /* записываем буфер в файл */
  WriteFile(hFile, lpBuffer, dwFileSize, &dwFileSize, NULL);

  /* освобождаем буфер */
  GlobalUnlock(lpBuffer);
  GlobalFree(hBuffer);

  /* закрываем файл */
  CloseHandle(hFile);
  return TRUE;
}
```

Как видите, все, что касается собственно оконной функции нашего редактора, ненамного сложнее кода, приведенного в книге ранее. Наибольший интерес представляют функции, активизирующие некоторые стандартные диалоги и функции работы с файлами. Давайте остановимся на них поподробнее.

Забегая немного вперед, скажем, что практически все стандартные диалоги поддерживаются библиотекой классов MFC. Настройка таких диалогов осуществляется примерно одинаково, как при использовании функций Win32 API, так и при использовании соответствующих классов библиотеки. Функции *GetOpenFileName* и *GetSaveFileName* очень показательны в этом смысле. И та и другая управляются при помощи подготавливаемой структуры OPENFILENAME. В части V этого тома подробно рассказано о поддержке библиотекой классов MFC стандартных диалогов, где в числе прочих рассмотрена и эта структура.

Что касается функций ввода/вывода[1] (в примере используются функции *CreateFile*, *GetFileSize*, *SetEndOfFile*, *ReadFile*, *WriteFile* и универсальная функция освобождения/закрытия некоторого объекта *CloseHandle*), то их в ряде случаев приходится использовать и при программировании MFC-приложений. Не будем подробнее говорить об этих функциях, важно лишь отметить тот факт, что файловый ввод/вывод в рамках Win32 API коренным образом отличается от Win16 API. (Например, для открытия файла вызывается функция *CreateFile* с флагом OPEN_EXISTING, а не функция *OpenFile*). Win32 API унифицирует работу с потоками данных различной, если хотите, физической природы. Это могут быть файлы, каналы, порты, стандартные устройства ввода/вывода и т. д. Кроме того, использование именно этого интерфейса задействует новые, более эффективные средства обмена с дисковыми накопителями, сетями и т. п.

Часть III

Библиотека классов Microsoft Foundation Classes (MFC)

7. Обзор
8. Макросы, глобальные функции и переменные

Глава 7

Обзор

Для выполнения какой-либо работы на профессиональном уровне недостаточно поверхностного знакомства с тем инструментарием, которым собираешься пользоваться. Необходимо ясно представлять себе внутреннюю структуру и логику работы используемого средства. В полной мере это относится к инструменту построения (создания) программных продуктов. Мы подходим к библиотеке классов MFC именно как к инструменту, который, взяв на себя очень большую часть черновой работы, требует от нас не просто знакомства с ним, но и глубокого изучения. Только знание всех нюансов построения и возможностей этой библиотеки позволит нам быстро и легко создавать программы любой степени сложности. В противном случае нет уверенности, что созданное приложение будет при любых условиях работать правильно — слишком много скрыто непосредственно от программиста, функции вызываются не только из недр Windows, но и из недр библиотеки. Ни о каком последовательном вызове функций речь не идет. Данные готовятся неизвестно где и кем и неизвестно когда обрабатываются. Какая уж тут уверенность!

Что же представляет собой библиотека Microsoft Foundation Classes (MFC)? Это множество классов, охватывающее большую часть функциональных возможностей Windows и предоставляющее разработчику не только комплекс очень мощных дополнительных классов, но и целые механизмы, которые, по сути, не нарушая идеологию операционной системы, существенно ее расширяют и... упрощают.

Перед создателями библиотеки стояла задача разработки объектно-ориентированного интерфейса для работы в среде Windows, который удовлетворял бы следующим основным целям проектирования программных продуктов:

- значительное уменьшение усилий при программировании приложений для Windows;
- скорость выполнения программ, написанных с использованием библиотеки, должна быть сопоставима со скоростью выполнения программ, написанных на языке C с использованием Win32 API;

- размер вспомогательного кода должен быть минимальным;
- способность напрямую вызывать любую С-функцию Win32 API;
- легкость использования Win32 API в C++ должна быть такая же, что и при использовании традиционного С.

Надо сказать, что поставленная задача была решена очень профессионально. Созданная библиотека классов предоставляет окна, блоки диалога, контексты устройств, общие объекты GDI, такие как битовые образы и кисти, элементы управления и многие другие стандартные элементы Windows. Основа программирования для Windows — обработка событий — предоставлена программистам в удобном и привычном виде. Классы библиотеки полностью вобрали в себя многочисленные операторы *switch*, которые так загромождают программы, написанные на языке С. Наряду с этим, вы можете совершенно свободно использовать как вызовы библиотеки классов, так и прямые вызовы Win32 API. Кроме того, за сравнительно небольшой промежуток времени корпорацией Microsoft было разработано несколько версий библиотеки MFC, которая становилась все мощнее и удобнее. Есть все основания считать, что развитие не остановится и на версии MFC 4.1. К тому же поддержка другими компаниями в рамках своих компиляторов и средств разработки программного обеспечения именно библиотеки MFC, а не OWL фирмы Borland International или своих дублирующих разработок позволяет сконцентрироваться именно на MFC. Но давайте рассмотрим все по порядку. Начнем с общего обзора библиотеки.

Иерархия классов MFC

На рис. 14 показана иерархия основных категорий классов библиотеки MFC. Каждый новый производный класс обладает как свойствами, унаследованными им от родительских классов, так и приобретает новые, характерные только для данного класса. На вершине иерархии находится единственный базовый класс *CObject*. Все остальные классы библиотеки MFC можно условно разбить на две группы, в зависимости от их отношения к классу *CObject*: производные и не производные от него.

Для того чтобы составить представление о структуре и возможностях библиотеки классов MFC, приведем краткое описание классов, составляющих эту библиотеку. Начнем с вершины представленной иерархии — класса *CObject*.



Рис. 14. Иерархия классов по категориям

CObject — основа всех классов

Большинство классов в библиотеке MFC являются производными от единственного класса *CObject* — корня иерархической структуры. Основное назначение этого класса заключается в предоставлении всем своим производным классам следующих возможностей:

- хранение информации о классе времени выполнения;
- поддержка сериализации и диагностики объекта.

К этим понятиям мы еще вернемся, а сначала рассмотрим данные и функции этого класса и тесно связанной с ним структуры *CRuntimeClass*, определенные в файле AFX.H.

```
class CObject
{
protected:
  CObject();
public:
  virtual ~CObject();
  void* PASCAL operator new(size_t nSize);
  void* PASCAL operator new(size_t, void* p);
  void  PASCAL operator delete(void* p);
public:
  BOOL IsSerializable() const;
  BOOL IsKindOf(const CRuntimeClass* pClass) const;

// Переопределяемые функции-члены
  virtual CRuntimeClass* GetRuntimeClass() const;
  virtual void Serialize(CArchive& ar);
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
public:
  static AFX_DATA CRuntimeClass classCObject;
};
```

Единственной переменной этого класса является статическая переменная *classObject* типа *CRuntimeClass*, которая хранит информацию об объекте времени выполнения, ассоциированным с классом *CObject*.

Краткое описание функций-членов приведено ниже.

CObject::**CObject**() —
: защищенный конструктор, вызывается при сериализации и динамическом создании объектов.

CRunTimeClass* CObject::**GetRuntimeClass**() —
: виртуальная функция, которая возвращает указатель на структуру CRuntimeClass, ассоциированную с данным объектом; в нашей работе нам еще не приходилось сталкиваться с необходимостью переопределения этой функции.

Прежде чем давать комментарии к следующей функции, необходимо дать небольшое пояснение широко используемому в библиотеке MFC термину "сериализация". *Сериализация* (Serialize) — это процесс сохранения текущего состояния объекта на каком-либо устройстве постоянного хранения (чаще всего в файле на диске) с возможностью последующего восстановления этого состояния.[1] Ключевым моментом здесь является то, что объект сам отвечает за способ и формат записи и чтения своего состояния. Для этого ему достаточно переопределить базовую операцию сериализации, которую поддерживает класс *CObject*, а именно функцию *Serialize*:

void CObject::**Serialize**(CArchive& ar) —
виртуальная функция, которая служит для чтения объекта из архива или его записи в архив; если вы хотите полностью использовать возможности, которые дает процесс сериализации, необходимо переопределить эту функцию для каждого класса, информацию о котором вы хотите сохранить, при этом сначала должна вызываться функция базового класса; более подробно к описанию этого процесса и тесно связанного с ним класса CArchive мы вернемся позже.

void CObject::**AssertValid**() —
виртуальная функция, которая проверяет внутреннее состояние объекта на истинность; в отладочной версии этой функции отрицательный результат проверки позволяет завершить программу выдачей сообщения о номере строки и имени файла, где эта ошибка произошла; при работе со своим классом следует переопределить эту функцию, причем в переопределенной версии первой должна вызываться функция базового класса; поскольку функция объявлена как const, нельзя изменять состояние объекта во время тестирования.

void CObject::**Dump**(CDumpContext& dc) —
виртуальная функция, которая сохраняет содержимое объекта в специальном объекте класса CDumpContext, в качестве которого чаще всего используется afxDump — предопределенный объект класса CDumpContext, позволяющий послать информацию в окно отладки; и функция и предопределенный объект доступны только в отладочной версии библиотеки.

BOOL CObject::**IsSerializable**() —
позволяет проверить возможность проведения сериализации объекта; возвращает значение TRUE, если в областях описания и реализации объекта использованы макросы DECLARE_SERIAL и IMPLEMENT_SERIAL.

BOOL CObject::**IsKindOf**(const CRuntimeClass* pClass) —
позволяет проверить, соответствует ли объект классу, переданному ей в качестве параметра, или какому либо производному от него; реализация этой функции нарушает принцип полиморфизма языка C++, поэтому не следует часто ее применять, лучше пользоваться имеющимися виртуальными функциями.

Класс *CObject* не поддерживает множественного наследования, т. е. классы, которые вы создаете, могут иметь только один *CObject* в качестве базового, и он должен быть "самым левым" в иерархии. Однако, он позволяет иметь

---

[1] Serialize дословно переводится как "преобразование в последовательную форму представления", слишком громоздко, поэтому будем использовать термин "сериализация".

"справа от него", в качестве базовых, несколько структур и классов, не являющихся производными от него. Например,

```
class A : public CObject        class B        class C
{};                             {};            {};
class D : public A, B, C
{};
```

Как уже отмечалось, каждый производный от *CObject* класс ассоциируется со структурой *CRuntimeClass*, которую можно использовать для получения информации времени выполнения об объекте или его базовом классе. Структура *CRuntimeClass* имеет следующее объявление:

```
struct CRuntimeClass
{
LPCSTR m_lpszClassName;
  int m_nObjectSize;
  UINT m_wSchema;
  CObject* (PASCAL* m_pfnCreateObject)();

#ifdef _AFXDLL
   CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
#else
   CRuntimeClass* m_pBaseClass;
#endif

  CObject* CreateObject();
  BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
  ...
};
```

Ниже приведено описание ее членов.

LPCSTR CRuntimeClass::**m_lpszClassName** — строка, содержащая имя класса.

int CRuntimeClass::**m_nObjectSize** — размер объекта в байтах.

UINT CRuntimeClass::**m_wSchema** —
номер схемы ("версии") сериализации, которая позволяет идентифицировать и обрабатывать данные, созданные более ранней версией программы.

CObject* CRuntimeClass::(PASCAL* **m_pfnCreateObject**)() —
указатель на конструктор по умолчанию, который создает объект вашего класса (если класс не поддерживает динамическое создание, то здесь хранится значение NULL).

CRuntimeClass* CRuntimeClass::(PASCAL* **m_pfnGetBaseClass**)()-
если вы работаете с динамической версией MFC, то здесь хранится указатель на функцию, которая возвращает указатель на структуру CRuntimeClass базового класса.

CRuntimeClass* CRuntimeClass::**m_pBaseClass** -
: используется, если вы работаете со статической версией MFC, и содержит указатель на структуру CRuntimeClass базового класса.

CObject* CRuntimeClass::**CreateObject**() –
: эта функция используется для создания объектов классов, которые поддерживают динамическое создание.

BOOL CRuntimeClass::**IsDerivedFrom**(const **CRuntimeClass*** pBaseClass) –
: позволяет определить, является ли класс, функция которого вызывается, производным от класса, указатель на который передается в качестве параметра.

Использование этой структуры должно сопровождаться обязательным включением в качестве информации времени выполнения одного из макросов DECLARE_DYNAMIC, DECLARE_DYNCREATE, DECLARE_SERIAL в области объявления и соответствующих им макросов IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE, IMPLEMENT_SERIAL в области реализации. Описание макросов приведено ниже в этой главе.

Наряду с рассмотренными функциями для получения структуры класса времени выполнения широко используется макрос RUNTIME_CLASS, который позволяет получить эту структуру по имени класса C++.

В заключении обзора корня иерархической структуры библиотеки MFC — класса *CObject* — рассмотрим четыре уровня функциональных возможностей, которые он нам предоставляет:

1. базовые функциональные возможности — отсутствует поддержка информации о классе времени выполнения и сериализации, но включается управление диагностикой памяти;
2. базовые функциональные возможности плюс поддержка информации о классе времени выполнения;
3. базовые функциональные возможности плюс поддержка информации о классе времени выполнения и динамическое создание;
4. базовые функциональные возможности плюс поддержка информации о классе времени выполнения, динамическое создание и сериализация;

Уровень функциональных возможностей задается вашим выбором определенных макросов объявления и реализации при создании классов, производных от *CObject*.

Информацию о классе времени выполнения *CObject* поддерживает посредством своей функции-члена *IsKindOf*, которая позволяет определить, относится ли объект к определенному классу или, может быть, является производным от него. Эта функция позволяет осуществить безопасное приведение типа в производном классе. Аргументом этой функции является объект класса *CRuntimeClass*, который можно получить, используя макрос RUNTIME_CLASS с именем класса. Функция *IsKindOf* не поддерживает множественного наследования или виртуальных классов, хотя, при необходимости, можно использовать множественное наследование для ваших классов, производных от классов библиотеки MFC.

CObject также поддерживает динамическое создание — создание объекта определенного класса во время выполнения. Объект создается функцией-членом *CreateObject* структуры *CRuntimeClass*, как показано в примере:

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CMyClass);
CObject* pObject = pRuntimeClass->CreateObject();
ASSERT(pObject->IsKindOf(RUNTIME_CLASS(CMyClass)));
```

Сериализация

Преобразование в последовательную форму и обратно, т. е. сериализация — метод, который позволяет сохранять и восстанавливать объекты классов, созданных на базе классов *CObject*. Этот метод можно разделить функционально на две составляющие.

С одной стороны, наличие определенной виртуальной функции (*Serialize*) позволяет унифицировать процесс сохранения/восстановления объектов. Впрочем, обычно для программ, написанных с использованием практически любого объектно-ориентированного языка. Для реализации этого процесса следует только переопределить виртуальную функцию, вызов которой по указателю на объект одного из базовых классов приведет к вызову нужной функции. Библиотека классов MFC идет несколько дальше, обеспечивая открытие потока, прежде всего файла, самостоятельно. Связывая этот поток с классом *CArchive*, библиотека обеспечивает тем самым удобный способ определения типа операции (чтение/запись) и целый набор функций для всех простейших типов данных (например, чисел) и библиотечных классов.

С другой стороны, сериализация поддерживает механизм динамического создания объектов неизвестного заранее типа. Например, приложение должно сохранять и восстанавливать некоторое количество объектов различного типа. Естественно, что для вызова соответствующего конструктора при восстановлении объекта, необходимо точно знать тип создаваемого объекта. Механизм сериализации делает это за программиста, сохраняя необходимую информацию самостоятельно. Единственное, что требуется — все классы сохраняемых и восстанавливаемых объектов должны базироваться на классе *CObject* и обеспечивать систему соответствующей информацией, используя макросы DECLARE_SERIAL и IMPLEMENT_SERIAL при объявлении и в реализации соответственно.

Поясним все вышесказанное на примере. Класс *CGraphicsView* имеет список указателей на объекты класса *CDrawObject* и производные от него. Класс *CDrawObject* основан на *CObject* и является абстрактным, а классы *CLine*, *CRectangle* и *CEllipse* являются производными от *CDrawObject* и поддерживают динамическое создание на основе сохраненной информации. Класс *CDrawObject* не может поддерживать динамическое создание в полном объеме, т. к. является абстрактным.

Рассмотрите объявление классов и реализацию их функций, обратив внимание на объявление конструкторов.

```
class CDrawObject : public CObject
{
protected: // вызывается только во время сериализации
  CDrawObject();
public:
  CDrawObject(BOOL Selected);
  virtual ~CDrawObject();

  ...

  virtual void Draw(CDC *pDC) = 0;
  virtual void Serialize(CArchive& ar);
private:
  BOOL m_Selected;
};

class CLine : public CDrawObject
{
protected: // вызывается только во время сериализации
  CLine();
  DECLARE_SERIAL(CLine)
public:
  CLine(CPoint beg, CPoint end);
  ~CLine();

  ...

  virtual void Draw(CDC *pDC);
  virtual void Serialize(CArchive& ar);
protected:
  CPoint m_beg;
  CPoint m_end;
};

class CRectangle : public CLine
{
protected: // вызывается только во время сериализации
  CRectangle();
  DECLARE_SERIAL(CRectangle)
public:
  CRectangle(CPoint beg, CPoint end);
  virtual ~CRectangle();

  ...

  virtual void Draw(CDC *pDC);
};
```

```
class CEllipse : public CLine
{
protected: // вызывается только во время сериализации
  CEllipse();
  DECLARE_SERIAL(CEllipse)
public:
  CEllipse(CPoint beg, CPoint end);
  virtual ~CEllipse();

  ...

  virtual void Draw(CDC *pDC);
};

class CGraphicsView : public CScrollView
{
protected: // вызывается только во время сериализации
  CGraphicsView();
  DECLARE_DYNCREATE(CGraphicsView)

  ...

public:
  virtual void Serialize(CArchive& ar);

  ...

private:
  CTypedPtrList <CPtrList, CDrawObject *> m_List;
};

CDrawObject::CDrawObject()
{
  m_Selected = FALSE;
}

CDrawObject::CDrawObject(BOOL Selected)
{
  m_Selected = Selected;
}

CDrawObject::~CDrawObject()
{
}

void CDrawObject::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    ar << m_Selected;
  }
  else
```

```
  {
    ar >> m_Selected;
  }
}

IMPLEMENT_SERIAL(CLine,CObject,1)

CLine::CLine()
{
}

CLine::CLine(CPoint beg, CPoint end) : CDrawObject(FALSE)
{
  m_beg = beg;
  m_end = end;
}

CLine::~CLine()
{
}

void CLine::Serialize(CArchive& ar)
{
  CDrawObject::Serialize(ar);
  if (ar.IsStoring())
  {
    ar <<  m_beg;
    ar <<  m_end;
  }
  else
  {
    ar >>  m_beg;
    ar >>  m_end;
  }
}

void CLine::Draw (CDC* pDC)
{
  ...
}

IMPLEMENT_SERIAL(CRectangle,CLine,1)

CRectangle::CRectangle()
{
}

CRectangle::CRectangle(CPoint beg, CPoint end) : CLine(beg, end)
{
}

CRectangle::~CRectangle()
{
}
```

```
void CRectangle::Draw(CDC* pDC)
{
  ...
}

IMPLEMENT_SERIAL(CEllipse,CLine,1)

CEllipse::CEllipse()
{
}

CEllipse::CEllipse(CPoint beg, CPoint end) : CLine(beg, end)
{
}

CEllipse::~CEllipse()
{
}

void CEllipse::Draw(CDC* pDC)
{
  ...
}

void CGraphicsView::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    ar << m_List.GetCount();
    POSITION pos = m_List.GetHeadPosition();
    while (pos != NULL)
      ar << m_List.GetNext(pos);
  }
  else
  {
    int nCount;
    ar >> nCount;
    while (nCount--)
    {
      CObject *pObject = NULL;
      ar >> pObject;
      m_List.AddTail(pObject);
    }
  }
}
```

Итак, каждый класс имеет конструкторы двух типов, во-первых, это конструкторы без параметров (реализация конструкторов по умолчанию), которые объявлены как защищенные, во-вторых, это общедоступные конструкторы, которые можно использовать для создания объектов по мере необходи-

мости. Защищенные конструкторы без параметров как раз и вызываются при восстановлении объектов в момент выполнения оператора чтения из архива.

```
ar >> pObject;
```

Собственно чтение в этот момент активизирует не только соответствующие конструкторы, но и функции *Serialize* уже созданных объектов. Например, вызов конструкторов и функций для восстановления объекта класса *CRectangle* происходит в следующем порядке:

CObject::CObject()

CDrawObject::CDrawObject()

CLine::CLine()

CRectangle::CRectangle()

CLine::Serialize()

Класс *CRectangle* не переопределяет функцию *Serialize* и, следовательно, используется реализация функции базового класса.

Далее представлено содержимое файла, в котором располагаются три объекта-прямоугольника и два объекта-линии в следующем порядке: прямоугольник, линия, прямоугольник, прямоугольник и линия. Одинарной чертой подчеркнуто описание типа объекта, двойной — ссылка на него в виде однобайтового числа и символа 'A'. Информация о типе объекта содержит только имя соответствующего класса, по которому и происходит вызов нужного конструктора во время исполнения программы, причем только один раз, далее тип определяется номером, указанным в ссылке. Не правда ли, удобно?

```
0000: 05 00 00 00 FF FF 01 00 0A 00 43 52 65 63 74 61 | . . .  . . .CRecta
0010: 6E 67 6C 65 26 00 00 00 19 00 00 00 58 00 00 00 | ngle&. . . . . X...
0020: 4B 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 | K. . . . . .
0030: 00 00 00 00 00 00 00 00 26 00 00 00 19 00 00 00 | . . . . . &. . . . .
0040: 58 00 00 00 4B 00 00 00 FF FF 01 00 05 00 43 4C | X...K...  . .CL
0050: 69 6E 65 26 00 00 00 19 00 00 00 58 00 00 00 4B | ine&. . . . . X...K
0060: 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 00 | . . . . . .
0070: 00 00 00 00 00 00 00 26 00 00 00 19 00 00 00 58 | . . . . . &. . . . . X
0080: 00 00 00 4B 00 00 00 01 80 26 00 00 00 19 00 00 | ...K... A&. . . . .
0090: 00 58 00 00 00 4B 00 00 00 01 00 00 00 00 00 00 | .X...K. . . . . .
00A0: 00 01 00 00 00 00 00 00 00 00 00 00 00 26 00 00 | . . . . . &..
00B0: 00 19 00 00 00 58 00 00 00 4B 00 00 00 01 80 26 | . . . X...K... A&
00C0: 00 00 00 19 00 00 00 58 00 00 00 4B 00 00 00 01 | . . . . . X...K. . .
00D0: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 | . . . . . .
00E0: 00 00 00 26 00 00 00 19 00 00 00 58 00 00 00 4B | ...&. . . . . X...K
00F0: 00 00 00 03 80 26 00 00 00 19 00 00 00 58 00 00 | ... A&. . . . . X..
0100: 00 4B 00 00 00 01 00 00 00 00 00 00 00 01 00 00 | .K. . . . . .
0110: 00 00 00 00 00 00 00 00 00 26 00 00 00 19 00 00 | . . . . . &. . . . .
0120: 00 58 00 00 00 4B 00 00 00                      |        .X...K...
```

Классы, определяющие архитектуру приложения

Классы этой категории поддерживают общие функциональные возможности большинства приложений. После создания каркаса приложения с использованием средства AppWizard ваша задача — наполнить его функциональными возможностями, специфичными для решения конкретной задачи. Обычно для этого создаются новые классы из классов архитектуры путем добавления новых и/или переопределением существующих функций-членов.

AppWizard генерирует несколько типов приложений, все из которых используют каркас в различных вариантах. Приложения, реализованные на базе SDI (однодокументного интерфейса) и MDI (многодокументного интерфейса), позволяют в полной мере использовать часть каркаса, называемую *архитектурой документ/представление* (document/view architecture). Другие типы приложений, такие как приложения на базе диалогового окна, формы или DLL, используют только некоторые особенности этой архитектуры.

Архитектура документ/представление включает один или несколько документов, их представлений и соответствующих фреймов окон. Объект шаблона документа ассоциируется с классами для каждого множества документ/представление/фрейм окна.

Все приложения на базе MFC имеют, по крайней мере, два объекта: объект "приложение", производный от класса *CWinApp*, и некоторый объект "главное окно", производный (часто не напрямую) от класса *CWnd*. (Наиболее часто главное окно порождается от классов *CFrameWnd*, *CMDIFrameWndalog* или *CDialog*, каждый из которых произведен от *CWnd*).

Приложения, которые используют архитектуру документ/представление содержат дополнительные объекты:

- объект-приложение, производный от класса *CWinApp*, как упоминалось выше;
- один или больше объектов-документов, произведенных от класса *CDocument*. Эти объекты служат для внутреннего представления данных, обрабатываемых в представлении. Они могут быть ассоциированы с файлом данных;
- один или больше объектов-представлений, произведенных от класса *CView*. Каждое представление является окном, которое присоединено к документу и ассоциировано с фреймом окна. Представление изображает и манипулирует данными, содержащимися в объекте-документе.

Кроме того, такие приложения содержат фреймы окон (произведенных от *CFrameWnd*) и шаблонов документов (произведенных от *CDocTemplate*).

Классы приложения и поддержки потоков

Каждое приложение имеет один и только один объект-приложение, который координирует работу других объектов при запуске программы.

Библиотека MFC поддерживает множество потоков, выполняемых внутри приложения. Но все приложения обязаны иметь, по крайней мере, один поток, называемый *главным*, который использует объект класса *CWinApp*.

Класс *CWinThread* инкапсулирует часть возможностей операционной системы для работы с потоками. Кроме того, MFC содержит классы объектов синхронизации, предоставляя интерфейс C++ объектам синхронизации Win32.

Классы приложения и потоков

CWinApp	инкапсулирует код инициализации, запуска и завершения приложения
CWinThread	базовый класс для всех потоков; его можно использовать непосредственно, или произвести от него свой класс, если необходимо выполнение каких-либо своих функций

Классы приложений ISAPI

CHttpFilter	определяет фильтры, посылающие выбранные запросы HTTP на сервер ISAPI; см. "Классы ISAPI"
CHttpServer	расширяет функциональные возможности сервера ISAPI; см. "Классы ISAPI"

Классы поддержки синхронизации

CSyncObject	базовый класс для всех классов поддержки синхронизации
CCriticalSection	класс, обеспечивающий исключительный доступ к некоторому объекту только одному потоку процесса
CSemaphore	класс, обеспечивающий заданное количество одновременных обращений к объекту
CMutex	класс, обеспечивающий исключительный доступ к объекту только для одного потока любого количества процессов
CEvent	класс для уведомления приложения о наступлении заданного события
CSingleLock	класс, обеспечивающий ожидание сигнализации объекта одного из классов CCriticalSection, CSemaphore, CMutex или CEvent
CMultiLock	класс, обеспечивающий ожидание сигнализации одного или всех заданных объектов классов CCriticalSection, CSemaphore, CMutex или CEvent

Связанные классы

CCommandLineInfo	анализирует командную строку приложения
CWaitCursor	меняет обычный курсор на "песочные часы"; используется для индикации выполнения длительных операций
CDockState	содержит дескрипторы постоянного хранилища состояния данных для панелей элементов управления
CRecentFileList	поддерживает список файлов, используемых в последнее время

Классы маршрутизации команд

Когда пользователь взаимодействует с приложением, выбирая при помощи мыши элементы меню или кнопки панели инструментов, приложение посылает сообщения из объекта интерфейса пользователя в соответствующий объект-получатель команды. Классы-получатели команд, производные от *CCmdTarget*, включают *CWinApp*, *CWnd*, *CDocTemplate*, *CDocument*, *CView* и классы, производные от них. Каркас приложения поддерживает автоматическую маршрутизацию команд, так что они могут быть обработаны большинством соответствующих активных объектов в приложении.

CCmdTarget	базовый класс для всех классов объектов, которые могут отправлять и получать сообщения
CCmdUI	предоставляет программный интерфейс для обновления (разрешение, запрещение, контроль и/или очистка) объектов интерфейса пользователя, таких как элементы меню или кнопки панелей инструментов; объект класса CCmdUI посылает вашим обработчикам команд команду на обновление интерфейса пользователя (ON_UPDATE_COMMAND_UI); этот процесс одинаков как для элементов меню, так и для кнопок

Классы документов

Объекты-документы, созданные из объектов-шаблонов документов, управляют данными приложения. Они взаимодействуют с объектами-представлениями, которые представляют клиентскую область окна, изображают данные документа и позволяют пользователю взаимодействовать с ними.

CDocument	базовый класс для документов, определяющих приложение
COleDocument	используется для реализации составного документа OLE; см. "Контейнерные классы OLE"
COleLinkingDoc	предоставляет инфраструктуру для организации связывания объектов; см. "Контейнерные классы OLE"
CRichEditDoc	содержит список элементов клиентов OLE; см. "Контейнерные классы OLE"
COleServerDoc	используется как базовый класс для классов документов приложения сервера; см. "Серверные классы OLE"

Связанные классы

Объекты-документы могут записывать свое состояние в устройство хранения и считывать его оттуда. Библиотека MFC предоставляет класс *CArchive*, который снабжен средствами передачи данных документов в хранилище и обратно:

CArchive	совместно с классом CFile реализует механизм непрерывного хранения объектов посредством сериализации

Документы могут также содержать OLE-объекты, абстрактным базовым классом для которых является класс *CDocItem*:

CDocItem	абстрактный базовый класс; объекты классов, производных от него, представляют части документов

Классы шаблонов документов

Объекты-шаблоны документов координируют создание объектов-документов, представлений и фреймов окон, когда создается новый документ и/или представление.

CDocTemplate	базовый класс для шаблонов документов; нет необходимости использовать этот класс непосредственно, лучше воспользоваться одним из его производных классов
CMultiDocTemplate	шаблон для документов в MDI
CSingleDocTemplate	шаблон для документов в SDI

Связанные классы

CCreateContext	структура, посылаемая из шаблона документа в функции создания окна для координации создания объектов-документов, представлений и фреймов окон

Окна, блоки диалога и элементы управления

Класс *CWnd* базовый класс для всех окон и его производные классы инкапсулируют объекты — окна Windows. Этот класс может использоваться как самостоятельно, так и в качестве базового для производства новых классов. Производные классы, имеющиеся в библиотеке классов, представляют различные свойства окон.

CWnd	базовый класс окон

Классы фреймов окон

В архитектуре документ/представление фреймы окон являются окнами, которые содержат представления. Они также поддерживают панели элементов управления, присоединенные к ним.

В приложениях с многодокументным интерфейсом главное окно является производным от *CMDIFrameWnd*, которое содержит фреймы документов, являющихся объектами *CMDIChildWnd.*

В приложениях с однодокументным интерфейсом главное окно является производным от *CFrameWnd* и содержит представление текущего документа.

CFrameWnd	базовый класс для главного окна однодокументного интерфейса; кроме того, является базовым для всех других классов фреймов окон
CMDIFrameWnd	базовый класс для главного окна многодокументного интерфейса
CMDIChildWnd	базовый класс для окна документа многодокументного интерфейса
CMiniFrameWnd	окно с заголовком половинной высоты, обычно показывающее границу плавающей панели инструментов
COleIPFrameWnd	предоставляет окно для представления, когда документ-сервер редактируется по месту; см. "Серверные классы OLE"

Связанные классы

CMenu	предоставляет интерфейс для доступа к меню приложения; он позволяет динамически манипулировать меню во время выполнения, например, когда необходимо добавить или удалить элемент меню в зависимости от контекста; несмотря на то, что наиболее часто меню используются с фреймами окон, можно также использовать их в блоках диалога и в дочерних окнах; этот класс инкапсулирует системный объект-меню

Классы представлений

Класс *CView* и его производные классы являются дочерними окнами, которые представляют клиентскую область фрейма окна. Представления отображают данные и позволяют пользователю вводить, редактировать или выбирать данные документа. Этот класс является базовым для представления специфических данных документа.

CView	базовый класс представлений
CScrollView	базовый класс для представления с возможностью автоматической прокрутки данных

Форматное представление и просмотр записей

Формы также обеспечивают прокрутку, но строятся на основе шаблонов блоков диалога.

Классы просмотра записей баз данных основываются на формах и имеют непосредственную связь с таблицами базы данных.

CFormView прокручиваемое представление, расположение которого определяется шаблоном блока диалога

CDaoRecordView отображает записи базы данных в элементах управления; см. "Классы для работы с DAO"

CRecordView предоставляет вид формы, непосредственно соединенной с объектом класса CRecordSet; см. "Классы для работы с ODBC"

Форматное представление на основе элементов управления

CCtrlView базовый класс для всех представлений, ассоциированных с элементами управления Windows

CEditView представление, которое содержит стандартный элемент управления Windows — поле, поддерживающее возможности редактирования, поиска, замены и прокрутки

CRichEditView представление, которое в дополнение к возможностям элемента управления EDIT поддерживает настройку шрифтов, цветов, форматирование параграфов и внедрение OLE-объектов

CListView представление, которое содержит элемент управления Windows — LIST BOX

CTreeView представление, позволяющее изображать пиктограммы и строки текста, иерархически упорядоченные в виде дерева

Связанные классы

CSplitterWnd окно, которое пользователь может разделять на несколько частей, размеры которых при необходимости можно изменять; этот класс позволяет иметь несколько представлений внутри единственного фрейма окна

CPrintInfo содержит информацию о печати или задании предварительного просмотра

CRichEditDoc поддерживает список элементов-клиентов OLE, которые имеются в CRichEditView; см. "Контейнерные классы OLE"

CRichEditCntrItem предоставляет доступ к элементам OLE, хранящимся в CRichEditView; см. "Контейнерные классы OLE"

Два последних класса совместно с *CRichEditView* реализуют возможности контейнера OLE.

Классы блоков диалога

Класс *CDialog* и его производные классы инкапсулируют функциональные возможности блока диалога. Поскольку блок диалога является специальным типом окна, то очевидно, что класс *CDialog* является производным от *CWnd*, и служит в качестве базового для всех блоков диалога, как модальных, так и немодальных. Кроме того, от него образованы классы стандартных блоков диалогов, таких как диалог для открытия или сохранения файла, выбора шрифта или цвета, инициализации операций поиска и замены или выполнения различных операций с OLE-объектами.

CDialog	базовый класс блоков диалога
CDataExchange	поддерживает обмен данными и проверку информации для блоков диалога

Стандартные диалоги

CCommonDialog	базовый класс для всех общих блоков диалога
CFileDialog	стандартный блок диалога для открытия и сохранения файла
CColorDialog	стандартный блок диалога для выбора цвета
CFontDialog	стандартный блок диалога для выбора шрифта
CFindReplaceDialog	стандартный блок диалога для операций поиска и замены
CPrintDialog	стандартный блок диалога для печати файла
CPageSetupDialog	инкапсулирует сервис, предоставляемый блоком диалога Windows, обеспечивающий задание параметров страницы с дополнительной поддержкой установки и модификации границ печати

Классы стандартных блоков диалога OLE

Эти классы обеспечивают решение общих задач OLE, используя для этого стандартные блоки диалога OLE. Они также предоставляют составляющие интерфейса пользователя для функциональных возможностей OLE.

COleDialog	используется в рабочем окне и содержит общую реализацию для всех блоков диалога OLE; все классы блоков диалога в категории интерфейса пользователя произведены от этого базового класса; он не может использоваться непосредственно
COleInsertDialog	изображает блок диалога Insert Object (Вставка объекта) для вставки нового связанного или внедренного объекта OLE
COlePasteSpecialDialog	изображает блок диалога Paste Special (Специальная вставка) для реализации команды Paste Special (Специальная вставка) меню Edit (Правка)

COleLinksDialog	изображает блок диалога Edit Links (Редактировать связи) для модификации информации о связанных элементах
COleChangeIconDialog	изображает блок диалога Change Icon (Смена пиктограммы) для изменения пиктограммы, ассоциированной со связанным или внедренным элементом OLE
COleConvertDialog	изображает блок диалога Convert (Конвертировать) для преобразования элементов OLE из одного типа в другой
COlePropertiesDialog	инкапсулирует общий блок диалога OLE Properties (Свойства OLE), который предоставляет удобный подход для отображения и модификации свойств элемента OLE, совместимого со стандартом Windows
COleUpdateDialog	изображает блок диалога Update (Обновить) для обновления всех связей в документе; блок диалога содержит линейный индикатор, для отображения степени завершения процесса обновления
COleChangeSourceDialog	изображает блок диалога Change Source (Изменить источник), для изменения связей приемника или источника
COleBusyDialog	представляет стандартные блоки диалога Server Busy (Сервер занят) и Server Not Responding (Сервер не отвечает), для обработки вызовов занятых приложений; обычно изображается автоматически при реализации COleMessageFilter

Классы свойств и мастеров

Окна свойств, известные также как диалоги со вкладками, наиболее эффективны, когда нужно объединить вместе большое число элементов управления в одном блоке диалога.

CPropertyPage	представляет отдельную страницу (вкладку) внутри окна свойств
CPropertySheet	представляет фрейм для отображения и координации множества вкладок
COlePropertyPage	изображает свойства элемента управления OLE с использованием графического интерфейса, похожего на блок диалога

Классы элементов управления

Классы элементов управления инкапсулируют ряд стандартных элементов управления Windows, а также включают дополнительные элементы — кнопки с битовыми образами и панели элементов управления.

Статический элемент управления

CStatic	статически отображаемое окно; используется в качестве метки, рамки или разделителя других элементов управления в блоках диалога или окнах; может также содержать графические образы

Элементы редактирования текста

CEdit	окно простейшего редактора текста
CRichEditCtrl	элемент управления, в котором пользователь может вводить и редактировать текст; в отличие от CEdit поддерживает форматирование символов и параграфов, а также объекты OLE

Элементы представления числовой информации

CSliderCtrl	элемент управления, содержащий бегунок, который пользователь может перемещать, выбирая одно из возможных значений
CSpinButtonCtrl	пара кнопок со стрелками, нажимая которые пользователь может увеличивать или уменьшать некоторое значение
CProgressCtrl	полоса, дискретно проградуированная слева направо и отображающая процесс выполнения операции
CScrollBar	полоса прокрутки

Кнопки

CButton	программный интерфейс для кнопок (BUTTON), флажков (CHECKBOX) или переключателей (RADIO BUTTON) в блоке диалога или окне
CBitmapButton	кнопка с битовым образом вместо текста

Списки

CListBox	список элементов, которые пользователь может просматривать и выбирать
CDragListBox	дополнительно к возможностям, предоставляемым классом CListBox, позволяет пользователю перемещать элементы внутри списка
CCheckListBox	список элементов с флажками, каждый из которых пользователь может установить или сбросить
CComboBox	элемент управления, объединяющий поле редактора текста со списком
CListCtrl	набор элементов, содержащих пиктограмму и текст
CTreeCtrl	иерархический (древовидный) список, где каждый элемент может содержать пиктограмму и текст

Панель инструментов и строка состояния

CToolBarCtrl представляет функциональные возможности общего элемента управления Windows — панель инструментов; в большинстве программ с использованием MFC вместо этого класса используется класс CToolBar

CStatusBarCtrl горизонтальное окно, обычно разделенное на области, в которых приложение может отображать информацию о состоянии; в большинстве программ с использованием MFC вместо этого класса используется класс CStatusBar

Другие элементы управления

CAnimateCtrl отдельный видео-клип

CToolTipCtrl небольшое всплывающее окно, в котором отображается единственная строка текста, описывающая назначение некоторого объекта приложения

CHeaderCtrl заголовок или метка для столбца

CTabCtrl элемент управления, реализующий вкладки, например для смены отображаемых страниц (по аналогии с закладками в блокноте)

CHotKeyCtrl поддерживает создание "горячей клавиши", которую пользователь может нажать для быстрого выполнения определенного действия

Связанные классы

CImageList предоставляет функциональные возможности списка образов Windows; в основном используется с классами CListCtrl и CTreeCtrl, реже с CTabCtrl; позволяет хранить множество битовых массивов или пиктограмм одного размера

Классы панелей элементов управления

Панели элементов управления присоединяются к фрейму окна. Они содержат кнопки, области отображения состояния или основываются на шаблонах диалога. Плавающие панели элементов реализуют присоединение их к объекту *CMiniFrameWnd*.

CControlBar базовый класс для панелей элементов управления, которые представляют собой окно, выровненное по углу фрейма окна; может содержать как оконные дочерние элементы управления, так и не оконные, например, кнопки панели инструментов

CToolBar представляет функциональные возможности общего элемента управления Windows — панель инструментов в более удобном для программиста виде, чем класс CToolBarCtrl

CStatusBar горизонтальное окно, обычно разделенное на области, в которых приложение может отображать информацию состояния; класс более удобен в использовании нежели CStatusBarCtrl

CDialogBar	предоставляет возможности, аналогично другим классам панелей элементов управления, но основывается на шаблонах диалога, и, соответственно, может содержать любые элементы управления

Классы для вывода на экран и печать

В Windows весь графический вывод осуществляется в при помощи контекстов устройства (DC — device context). Библиотека MFC инкапсулирует классы различных типов контекстов устройств. Кроме того имеются классы графических объектов, таких как битовые образы, кисти, палитры, перья и др.

Классы контекста устройств

Контекст устройства — это объект Windows, который содержит информацию об атрибутах рисования. Все вызовы для рисования осуществляются с использованием объекта контекста устройства, который представлен в библиотеке MFC базовым классом *CDC*. Дополнительные классы, производные от *CDC*, инкапсулируют специализированные функциональные возможности контекста устройства, включая поддержку метафайлов Windows.

CDC	базовый класс для контекстов устройств; используется непосредственно для доступа ко всем экранным и неэкранным контекстам устройств, таких как принтеры
CPaintDC	контекст экрана, используемый в функции OnPaint оконных классов; в конструкторе автоматически вызывается функция BeginPaint, а в деструкторе — EndPaint [1]
CClientDC	контекст экрана для клиентской области окна; используется, например, при рисовании в ответ на события от мыши
CWindowDC	контекст экрана для всего окна, включая клиентскую и неклиентскую области
CMetaFileDC	контекст устройства для метафайла Windows, который содержит последовательность команд GDI, необходимых для создания образа

Классы графических объектов

Эти классы определяют графические объекты, используемые для рисования в контексте устройства.

CGdiObject	базовый класс для объектов GDI
CBrush	инкапсулирует кисть GDI, которая может быть установлена текущей в контексте устройства; кисти используются для заливки замкнутых объектов

[1] См. главу 4, "Графический вывод".

CPen	инкапсулирует карандаш GDI; карандаши используются для рисования ограничивающих линий объектов
CFont	инкапсулирует шрифт GDI
CBitmap	инкапсулирует битовый образ GDI, предоставляя интерфейс для манипулирования им
CPalette	инкапсулирует палитру цветов GDI для использования в качестве интерфейса между приложением и цветом выходного устройства, такого как экран
CRgn	инкапсулирует регион GDI для манипулирования эллиптическими, многоугольными или нерегулярными областями внутри окна; используется совместно с функциями отсечения класса CDC

Связанные классы

CRectTracker	изображение и обработка интерфейса пользователя для изменения размеров и перемещения прямоугольных областей

Классы простых типов данных

Эти классы инкапсулируют координаты рисования, строки символов и информацию о дате и времени, поддерживая удобный синтаксис языка C++. Они широко используются в качестве параметров функций-членов классов Windows в библиотеке MFC.

Классы *CPoint*, *CSize* и *CRect* соответствуют структурам *POINT*, *SIZE RECT*, представленным в Win32 SDK (возможно использование как объектов этих классов C++, так и структур языка C).

CPoint	хранит пару координат (x, y)
CSize	хранит размеры или относительные координаты позиции
CRect	хранит координаты прямоугольной области
CString	хранит строку символов
CTime	хранит значение абсолютного времени и даты
COleDateTime	интерфейс к объектам OLE типа DATE; представляет значения времени и даты
CTimeSpan	хранит относительные значения времени и даты
COleDateTimeSpan	хранит относительные значения времени и даты для объектов OLE типа DATE
COleCurrency	интерфейс к объектам OLE типа CURRENCY, арифметического типа с фиксированной точкой (15 знаков до десятичной точки и 4 после)
COleVariant	интерфейс к объектам OLE типа VARIANT; данные этого типа могут храниться во многих форматах

Массивы, списки и ассоциативные списки

Для обработки агрегированных данных библиотека классов предоставляет группу классов коллекций — массивы, простые и ассоциативные списки, которые могут хранить различные объекты и предопределенные типы. Коллекции динамически изменяют свой размер. Эти классы могут использоваться в любом приложении, написанном как для Windows, так и нет. Однако, они очень полезны для реализации структур данных, которые определяют классы документов в каркасе приложения. На основе этих классов можно создавать различные специальные классы коллекций.

- *Массивы* — это одномерная структура данных, занимающая непрерывное место в памяти. Они поддерживают достаточно быстрый прямой доступ к заданному индексом элементу. Однако массивы неэффективны в случае возникновения необходимости вставить элемент. Размер массива может изменяться.
- *Списки* похожи на массивы, но хранятся по-другому. Каждый элемент списка включает указатель на предыдущий и следующий элементы, делая его двусвязным списком. Он позволяет очень быстро осуществлять удаление и вставку элементов, т. к. для этого необходимо только изменить несколько указателей. К недостаткам следует отнести большие издержки на операции поиска в списке.
- *Ассоциативные списки* обеспечивают взаимосвязь между ключевым значением и значением элемента данных. На базе ассоциативных списков библиотека классов MFC поддерживает список, называемый *картой сообщений*, который хранит указатели на функции их обработки.

Классы шаблонов массивов, списков и ассоциативных списков

Эти классы коллекций являются шаблонами, параметры которых определяют типы хранящихся объектов. Классы *CArray*, *CList* и *CMap* используют также глобальные вспомогательные функции, которые предварительно должны настраиваться.

CArray	шаблон класса для создания массивов произвольных типов
CList	шаблон класса для создания списков произвольных типов
CMap	шаблон класса для создания ассоциативных списков произвольных типов
CTypedPtrArray	шаблон класса для массивов указателей
CTypedPtrList	шаблон класса для списков указателей
CTypedPtrMap	шаблон класса для ассоциативных списков указателей

Готовые к использованию классы массивов

CByteArray	массив элементов типа BYTE
CWordArray	массив элементов типа WORD
CDWordArray	массив элементов типа DWORD
CUIntArray	массив элементов типа UINT
CObArray	массив указателей на объекты класса CObject или производных от него
CPtrArray	массив указателей типа void
CStringArray	массив объектов CString

Готовые к использованию классы списков

CObList	связанный список указателей на объекты класса CObject или производных от него
CPtrList	связанный список указателей типа void
CStringList	связанный список объектов CString

Готовые к использованию классы ассоциативных списков

CMapPtrToPtr	использует указатель типа void в качестве ключа для нахождения другого указателя типа void
CMapPtrToWord	использует указатель типа void в качестве ключа для нахождения данных типа WORD
CMapStringToOb	использует объект CString в качестве ключа для нахождения указателей на объекты класса CObject или производных от него
CMapStringToString	использует объект CString в качестве ключа для нахождения другого объекта CString
CMapStringToPtr	использует объект CString в качестве ключа для нахождения указателя типа void
CMapWordToOb	использует данные типа WORD для поиска указателя на объект класса CObject или производных от него
CMapWordToPtr	использует данные типа WORD для поиска указателя типа void

Классы для работы с файлами и базами данных

Эти классы позволяют хранить информацию в базе данных или в файле на диске. Они включают две группы классов для работы с базами данных на основе механизмов DAO и ODBC, которые имеют сходные функциональ-

ные возможности. Группа DAO реализована с использованием механизма Data Access Object — объект доступа к данным, группа ODBC использует механизм Open Database Connectivity — открытый доступ к базам данных. Кроме того, библиотека снабжена множеством классов для работы со стандартными файлами, потоками OLE и HTML.

Классы файлов ввода/вывода

Эти классы предоставляют интерфейс для традиционных дисковых файлов, потоков OLE и сокетов Windows. Все классы, производные от *CFile*, могут использоваться совместно с объектом класса *CArchive* для выполнения сериализации. Обычно нет необходимости образовывать из них свои классы. Если для создания приложения используется средство *AppWizard*, то по умолчанию предоставляется реализация обработчиков команд Open (Открыть), Save (Сохранить) и Save as (Сохранить как) меню File (Файл) для организации ввода/вывода (с использованием класса *CArchive*). В дальнейшем, возможно переопределить функцию *Serialize* для реализации необходимой сериализации документа.

CFile	предоставляет интерфейс для бинарных файлов на диске
CStdioFile	предоставляет интерфейс класса CFile с буферизацией потоков файлов на диске, обычно в текстовом режиме
CMemFile	предоставляет интерфейс класса CFile для файлов в памяти
CSharedFile	предоставляет интерфейс класса CFile для разделяемых файлов в памяти
COleStreamFile	использует интерфейс OLE IStream, предоставляя доступ класса CFile к составным файлам; этот класс (производный от CFile) разрешает сериализацию MFC с использованием структурированного хранилища OLE
CSocketFile	предоставляет интерфейс класса CFile для сокетов Windows

Связанные классы

CHtmlStream	обеспечивает кэширование вывода HTML; функциональные возможности похожи на предоставляемые классом CMemFile
CFileStatus	структура, содержащая информацию о файле, такую как имя, атрибуты, длина и т. д.

Классы для работы с DAO

Эти классы, работая с другими классами приложения, дают легкий доступ к базам данных, поддерживающим механизм DAO, основанный на тех же принципах, что и Microsoft Visual Basic и Microsoft Access. Кроме того, классы DAO обеспечивают доступ к разнообразным базам данных, использующим драйверы ODBC. Приложениям, которые используют базы данных

DAO, следует, по крайней мере, иметь в своем составе объекты классов *CDaoDatabase* и *CDaoRecordset*.

CDaoWorkspace	управляет наименованием и парольной защитой сеанса работы с базой данных от начала и до завершения
CDaoDatabase	обеспечивает соединение с базой данных, посредством которого возможно выполнять операции над данными
CDaoRecordset	представляет множество записей, выбранных из источника данных
CDaoRecordView	изображает записи базы данных в элементах управления
CDaoQueryDef	представляет определение запроса, обычно хранящегося в базе данных
CDaoTableDef	представляет определение таблицы базы данных или присоединенной таблицы
CDaoFieldExchange	обеспечивает обмен полей записей DAO, которые используются классами базы данных DAO; этот класс обычно непосредственно не используется

Связанные классы

CLongBinary	инкапсулирует дескриптор хранилища для бинарного объекта большого объема, такого как битовая карта; объекты этого класса используются для управления большими объемами данных, хранящимися в таблицах базы данных

Классы для работы с ODBC

Эти классы, работая с другими классами приложения, обеспечивают простой доступ к различным базам данных, использующим драйверы ODBC. Приложениям, которые используют базы данных ODBC, следует иметь в своем составе, по крайней мере, объекты классов *CDatabase* и *CRecordset*.

CDatabase	инкапсулирует соединение с источником данных, посредством которого возможно манипулирование данными
CRecordSet	объединяет множество записей, выбранных из источника данных; этот класс поддерживает переход от записи к записи, обновление записей (добавление, редактирование и удаление), ограничение выбора некоторым фильтром, сортировку и параметризованный выбор с информацией, имеющейся или рассчитываемой во время выполнения
CRecordView	предоставляет вид формы, непосредственно соединенный с объектом класса CRecordSet; механизм DDX (обмен данными с блоком диалога) позволяет обновлять данные; так же, как и все виды формы, вид записи базируется на ресурсах шаблона блока диалога; он поддерживает также перемещение между записями, их обновление и закрытие соответствующего объекта класса CRecordSet

CFieldExchange	обеспечивает обмен данными в обоих направлениях между источником данных и их представлением в приложении; требуется только для пользовательских типов данных

Классы для работы в Internet и сетях

Классы этой категории обеспечивают обмен информацией с другими компьютерами, используя механизмы ISAPI (Internet Server API) или сокетов. Они включают также множество классов для создания ISAPI-расширений и множество классов для манипулирования сокетами Windows.

Классы ISAPI

Механизм ISAPI реализует интерфейс для сервера Internet. Например, сервер ISAPI является сервером Windows NT для запуска информационного сервера Internet.

Фильтры HTTP обрабатывают запросы сервера. Они могут использоваться для следующих типов операций:

- аутентификация;
- сжатие данных;
- шифрация;
- регистрация.

Классы фильтров

CHttpFilter	определяет фильтры, посылающие выбранные запросы HTTP на сервер ISAPI
CHttpFilterContext	управляет содержимым для фильтров HTTP; это вспомогательный класс для множественной обработки конкурирующих запросов объекта класса CHttpFilter

Серверные классы

Классы обеспечивают обслуживание запросов сервера, включая CGI (Common Gateway Interface).

CHttpServer	расширяет функциональные возможности сервера ISAPI, обрабатывая запросы клиента, и включает в себя CGI
CHttpServerContext	управляет содержимым расширенного сервера ISAPI; это вспомогательный класс для множественной обработки конкурирующих запросов объекта класса CHttpServer

Классы сокетов Windows

Сокеты Windows предоставляют независимый сетевой протокол коммуникаций между двумя компьютерами. Сокеты могут быть синхронными (приложение ждет, когда освободится коммуникация) или асинхронными (приложение продолжает выполняться, пока действует коммуникация).

CAsyncSocket	инкапсулирует сокеты Windows API
CSocket	реализует то же, что и CAsyncSocket, но на более высоком уровне абстракции; используется для синхронных операций
CSocketFile	предоставляет интерфейс класса CFile для сокетов Windows

Классы для работы с OLE

Классы OLE работают с другими классами окна приложения, предоставляют легкий доступ к функциям OLE API. Используя OLE, можно:

- создавать документы OLE, которые позволяют пользователям создавать и редактировать документы, содержащие данные, которые созданы другими приложениями, включая текст, графику, звук или другие типы данных;
- создавать объекты OLE, которые можно внедрять в документы OLE;
- использовать механизм drag-and-drop для копирования данных между приложениями;
- использовать автоматизацию OLE для контроля одной программы из другой;
- создавать элементы управления OLE и контейнеры из таких элементов управления.

Контейнерные классы OLE

Эти классы используются в приложениях, использующих контейнеры. Оба класса *COleLinkingDoc* и *COleDocument* управляют коллекциями объектов класса *COleClientItem*, которые используются для представления каждого элемента OLE в документе, внедренного в него или связаного с ним.

COleDocument	используется для реализации составного документа OLE; может использоваться как базовый для поддержки документов с контейнерами
COleLinkingDoc	предоставляет инфраструктуру для организации связывания объектов
CRichEditDoc	содержит список элементов OLE; используется совместно с CRichEditView и CRichEditCntrItem

COleClientItem	класс элемента, который представляет сторону клиента при связывании и внедрении элементов OLE; свои элементы со стороны клиентов следует образовывать из этого класса
CRichEditCntrItem	представляет доступ к стороне клиента OLE, хранящегося в элементе управления RICH EDIT, когда используется совместно с CRichEditView и CRichEditDoc

Серверные классы OLE

Эти классы используются в серверных приложениях. Документы сервера могут являться контейнерами, которые поддерживают связывание.

COleServerDoc	используется как базовый класс для классов документов приложения сервера; объекты этого класса представляют собой основу поддержки сервера через взаимодействие с объектами класса COleServerItem; визуальные возможности становятся доступными при использовании архитектуры документ/представление
COleServerItem	используется для представления документа или его части, которые могут быть внедрены в другой документ или связаны с ним
COleIPFrameWnd	является окном для представления, когда документ-сервер редактируется по месту
COleResizeBar	предоставляет стандартный интерфейс пользователя для изменения размеров по месту; объекты этого класса всегда используются совместно с объектами класса COleIPFrameWnd
COleTemplateServer	используется для создания документов на базе архитектуры документ/представление; объект этого класса передает большую часть своей работы ассоциированному с ним объекту класса CDocTemplate

Классы OLE для передачи данных и drag-and-drop

Эти классы используются при передаче данных OLE. Они поддерживают передачу данных между приложениями с использованием буфера обмена и посредством механизма drag-and-drop (перетащить).

COleDropSource	контролирует операции drag-and-drop от начала до конца; этот класс позволяет определить, когда начинается и когда заканчивается операция переноса; он также изображает курсор обратной связи во время операции drag-and-drop
COleDataSource	используется, когда приложение предоставляет данные для передачи; может быть представлен как объектно-ориентированный объект буфера обмена
COleDropTarget	представляет цель операции drag-and-drop; объект этого класса передается в окно, определяет, имеются ли в нем какие-либо переносимые данные, и реализует действительную операцию переноса

COleDataObject используется как получающая сторона для объекта класса COleDataSource; он предоставляет доступ к данным, хранящимся в объекте класса COleDataSource

Классы автоматизации OLE

Эти классы поддерживают автоматизацию *клиентов* (приложений, которые могут быть проконтролированы другими приложениями). Автоматизация *серверов* (приложений, которые контролируют другие приложения) поддерживается посредством карт диспетчеризации.

COleDispatchDriver используется при вызове сервера-автомата из сервера-клиента; ClassWizard использует этот класс в качестве базового при создании классов для сервера-автомата, предоставляемого библиотекой типов

Классы, связанные с OLE

Это основные классы, которые используются, когда создаются элементы управления OLE. Класс *COleControlModule* в модуле элементов управления OLE похож на класс *CWinApp* в приложении. Каждый модуль содержит один или больше элементов управления OLE, которые представлены объектами класса *COleControl*. Эти элементы управления связываются со своими контейнерами, используя объекты *CConnectionPoint*. Классы *CPictureHolder* и *CFontHolder* инкапсулируют интерфейс OLE для рисунков и шрифтов, в то время как классы *COlePropertyPage* и *CPropertyExchange* помогают реализовать страницы свойств для созданных элементов управления.

COleControlModule заменяет класс CWinApp для модуля элемента управления OLE; он предоставляет функции-члены для инициализации модуля элементов управления

COleControl наследует все функциональные возможности объекта окна Windows и дополнительные возможности, специфичные для OLE, такие как запуск события и доступ к методам поддержки и свойствам

CConnectionPoint определяет специальный тип интерфейса, используемого для соединения с другими объектами OLE, называемый *точкой соединения*; точка соединения реализует интерфейс внешнего выхода, который в состоянии инициировать действия других объектов, таких как запуск событий и изменение уведомлений

CPictureHolder инкапсулирует функциональные возможности рисованных объектов Windows и интерфейса OLE IPicture; используется для реализации общих свойств элемента управления OLE Picture

CFontHolder инкапсулирует функциональные возможности шрифтовых объектов Windows и интерфейса OLE IFont; используется для реализации имеющихся свойств элемента управления OLE Font

CPropExchange	поддерживает реализацию постоянных свойств для элемента управления OLE, аналогично классу CDataExchange для блоков диалога
COleObjectFactory	используется для создания элементов, которые запрашиваются из других OLE-контейнеров; этот класс служит базовым для большинства специфических типов "фабрик объектов", включая COleTemplateServer
COleMessageFilter	используется для управления соглашениями с *облегченными вызовами удаленных процедур* (Lightweight Remote Procedure Calls, LRPC)

Классы для отладки и обработки исключений

Эти классы обеспечивают поддержку обработки исключительных ситуаций в момент динамического распределения памяти при отладке и для передачи информации из функции, где произошло исключение, в функцию его обработки.

Классы поддержки отладки

CDumpContext	служит приемником для дампа диагностики
CMemoryState	структура, которая предоставляет мгновенный снимок используемой памяти; позволяет сравнивать ранний и поздний снимки памяти
CRunTimeClass	класс поддержки контроля типов времени исполнения

Классы обработки исключений

Библиотека классов предоставляет механизм обработки исключений, базирующийся на классе *CException*. Использование объектов этих классов обеспечивает включение в код приложения обработки исключений.

CException	базовый класс для обработки исключений
CArchiveException	обработка исключений при работе с архивами
CDaoException	обработка исключений при ошибках в операциях работы с базами данных на основе DAO
CDBException	обработка исключений при ошибках в операциях работы с базами данных на основе ODBC
CFileException	обработка исключений при работе с файлами

CMemoryException	обработка исключений при работе с памятью
CNotSupportedException	обработка исключений, возникающих при попытке использования не поддерживаемых возможностей
COleException	обработка исключений при работе с механизмом OLE; этот класс используется и для сервера и для контейнера
COleDispatchException	обработка исключений при ошибках, возникающих во время автоматизации OLE
CResourceException	обработка исключений, возникающих при загрузке ресурсов Windows
CUserException	обработка исключений, используемая при остановке операций, инициированных пользователем

Глава 8

Макросы, глобальные функции и переменные

Знакомство с библиотекой будет не полным и не даст существенных преимуществ при работе с ней, если мы не рассмотрим включенные в нее макросы, глобальные функции и переменные.

Все макросы, глобальные функции и переменные можно разбить на несколько категорий: основные, работа с базами данных, работа в Internet, OLE и элементы управления OLE. Поскольку в данном томе мы не останавливаемся подробно на работе с базами данных, в сети и с использованием OLE (этим вопросам будут посвящены следующие тома книги), то ограничимся обзором основных макросов, глобальных функций и переменных. Начнем, естественно, с начала. Имена практически всех глобальных функций начинаются с префикса *Afx*. Исключение составляют большинство функций работы с базами данных и функции, обеспечивающие обмен данными. Во всех без исключения именах глобальных переменных применяется префикс *afx*, а имена всех макросов записаны заглавными буквами.

Типы данных

Большинство типов данных полностью соответствует представленным в SDK (Software Development Kit), однако есть и специфичные для MFC.

BOOL	булевское значение
BSTR	32-битный указатель на символ
BYTE	8-битное беззнаковое целое
COLORREF	32-битное значение, используемое как значение цвета
DWORD	32-битное беззнаковое целое или адрес
LONG	32-битное целое со знаком
LPARAM	32-битное значение, посылаемое в качестве параметра в оконную процедуру или функцию обратного вызова[1]

[1] *Функция обратного вызова* или CallBack-функция вызывается операционной системой по указателю, переданному ей ранее приложением.

LPCSTR	32-битный указатель на постоянную строку символов
LPSTR	32-битный указатель на строку символов
LPCTSTR	32-битный указатель на постоянную строку символов, которая переносима в Unicode и DBCS
LPTSTR	32-битный указатель на строку символов, которая переносима в Unicode и DBCS
LPVOID	32-битный указатель на неопределенный тип данных
LRESULT	32-битное значение, возвращаемое из оконной процедуры или функции обратного вызова
UINT	32-битное беззнаковое целое для Win32 и 16-битное для Windows 3.x
WNDPROC	32-битный указатель на оконную процедуру
WORD	16-битное беззнаковое целое
WPARAM	значение, посылаемое в качестве параметра в оконную процедуру или функцию обратного вызова, 32-битное для Win32 и 16-битное для Windows 3.x.
POSITION	32-битное беззнаковое целое значение, используемое для обозначения позиции элемента в коллекции
LPCRECT	32-битный указатель на немодифицируемую структуру *RECT*
HINSTANCE	32-битное беззнаковое целое для идентификации дескриптора экземпляра приложения

Имеются также некоторые другие типы данных, которые будут описаны в соответствующих разделах.

Получение информации о приложении

Когда вы пишете приложение с использованием классов MFC, вы создаете единственный объект, производный от *CWinApp*. Для получения информации об этом объекте и связанных с ним параметрах из любого места программы разработчики предоставили следующие функции:

CWinApp* **AfxGetApp**() –
возвращает указатель на единственный объект CWinApp приложения.

LPCTSTR **AfxGetAppName**() –
возвращает строку, заканчивающуюся нулем, содержащую имя приложения.

HINSTANCE **AfxGetInstanceHandle**() –
возвращает дескриптор текущего приложения для исполняемого файла (.EXE); в случае, если она вызвана из динамически подключаемой библиотеки, связанной с USRDLL-версией MFC, то возвращается HINSTANCE этой библиотеки DLL.

CWnd* **AfxGetMainWnd**() –
возвращает значение переменной m_pMainWnd объекта приложения, если оно не является сервером OLE, или указатель на активное главное окно приложения в противном случае.

HINSTANCE **AfxGetResourceHandle**() –
возвращает дескриптор загруженных по умолчанию ресурсов приложения.

void **AfxSetResourceHandle**(HINSTANCE hInstResource) —
устанавливает дескриптор экземпляра приложения или модуля .EXE или .DLL, из которого будут загружены ресурсы приложения.

LPCTSTR AFXAPI AfxRegisterWndClass(
UINT nClassStyle,
HCURSOR hCursor = 0,
HBRUSH hbrBackground = 0,
HICON hIcon = 0) —
возвращает строку, заканчивающуюся нулем, которая содержит имя зарегистрированного класса. В качестве параметров в эту функцию передаются: nClassStyle — стиль или комбинация стилей класса окна Windows, которые будут рассмотрены при описании регистрации класса окна, hCursor — дескриптор ресурса курсора, hbrBackground — дескриптор ресурса кисти, определяющей цвет фона, hIcon — дескриптор ресурса пиктограммы.

BOOL AFXAPI **AfxRegisterClass**(WNDCLASS* lpWndClass) —
возвращает значение TRUE при успешной регистрации класса окна и FALSE при ошибке. В качестве параметра в функцию передается указатель на структуру WNDCLASS, которая должна содержать информацию, необходимую для успешной регистрации. При использовании этой функции для регистрации класса окна в модуле DLL следует иметь в виду, что имя зарегистрированного класса автоматически удаляется из системы, когда библиотека выгружается из памяти.

HINSTANCE AFXAPI **AfxLoadLibrary**(LPCTSTR lpszModuleName) —
загружает в память модуль (файл .DLL или .EXE), имя которого передается в качестве параметра lpszModuleName. Если загрузка прошла успешно, то возвращается дескриптор модуля и увеличивается счетчик ссылок для этой библиотеки; в противном случае возвращается NULL.

BOOL AFXAPI **AfxFreeLibrary**(HINSTANCE hInstLib) —
уменьшает счетчик ссылок для загруженного модуля DLL, определяемого параметром hInstLib. Когда счетчик ссылок становится равным нулю, модуль выгружается из адресного пространства текущего процесса.

Модель объекта времени выполнения

Как уже отмечалось в предыдущей главе, все классы, производные от *CObject* и *CRuntimeClass* приобретают определенные возможности по обслуживанию объектов, включающие доступ к информации времени выполнения о классе, сериализацию и динамическое создание объекта.

Здесь описаны макросы, которые облегчают использование этих возможностей.

DECLARE_DYNAMIC(className) —
разрешает доступ к информации времени выполнения об объекте класса (должен использоваться при объявлении класса); className — действительное C++-имя класса без кавычек.

IMPLEMENT_DYNAMIC(className, baseClassName) —
генерирует код, необходимый для динамического получения информации времени выполнения об объекте класса, производного от CObject (должен использоваться в реализации класса).

DECLARE_DYNCREATE(className) –
разрешает динамическое создание объекта, производного от CObject, и получение доступа к информации времени выполнения об этом объекте (должен использоваться при объявлении класса).

IMPLEMENT_DYNCREATE(className, baseClassName) –
разрешает динамическое создание и доступ к информации класса времени выполнения (должен использоваться в реализации класса). Приложение использует эту возможность для динамического создания нового объекта, например, когда требуется прочитать объект из файла на диске во время процесса сериализации (должен использоваться в реализации класса).

При использовании этих макросов вы получаете возможность использования макроса RUNTIME_CLASS и функции *CObject::IsKindOf* для определения класса вашего объекта во время выполнения.

DECLARE_SERIAL(className) –
генерирует код C++, необходимый для сериализации и доступа к информации класса времени выполнения об объекте класса, производного от CObject (должен использоваться при объявлении класса). Определение этого макроса включает в себя все функциональные возможности, предоставляемые обоими макросами DECLARE_DYNAMIC и DECLARE_DYNCREATE.

IMPLEMENT_SERIAL(className, baseClassName, wSchema) –
генерирует код C++, необходимый для сериализации и доступа к информации класса времени выполнения об объекте класса, производного от CObject (должен использоваться в реализации класса). wSchema – переменная типа UINT, определяющая "номер версии" приложения, которой может быть закодирован архив, для возможности применения процесса десериализации. Этот номер не должен быть равен –1.

RUNTIME_CLASS(className) –
для объектов, производных от CObject, и объявленных с использованием макросов DECLARE_DYNAMIC, DECLARE_DYNCREATE или DECLARE_SERIAL возвращает указатель на структуру CRuntimeClass, которая соответствует имени класса className.

Диагностика объектов

Диагностический сервис, предоставляемый библиотекой MFC, значительно упрощает отладку приложений. Он включает макросы и глобальные функции, которые позволяют отслеживать распределение памяти, содержимое дампа объекта и печать отладочной информации во время выполнения.

Основные макросы

ASSERT(booleanExpression) –
прерывает выполнение программы, если вычисляемое выражение booleanExpression равно FALSE, и печатает сообщение об ошибке.

ASSERT_KINDOF(className, pObject) —
проверяет, является ли pObject объектом класса className, где className — имя класса, производного от класса CObject; этот макрос работает только в случае, если в области объявления класса используется макрос DECLARE_DYNAMIC или DECLARE_SERIAL.

ASSERT_VALID(pObject) —
используется для проверки внутреннего состояния pObject на доступность, проверяет pObject на NULL и вызывает затем его функцию-член AssertValid. Если хотя бы одна проверка приводит к возникновению ошибки, то макрос выводит сообщение, аналогично ASSERT.

TRACE(exp) —
позволяет вывести на экран форматированную строку, определяемую выражением exp, аналогично функции printf для С-программ под MS-DOS, например:

```
TRACE("Проверка вывода строк %s и чисел %d",
      "Отладка программы", 5)
```

TRACE0,..., **TRACE3** —
упрощенные версии макроса TRACE, позволяющие вывести форматированную строку с числом аргументов от 0 до 3.

Перечисленные макросы работают только в отладочной версии библиотеки.

VERIFY(booleanExpression) —
действует аналогично макросу ASSERT, но для рабочей версии библиотеки.

Основные глобальные переменные

CDumpContext **afxDump** —
предопределенная переменная, позволяющая послать информацию в окно отладчика.

BOOL **afxTraceEnable** —
используется для разрешения или запрещения работы макроса TRACE.

Форматирование строк и окно сообщения

Эта группа содержит три глобальные функции: две — для работы со строками и одну — для вывода произвольного сообщения в специальный блок диалога, называемый *окном сообщения*.

void **AfxFormatString1**(
CString& rString,
UINT nIDS,
LPCTSTR lpsz) —
загружает строку из ресурсов, определяемую идентификатором *nIDS*, в объект *rString* с одновременной заменой шаблона "%1" на строку, указанную в *lpsz*. Например, если в ресурсах определена строка "Для профессионального программирования 32-разрядных приложений читайте книгу %1", а *lpsz* указывает на строку "<Программирование для Windows 95 и Windows NT с использованием библиотеки MFC>", то после выполнения этой функции *rString* будет содер-

жать строку "Для профессионального программирования 32-разрядных приложений читайте книгу <Программирование для Windows 95 и Windows NT с использованием библиотеки MFC>".

void **AfxFormatString2**(
CString& rString,
UINT nIDS,
LPCTSTR lpsz1,
LPCTSTR lpsz2) —
аналогична функции AfxFormatString1, но использует два шаблона — "%1" и "%2", — и две строки для подстановки, соответственно.

Последняя функция этой группы имеет две формы: со строкой текста и с идентификатором ресурса:

int **AfxMessageBox**(
LPCTSTR lpszText,
UINT nType = MB_OK,
UINT nIDHelp = 0)

или

int **AfxMessageBox**(
UINT nIDPrompt,
UINT nType = MB_OK,
UINT nIDHelp = (UINT) –1) —
выводят на экран окно сообщения. Первая форма этой функции выводит в окно сообщения строку *lpszText* и использует идентификатор *nIDHelp* для перехода к соответствующей теме справки при нажатии клавиши <F1>. Вторая форма функции использует для определения текста идентификатор строки ресурса, задаваемый параметром *nIDPrompt*. Если для *nIDHelp* задано значение по умолчанию –1, то *nIDPrompt* задает тему справки.

Обе эти функции возвращают одно из следующих значений:

- IDABORT — выбрана кнопка Abort (Стоп)
- IDCANCEL — выбрана кнопка Cancel (Отмена) или нажата клавиша <Esc>
- IDIGNORE — выбрана кнопка Ignore (Пропустить)
- IDNO — выбрана кнопка No (Нет)
- IDOK — выбрана кнопка OK
- IDRETRY — выбрана кнопка Retry (Повтор)
- IDYES — выбрана кнопка Yes (Да)
- 0 — окно сообщения не может быть создано

Параметр *nType* определяет тип окна сообщения и может принимать одно из значений:

- MB_ABORTRETRYIGNORE — содержит три кнопки Abort (Стоп), Retry (Повтор) и Ignore (Пропустить)

- MB_OK — содержит только кнопку OK
- MB_OKCANCEL — содержит две кнопки OK и Cancel (Отмена)
- MB_RETRYCANCEL — содержит две кнопки Retry (Повтор) и Cancel (Отмена)
- MB_YESNO — содержит две кнопки Yes (Да) и No (Нет)
- MB_YESNOCANCEL — содержит три кнопки Yes (Да), No (Нет) и Cancel (Отмена)

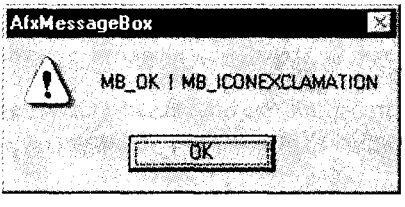
Кроме того, можно использовать следующие константы для определения степени модальности блока сообщений:

- MB_APPLMODAL — модальность на уровне пользователя
- MB_SYSTEMMODAL — модальность на уровне системы
- MB_TASKMODAL — модальность на уровне задачи

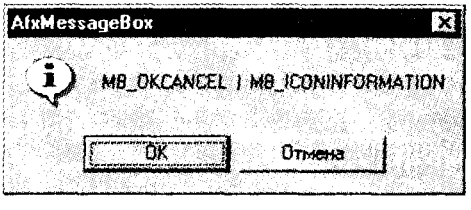
Более подробно эти вопросы будут рассмотрены при описании блоков диалога в части VI книги.

Помимо кнопок, в окне сообщения можно разместить одну из предопределенных (системных) пиктограмм:

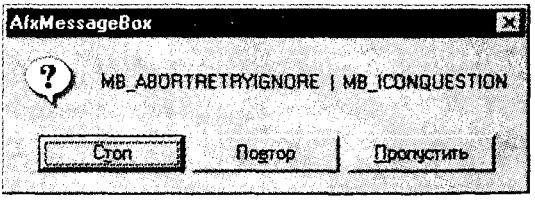
- MB_ICONEXCLAMATION — восклицательный знак



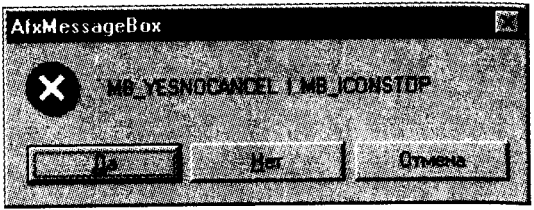
- MB_ICONINFORMATION — символ "i" (Информация)



- MB_ICONQUESTION — знак вопроса



- MB_ICONSTOP — пиктограмма "стоп"



При использовании в блоке сообщений более одной кнопки можно также установить кнопку по умолчанию:

- MB_DEFBUTTON1 — первая кнопка по умолчанию
- MB_DEFBUTTON2 — вторая кнопка по умолчанию
- MB_DEFBUTTON3 — третья кнопка по умолчанию

Часть IV

Программирование с использованием библиотеки классов MFC

9. Основные составляющие приложения на базе библиотеки классов MFC

10. Обработка сообщений в библиотеке классов MFC

Глава 9

Основные составляющие приложения на базе библиотеки классов MFC

Цель этой главы — проанализировать общую структуру программы для Windows, написанной с использованием тех возможностей, которые предоставляет библиотека классов MFC, и показать тесную взаимосвязь между классами библиотеки и интерфейсом прикладного программирования Win32 API.

Начнем с рассмотрения полного текста первого минимального приложения First, создающего простое окно с заголовком, бордюром и системным меню.

```
//////////////////////////////////////////////////////////
// Файл First.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////
#ifndef __AFXWIN_H__
  #error сначала надо включить 'stdafx.h' для поддержки PCH
#endif

class CFirstApp : public CWinApp
{
public:
  CFirstApp();

// Переопределяемые функции
  //{{AFX_VIRTUAL(CFirstApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
};
//////////////////////////////////////////////////////////
// Файл First.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////

#include "stdafx.h"
```

```
#include "First.h"
#include "MainFrm.h"

// CFirstApp
CFirstApp::CFirstApp()
{
}

// Может быть создан один и только один объект CFirstApp
CFirstApp theApp;

// Инициализация объекта CFirstApp
BOOL CFirstApp::InitInstance()
{
  const char *szWndClass = AfxRegisterWndClass(
    CS_HREDRAW | CS_HREDRAW,
    LoadStandardCursor(IDC_ARROW),
    (HBRUSH)COLOR_BACKGROUND + 1,
    LoadIcon(IDR_MAINFRAME));

  CMainFrame *pWnd = new CMainFrame;

  pWnd->CreateEx(
    0,                              // дополнительные стили
    szWndClass,                     // класс окна
    "Первая программа",             // заголовок
    WS_OVERLAPPEDWINDOW,            // стиль
    CW_USEDEFAULT, CW_USEDEFAULT,   // левый верхний угол (x, y)
    CW_USEDEFAULT, CW_USEDEFAULT,   // размеры (cx, cy)
    NULL,                           // родительское окно
    NULL);                          // идентификатор меню
  pWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd = pWnd;

  return TRUE;
}

////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

class CMainFrame : public CWnd
{
public:
  CMainFrame();

public:
virtual ~CMainFrame();
};
```

```
////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

#include "stdafx.h"
#include "First.h"
#include "MainFrm.h"

// Конструктор/деструктор CMainFrame
CMainFrame::CMainFrame()
{
}

CMainFrame::~CMainFrame()
{
}

////////////////////////////////////////////////////////
// Файл Stdafx.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

// stdafx.h : заголовочный файл, подключающий стандартные,
// наиболее часто используемые заголовочные файлы
#define VC_EXTRALEAN // Исключает редко используемые части
                     // заголовочных файлов Windows
#include <afxwin.h>  // Сердцевина MFC и стандартные компоненты

////////////////////////////////////////////////////////
// Файл Stdafx.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

// stdafx.cpp : исходный файл, который подключает только
// стандартные включаемые файлы
// First.pch : файл прекомпилированных заголовков
// stdafx.obj : файл, содержащий прекомпилированные типы информации

#include "stdafx.h"

////////////////////////////////////////////////////////
// Файл Resource.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Template.rc
//
// Next default values for new objects
//
```

```
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1010
#define _APS_NEXT_SYMED_VALUE           103
#endif
#endif

//////////////////////////////////////////////////////////
// Файл First.rc
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////

...

//////////////////////////////////////////////////////////

// Icon
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.

IDR_MAINFRAME           ICON    DISCARDABLE     "res\\First.ico"
#endif    // Russian resources

...
```

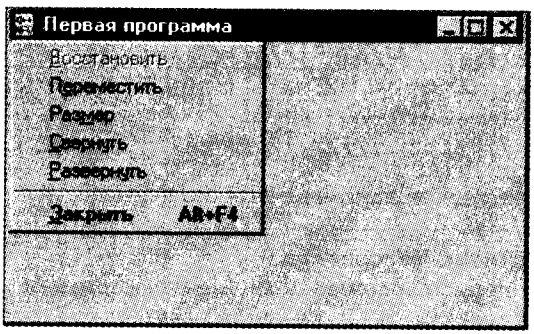


Рис. 15. Главное окно программы First

Программа First строит обычное, традиционное окно, хорошо знакомое всем пользователям системы Windows (рис. 15). Его можно переместить по экрану, закрыть, свернуть (уменьшить до размера пиктограммы) или развернуть (увеличить до размеров экрана), а также изменить его размер.

Основное назначение этого примера в том, что он позволяет нам познакомиться с принципами построения программ на базе библиотеки классов MFC.

Программа имеет два класса объектов: *CFirstApp* и *CMainWnd*. Первый позволяет создать объект "приложение", а второй — объект "главное окно". Каждое приложение, которое использует библиотеку MFC, может содержать только один объект "приложение", который конструируется вместе с другими глобальными объектами C++ (определение — *CFirstApp theApp*) и уже доступен, когда Windows вызывает функцию *WinMain*. На данном этапе знакомства это самый существенный момент для успешного программиро-

вания для Windows на базе MFC, и если вы его усвоили, то можно переходить к обсуждению имен, используемых библиотекой для классов, функций и переменных.

Соглашения об именах MFC

В качестве префикса, обозначающего имя класса, библиотека MFC использует заглавную букву "C" от слова "class" (класс), за которой идет имя, характеризующее его назначение. Например, *CWinApp* — класс, определяющий приложение, *CWnd* — базовый класс всех оконных объектов, *CDialog* — класс блоков диалога и т. д. Мы также будем придерживаться этого соглашения в приводимых примерах. Например, программа First содержит определения двух классов: *CFirstApp* и *CMainFrame*.

Для имен функций-членов классов используется три способа. При одном имя объединяет глагол и существительное, например, *LoadIcon* (Загрузить пиктограмму) или *DrawText* (Нарисовать текст). При втором имя функции-члена состоит только из существительного, например, *DialogBox* (Блок диалога). Для функций, предназначенных для преобразования одного типа в другой, обычными являются такие имена, как *XtoY* (Из X в Y).

Для членов классов библиотеки MFC принят следующий способ назначения имен: обязательный префикс *m_* (от class **m**ember — член класса), за которым идет префикс, характеризующий тип данных, и завершается все содержательным именем переменной, например, *m_pMainWnd*, где *p* — префикс, описывающий указатель (для переменных-не членов класса префикс *m_* не используется, чтобы можно было отличить обычные переменные от членов любых классов, а не только классов библиотеки MFC). В своих примерах мы придерживаемся аналогичного способа формирования имен и рекомендуем вам поступать так же.

Включаемые файлы

Вы, наверное, уже обратили внимание на то, что одной из первых в нашей программе стоит строка *#include "stdafx.h"*. Этот включаемый файл (или include-файл) служит для подключения к программе необходимых ей библиотечных файлов. При написании больших приложений вам обязательно потребуется использование заранее откомпилированных заголовочных файлов, что поможет существенно сократить требуемое время компиляции. Прекомпилированные заголовки полезны, если есть несколько заголовков, которые должны быть включены во все или в большинство ваших исходных файлов и не меняются между компиляциями. Для того чтобы продемонстрировать способ сокращения времени компиляции, мы использовали прекомпилируемый файл STDAFX.H, который должен включать в себя все наиболее часто используемые заголовочные файлы. В данном случае это

единственный большой (143 Кбайта) файл AFXWIN.H, который содержит описания основных классов библиотеки и сводит воедино все включаемые файлы, необходимые для базового функционирования MFC: AFX.H, содержащий описания классов общего назначения, макросы, базовые типы данных MFC и подключающий стандартные библиотеки времени выполнения языка C, и файл AFXRES.H, куда включены стандартные идентификаторы ресурсов. По мере изложения материала мы ближе рассмотрим классы, определяемые во включаемых файлах MFC. Здесь же отметим изменения, которые претерпел основной включаемый файл Windows — WINDOWS.H. Как и раньше, каждое приложение Windows должно содержать этот файл. Отличие заключается в том, что в нынешнем виде он уже не представляет собой огромного "монстра", а разбит на четыре сравнительно небольших файла: WINDEF.H, содержащий определения базовых типов; WINBASE.H, описывающий базовые функции 32-битного API; WINGDI.H и WINUSER.H, включающие объявления процедур GDI и USER, а также определения констант и макросов для соответствующих модулей системы.

В заключение обсуждения стандартных включаемых файлов скажем несколько слов об определении *#define VC_EXTRALEAN*, использование которого позволяет исключить из процесса компиляции редко используемые части заголовочных файлов Windows (для C- и C++-программ, не использующих MFC, используется определение *#define WIN32_EXTRA_LEAN*).

Функция WinMain

А где же функция *WinMain*, которая является точкой входа в любую программу написанную для Windows? [1] Пусть вас не удивляет, что в представленном листинге вы не найдете даже упоминания об этой функции. Дело в том, что создатели библиотеки MFC проделали большую работу и избавили вас от написания многих строк вспомогательного кода, предоставив возможность заниматься только решением конкретной задачи. Однако, при этом библиотека скрывает от программиста и часть того, что заставляет Windows работать. По этой причине мы будем углубляться в Windows немного больше, чем может показаться необходимым. Мы уверены, что

---

[1] На самом деле функция *WinMain* операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки времени выполнения C/C++. Компоновщик Visual C++ знает, что она имеет имя *_WinMainCRTStartup* и отвечает за выполнение таких операций как:

- поиск указателя на полную командную строку нового процесса,
- поиск указателя на переменные окружения нового процесса,
- инициализация глобальных переменных из библиотеки времени выполнения языка C, доступ к которым обеспечивается включением файла STDLIB.H,
- инициализация блока памяти (heap), используемой функциями выделения памяти и процедурами низкоуровневого ввода/вывода языка C,
- вызов функции *WinMain* библиотеки MFC.

приобретенное понимание позволит вам с большей легкостью использовать возможности библиотек MFC и SDK. Сказанное в полной мере относится к функции *WinMain*. Что же предлагают нам разработчики MFC в плане ее реализации?

Давайте вместе подробно рассмотрим отрывок исходного текста из библиотеки.

```
int AFXAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow)
{
  int nReturnCode = -1;
  CWinApp* pApp = AfxGetApp();

// Внутренняя инициализация
  if (!AfxWinInit(hInstance, hPrevInstance,
    lpCmdLine, nCmdShow))
    goto InitFailure;

// Глобальная инициализация приложения
  if (!pApp->InitApplication())
    goto InitFailure;

// Специфическая инициализация экземпляра приложения
  if (!pApp->InitInstance())
  {
    if (pApp->m_pMainWnd != NULL)
    {
      TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
      pApp->m_pMainWnd->DestroyWindow();
     }
    nReturnCode = pApp->ExitInstance();
   goto InitFailure;
  }

  nReturnCode = pApp->Run();
  InitFailure:
  AfxWinTerm();

  return nReturnCode;
}
```

Мы не будем повторять описание параметров, которые система Windows передает в программу. Этот вопрос рассматривался нами достаточно подробно в первой части. Вместо этого еще раз подчеркнем, что объект нашего приложения (класса *CFirstApp*) является глобальным, единственным и уже создан к моменту вызова *WinMain*. Глобальная функция *AfxGetApp*, реализо-

ванная в библиотеке MFC, как раз и занимается тем, что возвращает указатель на объект "приложение" (точнее, на его базовый класс):

```
CWinApp* pApp = AfxGetApp();
```

Для лучшего понимания тех задач, которые решает *WinMain* рассмотрим структуру этой функции, упрощенное условное изображение которой представлено на рис. 16.

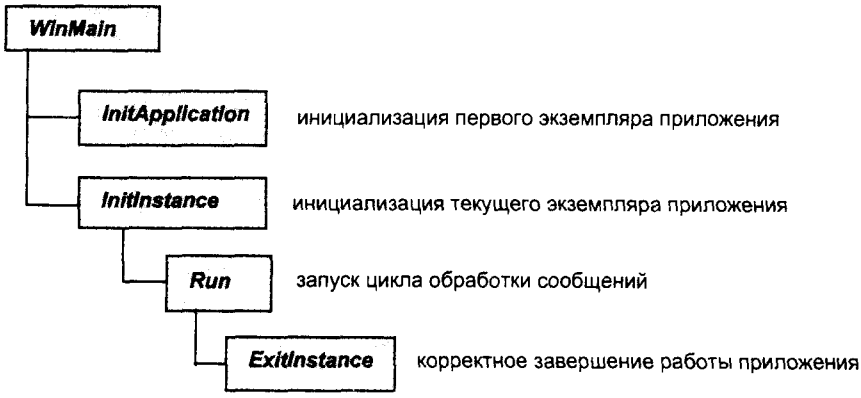
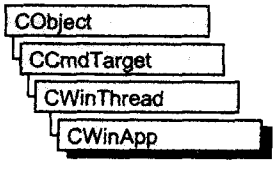


Рис. 16. Упрощенная структура функции WinMain

Все представленные здесь функции являются функциями-членами класса *CWinApp* — базового для нашего объекта "приложение" и, чтобы двигаться дальше, пришло время рассмотреть его подробнее.

Класс CWinApp



Класс *CWinApp* является базовым классом, из которого мы образуем обязательный объект-приложение Windows. Основными задачами объекта этого класса являются инициализация и создание главного окна, а затем опрос системных сообщений. Если вспомнить, что процесс — каждый выполняемый экземпляр приложения (в предыдущих версиях применялся термин "задача"), то можно сказать, что объект класса *CWinApp* отвечает за создание процесса в системе. Например, щелчком на пиктограмме WordPad вы запускаете процесс, который вызывает приложение WordPad. В действительности все происходит несколько сложнее (одновременно формируется первичный поток, за создание и функционирование которого отвечает класс *CWinThread* — базовый для *CWinApp*), но на данном этапе знакомства с библиотекой MFC нам достаточно такого упрощенного представления.

Поскольку без использования этого класса нельзя создать ни одно приложение на базе библиотеки MFC, то начнем с рассмотрения основных членов и функций-членов этого класса, определенных в AFXWIN.H. Члены класса *CWinApp*:

LPTSTR CWinApp::**m_pszAppName** –
определяет имя приложения (общедоступная переменная).

HINSTANCE CWinApp::**m_hInstance** –
копия параметра hInstance, передаваемого WinMain с логическим номером (дескриптором) приложения. Может быть получен с помощью глобальной функции AfxGetInstanceHandle (общедоступная переменная).

HINSTANCE CWinApp::**m_hPrevInstance** –
копия параметра hPrevInstance, передаваемого WinMain. В отличие от предыдущих версий Windows, где этот параметр указывал на предыдущий экземпляр приложения, для Win32 он всегда устанавливается в NULL. Различные варианты идентификации запущенных экземпляров программы, кроме уже встречавшихся, будут рассмотрены позже (общедоступная переменная).

LPTSTR CWinApp::**m_lpCmdLine** –
копия параметра lpCmdLine, передаваемого WinMain для указания на командную строку приложения (общедоступная переменная).

int CWinApp::**m_nCmdShow** –
копия параметра nCmdShow, передаваемого WinMain, который определяет режим первоначального отображения главного окна на экране (общедоступная переменная).

LPTSTR CWinApp::**m_pszExeName** –
содержит имя исполняемого модуля приложения без расширения. В общем случае не совпадает с m_pszAppName (общедоступная переменная).

LPTSTR CWinApp::**m_pszHelpFilePath** –
содержит полное имя пути к файлу справки приложения. По умолчанию инициализируется именем приложения с расширением .HLP. При изменении места расположения и/или имени файла справки необходимо переопределить эту переменную. Обычно это делается в переопределенной функции приложения InitInsnance (общедоступная переменная).

LPTSTR CWinApp::**m_pszProfileName** –
содержит имя INI-файла приложения (общедоступная переменная).

LPCTSTR CWinApp::**m_pszRegistryKey** –
определяет полный ключ реестра, где хранится профиль установок приложения (общедоступная переменная).

В заключении перечисления членов класса *CWinApp* рассмотрим один член упоминавшегося класса *CWinThread*, который нам понадобится раньше, чем мы займемся потоками:

CWnd* CWinThread::**m_pMainWnd** –
в самом общем случае служит для хранения указателя на объект главного окна текущего потока. Если этот поток является основным (первичным) потоком приложения, то в этой переменной хранится указатель на объект главного окна

приложения. Библиотека MFC автоматически завершает поток, когда закрывается окно, на которое ссылается m_pMainWnd. В случае основного (первичного) потока завершается работа всего приложения (общедоступная переменная).[1]

Оставшиеся члены класса *CWinApp* будут рассмотрены ниже, при обсуждении соответствующих тем.

Перечислим основные функции-члены класса *CWinApp.*

Поскольку создание объекта начинается с конструктора, то его и рассмотрим в первую очередь:

CWinApp::**CWinApp**(
LPCTSTR lpszAppName = NULL) —

параметр lpszAppName — указатель на текстовую строку, которая содержит имя приложения, используемое системой Windows. Если этот аргумент отсутствует или равен NULL (по умолчанию), то для формирования имени приложения конструктор использует строку из файла ресурсов с идентификатором AFX_IDS_APP_TITLE (в ней вы можете указать имя главного окна вашего приложения), а при ее отсутствии — имя исполняемого файла. Этот параметр конструктор запоминает в члене класса m_pszAppName.

Следующая группа отвечает за инициализацию и содержит функции, отвечающие за внешний вид приложения и получение информации из INI-файла или системного реестра. Сюда входят:

void CWinApp::**LoadStdProfileSettings**(
UINT nMaxMRU = _AFX_MRU_COUNT) —

загружает список последних использованных (MRU — most recently used) файлов и последнее состояние предварительного просмотра. Если nMaxMRU = 0, то список файлов не поддерживается. Параметр _AFX_MRU_COUNT определяется в файле AFXWIN.H директивой препроцессора #define _AFX_MRU_COUNT 4.

void CWinApp::**SetDialogBkColor**(
COLOREF clrCtlBk = RGB(192, 192, 192),
COLOREF clrCtlText = RGB(0, 0, 0)) —

устанавливает цвет фона (clrCtlBk) и текста (clrCtlText) для блоков диалога и окон сообщений, используемых в приложении. Макрос RGB комбинирует интенсивности (в диапазоне от 0 до 255) трех цветов (красного, зеленого и синего) для получения требуемого цвета и определен следующим образом:

COLORREF **RGB**(int red, int green, int blue)

BOOL CWinApp::**Enable3dControlsStatic**()

и

BOOL CWinApp::**Enable3dControls**() —

загружают CTL3D32.DLL для того, чтобы можно было выводить в блоках диалога и окнах элементы управления, которые имеют трехмерный внешний вид. В этом случае отпадает необходимость в вызове функции SetDialogBkColor. Первый ва-

[1] В версиях Visual C++ до 4.0 эта переменная принадлежала классу *CWinApp.*

риант функции используется при работе со статической, а второй — с динамической библиотекой. MFC автоматически поддерживает трехмерное представление для объектов следующих классов: CDialog, CDialogBar, CFormView, CPropertyPage, CPropertySheet, CControlBar и CToolBar. При необходимости получения трехмерного эффекта для других элементов управления следует непосредственно вызывать функции библиотеки API CTL3D32.DLL.

void CWinApp::**SetRegistryKey**(
LPCTSTR lpszRegistryKey)

и

void CWinApp::**SetRegistryKey**(
UINT nIDRegistryKey) —

заставляют приложение сохранять свои установки в системном реестре вместо INI-файла, устанавливая для этого член класса m_pszRegistryKey в значение, передаваемое в качестве параметра функции. При вызове этой функции список MRU-файлов также сохраняется в реестре. Параметры: lpszRegistryKey — указатель на строку, содержащую имя ключа; nIDRegistryKey — идентификатор/индекс ключа в реестре. Ключ реестра обычно хранится в следующем формате:

HKEY_CURRENT_USER\Software\<Название Организации>\
<ИмяПриложения>\<ИмяСекции>\<ИмяПеременной>

void CWinApp::**EnableShellOpen**() —

разрешает пользователям приложения открывать файлы данных, обрабатываемые им, двойным щелчком на имени файла. Используется совместно с функцией RegisterShellFileTypes.

void CWinApp::**RegisterShellFileTypes**() —

регистрирует все типы документов вашего приложения в регистрационной базе данных. Эту функцию следует вызывать после формирования всех типов шаблонов документов, с которыми вы будете работать. Если регистрационная база данных уже ассоциирует расширение имени файла с другим типом файла, то новое соответствие не устанавливается. Используется совместно с функцией EnableShellOpen.

Функции следующей группы (операции) позволяют определить курсор и пиктограмму приложения, получить из INI-файла или записать в него необходимую информацию, произвести анализ параметров, передаваемых из командной строки, и выполнить некоторые другие операции:

HCURSOR CWinApp::**LoadCursor**(
LPCTSTR lpszResourceName)

и

HCURSOR CWinApp::**LoadCursor**(
UINT nIDResource) —

загружают ресурс курсора, определяемый строковой константой lpszResourceName или номером идентификатора ресурса nIDResource, из текущего выполняемого файла. Курсор загружается в память, если он не был предварительно туда загружен; в последнем случае функции возвращают дескриптор существующего ресурса. Для загрузки предопределенных курсоров Windows следует использовать функции LoadStandardCursor или LoadOEMCursor.

HICON CWinApp::**LoadIcon**(
LPCTSTR lpszResourceName)

и

HICON CWinApp::**LoadIcon**(
UINT nIDResource) —

действие этих функций и параметры аналогичны LoadCursor, но для определения пиктограммы приложения. Для загрузки предопределенных значков Windows следует использовать функции LoadStandardIcon или LoadOEMIcon.

void CWinApp::**ParseCommandLine**(
CCommandLineInfo& rCmdInfo) —

осуществляет анализ командной строки и посылает параметры в член ParseParam класса CCommandLineInfo. Перед вызовом следует создать локальный объект класса CCommandLineInfo и предать ссылку на него в качестве параметра функции. При использовании AppWizard это производится автоматически.

BOOL CWinApp::**ProcessShellCommand**(
CCommandLineInfo& rCmdInfo) —

обрабатывает аргументы командной строки и флаги.

UINT CWinApp::**GetProfileInt**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
int nDefault) —

при успешном завершении возвращает целое значение, заданное текстовой строкой lpszEntry внутри секции, определяемой строкой текста lpszSection, из INI-файла приложения или реестра, если функция не нашла запись, то возвращается значение параметра nDefault. Эта функция поддерживает шестнадцатеричную нотацию для значений в INI-файле или реестре.

BOOL CWinApp::**WriteProfileInt**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
int nValue) —

записывает целое значение nValue в INI-файл приложения или реестр, в секцию, заданную строкой lpszSection, на место записи, определяемой текстовой строкой lpszEntry. Если секция и/или запись не определены, то они создаются этой функцией.

CString CWinApp::**GetProfileString**(
LPCTSTR lpszSection,
LPCTSRT lpsz Entry,
LPCTSTR lpszDefault = NULL) —

возвращает значение строки из INI-файла приложения или реестра или lpszDefault, если строка не найдена. Максимальная поддерживаемая длина строки определяется параметром _MAX_PATH и составляет 256 символов, включая завершающий нуль. Если lpszDefault = NULL, то возвращается пустая строка. По принципу действия аналогична функции GetProfileInt.

BOOL CWinApp::**WriteProfileString**(
LPCTSTR lpszSection,
LPCTSRT lpsz Entry,
LPCTSTR lpszValue) —

записывает строку lpszValue в определенную секцию INI-файла приложения или реестра, аналогично WriteProfileInt для целых чисел.

BOOL CWinApp::**GetProfileBinary**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
LPBYTE* ppData,
UINT* pBytes) —
используется для чтения бинарного массива данных из INI-файла приложения или реестра, аналогично функции GetProfileInt для целых чисел. В случае успеха возвращает TRUE, параметр ppData содержит указатель на прочитанный массив данных, длина которого содержится по адресу pBytes. Память под данные, отведенную функцией GetProfileBinary, необходимо самостоятельно освободить после использования.

BOOL CWinApp::**WriteProfileBinary**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
LPBYTE pData,
UINT nBytes) —
используется для записи в INI-файл приложения или реестр бинарного массива данных, заданного адресом (pData) и длиной (nBytes).

Следующая важная группа членов-функций, которую мы должны рассмотреть для того, чтобы двигаться дальше — переопределяемые (виртуальные) функции. В документации на MFC обращается внимание на следующие функции этой группы: *InitInstance, Run, ExitInstance и OnIdle.*

virtual BOOL CWinApp::**InitInstance**() —
отвечает за инициализацию каждого нового экземпляра приложения.

Если посмотреть поставляемые исходные тексты, то можно увидеть, что в своем первозданном виде эта функция не делает "ничего":

```
BOOL CWinApp::InitInstance()
{
  return TRUE;
}
```

и наполнение ее содержанием целиком является задачей программиста. Обычно это то место, где конструируется объект "главное окно" приложения, загружаются стандартные настройки из INI-файла или из реестра Windows, обрабатывается командная строка текущего экземпляра приложения, создается новый или открывается существующий документ, регистрируются шаблоны документов, создаются сами документы, их представления (view) и ассоциированные с ними окна (этот процесс будет подробно рассмотрен в следующем томе). После создания главного окна переменная *CWinThread::m_pMainWnd* устанавливается равной значению указателя на это окно. Это единственная функция-член *CWinApp*, которую вы должны обязательно переопределить для того, чтобы ваше приложение что-нибудь делало.

virtual BOOL CWinApp::**ExitInstance**() —

вызывается только из функции *Run* для завершения работы текущего экземпляра приложения. Реализованная версия этой функции записывает сохраняемые настройки в INI-файл приложения. Если есть необходимость в проведении специальной обработки при завершении приложения, такой как возврат памяти, распределенной во время работы, очистка контекста устройства или других ресурсов, то следует переопределить эту функцию. Освобождение стандартных элементов, таких как документы и их представления (view), производится стандартными средствами библиотеки классов MFC и не должно беспокоить разработчиков программного обеспечения.

virtual int CWinApp::**Run**() —

запускает цикл обработки сообщений, в котором получаются и распределяются сообщения Windows до тех пор, пока не поступит сообщение WM_QUIT, инициирующее завершение работы приложения. Все поступающие сообщения направляются в функцию PreTranslateMessage и затем в стандартную функцию Windows TranslateMessage для поддержки работы с клавиатурой. В заключение вызывается функция Windows DispatchMessage. Если очередь сообщений пуста, то Run вызывает виртуальную функцию OnIdle для выполнения фоновой обработки. Эта функция не требует переопределения, хотя и допускает его.

virtual BOOL CWinApp::**OnIdle**(LONG lCount) —

вызывается в цикле обработки сообщений, когда очередь сообщений приложения пуста. Функция, выполняемая по умолчанию, обновляет команды объектов интерфейса пользователя, такие как элементы меню и кнопки панелей инструментов, и очищает внутренние структуры данных. Если вы переопределяете эту функцию, то в новой версии необходимо сначала вызвать базовую функцию CWinApp::OnIdle с параметром lCount. Параметр lCount увеличивается перед каждым вызовом функции, т. е. всякий раз, когда очередь сообщений приложения пуста, и сбрасывается в 0 при обработке нового сообщения. Этот параметр можно использовать для определения различных режимов обработки или относительного времени "простоя" приложения.

virtual void CWinApp::**HideApplication**() —

скрывает приложение до закрытия открытых документов.

virtual void CWinApp::**DoWaitCursor**(
int nCode) —

эта функция вызывается приложением при создании объекта класса CWaitCursor, выполнении функций CCmdTarget::BeginWaitCursor, CCmdTarget::EndWaitCursor и CCmdTarget::RestoreWaitCursor. Реализация по умолчанию выводит курсор в виде песочных часов и поддерживает счетчик ссылок. Если его значение положительно, то курсор отображается, а если отрицательно, то нет. Параметр nCode определяет появление или скрытие курсора "песочные часы": nCode = 1 — курсор появляется, nCode = 0 — курсор восстанавливается без увеличения счетчика ссылок и nCode = –1 — курсор убирается с экрана. При переопределении функции можно изменить вид курсора, добавить дополнительную обработку на время его изображения и т. д.

*virtual int CWinApp::**DoMessageBox**(*
LPCTSTR lpszPrompt,
UINT nType,
UINT nIDPrompt) —

вызывается глобальной функцией AfxMesageBox для вывода на экран окна сообщений; непосредственный вызов этой функции не рекомендуется. Параметр

lpszPrompt определяет адрес строки текста, выводимой в окно; nType — стиль окна сообщений; nIDPrompt — индекс строки контекстной справки, задаваемый в файле ресурсов. Можно переопределить эту функцию для настройки широкого круга процессов, вызывающих AfxMessageBox.

Мы не затронули некоторые имеющиеся виртуальные функции-члены, которые определяют работу с документами. Это большая, и во многом основная для работы с библиотекой классов MFC, тема, которой будет посвящена отдельная часть следующего тома.

Теперь, после подробного знакомства с классом *CWinApp*, можно вернуться к нашей первой программе.

Минимальная программа для Windows

Напомним еще раз основные этапы создания и работы нашей первой программы.

1. Создается единственный глобальный объект класса приложения *CFirstApp* — *theApp*. Программа First использует класс *CWinApp* как базовый для собственного класса *CFirstApp*, который имеет две функции — конструктор и функцию, переопределяющую аналогичную функцию родительского класса:

```
class CFirstApp : public CWinApp
{
public:
  CFirstApp();

// Переопределяемые
// ClassWizard описывает здесь виртуальные функции
  //{{AFX_VIRTUAL(CFirstApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
};
```

2. После создания объекта *theApp* операционная система вызывает функцию *WinMain* — точку входа в программу, написанную для Windows. Реализация этой функции в библиотеке MFC выполняет следующие действия:

   - определяет указатель на объект "приложение"

     ```
     CWinApp* pApp = AfxGetApp();
     ```

   - вызывает функцию, отвечающую за инициализацию первого экземпляра вашей программы

     ```
     pApp->InitApplication();
     ```

   - вызывает функцию, отвечающую за инициализацию текущего экземпляра приложения

     ```
     pApp->InitInstance();
     ```

- запускает цикл обработки сообщений

```
pApp->Run();
```

- завершает функцию *WinMain* при завершении работы приложения.

При всей своей лаконичности функция *WinMain* имеет все составляющие, необходимые для выполнения программы в Windows. Из этих составляющих наибольший интерес представляют инициализация текущего экземпляра и цикл обработки сообщений, поэтому рассмотрим их подробнее. Любая программа, написанная на базе библиотеки классов MFC, переопределяет функцию *InitInstance*. В примере First это выглядит так:

```
BOOL CFirstApp::InitInstance()
{
  const char *szWndClass = AfxRegisterWndClass(
    CS_HREDRAW | CS_HREDRAW,
    LoadStandardCursor(IDC_ARROW),
    (HBRUSH)COLOR_BACKGROUND + 1,
    LoadIcon(IDR_MAINFRAME));

  CMainWnd *pWnd = new CMainWnd;

  pWnd->CreateEx(
    0,                          // дополнительные стили
    szWndClass,                 // класс окна
    "Первая программа",         // заголовок
    WS_OVERLAPPEDWINDOW,        // стиль
    CW_USEDEFAULT,              // координата (x) верхнего левого угла
    CW_USEDEFAULT,              // координата (y) верхнего левого угла
    CW_USEDEFAULT, CW_USEDEFAULT,    // размеры окна (cx, cy)
    NULL,                            // родитель
    NULL);                           // идентификатор меню
  pWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd = pWnd;

  return TRUE;
}
```

Основное назначение этой функции, как вы помните, заключается в проведении некоторых инициализирующих действий (в нашей функции отсутствуют) и создание главного окна приложения, для чего необходимо, прежде всего, зарегистрировать класс нашего окна (подробно к вопросу регистрации мы вернемся в следующем разделе):

```
const char *szWndClass = AfxRegisterWndClass(
  CS_HREDRAW | CS_HREDRAW,
  LoadStandardCursor(IDC_ARROW),
  (HBRUSH)COLOR_BACKGROUND + 1,
  LoadIcon(IDR_MAINFRAME));
```

а затем создать экземпляр одного из оконных классов MFC:

```
CMainWnd *pWnd = new CMainWnd;
```

и создать собственно окно.

После того как объект окна MFC создан, указатель на этот объект присваивается переменной *CWinThread::m_pMainWnd* (как уже отмечалось, такое присвоение необходимо для того, чтобы иметь возможность корректно завершить приложение, когда будет закрыто его главное окно):

```
m_pMainWnd = pWnd;
```

можно создавать окно Windows и выводить его на экран

```
m_pMainWnd->ShowWindow(m_nCmdShow);
```

По завершении инициализации основная работа объекта "приложение" состоит в том, чтобы обеспечить передачу сообщений между пользователем и оконным объектом.[1] К этим вопросам мы еще вернемся, а пока нам надо рассмотреть процессы регистрации оконного класса и создания окна.

Регистрация класса окна

Любое окно Windows принадлежит одному из существующих в системе классов, который должен быть определен до того, как окно будет отображено на экране. Класс окна задает наиболее общие свойства окон, например форму курсора при перемещении его в область окна или имя меню, определенного для окон этого класса.

В библиотеке MFC для выполнения операции регистрации определены две глобальные функции — *AfxRegisterClass* и *AfxRegisterWndClass*. Первая требует заполнения полей структуры WNDCLASS, а вторая делает это сама, запрашивая только некоторые параметры. Но обо всем по порядку.

В Win32 для регистрации класса окна используются два типа структур: привычная *WNDCLASS* и новая *WNDCLASSEX*, которая отличается наличием двух полей — *cbSize* (определяет размер структуры в байтах и вычисляется как *sizeof(WNDCLASS)*) и *hIconSm* (задает идентификатор маленькой пиктограммы, которая будет отображаться области системного меню; в Windows NT это поле не используется и должно быть установлено в NULL).

---

[1] Для тех читателей, которым не терпится посмотреть на результаты своего труда и запустить эту программу, мы предлагаем обратиться к приложению, где подробно описан процесс создания проектов различных типов. После создания пустого проекта следует выбрать пункт Settings (Установки) в меню Build (Построение) для вывода на экран блока диалога Project Settings (Установки проекта). Затем, на вкладке General (Общие) необходимо заменить установку Not Using MFC (Без использования MFC) на любую другую в зависимости от того, с какой библиотекой вы предпочитаете работать — с динамической или со статической. Закончив установку параметров, можно выполнять построение и запуск вашего приложения, не забывая включать создаваемые файлы в проект.

Введение новой структуры пока не привело к ее широкому использованию и в MFC по-прежнему используется *WNDCLASS*, которая однако сменила "место прописки" и теперь располагается в WINUSER.H:

```
typedef struct tagWNDCLASSA {
  UINT style;
  WNDPROC lpfnWndProc;
  int cbClsExtra;
  int cbWndExtra;
  HINSTANCE hInstance;
  HICON hIcon;
  HCURSOR hCursor;
  HBRUSH hbrBackground;
  LPCSTR lpszMenuName;
  LPCSTR lpszClassName;
} WNDCLASS;
```

Рассмотрим поля этой структуры:

- *lpszClassName* — указатель на строку, содержащую имя класса. Поскольку после регистрации класс становится доступен всем другим приложениям Windows, имя класса должно быть уникальным;
- *lpfnWndProc* — определяет адрес функции окна, которая обрабатывает сообщения для окон данного класса. В программах на базе MFC эта функция следит за вызовом правильной функции-обработчика сообщений;
- *hInstance* — сообщает Windows о том, кто создает определение класса. Это необходимо для служебных действий внутри Windows. Когда завершается последний экземпляр программы, Windows удаляет все связанные определения классов;
- *cbClsExtra* — задает число байт, которое необходимо дополнительно запросить у Windows под эту структуру для хранения собственных данных, присоединенных к классу;
- *cbWndExtra* — задает число байт, которое необходимо дополнительно запросить у Windows для размещения всех структур, создаваемых совместно с данным классом, для хранения собственных данных, присоединенных к окну;
- *hIcon* — определяет пиктограмму, которая будет использоваться для изображения приложения, например на панели задач. Вы можете создать пиктограмму сами или воспользоваться какой-либо из предопределенных:

IDI_APPLICATION	стандартная пиктограмма для приложения
IDI_HAND	"стоп"
IDI_QUESTION	"вопросительный знак"
IDI_EXCLAMATION	"восклицательный знак"
IDI_ASTERISK	"информация"

- *hCursor* — идентифицирует курсор мыши, используемый в данном окне по умолчанию. Если нет желания создавать свой курсор, то можно воспользоваться одним из предоставляемых системой:

IDC_ARROW	стрелка
IDC_IBEAM	вертикальная черта
IDC_WAIT	"песочные часы"
IDC_CROSS	перекрестье
IDC_UPARROW	вертикальная стрелка
IDC_SIZE	четыре стрелки, указывающие в разные стороны
IDC_ICON	курсор, используемый при переносе файлов
IDC_SIZENWSE	двунаправленная стрелка "северо-запад/юго-восток"
IDC_SIZENESW	двунаправленная стрелка "северо-восток/юго-запад"
IDC_SIZEWE	двунаправленная стрелка "восток-запад"
IDC_SIZENS	двунаправленная стрелка "север-юг"

- *hbrBackground* — задает цвет фона для окна. Для его определения можно воспользоваться любой из 20 системных констант цвета, которые определены в файле WINUSER.H следующим образом:

```
#define COLOR_BACKGROUND     1
...
#define COLOR_APPWORKSPACE   12
```

  При использовании этих констант к их значению обязательно нужно добавлять единицу, т. к. первое значение для них равно нулю, а ноль является недействительным значением для логического номера кисти. Кроме того необходимо выполнить приведение к типу HBRUSH, иначе компилятор выдаст ошибку;

- *lpszMenuName* — указатель на имя меню окна, определенное в файле ресурсов;

- *style* — определяет свойства окна, которые могут комбинироваться при помощи оператора | (ИЛИ). При программировании для Windows этот термин относится к набору параметров, каждый из которых управляется одним или двумя битами. Если этому полю присвоить значение NULL, то Windows автоматически установит значения по умолчанию. Для того чтобы больше не возвращаться к этому вопросу, приведем здесь список допустимых стилей:

CS_BYTEALIGNCLIENT	оказывает влияние на ширину окна, задавая выравнивание клиентской области по границе байтов; такое выравнивание позволяет увеличить производительность операций рисования в окне

CS_BYTEALIGNWINDOW	устанавливает выравнивание окна по границе байтов; такое выравнивание позволяет увеличить производительность некоторых типов операций, таких как перемещение и изменение размеров окна, рисование пунктов меню
CS_CLASSDC	обеспечивает контекст устройства, используемый совместно всеми окнами в классе
CS_DBLCLKS	служит для установки таймера после получения первого сообщения о нажатии кнопки мыши; сообщение о двойном щелчке будет сформировано только в том случае, если второе нажатие произойдет до истечения определенного времени таймера; обеспечивает всем окнам класса возможность воспринимать сообщения о двойном щелчке на одной из кнопок мыши
CS_GLOBALCLASS	позволяет приложению создавать окно этого класса, не считаясь со значением параметра *hInstance;* если вы не определите этот стиль, то параметр *hInstance*, передаваемый в функции создания окна, должен быть таким же, как передаваемый в функцию *RegisterClass;* другими словами, установка этого стиля позволяет создавать оконные классы, предназначенные для совместного использования несколькими программами
CS_HREDRAW	определяет, что окно будет перерисовываться, как только изменится его горизонтальный размер
CS_VREDRAW	определяет, что окно будет перерисовываться, как только изменится его вертикальный размер
CS_NOCLOSE	запрещает пункт Close (Закрыть) системного меню; этот стиль следует использовать с окнами, имеющими системное меню, которые не могут быть закрыты пользователем
CS_OWNDC	предоставляет частный контекст устройства для каждого окна в данном классе; этот тип контекста устройства наиболее расточителен в смысле использования памяти, однако обеспечивает самую высокую скорость реакции
CS_PARENTDC	устанавливает, что дочерние окна не могут рисовать в своем родительском окне; этот стиль обычно применяется для предварительно определенных классов, необходимых для создания элементов управления блоков диалога
CS_SAVEBITS	предписывает системе сохранять копию части экрана, поврежденной окном. После удаления окна поврежденная часть может быть восстановлена. При этом Windows не посылает сообщения WM_PAINT для перерисовки области, которую занимало окно. Этот стиль применяется для небольших окон (например, меню или блоков диалога), которые появляются на экране на короткие промежутки времени

Теперь, когда вы ознакомились с полями структуры *WNDCLASS* можно их осмысленно заполнить и, передав адрес структуры в качестве параметра в функцию *AfxRegisterClass*, произвести регистрацию класса окна Windows. Данный подход ничем не отличается от того, который применяется при программировании с использованием Win32 API. Совершенно естественно, что в библиотеке есть функция, которая выполняет большую часть работы по регистрации класса окна. Это описанная ранее глобальная функция *AfxRegisterWndClass*, которая позволяет пользователю определить стиль, цвет фона, курсор и пиктограмму для окон, создаваемых на базе этого класса. Остальные поля структуры класса заполняются автоматически. Обе функции возвращают имя зарегистрированного класса.

Однако, чаще всего вообще не нужно думать о методе регистрации. Дело в том, что в библиотеке MFC созданы предопределенные классы окон, достаточные практически во всех возможных случаях:

"AfxWnd"	используется для всех дочерних окон, создаваемых при помощи функции CWnd::Create; для этого класса определен курсор в виде стандартной стрелки и стиль CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW
"AfxFrameOrView" [1]	используется для стандартных окон, созданных на базе классов CFrameWnd и CMDIChildWnd, и всех представлений; для этого класса определен курсор в виде стандартной стрелки, стиль CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW, цвет фона WINDOW_COLOR и пиктограмма для окон SDI
"AfxMDIFrame" [2]	используется для окон MDI, созданных при помощи функции CMDIFrame::Create; для этого класса определен курсор в виде стандартной стрелки, стиль CS_DBLCLKS и пиктограмма для окон MDI
"AfxControlBar"	используется для стандартных элементов управления; для этого класса определен курсор в виде стандартной стрелки и серый цвет фона (COLOR_BTNFACE)

Теперь, когда класс окна зарегистрирован, можно переходить непосредственно к созданию главного окна нашего приложения.

---

[1] Если для окон многодокументного интерфейса вы захотите использовать различные пиктограммы, то следует зарегистрировать дополнительные классы окон или использовать функцию *CFrameWnd::LoadFrame*, которая автоматически регистрирует класс, используя стандартные атрибуты "*AfxFrameOrView*". Параметр, передаваемый в эту функцию определяет идентификатор ресурса, содержащего пиктограмму. Более подробно об этом мы расскажем при описании класса *CFrameWnd*.

[2] Для клиентской области окна, созданного на базе класса *CMDIFrameWnd*, значения курсора и цвета фона не определяются. Дело в том, что эта область "покрывается" специальным окном MDICLIENT, которое использует только стандартные цвета и типы курсора. Более того, фирма Microsoft не поощряет создания каких-либо подклассов MDICLIENT.

Создание окна

Каждая программа на базе MFC, которую вы создаете, будет содержать, по крайней мере, один класс, определяющий оконный объект. Приложение First определяет свой класс *CMainFrame*, который базируется на одном из предопределенных классов, а именно *CWnd*, и полностью использует тот прочный фундамент, который заложила библиотека MFC, не добавляя ничего своего:

```
class CMainFrame : public CWnd
{
public:
  CMainFrame();
  virtual ~CMainFrame();
};
```

В действительности все еще проще, поскольку этот класс можно было вообще не определять, т. к. обе его функции-члены пустые. Мы включили его только для наглядности. Всю работу выполняет базовый класс нашего окна *CWnd*. Еще раз повторим — окно создается в два-три шага: сначала определяется оконный класс, затем создается объект "окно", и наконец окно делается видимым.

Теперь о создании собственно окна Windows. Мы использовали для этой цели функцию *CWnd::CreateEx*:

```
pWnd->CreateEx(
  0,                          // дополнительные стили
  szWndClass,                 // имя класса
  "Первая программа",         // заголовок
  WS_OVERLAPPEDWINDOW,        // стиль
  CW_USEDEFAULT,              // координата (x) левого верхнего угла
  CW_USEDEFAULT,              // координата (y) левого верхнего угла
  CW_USEDEFAULT, CW_USEDEFAULT, // размеры окна (cx, cy)
  NULL,                         // родитель
  NULL);                        // идентификатор меню
```

В дополнение к *CreateEx* класс *CWnd* имеет вторую функцию для создания окон — *Create*. Эти функции практически идентичны и имеют два отличия: во-первых, функция *Create* позволяет создавать лишь дочерние окна, в то время как *CreateEx* — также перекрывающиеся и вспомогательные, и, во-вторых, *CreateEx* содержит один дополнительный параметр (*Ex* означает Extended, т. е. расширенный). Ввиду того что нам надо было создать главное окно программы, которое не может быть дочерним, мы воспользовались функцией *CreateEx*. Параметры обеих функций подробно описаны в главе 11, где речь идет о классах окон библиотеки MFC.

После возврата из *CreateEx* система Windows создает у себя всю внутреннюю информацию, необходимую для сопровождения окна, т. е. вносит запись в базу, содержащую информацию об окнах. Но в большинстве случаев окно на экране еще не появляется — требуется вызвать функцию *ShowWindow*, которая также принадлежит классу *CWnd*:

```
m_pMainWnd->ShowWindow(m_nCmdShow)
```

В зависимости от значения *m_nCmdShow* программа может при вызове иметь свернутое, обычное или развернутое окно. Вызовом *ShowWindow* завершается инициализация оконного объекта. Такая последовательность действий при инициализации типична для приложений Windows.

Глава 10

Обработка сообщений в библиотеке классов MFC

Цикл обработки сообщений MFC

При рассмотрении нашей первой программы, мы совершенно не затронули такого важного вопроса, как обработка сообщений. И дело тут, как вы уже догадались, в том, что библиотека MFC опять взяла на себя решение этой важнейшей задачи. Ниже представлена функция *Run* из класса *CWinThread*, который отвечает за работу с потоками:

```
int CWinThread::Run()
{
  // переменные для отслеживания состояния времени ожидания
  BOOL bIdle = TRUE;
  LONG lIdleCount = 0;

  // получение и распределение сообщений, пока не получено
  // сообщение WM_QUIT
  for (;;)
  {
    // фаза1: проверка возможности работы в период ожидания
    while(bIdle &&
      !::PeekMessage(&m_msgCur, NULL,
      NULL, NULL, PM_NOREMOVE))
    {
      // вызов OnIdle, пока установлена переменная bIdle
      if (!OnIdle(lIdleCount++))
        bIdle = FALSE; // присвоить состояние "нет ожидания"
    }
    // фаза2: подкачка сообщений, пока возможно
    do
    {
      // подкачка сообщения, и выход по WM_QUIT
      if (!PumpMessage())
        return ExitInstance();
```

```
        // сброс состояния "нет ожидания" после подкачки
        // "нормального" сообщения
        if (IsIdleMessage(&m_msgCur))
        {
          bIdle = TRUE;
          lIdleCount = 0;
        }
      }
      while(::PeekMessage(&m_msgCur, NULL,
        NULL, NULL, PM_NOREMOVE));
    }
}
```

Хотя показанный цикл внешне и отличается от стандартного (см. главу 6), но практически по всем компонентам они совпадают. Сообщения выбираются из очереди при помощи функции *PeekMessage*, а работу по поддержке клавиатуры, выбору оконной процедуры и передаче в нее сообщений выполняет недокументированная функция *PumpMessage* из класса *CWinThread* (вы можете в этом убедиться, если изучите ее исходный текст в файле ..\MFC\SRC\THRDCORE.CPP).

Основное отличие заключается в наличии специального механизма, позволяющего перехватить у системы циклы фоновой обработки путем вызова переопределяемой функции *OnIdle* класса вашего приложения. Такая возможность появляется, когда приложение принимает сообщения с помощью *PeekMessage*. Если очередь сообщений пуста, то функция возвращает FALSE. Отметим, что цикл обработки сообщений MFC требует специальной обработки сообщения WM_QUIT, поскольку *PeekMessage* не реагирует на это сообщение так, как это делает функция *GetMessage*. Эту работу выполняет упомянутая функция *PumpMessage*.

И еще несколько слов в заключение. Представленный цикл включает две фазы: первая предназначена для внутреннего употребления библиотекой MFC, а во время второй вы получаете возможность выполнять любые задачи. Более того, если очередь приложения остается пустой "достаточно длительное время", то вы можете получить дополнительные вызовы функции *OnIdle* в течение одного цикла фоновой обработки. При этом параметр *lIdleCount* позволяет отслеживать номер вызова. Но следует иметь в виду, что при поступлении очередного сообщения *lIdleCount* сбрасывается в нуль.

Конечно, не всякая программа в системе Windows требует фоновой обработки, и можно просто игнорировать эту возможность. Тем более, что реализация функции *OnIdle* в классе *CWinApp* обновляет объекты интерфейса пользователя и очищает внутренние структуры данных, т. е. выполняет за вас всю черновую работу.

Однако, рассмотренный цикл обработки сообщений не объясняет, каким образом организуется обработка сообщений. Неужели необходимо пере-

определять оконную процедуру? Для ответа на этот вопрос необходимо рассмотреть, как организована обработка сообщений в библиотеке классов MFC.

Категории сообщений

В библиотеке MFC принята классификация сообщений, отличная от рассмотренной ранее. Все возможные сообщения разделены на три основные категории:

- сообщения Windows,
- извещения элементов управления,
- командные сообщения (команды).

В первую категорию входят сообщения, имена которых начинаются с префикса WM_, за исключением WM_COMMAND. Все сообщения этой категории предназначены для обработки окнами и представлениями и часто содержат параметры, которые определяют алгоритм обработки того или иного сообщения. Сюда входят, например, аппаратные сообщения, сообщения обслуживания окна и т. д.

Вторая категория включает *извещения* (notification messages) от элементов управления и других дочерних окон, направляемые своим родительским окнам. Например, элемент управления LISTVIEW посылает своему родительскому окну сообщение WM_COMMAND, содержащее код извещения LVN_SETDISPINFO, когда требуется обновить информацию об элементах списка. Оконная процедура отвечает на полученное извещение заполнением структуры LV_DISPINFO и передачей ее обратно в элемент управления. Механизмы передачи извещений и других WM_-сообщений аналогичны, с одним единственным исключением. Извещение BN_CLICKED, посылаемое элементом управления BUTTON, когда пользователь щелкает по нему, трактуется как командное сообщение и передается аналогично другим командам.

Сообщения этих двух категорий предназначены для объектов классов, образованных от базового класса *CWnd*, т. е. имеющих дескриптор HWND окна Windows. Их обработка осуществляется в соответствующей оконной процедуре.

Наконец, третья категория охватывает все сообщения WM_COMMAND, называемые *командами* (или *командными сообщениями*), от объектов интерфейса пользователя, который включает меню, кнопки панелей инструментов и клавиши-акселераторы. Обработка команд отличается от обработки других сообщений и может производиться множеством объектов, включающим документы, шаблоны документов и сам объект "приложение".

Но независимо от способа разделения сообщений по тем или иным категориям, они существуют для того, чтобы их обрабатывали, т. е. при получении

сообщения некоторая часть программы должна выполнить определенную последовательность действий. При рассмотрении оконной процедуры говорилось, что поступающие в нее сообщения обрабатываются индивидуально. Там же мы упомянули о существовании специального типа функций, каждая из которых отвечает за обработку одного и только одного сообщения. Аналогичный подход реализован и в библиотеке классов MFC, где для обработки каждого отдельного сообщения используется специальная функция-обработчик. Все функции-обработчики являются функциями какого-либо класса, и для них используются названия: *функция-член обработчик сообщения*, *функция обработчик сообщения* или просто *обработчик сообщения*.[1] Принципы работы обработчиков сообщений не изменяются от того, к какой категории относится сообщение, однако механизмы их вызова различны. Но прежде чем рассматривать механизмы посылки и получения сообщений, необходимо понять, каким образом сопоставляются сообщение и соответствующий ему обработчик.

Карта сообщений

Библиотека классов MFC предоставляет альтернативу многочисленным операторам switch, используемым в традиционных Windows-программах, для обработки сообщений, посылаемых окну.[2] Взаимосвязь сообщений и их обработчиков может быть определена таким образом, что когда сообщение поступает в оконную процедуру, автоматически вызывается соответствующий обработчик. Реализация такой взаимосвязи основана на понятии *карты сообщений* (message map).

Карта сообщений представляет собой механизм пересылки сообщений и команд Windows в окна, документы, представления и другие объекты приложения, реализованного на базе MFC. Такие карты преобразуют сообщения Windows, извещения элементов управления, а также команды меню, кнопок панелей инструментов, акселераторов клавиатуры в функции соответствующих классов, которые их обрабатывают. Эта отличительная черта карты сообщений реализована по аналогии с виртуальными функциями C++, но имеет дополнительные преимущества, не доступные для них. Каждый класс, который может получить сообщение, должен иметь свою карту сообщений, для того, чтобы иметь возможность соответствующим образом обрабатывать сообщения. При этом следует иметь в виду, что карта сообщений должна определяться вне какой-либо функции или объявления класса. Она также не может размещаться внутри C-блока.

Для определения карты сообщений необходимо использовать три макроса: BEGIN_MESSAGE_MAP, END_MESSAGE_MAP и DECLARE_MESSAGE_MAP.

---

[1] Мы будем использовать последнее из перечисленных названий.

[2] С появлением файла заголовков WINDOWSX.H, а, точнее, ряда макроопределений, стал возможным отказ от операторов switch в традиционных программах на языке C.

Макрос DECLARE_MESSAGE_MAP располагается в конце[1] объявления класса, использующего карту сообщений:

```
class CTheClass : public CBaseClass
{
// объявления членов класса
protected:
  //{{AFX_MSG_MAP(TheClass)
  afx_msg void OnPaint();
  //}}AFX_MSG_MAP,
DECLARE_MESSAGE_MAP()
};
```

Он используется для объявления трех специальных членов класса:

- частного массива компонент AFX_MSGMAP_ENTRY — *_messageEntries*,
- защищенной структуры AFX_MSGMAP — *messageMap*, которая указывает на массив *_messageEntries*,
- защищенной виртуальной функции — *GetMessageMap*, которая возвращает адрес *messageMap*.

Структура карты сообщений достаточно проста и представляет собой набор макросов, заключенных в специальные "операторные скобки":[2]

```
BEGIN_MESSAGE_MAP(CTheClass, CBaseClass)
  //{{AFX_MSG_MAP(CTheClass)
   ON_WM_CREATE()
   ON_WM_DESTROY()
   ON_COMMAND(ID_CHAR_BOLD, OnCharBold)
   ON_UPDATE_COMMAND_UI(ID_CHAR_BOLD, OnUpdateCharBold)
  //}}AFX_MSG_MAP
  ON_NOTIFY(FN_GETFORMAT, ID_VIEW_FORMATBAR, OnGetCharFormat)
  ON_NOTIFY(FN_SETFORMAT, ID_VIEW_FORMATBAR, OnSetCharFormat)
END_MESSAGE_MAP()
```

Как видите, начинается она с выполнения макроса:

```
BEGIN_MESSAGE_MAP(CTheClass, CBaseClass)
```

который имеет следующие параметры: *CTheClass* — задает имя класса, владельца карты сообщений; *CBaseClass* — определяет имя базового класса.

---

[1] Если вы захотите объявить какие-либо члены класса после этого макроса, то для них нужно определить новый тип доступа (*public*, *private* или *protected*).

[2] При использовании специальных средств автоматической генерации программного кода *AppWizard* и *ClassWizard*, достаточно подробно описанных в приложении, все три описанных макроса, а также специальные скобки для *ClassWizard* (//{{AFX_MSG_MAP(CTheClass) и //{{AFX_MSG_MAP) формируются автоматически при создании оконного класса.

Заканчивается карта сообщений вызовом макроса END_MESSAGE_MAP, имеющего еще более простой формат:

```
END_MESSAGE_MAP()
```

Между этими двумя вызовами располагаются специальные макросы, называемые *компонентами карты сообщений*. Вот они-то, собственно, и позволяют сопоставить сообщение с конкретным обработчиком.

Компоненты карты сообщений

Для каждого стандартного сообщения Windows определен свой макрос в форме:

```
ON_WM_XXX     // XXX — имя сообщения, например, ON_WM_PAINT.
```

Имена обработчиков определяются при распаковке параметров каждого сообщения Windows на основе простого соглашения; имя всегда начинается с префикса On, за которым следует имя соответствующего сообщения Windows (без префикса WM_), записанное строчными буквами. Например, для сообщения WM_PAINT в классе *CWnd* определен обработчик

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

а для сообщения WM_LBUTTONUP — обработчик

```
afx_msg void OnLButtonUp(
  UINT nFlags,
  CPoint point);
```

За формирование параметров, таких как *nFlags* и *point*, необходимых для каждого конкретного сообщения, отвечает соответствующий макрос. Описание всех обработчиков стандартных сообщений Windows можно найти в файле AFXWIN.H в объявлении класса *CWnd*, а также в приложении, приведенном в конце этой книги.

Командные сообщения Windows от меню, акселераторов и кнопок панелей инструментов обрабатываются макросом

```
ON_COMMAND(id, memberFn)
```

Этот макрос использует в качестве параметров идентификатор команды *id* и произвольное имя обработчика команды *memberFn*. Прототип обработчика должен быть описан в соответствующем классе и иметь вид:

```
afx_msg void memberFn();
```

Макрос ON_COMMAND_EX, определяющий расширенную форму обработчика, является надмножеством над функциональными возможностями макроса ON_COMMAND. Такие обработчики команд принимают единственный параметр типа UINT, содержащий идентификатор команды, и

возвращают значение типа BOOL: если возвращается FALSE, то команда передается для обработки другому объекту "получателю" команды; в противном случае обработка завершается.

Команды обновления выполняются посредством того же самого механизма, что и для "обычных" команд, только при этом используется макрос

```
ON_UPDATE_COMMAND_UI(id, memberFn)
```

Обработчики таких команд имеют вид:

```
afx_msg void memberFn(CCmdUI *pCmdUI);
```

В обоих этих случаях, обработчик команды — *memberFn*, будет вызван только в случае, если в оконную процедуру поступит сообщение, которое в параметре *wParam* содержит идентификатор команды, совпадающий с *id*.

Следующая группа макросов позволяет сопоставить команды, посылаемые элементами управления и другими дочерними окнами своим родителям, с соответствующим обработчиком.

Для обработки извещений от элементов управления применяется макрос

```
ON_CONTROL(wNotifyCode, id, memberFn)
```

Этот макрос использует в качестве параметров код извещения *wNotifyCode* от элемента управления, идентификатор элемента управления *id* и произвольное имя обработчика этой команды *memberFn*, прототип которого должен быть описан в соответствующем классе и иметь вид:

```
afx_msg void memberFn();
```

Так же, как и для обработчиков команд, для каждого извещения необходимо использовать один, и только один макрос ON_CONTROL. Обработчик *memberFn* вызывается только в том случае, если код извещения от элемента управления *wNotify*, такой как BN_CLICKED, совпадает с кодом извещения, определенным в компоненте карты сообщений, и значение параметра *id* совпадает с идентификатором элемента управления.

Помимо рассмотренного, общего для всех элементов управления и дочерних окон, макроса, для кнопок (BUTTON), элементов редактирования (EDIT), списков (LISTBOX) и комбинированных списков (COMBOBOX) используются специальные макросы, например

```
ON_BN_CLICKED(id, memberFn)
ON_EN_SETFOCUS(id, memberFn)
ON_LBN_DBLCLK(id, memberFn)
```

и многие другие. Полный список макросов этой группы можно найти в документации или в файле AFXMSG.H. Их принцип действия полностью совпадает с принципом действия макроса ON_CONTROL.

С появлением в Windows 95 и Windows NT более сложных общих элементов управления (common controls), имеющихся макросов оказалось недостаточ-

но, т. к. для них вместе с сообщением зачастую должны быть посланы некоторые данные. В связи с этим общие элементы управления Windows используют более мощное сообщение WM_NOTIFY, а в версиях Visual C++, начиная с 4.0, имеют прямую поддержку для этого сообщения на базе макроса

```
ON_NOTIFY(wNotifyCode, id, memberFn)
```

Параметры этого макроса полностью аналогичны параметрам уже рассмотренного макроса ON_CONTROL. Единственное, но существенное отличие заключается в параметрах обработчика *memberFn*:

```
afx_msg void memberFn(
  NMHDR *pNotifyStruct,
  LRESULT *result);
```

В качестве параметров обработчика используются: *pNotifyStruct* — указатель на структуру типа NMHDR, или большую, включающую ее в себя; *result* — указатель на переменную, в которую должно быть записано возвращаемое значение.

Подробно работа с сообщением WM_NOTIFY и поля структуры NMHDR описываются в главе 13, при обсуждении панелей элементов управления. Здесь мы приведем только список извещений, общих для всех дополнительных элементов управления: [1]

NM_CLICK	пользователь нажал левую кнопку мыши в элементе управления
NM_DBLCLK	пользователь дважды щелкнул левой кнопкой мыши в элементе управления
NM_RCLICK	пользователь нажал правую кнопку мыши в элементе управления
NM_DBLCLK	пользователь дважды щелкнул правой кнопкой мыши в элементе управления
NM_RETURN	пользователь нажал клавишу <Enter>, когда элемент управления имел фокус ввода
NM_SETFOCUS	элемент управления получил фокус ввода
NM_KILLFOCUS	элемент управления потерял фокус ввода
NM_OUTOFMEMORY	элемент управления не может завершить какую-либо операцию из-за нехватки памяти

Если есть необходимость использования одного обработчика для нескольких команд или извещений от элементов управления, то можно воспользоваться соответствующими макросами

[1] Извещения, определенные для каждого элемента управления, будут описаны при обсуждении соответствующего элемента управления.

```
ON_COMMAND_RANGE(idFirst, idLast, memberFn)
ON_CONTROL_RANGE(wNotifyCode, idFirst, idLast, memberFn)
ON_NOTIFY_RANGE(wNotifyCode, idFirst, idLast, memberFn)
```

Эти макросы используют в качестве параметров два идентификатора *idFirst* и *idLast*, которые составляют непрерывный ряд. Обработчик *memberFn* будет вызываться всякий раз, когда в оконную процедуру поступит сообщение, которое в параметре *wParam* содержит идентификатор команды, лежащий в границах, определяемых *idFirst* и *idLast*. Прототип обработчика должен иметь вид

```
afx_msg void memberFn(UINT nID);
```

для первых двух и

```
afx_msg void memberFn(
  UINT nID,
  NMHDR *pNotifyStruct,
  LRESULT *result);
```

для последнего макроса.

Аналогичный макрос существует и для команд обновления:

```
ON_UPDATE_COMMAND_UI_RANGE(idFirst, idLast, memberFn)
```

Обработчик, как и для макроса ON_UPDATE_COMMAND_UI, имеет единственный параметр — указатель на объект *CCmdUI*:

```
afx_msg void memberFn(CCmdUI *pCmdUI);
```

Для извещений, по аналогии с макросом ON_COMMAND_EX, определены и макросы:

```
ON_NOTIFY_EX(nCode, id, memberFn)
ON_NOTIFY_EX_RANGE(wNotifyCode, idFirst, idLast, memberFn)
```

Команды, определяемые пользователем, включаются в карту сообщений при помощи макроса

```
ON_MESSAGE(WM_NAMEMSG, OnNameMsg)
```

В качестве параметров этот макрос использует номер (WM_NAMEMSG) и имя (*OnNameMsg*) обработчика сообщения. Очевидно, что необходимо определить как номер сообщения, так и его функцию обработки:

```
#define WM_NAMEMSG (WM_USER + 100)
afx_msg LRESULT OnNameMsg(
 WPARAM wParam,
 LPARAM lParam
);
```

Ясно, что в этом случае всю ответственность по использованию параметров *wParam* и *lParam* берет на себя программист.

Сообщения, определяемые пользователем, включаются в карту при помощи макроса

```
ON_REGISTERED_MESSAGE(nMessageVariable, memberFn)
```

В качестве параметров этот макрос использует идентификатор зарегистрированного сообщения *nMessageVariable* и имя обработчика *memberFn*. Обработчик должен иметь следующий прототип:

```
afx_msg LRESULT memberFn(
  WPARAM wParam,
  LPARAM lParam);
```

Для получения уникального идентификатора сообщения необходимо вызвать функцию Windows *RegisterWindowMessage*, описанную ниже в этой главе.

Теперь, ознакомившись с общей структурой карты сообщений и ее компонентами, можно переходить к вопросу о том, в какой последовательности и кем осуществляется поиск обработчика сообщения или команды. Если внимательно посмотреть на иерархию классов библиотеки MFC, приведенную в приложении 1, то можно заметить, что все классы, которые могут получать сообщения, являются производными от класса *CCmdTarget*.

Класс CCmdTarget

Подобно тому как класс *CObject* является базовым для практически всех классов библиотеки MFC, класс *CCmdTarget* выступает в качестве базового для всех классов, которые могут работать с сообщениями.

Начнем с рассмотрения функций, входящих в состав класса *CCmdTarget*, и определенных в файле AFXWIN.H:[1]

void CCmdTarget::**BeginWaitCursor**() –
: изображает курсор в виде песочных часов. Обычно вызывается, чтобы показать пользователю, что выполняется длительная операция, такая как загрузка файла или сохранение документа в файле на диске.

void CCmdTarget::**EndWaitCursor**() –
: восстанавливает предыдущее изображение курсора, которое было изменено на "песочные часы" при вызове функции BeginWaitCursor.

void CCmdTarget::**RestoreWaitCursor**() –
: восстанавливает соответствующий курсор в виде песочных часов, после того как он был заменен на системный.

---

[1] Как обычно, мы не рассматриваем члены и функции-члены классов, которые используются для работы с использованием OLE, т. к. это выходит за рамки нашей темы. Этому и некоторым другим аспектам программирования для операционных систем Windows 95 и Windows NT мы планируем посвятить следующие наши книги.

```
virtual BOOL CCmdTarget::OnCmdMsg(
  UINT nID,
  int nCode,
  void* pExtra,
  AFX_CMDHANDLERINFO *pHandlerInfo) —
```

устанавливает маршрут и распределяет командные сообщения. Если рассмотреть поставляемые исходные тексты, то видно, что это единственная функция (не считая оконной процедуры по умолчанию), которая вызывается в оконной процедуре. Это означает, что именно она занимается поиском соответствующих обработчиков сообщений.

В качестве параметров функция *OnCmdMsq* принимает: идентификатор команды — *nID*; код извещения — *nCode*; значение параметра *pExtra* определяется значением параметра *nCode*; *pHandlerInfo* — обычно этот параметр равен NULL, в противном случае функция сама заполняет элементы *pTarget* и *pmf* структуры *pHandlerInfo*.

Если функция нашла своего адресата, который обработает сообщение, то она возвращает значение TRUE, в противном случае FALSE.

Таким образом, основным назначением функции *OnCmdMsg* является распределение команд по другим объектам. В принципе, можно переопределить эту функцию. Но при этом не следует забывать, что на вас перекладывается задача указания следующего получателя команды, чтобы не прервалась цепочка передачи сообщений.

К практическим аспектам обработки сообщений Windows и извещений элементов управления мы приступим, когда ознакомимся с имеющимися в библиотеке MFC классами, отвечающими за работу с окнами, а пока займемся рассмотрением достаточно важного вопроса — каким образом осуществляется поиск обработчика команды.

Стандартный маршрут команды

Когда *Run* — функция класса *CWinThread* — получает сообщение, она направляет его в соответствующее окно-получатель сообщения, где оно будет обрабатываться. Несколько иначе все выглядит для команд. Поскольку они формируются в результате взаимодействия пользователя с приложением, то, обычно, команда начинает свой путь к получателю от главного окна приложения. Напомним, что каждый объект, способный получать сообщения, располагает своей картой сообщений. Что же происходит, когда какой-либо объект получает сообщение? А все очень просто. Имея свою карту сообщений получатель ищет в ней поступившую команду (или сообщение) и в случае успеха запускает соответствующий обработчик. Сразу же возникает вопрос: а если не найдет? Ну что ж, в этом случае поиск переносится в карту сообщений базовых классов, а если и там нет ассоциированного обработчика, то главное окно передает эту команду следующему кандидату на обработку. Для его определения в библиотеке MFC определен стандарт-

ный маршрут, по которому передается команда. Этот процесс продолжается до тех пор, пока команда не будет выполнена или отправлена оконной процедуре, действующей по умолчанию.

Порядок, в котором адресаты передают команды, "зашит" в библиотеке классов MFC и представляет собой следующую последовательность шагов:

1. Команду получает фрейм MDI (объект класса *CMDIFrameWnd* или производного от него). Порядок получателей команды:
   - активное окно класса *CMDIChildWnd*,
   - сам объект класса *CMDIFrameWnd*,
   - приложение-объект класса *CWinApp* или производного от него.
2. Команду получает фрейм документа (объект класса *CFrameWnd*, *CMDIChildWnd* или производного от одного из них). Порядок получателей команды:
   - активное представление (view),
   - текущий фрейм документа,
   - приложение-объект класса *CWinApp* или производного от него.
3. Команду получает представление. Порядок получателей команды:
   - текущее представление,
   - документ, присоединенный к представлению.
4. Команду получает документ. Порядок получателей команды:
   - текущий документ,
   - шаблон документа, присоединенный к нему.
5. Команду получает блок диалога. Порядок получателей команды:
   - текущий блок диалога,
   - окно, которое владеет этим блоком диалога,
   - приложение-объект класса *CWinApp* или производного от него.

Очевидно, что представленные здесь переходы взаимозависимы. Например, если команду получает документ, присоединенный к текущему представлению, то следующим объектом, которому будет передана команда, является текущий документ, а затем его шаблон.

Чтобы сказанное стало более понятным, рассмотрим небольшой пример. Пользователь работает с MDI-приложением и хочет выделить весь текущий документ (для реализации этого действия в объекте класса документа приложения предусмотрен специальный обработчик). Он выбирает пункт меню Edit\Select All (Правка\Выделить все), с которым ассоциирована команда ID_EDIT_SELECT_ALL. Первым эту команду получает главное окно приложения, которое дает возможность обработать его текущему активному дочернему окну MDI. До того как дочернее окно просмотрит свою карту сообщений, оно дает возможность обработать команду своему

представлению. Представление проверяет свою карту сообщений и, поскольку не находит там обработчика, направляет команду в ассоциированный с ним документ. Теперь уже документ ищет обработчик в своей карте сообщений, находит его, и начинает обрабатывать команду. Если бы мы не реализовали обработчик в нашем объекте-документе, то команда была бы направлена в шаблон, и в случае неудачи поиск продолжается в объекте класса *CMDIFrameWnd*, а затем в объекте класса приложения. Наглядно описанная последовательность действий представлена на рис. 17. Для команды, совершающей этот или любой другой маршрут, вызывается метод *OnCmdMsg* следующего объекта.

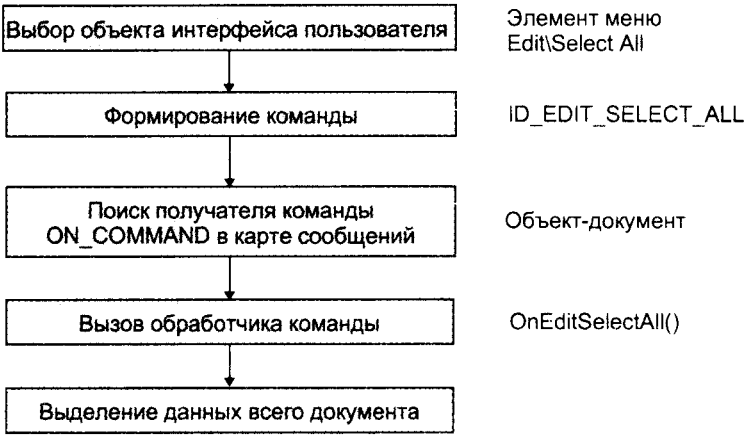


Рис. 17. Пример обработки действия пользователя

Стандартные сообщения Windows, в отличие от команд, не используют какой-либо маршрут для поиска обработчика, т. к. они посылаются непосредственно в оконную процедуру того объекта, который должен соответствующим образом отреагировать на них. Но и в этом случае каждый оконный класс, образованный непосредственно или косвенно от класса *CWnd*, имеет собственную карту сообщений, которая устанавливает связи между сообщениями и соответствующими обработчиками. Если соответствующий обработчик не найден в карте сообщений некоторого класса, то он ищется в карте сообщений базового для него класса. При этом более эффективным является использование механизма виртуальных функций, которые определены в классе *CWnd* для стандартных сообщений Windows.

Команды обновления и класс CCmdUI

Когда пользователь, например, раскрывает меню, каждый его элемент "должен знать" каким он будет отображаться — доступным, отмеченным или нет. Получатель команды предоставляет ему эту информацию, реализуя

обработчик сообщения ON_UPDATE_COMMAND_UI, внутри которого используется специальный класс *CCmdUI*, определяющий или элемент меню, или кнопку панели инструментов, или любой другой объект интерфейса пользователя, способный формировать команды. Этот обработчик вызывается до вывода элементов интерфейса пользователя на экран (или их перерисовки), что позволяет задать их внешний вид. Маршрут, по которому ищется обработчик, совпадает с тем, который действует для "обычных" командных сообщений (рис. 18)

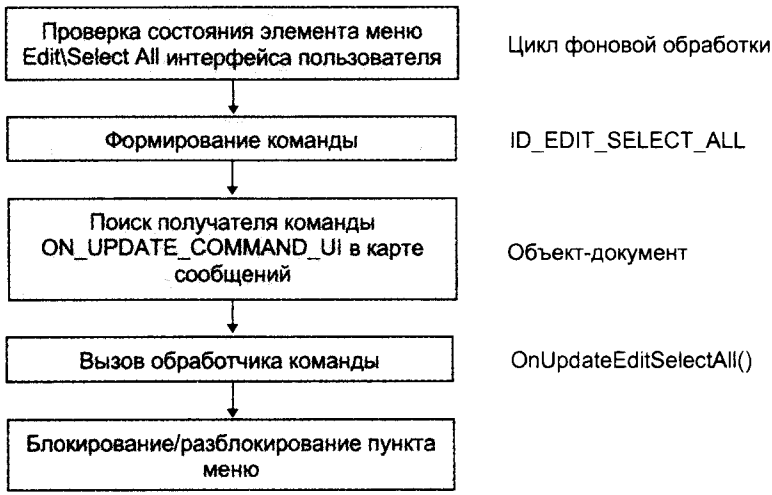


Рис. 18. Пример обновления элемента меню

Рассмотрим те возможности, которые заложены в классе *CCmdUI* для обновления элементов интерфейса пользователя. Начнем с членов класса:

UINT CCmdUI::**m_nID** — идентификатор объекта интерфейса пользователя.

UINT CCmdUI::**m_nIndex** — индекс объекта интерфейса пользователя.

CMenu CCmdUI::**m_pMenu** —
указатель на меню, представляемое объектом CCmdUI. Если представляется не меню, то этот параметр равен NULL.

CMenu CCmdUI::**m_pSubMenu** —
указатель на имеющееся подменю, представляемое объектом CCmdUI. Если представляется не меню, то этот параметр равен NULL. Если подменю является раскрывающимся, то параметр m_nID содержит идентификатор его первого элемента.

CWnd* CCmdUI::**m_pOther** —
указатель на объект окна, который послал извещение, например, панель элементов управления или строка состояния. Если извещение пришло от меню или не от оконного объекта, то этот параметр равен NULL.

Кроме представленных членов у класса имеются 5 функций:

virtual void CCmdUI::**Enable**(BOOL bOn = TRUE) —
блокирует (bOn = FALSE) или разблокирует (bOn = TRUE) элемент интерфейса пользователя, ассоциированный с этой командой.

virtual void CCmdUI::**SetCheck**(int nCheck = 1) —
устанавливает элемент интерфейса пользователя в одно из возможных состояний в зависимости от значения передаваемого параметра: 0 — непомеченное, 1 — помеченное, 2 — неопределенное. Последнее состояние применимо только к кнопкам панелей инструментов.

virtual void CCmdUI::**SetRadio**(BOOL bOn = TRUE) —
блокирует (bOn = FALSE) или разблокирует (bOn = TRUE) группу переключателей.

virtual void CCmdUI::**SetText**(LPCTSTR lpszText) —
устанавливает текст для данного элемента интерфейса пользователя.

void CCmdUI::**ContinueRouting**() —
позволяет направить команду на обработку дальше по маршруту. Эта функция действует совместно с обработчиком команды ON_COMMAND_EX, который возвращает FALSE.

Практические аспекты использования перечисленных здесь возможностей будут рассмотрены при обсуждении вопросов организации интерфейса пользователя.

В заключение общего обзора принципов работы с сообщениями, заложенными в библиотеке классов MFC, и несколько забегая вперед, рассмотрим основные функции, которые предназначены для работы с сообщениями.

Функции для работы с сообщениями

Понимание работы с сообщениями крайне важно для написания правильных и эффективных программ для Windows. Поэтому рассмотрим те основные функции, которые вы можете (и должны) широко использовать:

static const MSG* CWnd::**GetCurrentMessage**() —
возвращает указатель на структуру *MSG*, которая содержит информацию о текущем сообщении. Может быть вызвана только из обработчика, заданного макросом ON_MESSAGE.

LRESULT CWnd::**Default**() —
служит для вызова оконной процедуры, выполняемой по умолчанию.

virtual BOOL CWnd::**PreTranslateMessage**(MSG *pMsg) —
используется классом *CWinApp* для фильтрации оконных сообщений до того, как они будут распределены функциями Windows TranslateMessage и DispatchMessage.

LRESULT CWnd::**SendMessage**(
UINT msg,
WPARAM wPrm = 0,
LPARAM lPrm = 0) –
посылает сообщение в объект CWnd, непосредственно вызывая оконную процедуру, и не выходит из нее, пока та не обработает сообщение.

BOOL CWnd::**PostMessage**(
UINT msg,
WPARAM wPrm = 0,
LPARAM lPrm = 0) –
помещает сообщение в очередь приложения и заканчивает работу без ожидания его обработки. Из очереди сообщения извлекаются функциями Windows GetMessage или PeekMessage.

BOOL CWnd::**SendNotifyMessage**(
UINT msg,
WPARAM wPrm = 0,
LPARAM lPrm = 0) –
посылает в окно определенное извещение. Если окно было создано текущим потоком, то функция для обработки извещения вызывает оконную процедуру и не завершается, пока оно не будет обработано. Если же окно было создано другим потоком, то послав извещение в оконную процедуру, функция завершает свою работу.

UINT ::**RegisterWindowMessage**(LPCTSTR lpString) –
позволяет получить идентификатор (в диапазоне от 0xC000 до 0xFFFF) нового уникального сообщения, который может затем использоваться при вызовах функций SendMessage или PostMessage. В качестве параметра lpString она использует указатель на строку, определяющую имя сообщения. Эта функция должна использоваться для предотвращения возможных конфликтов в случае, когда нужно связать между собой два (или больше) приложения. Зарегистрированное сообщение действует до окончания сеанса работы Windows. Обработчик сообщения ассоциируется с ним посредством специального макроса ON_REGISTERED_MESSAGE, например, как показано ниже:

```
class CUserWnd : public CParentWnd
{
...
  //{{AFX_MSG_MAP(CUserWnd)
    afx_msg LRESULT OnSample(WPARAM wParam, LPARAM lParam);
  //}}AFX_MSG_MAP
DECLARE_MESSAGE_MAP()
};

UINT WM_SAMPLE = RegisterWindowMessage("MESSAGE_SAMPLE");

BEGIN_MESSAGE_MAP(CUserWnd, CParentWnd)
  //{{AFX_MSG_MAP(CUserWnd)
    ON_REGISTERED_MESSAGE(WM_SAMPLE, OnSample)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Наиболее интересными из перечисленных являются две функции: *SendMessage* и *PostMessage* (рис. 19):

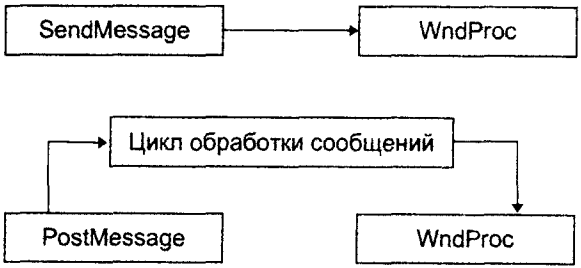


Рис. 19. Посылка сообщений функциями SendMessage и PostMessage

Основная цель этих функций — посылка сообщения в окно своего или некоторого другого приложения. Но на этом их совпадения заканчиваются. Дело в том, что функция *SendMessage* явно вызывает процедуру того окна, которое ассоциировано с объектом, вызывающим ее, и не выходит из процедуры, пока та не обработает сообщение. Другими словами, выполнение программы не будет продолжено до тех пор, пока сообщение не будет обработано.

В отличие от нее, функция *PostMessage* не вызывает явно оконную процедуру, а посылает сообщение в очередь сообщений приложения, откуда оно выбирается с помощью функции *GetMessage* или *PeekMessage*. Выполнение самой функции *PostMessage* при этом завершается и работа приложения может быть продолжена.

Но от теории, пусть даже и очень важной, пора переходить к практике. Неоднократное упоминание классов оконных объектов (и даже некоторое знакомство с ними) — одной из важнейших составляющих библиотеки классов MFC, диктует необходимость их скорейшего рассмотрения.

Часть V

Создание однодокументных и многодокументных приложений на базе библиотеки MFC

11. Классы окон библиотеки MFC

12. Создание многодокументных приложений

13. Панели элементов управления

Глава 11

Классы окон библиотеки MFC

Вспомним, что Windows — это операционная система, которая, прежде всего, работает с окнами. Пришло время подробно познакомиться с ними.

Объекты оконных классов библиотеки MFC, естественно, отличаются от окон Windows, но тем не менее тесно с ними связаны. Полное и правильное понимание взаимосвязей между ними является залогом успеха при создании приложений для Windows.

Как и любой другой объект C++, объект "окно" создается и уничтожается, соответственно, конструктором и деструктором класса. Это с одной стороны. А с другой, окно Windows — это внутренняя структура данных операционной системы, которая формируется функцией интерфейса прикладного программирования Win32 API *Create*, а разрушается функцией *DestroyWindow*. В системе Windows каждое окно идентифицируется своим дескриптором, значение которого хранится в члене *m_hWnd* объекта *CWnd* (рис. 20).

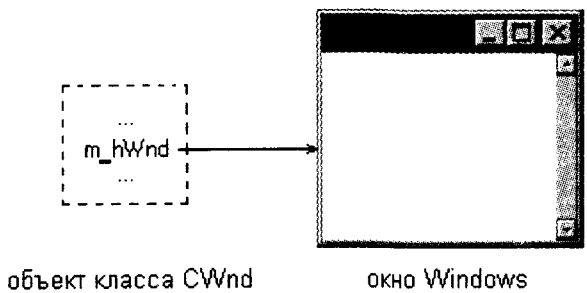


Рис. 20. Объект класса CWnd связан с окном Windows через дескриптор m_hWnd

Непосредственно переходить к созданию и работе с окнами нельзя, на наш взгляд, без подробного рассмотрения типов окон, имеющихся в Windows.

Окна, определенные в системе Windows

Существует всего три основных типа окон:

- перекрывающиеся (overlapped),
- всплывающие или вспомогательные (popup),
- дочерние (child),

из которых программист может создавать множество самых разнообразных объектов, комбинируя предопределенные биты стиля, имена которых имеют префикс WS_ (от window style — стиль окна). Рассмотрим более подробно эти биты, их влияние на внешний вид и некоторые свойства окон:

WS_BORDER	окно имеет рамку без заголовка
WS_CAPTION	окно имеет заголовок и рамку; как правило, этот стиль используется для перекрывающихся окон и не может применяться совместно со стилем WS_DLGFRAME
WS_CHILD	Дочернее окно; не может использоваться совместно со стилем WS_POPUP
WS_CLIPCHILDREN	исключает область, занятую дочерним окном, из области рисования родительского окна; используется только для родительских окон
WS_CLIPSIBLINGS	исключает все другие дочерние окна из своей области рисования; другими словами, если дочерние окна перекрываются, а этот стиль не указан, то при изменении рабочей области одного из окон могут быть испорчены рабочие области других дочерних окон; этот стиль используется только вместе со стилем WS_CHILD
WS_DISABLED	неактивное окно, т. е. сразу после его создания оно недоступно для ввода с клавиатуры или с помощью мыши
WS_DLGFRAME	окно имеет двойную рамку и не имеет заголовка
WS_GROUP	определяет первый элемент управления группы окон, к которым пользователь может переходить при помощи клавиш со стрелками; все элементы управления, определенные с этим стилем, заканчивают текущую и начинают новую группу (одна группа заканчивается там, где начинается другая)
WS_HSCROLL	окно имеет горизонтальную полосу прокрутки[1]

[1] Полосы прокрутки, определяемые как свойства окон, отличаются от полос прокрутки — элементов управления: так, если установить нулевой диапазон полос прокрутки окна, то Windows автоматически уберет их! Более подробно полосы прокрутки рассматриваются в главах, посвященных элементам управления.

WS_VSCROLL	окно имеет вертикальную полосу прокрутки
WS_MAXIMIZE	окно будет отображено в развернутом виде
WS_MAXIMIZEBOX	окно имеет кнопку "Развернуть"; если окно является дочерним окном элемента управления, то этот флаг стиля используется под другим именем — WS_TABSTOP
WS_MINIMIZE	окно будет отображено в виде пиктограммы; используется только со стилем WS_OVERLAPPED
WS_MINIMIZEBOX	окно имеет кнопку "Свернуть"; если окно является дочерним окном элемента управления, то этот флаг стиля используется под другим именем — WS_GROUP
WS_OVERLAPPED	окно является перекрывающимся; обычно имеет заголовок и рамку
WS_POPUP	всплывающее окно; не может использоваться совместно со стилем WS_CHILD.
WS_SYSMENU	окно имеет пиктограмму системного меню в полосе заголовка
WS_TABSTOP	назначается одному или нескольким элементам управления для того, чтобы между ними можно было перемещаться с помощью клавиши табуляции <Tab>
WS_THICKFRAME	окно имеет утолщенную рамку, при помощи которой можно изменять размер окна
WS_VISIBLE	окно становится видимым сразу после создания; этот стиль в основном используется для окон диалога

Помимо перечисленных стилей окон Windows, в файле WINUSER.H определены стили, являющиеся комбинацией наиболее часто используемых стилей.

WS_OVERLAPPEDWINDOW	комбинация стилей WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX
WS_POPUPWINDOW	комбинация стилей WS_POPUP, WS_BORDER и WS_SYSMENU

При работе с окнами часто также используются дополнительные или расширенные биты стиля:

WS_EX_ABSPOSITION	определяет, что позиция окна задается в абсолютных единицах относительно левого верхнего угла экрана
WS_EX_ACCEPTFILES	определяет, что окно допускает перетаскивание файлов (drag-and-drop)

WS_EX_CLIENTEDGE	придает "трехмерный эффект" границе окна, клиентская область при этом как бы вдавлена в окно
WS_EX_CONTEXTHELP	включает знак вопроса в заголовок окна; когда пользователь нажимает на этот знак, то курсор изменяется на указатель в виде знака вопроса; если теперь выбрать какое-либо дочернее окно — оно получит сообщение WM_HELP
WS_EX_CONTROLPARENT	позволяет пользователю перемещаться по дочерним окнам родительского окна, используя для этого клавишу табуляции <Tab>
WS_EX_DLGMODALFRAME	окно имеет удвоенную границу, что при использовании совместно со стилем WS_CAPTION позволяет создавать окно с полосой заголовка
WS_EX_LEFT	устанавливает для окна свойство левого выравнивания (по умолчанию)
WS_EX_LEFTSCROLLBAR	если имеется полоса прокрутки, то она располагается с левой стороны рабочей (клиентской) области окна
WS_EX_LTRREADING	используется по умолчанию и определяет естественный порядок (слева направо) для большинства языков ввода и чтения текста
WS_EX_MDICHILD	дочернее окно MDI
WS_EX_NOPARENTNOTIFY	определяет, что дочернее окно, созданное с этим стилем, не посылает сообщения WM_PARENTNOTIFY своему родительскому окну при создании или уничтожении
WS_EX_RIGHT	в зависимости от класса окна устанавливает для него групповое свойство правого выравнивания
WS_EX_RIGHTSCROLLBAR	если имеется полоса прокрутки, то она располагается с правой стороны рабочей (клиентской) области окна; используется по умолчанию
WS_EX_RTLREADING	определяет порядок ввода и чтения текста справа налево
WS_EX_SMCAPTION	окно имеет уменьшенную высоту полосы заголовка; используется при создании "плавающих" панелей инструментов
WS_EX_STATICEDGE	окно имеет трехмерный стиль границы, заданный для использования в элементах, к которым пользователь не получает доступа (статические элементы управления)
WS_EX_TOOLWINDOW	окно, которое используется для создания "плавающей" панели инструментов; такое окно имеет укороченную полосу заголовка и не отображается на панели задач

WS_EX_TOPMOST	окно будет располагаться поверх всех окон (созданных без этого стиля); для установки или сброса этого бита можно использовать функцию SetWindowPos
WS_EX_TRANSPARENT	окно является прозрачным, т. е. не закрывает другие окна, расположенные "под ним"; сообщение WM_PAINT такое окно получает до того, как это сообщение получают все окна того же уровня, но расположенные "под ним"
WS_EX_WINDOWEDGE	окно имеет приподнятую границу для создания "трехмерного эффекта"

Так же как в случае основных стилей, для расширенных определены два комбинированных дополнительных стиля (WINUSER.H):

WS_EX_OVERLAPPEDWINDOW	комбинация стилей WS_EX_CLIENTEDGE и WS_EX_WINDOWEDGE
WS_EX_PALETTEWINDOW	комбинация стилей WS_EX_WINDOWEDGE, WS_EX_SMCAPTION и WS_EX_TOPMOST

После того как мы перечислили все возможные стили окон, осталось сказать несколько слов об упомянутых выше основных типах окон Windows. Напомним, что их всего три.

- **Перекрывающиеся окна.** Это основной, наиболее универсальный тип окон Windows. Для их создания чаще всего используется стиль WS_OVERLAPPEDWINDOW. Главное окно приложения, как правило, имеет именно этот тип.
- **Вспомогательные или всплывающие окна.** Чаще всего этот тип окон создается с использованием стиля WS_POPUP.[1] Обычно они отображают какую-либо информацию короткого промежутка времени. Наиболее часто окна этого типа используются для отображения диалоговых окон или окон сообщения. Основное их отличие от других окон заключается в том, что, даже если они имеют родительское окно, то все равно всегда отображаются поверх всех окон на экране, выскакивая, как поплавки, наверх даже тогда, когда пользователь делает активным другое окно. Для окон этого типа, как правило, организуется своя оконная процедура. Они могут иметь, а могут и не иметь родителя. Если такое окно не имеет родительского, то оно является совершенно независимым от создавшего его окна и по своим свойствам практически неотличимо от перекрывающихся окон. Поведение всплывающего окна, имеющего родителя, зависит от того, что происходит с последним. Когда главное окно минимизируется, всплывающее окно "без родителя" скрывается, а "с родителем" остается на экране сверху. Подчеркнем еще один важный момент: когда мы гово-

[1] Наиболее полным семантическим эквивалентом английского термина "popup window" является термин "окно-поплавок". Мы будем называть такие окна *всплывающими*.

рим, что вспомогательное окно *имеет родителя*, это совсем не означает, что оно является *дочерним*.

- **Дочерние окна.** Окна этого типа создаются тогда, когда у приложения уже есть главное окно. Такие окна связаны некоторыми характеристиками (как бы подчинены) с тем окном, из которого они были созданы, отсюда и название. Назначение этих окон может быть самым разнообразным, начиная от простого деления родительского окна на области, до организации многодокументного интерфейса. Все элементы управления также являются дочерними окнами. Из основных свойств этого типа окон отметим следующие. Дочерние окна никогда не отображаются вне своего родительского окна ни в раскрытом виде, ни в виде пиктограммы — они как бы целиком принадлежат родителю. Располагаются они в родительском окне относительно верхнего левого угла его рабочей (клиентской) области. Более того, при перемещении родительского окна по экрану его дочерние окна перемещаются вместе с ним. И, наконец, дочернее окно никогда не может стать активным.[1]

Какие же возможности предоставляет нам библиотека MFC в плане создания и работы с окнами? Если обратимся к иерархии классов MFC (рис. 21), то вспомним, что все классы, имеющие какое-либо отношение к окнам, являются производными (прямо или косвенно) от единственного класса *CWnd*.

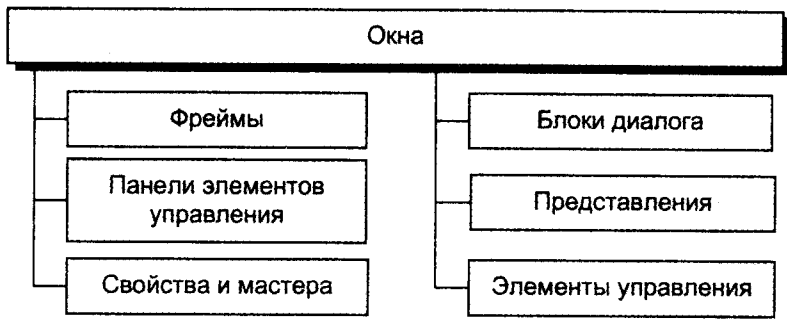


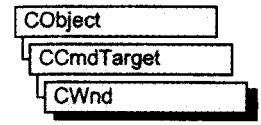
Рис. 21. Иерархия оконных классов

Этот класс, а также специальные производные от него классы, включая *CFrameWnd*, *CMDIFrameWnd*, *CMDIChildWnd*, *CView* и *CDialog*, имеющиеся в библиотеке, спроектированы таким образом, чтобы сделать за нас большую часть обязательной, но рутинной работы с различными типами окон Windows. Но начать надо с рассмотрения возможностей базового класса.

---

[1] Если вы создаете окно-элемент управления, то вместе со стилем WS_CHILD можно указать дополнительные флаги, характерные для того или иного элемента управления. С конкретными значениями вы ознакомитесь при обсуждении соответствующих элементов управления.

Класс CWnd



Класс *CWnd* — это базовый класс для всех окон, созданных на базе библиотеки MFC. Помимо предоставления всем своим производным классам большого числа функций для работы с окнами, этот класс служит для создания разнообразных дочерних окон. Начнем со знакомства с теми возможностями, которые заложены в него разработчиками.

Из имеющихся членов этого класса для нас, на данном этапе знакомства, интерес представляет один:

HWND CWnd::***m_hWnd*** –

дескриптор окна Windows, присоединенного к объекту класса *CWnd*. Это достаточно важный параметр, особенно для главного окна приложения, т. к. он позволяет библиотеке взять на себя всю работу для корректного завершения приложения, когда пользователь закрывает его главное окно.

Класс *CWnd* содержит практически все функции, необходимые для работы с окнами. Таких функций более двухсот[1] и их можно разбить на несколько категорий, каждая из которых включает функции, направленные на решение определенного круга задач. Приведем список категорий с краткой характеристикой входящих в них функций:

- **Конструктор.** Создает объект класса без создания окна Windows.
- **Инициализация.** В эту категорию входят функции: отвечающие за создание окна Windows (соответствующей ему внутренней структуры) и присоединение его к объекту класса, — *Create* и *CreateEx*; виртуальная функция *PreCreateWindow*, позволяющая зарегистрировать используемый по умолчанию класс окна "AfxWnd"; функции, позволяющие узнать текущий обычный или расширенный стиль окна — *GetStyle* и *GetExStyle*; функции, позволяющие присоединить или отсоединить окно Windows от объекта класса *CWnd* — *Attach* и *Detach*; функции для получения дескриптора окна Windows (*GetSafeWnd*), или указателя на объект по дескриптору (*FromHandle* и *FromHandlePermanent*); функции для работы с подклассами окна — *PreSubclassWindow* и *SubclassWindow*; а также некоторые другие функции.
- **Состояние окна.** Сюда входят функции, позволяющие: определить (*IsWindowEnable*) или изменить (*EnableWindow*) доступ к окну; узнать или назначить активное окно (обычное — *GetActiveWnd* и *SetActiveWnd*, окно переднего плана — *GetForegroundWindow* и *SetForegroundWindow*); получить или установить фокус ввода с клавиатуры (*GetFocus* и *SetFocus*) или от мыши (*GetCapture* и *SetCapture*), а также дескриптор пиктограммы, ассо-

[1] Полный список функций класса *CWnd* приведен в приложении.

циированной с окном (*GetIcon* и *SetIcon*); модифицировать расширенный или обычный стили окна (*ModifyStyle* и *ModifyStyleEx*); получить указатель на объект "рабочего стола" Windows.

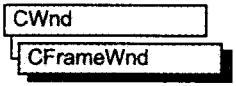
- **Изменение размеров и позиции окна.** Включает две группы функций. Функции, служащие для определения или установки состояния окна: видимое/невидимое (*GetWindowPlacement* и *SetWindowPlacement*); свернуто (*IsIconic*) или развернуто (*IsZoomed*). И, собственно, функции, позволяющие узнать или изменить размеры и/или позицию окна: *MoveWindow* — перемещение окна с изменением размеров; *SetWindowPos* — перемещение окна с изменением размеров и расположения относительно других окон; *ArrangeIconicWindows* — упорядочить все свернутые дочерние окна; *BringWindowToTop* — делает окно старшим в Z-порядке; *GetWindowRect* — определяет размеры всего окна; *GetClientRect* — определяет размеры рабочей области окна.
- **Доступ к окну.** Из функций этой категории мы пока хотели бы отметить следующие: *FindWindow* — позволяет определить существует ли в системе окно, заданное именем класса и именем окна (оба параметра являются необязательными), т. е. запущено ли приложение, имеющее искомое окно; *CenterWindow* — центрирует положение окна относительно его родителя. Кроме представленных, категория содержит еще большую группу функций, которые используются значительно реже и поэтому будут рассмотрены не все.
- **Обновление-перерисовка.** Из функций этой категории мы представим только наиболее часто используемые: *UpdateWindow* — обновляет рабочую область, посылая сообщение WM_PAINT; *Invalidate* — задает область модификации для последующей перерисовки при обработке ближайшего сообщения WM_PAINT; *ShowWindow* — отображает окно на экране, режим отображения задается передаваемым параметром. Кроме вышеперечисленных, сюда же можно отнести группу функций для работы с контекстами устройств: *BeginPaint*, *EndPaint*, *Print*, *PrintClient*, *GetDC*, *ReleaseDC* и *GetWindowDC*, которые подробно будут описаны при обсуждении вывода в окна графической информации.
- **Преобразование координат.** Служат для взаимного преобразования рабочих и экранных координат текущего окна.
- **Работа с текстом.** В эту категорию входят функции, которые позволяют установить или получить текст, ассоциированный с окном (*SetWindowText* и *GetWindowText*) и его длину (*GetWindowTextLenght*), а также используемый шрифт (*SetFont* и *GetFont*).
- **Работа с сообщениями.** Функции этой категории будут рассмотрены более подробно, т. к. они важны для понимания принципов работы всей системы Windows и ваших приложений в частности.

Оставшиеся категории функций имеют достаточно специфическую направленность и будут обсуждаться в соответствующих разделах книги. К ним

относятся следующие категории: функции для работы с полосами прокрутки, каретками, элементами блока диалога, меню, таймерами, всплывающими подсказками, а также функции для работы с буфером обмена и обработчики сообщений.

Класс *CWnd* служит для выполнения двух основных целей: предоставление интерфейса всем другим оконным классам библиотеки MFC и создание дочерних окон Windows. В связи с этим мы ограничились здесь простым перечислением основных реализованных в классе функций. Многие из них мы рассмотрим при обсуждении конкретных примеров программ, а полный список функций этого класса приведен в приложении 2.

Класс CFrameWnd



В отличии от своего базового класса *CFrameWnd* служит для создания перекрывающихся или всплывающих окон и поддерживает однодокументный интерфейс Windows (SDI). С этим классом (и производными от него) тесно связано понятие *фрейма* (frame window). Под ним понимается окно, которое координирует взаимодействие приложения с документом и его представлением.[1] Оно отражается на экране в виде рамки, ограничивающей представление, с необязательной строкой состояния и стандартными элементами управления окном. Это окно отвечает за управление размещением своих дочерних окон и других элементов рабочей области. Кроме того, фрейм переадресует команды своим представлениям и может отвечать на извещения от элементов управления окна.

Этот класс спроектирован таким образом, чтобы взять на себя задачу выполнения всех основных функций приложения Windows на базе интерфейса SDI:

- передает команды и многие общие извещения в активное текущее представление,
- управляет размещением панелей элементов управления, представлений и других дочерних окон внутри своей рабочей области. Во время цикла ожидания обновляет панель инструментов и кнопки панелей элементов управления. Кроме того, имеется обработка команд по умолчанию для показа или скрытия панели инструментов и строки состояния,
- управляет полосой основного меню. Когда изображается раскрывающееся меню, фрейм использует механизм UPDATE_COMMAND_UI для установки элементов меню в состояние "доступен", "заблокирован" или "помечен", а также для добавления и удаления элементов. Когда пользо-

[1] Ввиду широкого использования архитектуры документ/представление (document/view) мы посвятим ей отдельную часть второго тома.

ватель выбирает элемент меню, фрейм обновляет строку состояния, выводя туда соответствующее заданное сообщение,

- имеет возможность использовать специальную *таблицу назначений клавиш* (accelerator table), которая автоматически транслирует комбинации клавиш в команды,
- имеет необязательный идентификатор справки, который используется для контекстно-зависимой справочной системы. Управляет самим механизмом контекстно-зависимой справочной системы (вызов комбинацией клавиш <Shift>+<F1>) и режимом предварительного просмотра перед печатью,
- поддерживает выполнение операций drag-and-drop для работы файлами. На перенос зарегистрированных и ассоциированных с приложением типов файлов он реагирует генерацией требования открыть канал динамического обмена данными (DDE),
- если фрейм является главным окном приложения (определяется значением *CWinThread::m_pMainWnd*), то когда пользователь закрывает приложение, он предлагает ему сохранить модифицированные документы.

В классе определены два основных члена:

BOOL CFrameWnd::**m_bAutoMenuEnable**,
: значение, которого равное FALSE приведет к автоматической блокировке тех элементов меню (и соответствующих кнопок панели инструментов), для которых не определены обработчики ON_UPDATE_COMMAND_UI или ON_COMMAND. По умолчанию устанавливается в TRUE.

static AFX_DATA const CRect CFrameWnd::**rectDefault** –
: Определяет константный объект класса CRect,который предписывает Windows самостоятельно определить размеры и положение создаваемого окна и определен следующим образом:

```
const AFX_DATADEF CRect CFrameWnd::rectDefault(
  CW_USEDEFAULT,
  CW_USEDEFAULT,
  0,
  0
);
```

Функции-члены этого класса сгруппированы по следующим категориям:

- **Конструктор.** Как обычно, конструктор создает объект класса, но не создает окно.
- **Инициализация.** Сюда входят: две функции, отвечающие за создание окна Windows и присоединение его к текущему объекту класса — *Create* и *LoadFrame*; функция, загружающая таблицу акселераторов — *LoadAccelTable*; группа функций для управления инициализацией панелей элементов управления (*LoadBarState* — загружает сохраненные в INI-файле или реестре установки панели, *SaveBarState* — сохраняет текущие установки

панели в INI-файле или в реестре, *SetDockState* — сохраняет размещение текущего окна относительно главного, *GetDockState* — восстанавливает размещение текущего окна относительно главного (для плавающих панелей) и *ShowControlBar* — управляет режимом отображения панели элементов управления).

- **Операции.** В эту категорию входит большое число функций для работы с документами и их представлениями, с панелями элементов управления, строкой состояния и для управления модальностью фрейма окна.

Помимо своего основного назначения — поддержки SDI-приложений — этот класс служит базовым для классов *CMDIFrameWnd* и *CMDIChildWnd*, которые созданы специально для организации приложений на базе MDI.

Еще раз подчеркнем важный момент. На базе этого класса легко и удобно создавать фреймы окон. Для того чтобы существенно расширить эти возможности, в библиотеке реализованы специальные классы, базирующиеся на понятии "представление".

Создание окна и библиотека MFC

Пора подробнее рассмотреть процесс создания окон. Рассмотрим приложение-пример Styles, показывающее 10 дочерних окон с различными комбинациями оконных стилей в каждом. При щелчке в рабочей области окна левой кнопкой мыши, на экран выводится окно информационного сообщения, где перечислены стили, с которыми создано окно.

Мы приводим полностью все файлы, содержащие код этого приложения. Основа (или каркас) приложения создавалась с использованием средств AppWizard и ClassWizard, которые кроме самого кода добавили и комментарии. Комментарии приведены без изменений и, что естественно, на английском языке. После исходного текста приведен его подробный анализ. На рис. 22 представлен внешний вид всех окон приложения.

```
////////////////////////////////////////////////////////////
// Файл Styles.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

// main header file for the STYLES application

#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols
```

```
/////////////////////////////////////////////////////////////////
// CStylesApp:

class CStylesApp : public CWinApp
{
public:
  CStylesApp();

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CStylesApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL

  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////
// Файл Styles.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// Defines the class behaviors for the
// application.

#include "stdafx.h"
#include "Styles.h"
#include "MainFrm.h"

/////////////////////////////////////////////////////////////////
// CStylesApp

BEGIN_MESSAGE_MAP(CStylesApp, CWinApp)
  //{{AFX_MSG_MAP(CStylesApp)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////
// CStylesApp construction
CStylesApp::CStylesApp()
{
}

/////////////////////////////////////////////////////////////////
// The one and only CStylesApp object
CStylesApp theApp;

/////////////////////////////////////////////////////////////////
// CStylesApp initialization
BOOL CStylesApp::InitInstance()
{
  CMainFrame *pMainWnd = new CMainFrame;
```

```
  m_pMainWnd = pMainWnd;
  pMainWnd->LoadFrame(IDR_MAINFRAME);

  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;

  pMainWnd->SetWindowPos(
    NULL,
    cx,
    cy,
    6*cx,
    2*cy,
    SWP_NOZORDER | SWP_SHOWWINDOW);
  return TRUE;
}

/////////////////////////////////////////////////////////////////
// CStylesApp commands

/////////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

// interface of the CMainFrame class
/////////////////////////////////////////////////////////////////

#include "ChildWindows.h"

class CMainFrame : public CFrameWnd
{
public:
  CMainFrame();

// Attributes
public:
  CChildWnd *apChildWnd[5];
  CPopupWnd *apPopupWnd[5];

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  //}}AFX_VIRTUAL

// Implementation
public:
  virtual ~CMainFrame();

// Generated message map functions
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
```

```
  afx_msg void OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

// implementation of the CMainFrame class
//

#include "stdafx.h"
#include "Styles.h"
#include "MainFrm.h"

/////////////////////////////////////////////////////////////////
// CMainFrame

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_WM_CREATE()
  ON_WM_GETMINMAXINFO()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

extern char szNameWnd[128];
CMainFrame::CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    apChildWnd[i] = NULL;
    apPopupWnd[i] = NULL;
  }
}

CMainFrame::~CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    if(apChildWnd[i] != NULL)
    {
      delete apChildWnd[i];
    }
  }
}
```

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszName = "Главное окно приложения \"Стили\"";

  return CFrameWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////
// CMainFrame message handlers

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  int i;
  for(i = 0; i < 5; i++)
  {
    apChildWnd[i] = new CChildWnd;
    apChildWnd[i]->Create(NULL,
      NULL, WS_VISIBLE,
      CRect(0, 0, 0, 0), this, (WORD)i);
    ::SetWindowLong(apChildWnd[i]->GetSafeHwnd(),
      GWL_USERDATA, i);
  }

  for(i = 0; i < 4; i++)
  {
    apPopupWnd[i] = new CPopupWnd;
    apPopupWnd[i]->Create(NULL, NULL, 0,
      rectDefault, this);
    ::SetWindowLong(apPopupWnd[i]->GetSafeHwnd(),
      GWL_USERDATA, i);
    sprintf(szNameWnd, "Окно %d", i);
    apPopupWnd[i]->SetWindowText(szNameWnd);
  }

  apPopupWnd[4] = new CPopupWnd;
  apPopupWnd[4]->Create(NULL, NULL, 0, rectDefault);

  ::SetWindowLong(apPopupWnd[4]->GetSafeHwnd(),
    GWL_USERDATA, 4);
  sprintf(szNameWnd, "Окно %d", 4);
  apPopupWnd[i]->SetWindowText(szNameWnd);

  ::SetClassLong(apChildWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_IBEAM));

  ::SetClassLong(apPopupWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_UPARROW));
```

```
  HBRUSH hNewBrush = ::CreateSolidBrush(RGB(255, 255, 220));
  ::SetClassLong(apPopupWnd[3]->GetSafeHwnd(),
    GCL_HBRBACKGROUND, (LONG)hNewBrush);

  for(i = 0; i < 5; i++)
  {
    apPopupWnd[i]->ShowWindow(SW_SHOWNORMAL);
    apPopupWnd[i]->UpdateWindow();
  }
  return 0;
}

void CMainFrame::OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI)
{
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  lpMMI->ptMinTrackSize.x = 6*cx;
  lpMMI->ptMinTrackSize.y = 2*cy;
  lpMMI->ptMaxTrackSize.x = 6*cx;
  lpMMI->ptMaxTrackSize.y = 2*cy;
  lpMMI->ptMaxPosition.x  = cx;
  lpMMI->ptMaxPosition.y  = cy;
  CFrameWnd::OnGetMinMaxInfo(lpMMI);
}

/////////////////////////////////////////////////////////////
// Файл ChildWindows.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// header file
//

/////////////////////////////////////////////////////////////
// CChildWnd window

class CChildWnd : public CWnd
{
// Construction
public:
  CChildWnd();

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CChildWnd)
  protected:
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  //}}AFX_VIRTUAL

// Implementation
public:
  virtual ~CChildWnd();
```

```
// Generated message map functions
protected:
  //{{AFX_MSG(CChildWnd)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////
// CPopupWnd frame

class CPopupWnd : public CFrameWnd
{
// Construction
public:
  CPopupWnd();

// Implementation
public:
  virtual ~CPopupWnd();

// Generated message map functions
  //{{AFX_MSG(CPopupWnd)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////
// Файл ChildWindows.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

// implementation file
//

#include "stdafx.h"
#include "Styles.h"
#include "ChildWindows.h"
int nCount = 0;

DWORD adwStyles[] =
{
  WS_CAPTION | WS_CLIPSIBLINGS,
  WS_CAPTION | WS_SYSMENU | WS_HSCROLL | WS_VSCROLL |
 WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_CLIPSIBLINGS,
  WS_THICKFRAME | WS_CLIPSIBLINGS,
  WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
    WS_CLIPSIBLINGS,
```

```
  WS_CAPTION | WS_SYSMENU | WS_HSCROLL | WS_VSCROLL |
    WS_CLIPSIBLINGS,
  WS_POPUP,
  WS_POPUP | WS_CAPTION | WS_THICKFRAME,
  WS_POPUP | WS_CAPTION | WS_SYSMENU |
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX,
  WS_POPUP | WS_CAPTION | WS_SYSMENU |
    WS_HSCROLL | WS_VSCROLL |WS_DISABLED,
  WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
    WS_HSCROLL | WS_VSCROLL
};

char *aszStyles[] =
{
  "WS_CAPTION | WS_CLIPSIBLINGS",
  "WS_CAPTION | WS_SYSMENU | WS_HSCROLL | WS_VSCROLL |\
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_CLIPSIBLINGS",
  "WS_THICKFRAME | WS_CLIPSIBLINGS",
  "WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX |\
    WS_MAXIMIZEBOX | WS_CLIPSIBLINGS",
  "WS_CAPTION | WS_SYSMENU | WS_HSCROLL |\
    WS_VSCROLL | WS_CLIPSIBLINGS",
  "WS_POPUP",
  "WS_POPUP | WS_CAPTION | WS_THICKFRAME",
  "WS_POPUP | WS_CAPTION | WS_SYSMENU |\
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX",
  "WS_POPUP | WS_CAPTION | WS_SYSMENU |\
    WS_HSCROLL | WS_VSCROLL",
  "WS_OVERLAPPPED | WS_CAPTION | WS_SYSMENU |\
    WS_HSCROLL | WS_VSCROLL"
};

char szNameWnd[128];

////////////////////////////////////////////////////////////////
// CChildWnd

CChildWnd::CChildWnd(){
}

CChildWnd::~CChildWnd(){
}

BEGIN_MESSAGE_MAP(CChildWnd, CWnd)
  //{{AFX_MSG_MAP(CChildWnd)
  ON_WM_CREATE()
  ON_WM_LBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
////////////////////////////////////////////////////////////
// CChildWnd message handlers

BOOL CChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= adwStyles[nCount];
  sprintf(szNameWnd, "Окно %d", nCount);
  cs.lpszName = szNameWnd;

  return CWnd::PreCreateWindow(cs);
}

int CChildWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  MoveWindow(nCount++ * 118, 24, 116, 96);
  return 0;
}

void CChildWnd::OnLButtonUp(UINT nFlags, CPoint point)
{
  WORD nWnd = (WORD)::GetWindowLong(this->GetSafeHwnd(),
    GWL_USERDATA);
  sprintf(szNameWnd, "Дочернее окно: %d", nWnd);
  MessageBox(aszStyles[nWnd], szNameWnd);
  CWnd::OnLButtonUp(nFlags, point);
}

////////////////////////////////////////////////////////////
// CPopupWnd

CPopupWnd::CPopupWnd()
{
}

CPopupWnd::~CPopupWnd()
{
}

BEGIN_MESSAGE_MAP(CPopupWnd, CFrameWnd)
  //{{AFX_MSG_MAP(CPopupWnd)
  ON_WM_CREATE()
  ON_WM_LBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////
// CPopupWnd message handlers

int CPopupWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  ModifyStyle((DWORD)-1, adwStyles[nCount]);
  int n = nCount++ - 5;
  MoveWindow(n * 144, 24, 136, 120);
  return 0;
}

void CPopupWnd::OnLButtonUp(UINT nFlags, CPoint point)
{
  WORD nWnd = (WORD)::GetWindowLong(this->GetSafeHwnd(),
    GWL_USERDATA);

  sprintf(szNameWnd, "Всплывающее окно: %d", nWnd);
  MessageBox(aszStyles[nWnd + 5], szNameWnd);
  CFrameWnd::OnLButtonUp(nFlags, point);
}

////////////////////////////////////////////////////////////
// Файл stdafx.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

// source file that includes just the standard
// includes
// Styles.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information
#include "stdafx.h"

////////////////////////////////////////////////////////////
// Файл stdafx.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

// include file for standard system include files,
// or project specific include files that are used
// frequently, but are changed infrequently
//
#define VC_EXTRALEAN        // Exclude rarely-used stuff from
                            // Windows headers
#include <afxwin.h>         // MFC core and standard components

////////////////////////////////////////////////////////////
// Файл Styles.rc
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

//Microsoft Developer Studio generated resource script.
//
```

```
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS

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

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

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

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

//////////////////////////////////////////////////////////////
// TEXTINCLUDE

1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END

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

3 TEXTINCLUDE DISCARDABLE
BEGIN
  "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
  "#define _AFX_NO_OLE_RESOURCES\r\n"
  "#define _AFX_NO_TRACKER_RESOURCES\r\n"
  "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
  "\r\n"
  "#if !defined(AFX_RESOURCE_DLL)||defined(AFX_TARG_ENU)\r\n"
  "#ifdef _WIN32\r\n"
  "LANGUAGE 9, 1\r\n"
  "#pragma code_page(1252)\r\n"
  "#endif\r\n"
  "#include ""res\\Styles.rc2""  // non-Microsoft Visual C++
                                //edited resources\r\n"
  "#include ""afxres.rc""       // Standard components\r\n"
  "#endif\0"
END
```

```
#endif    // APSTUDIO_INVOKED

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

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
   MENUITEM "E&xit",                          ID_APP_EXIT
  END
END

#ifndef _MAC

//////////////////////////////////////////////////////////////
// Version

VS_VERSION_INFO VERSIONINFO
  FILEVERSION 1,0,0,1
  PRODUCTVERSION 1,0,0,1
  FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
  FILEFLAGS 0x1L
#else
  FILEFLAGS 0x0L
#endif
  FILEOS 0x4L
  FILETYPE 0x1L
  FILESUBTYPE 0x0L
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "040904B0"
    BEGIN
      VALUE "CompanyName", "\0"
      VALUE "FileDescription", "STYLES MFC Application\0"
      VALUE "FileVersion", "1, 0, 0, 1\0"
      VALUE "InternalName", "STYLES\0"
      VALUE "LegalCopyright", "Copyright © 1996\0"
      VALUE "LegalTrademarks", "\0"
      VALUE "OriginalFilename", "STYLES.EXE\0"
      VALUE "ProductName", "STYLES Application\0"
      VALUE "ProductVersion", "1, 0, 0, 1\0"
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x409, 1200
  END
END
```

```
#endif    // !_MAC

/////////////////////////////////////////////////////////////////////////////
// Icon

IDR_MAINFRAME    ICON    DISCARDABLE     "res\\Styles.ico"

/////////////////////////////////////////////////////////////////////////////
// String Table

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME "Styles\n\nStyles\n\n\nStyles.Doc\nStyles"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  AFX_IDS_APP_TITLE       "Приложение Styles"
  AFX_IDS_IDLEMESSAGE     "Готовность"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_APP_EXIT             "Завершение работы приложения\nВыход"
END

#endif    // Russian resources

/////////////////////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED

// Generated from the TEXTINCLUDE 3 resource.
//
#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE 9, 1
#pragma code_page(1252)
#endif
#include "res\Styles.rc2"  // non-Microsoft Visual C++
                           // edited resources
#include "afxres.rc"       // Standard components
#endif

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

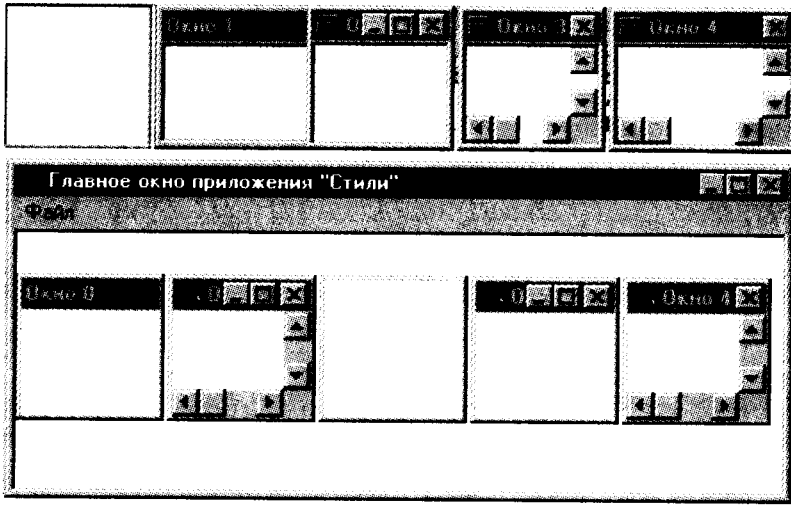


Рис. 22. Различные типы окон приложения Styles

Создание главного окна SDI-приложения

В рассмотренной ранее программе First мы образовали класс нашего окна из базового *CWnd*. Это допустимый, но очень неэффективный путь. Пришло время познакомить вас с одним из возможных подходов для создания главного окна SDI-приложения.[1]

Прежде всего обратим внимание на базовый класс нашего главного окна:

```
class CMainFrame : public CFrameWnd
{
...
};
```

Почему *CFrameWnd*? Дело в том, что этот класс спроектирован таким образом, чтобы предоставить максимум удобств для работы с перекрывающимися и всплывающими окнами. А, как вы помните, в качестве главного окна приложения наиболее приемлемым является именно перекрывающееся окно. Рассмотрим теперь подробно функцию *CStylesApp::InitInsnance* где, собственно, и начинается создание главного окна:

```
BOOL CStylesApp::InitInstance()
{
  CMainFrame *pMainWnd = new CMainFrame;
  m_pMainWnd = pMainWnd;
```

[1] Второй, еще более эффективный, подход будет представлен при знакомстве с архитектурой документ/представление.

```
  pMainWnd->LoadFrame(IDR_MAINFRAME);
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  pMainWnd->SetWindowPos(NULL,
    cx,
    cy,
    6*cx,
    2*cy,
    SWP_NOZORDER | SWP_SHOWWINDOW);
  return TRUE;
}
```

Процесс создания окна состоит из трех шагов. Прежде всего необходимо создать объект "главное окно":

```
CMainFrame *pMainWnd = new CMainFrame;
```

Следующий шаг состоит в вызове специальной функции, которая создает окно Windows и присоединяет его к нашему объекту:

```
pMainWnd->LoadFrame(IDR_MAINFRAME);
```

И, наконец, следует отобразить окно на экране, чтобы пользователь мог с ним взаимодействовать:

```
pMainWnd->SetWindowPos(NULL, cx, cy, 6*cx, 2*cy,
  SWP_NOZORDER | SWP_SHOWWINDOW);
```

Рассмотрим сначала процесс создания окна Windows, или, используя терминологию MFC, фрейма окна.

Из двух функций, реализованных в классе *CFrameWnd*, мы остановили свой выбор на виртуальной функции *CFrameWnd::LoadFrame*, которая берет на себя задачу регистрации оконного класса, используя для этого предопределенный шаблон, и извлекает из файла ресурсов, задаваемого идентификатором, остальные необходимые для создания окна Windows параметры:

```
virtual BOOL CFrameWnd::LoadFrame(
  UINT nIDResource,
  DWORD dwDefaultStyle = WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
  CWnd *pParentWnd = NULL,
  CCreateContext *pContext = NULL)
```

Функция возвращает значение TRUE, если создание фрейма прошло успешно, и FALSE в противном случае. Она имеет следующие параметры: *nIDResource* — идентификатор разделяемых ресурсов, ассоциированных с фреймом, который определяет меню, таблицу акселераторов, пиктограмму и строку, помещаемую в заголовок окна; *pParentWnd* — указатель на родительское окно данного фрейма (необязательный параметр); *pContext* — указатель на структуру *CCreateContex*, которая определяет объекты, связанные с

данным фреймом; *dwDefaultStyle* — стиль фрейма (необязательный параметр), дополнительно к стандартным могут быть использованы следующие стили:

FWS_ADDTOTITLE	в конец заголовка окна будет добавлена дополнительная текстовая строка
FWS_PREFIXTITLE	перед заголовком окна будет помещено имя документа
FWS_SNAPTOBAR	позволяет подогнать размеры окна под размеры панели элементов управления

Основное назначение этой функции — максимальное упрощение процесса создания фрейма окна.[1] Используя идентификатор ресурсов, функция извлекает из них необходимые параметры и ... вызывает функцию *CFrameWnd::Create*, к рассмотрению которой мы и переходим:

```
BOOL CFrameWnd::Create(
  LPCTSTR lpszClassName,
  LPCTSTR lpszWindowName,
  DWORD dwStyle = WS_OVERLAPPEDWINDOW,
  const RECT &rect = rectDefault,
  CWnd *pParentWnd = NULL,
  LPCTSTR lpszMenuName = NULL,
  DWORD dwExStyle = 0,
  CCreateContext *pContext = NULL)
```

Функция возвращает значение TRUE, если создание прошло успешно, и FALSE в противном случае. Многие из параметров, передаваемых в функцию, вам уже должны быть хорошо знакомы, поэтому мы их только перечислим, рассматривая только те, которые не совпадают с уже описанными для функции *CFrameWnd::LoadFrame*: *lpszClassName* — адрес символьной строки, содержащей зарегистрированное имя класса, если этот параметр равен NULL, то в качестве шаблона используются атрибуты предопределенного класса "AfxFrameOrView"; *lpszWindowName* — адрес символьной строки, содержащей имя конкретного окна, этот же текст помещается в заголовок окна; *dwStyle* — стиль окна (необязательный параметр); *rect* — ссылка на структуру, содержащую размеры и координаты окна относительно его родителя (необязательный параметр); *lpszMenuName* — адрес символьной строки, содержащей имя ресурса меню, используемого в окне, если меню задано числовым идентификатором, то для получения строки следует использовать макрос Windows MAKEINTRESOURCE (необязательный параметр); *dwExStyle* — определяет атрибуты расширенного стиля (необязательный параметр).

Функция загружает ресурс меню и вызывает функцию *CWnd::CreateEx*, которая регистрирует имя оконного класса, заполняя структуру *WNDCLASS*,

---

[1] При использовании функции *CFrameWnd::LoadFrame* единственным обязательным ресурсом является меню. Если его не определить, то программа будет выдавать сообщение об ошибке.

и вызывает виртуальную функцию *PreCreateWindow*. В процессе выполнения функции Windows посылает в окно следующие сообщения: WM_GETMINMAXINFO, WM_NCCREATE, WM_NCCALKSIZE и WM_CREATE. Если при создании окна был определен стиль WS_VISIBLE, то в окно дополнительно посылаются все сообщения, необходимые для активизации и отображения окна на экране. Таким образом, переопределяя функцию *PreCreateWindow* и обработчик любого из представленных сообщений Windows, можно изменять параметры оконного класса и собственно окна, как до, так и после создания окна Windows.

Рассмотрим, как этот механизм реализован в нашей программе. Начнем с функции *CMainFrame::PreCreateWindow*:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszName = "Главное окно приложения \"Стили\"";
  return CFrameWnd::PreCreateWindow(cs);
}
```

Единственный параметр, передаваемый в эту функцию, — ссылка на структуру *CREATESTRUCT*, которая определяет параметры инициализации, посылаемые в оконную процедуру, и содержит следующие поля:

```
typedef struct tagCREATESTRUCT{
  LPVOID lpCreateParams;
  HANDLE hInctance;
  HMENU hMenu;
  HWND hwndParent;
  int cy;
  int cx;
  int y;
  int x;
  LONG style;
  LPCSTR lpszName;
  LPCSTR lpszClass;
  DWORD dwExStyle;
} CREATESTRUCT;
```

Назначение полей этой структуры: *lpCreateParams* — указатель на данные, используемые при создании окна; *hInstance* — дескриптор модуля, который владеет окном; *hMenu* — идентификатор меню, используемого окном; *hwndParent* — дескриптор окна-родителя, если этот параметр равен NULL, то новое окно является окном верхнего уровня; *cy*, *cx*, *y*, *x* — целые числа, задающие соответственно ширину и высоту окна, а также его положение относительно своего владельца; *lpszName* — символьная строка, задающая имя окна; *lpszClass* — символьная строка, содержащая имя класса, которому принадлежит окно; *dwExStyle* — определяет атрибуты расширенного стиля окна.

Мы использовали эту функцию, чтобы не заниматься процедурой регистрации класса окна, возложив ее выполнение на библиотеку, поскольку принятые по умолчанию установки нас вполне устраивают. Единственное, что нас не устраивает, это заголовок окна, который мы и переопределили:

```
cs.lpszName = "Главное окно приложения \"Стили\"";
```

Это один из возможных способов присвоения имени окну.[1]

После того как окно Windows создано и присоединено к объекту фрейма окна, в ответ на сообщение WM_CREATE вызывается обработчик сообщения *OnCreate* (механизм вызова был нами подробно описан при обсуждении механизма работы с сообщениями в библиотеке MFC):

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  return 0;
}
```

Эта функция — одно из возможных мест, где создаются и присоединяются к окну произвольные элементы пользовательского интерфейса. Для правильного завершения процесса создания окна не забывайте вызывать перед какими-либо своими действиями обработчик базового класса *CFrameWnd::OnCreate*.

После того как фрейм создан, его необходимо вывести на экран. Библиотека предоставляет несколько функций для реализации этого процесса. Для отображения нашего главного окна мы остановились на одной из них — функции *CWnd::SetWindowPos*:

```
BOOL CWnd::SetWindowPos(
  const CWnd *pWndInsertAfter,
  int x,
  int y,
  int cx,
  int cy,
  UINT nFlags)
```

Вызов этой функции позволяет изменить как размеры, так и расположение дочерних и всплывающих окон относительно экрана и других окон. Она имеет следующие параметры: *pWndInsertAfter* — указывает на объект *CWnd*, после которого будет расположено окно, вместо этого указателя можно использовать одно из следующих предопределенных значений, задающих расположение окна относительно других окон:

---

[1] Не забывайте вызывать функцию базового класса *CFrameWnd::PreCreateWindow* для проведения правильных инициализирующих действий.

wndBottom	разместить окно под другими окнами (внизу стека Z-порядка); если окно имело статус wndTopMost, то этот статус теряется — удаляется флаг расширенного стиля WS_EX_TOPMOST
wndTop	разместить окно над другими окнами (на вершине стека Z-порядка)
wndTopMost	разместить окно над всеми другими окнами, имеющими статус wndTopMost; эта позиция сохраняется (флаг расширенного стиля WS_EX_TOPMOST остается установленным) даже когда окно находится в неактивном состоянии
wndNoTopMost	перемещает окно выше всех окон, имеющих статус wndTop, но ниже окон, имеющих статус wndTopMost

x, *y*, *cx*, *cy* — задают новые значения, соответственно, горизонтального и вертикального расположения окна, его ширины и высоты; *nFlags* — определяет параметры размера и позиции окна и может принимать одно из следующих значений:

SWP_DRAWFRAME	задает рамку вокруг окна, определенную в классе окна
SWP_HIDEWINDOW	делает окно невидимым
SWP_NOACTIVATE	блокирует активизацию окна
SWP_NOMOVE	предписывает окну сохранение текущей позиции; параметры *x* и *y* игнорируются
SWP_NOREDRAW	предписывает не перерисовывать окно; после перемещения приложение должно перерисовать окно самостоятельно
SWP_NOSIZE	задает сохранение текущих размеров; параметры *cx* и *cy* игнорируются
SWP_NOZORDER	задает сохранение текущего расположения окна относительно других окон, при этом параметр pWndInsertAfter игнорируется
SWP_SHOWWINDOW	отображает окно

Примечание

Windows располагает окна на экране согласно их положению в так называемом Z-порядке, который определяет позицию окна в стеке перекрывающихся окон. Стек окна ориентирован вдоль воображаемой оси (Z-оси) внутрь экрана. Окно, расположенное на вершине стека, перекрывает все другие окна, а расположенное внизу — перекрывается всеми другими окнами. Приложение не может активизировать окно без того, чтобы не перенести его на самый верх Z-порядка. Кроме того, расположение окна относительно экрана и других окон может изменяться без каких-либо ограничений.

Мы использовали эту функцию для указания размеров и расположения окна, а также для того, чтобы сделать его видимым.[1]

Отложим пока обсуждение вопросов создания и разрушения объекта класса *CStylesApp* до тех пор, пока не рассмотрим, как создаются дочерние окна.[2]

Создание дочерних окон

Теперь, после создания основного окна приложения, можно переходить к созданию некоторого дополнительного пространства для отображения информации или для получения команд от пользователя. Каждому "ребенку", как в свое время и основному окну, можно приписать конкретный стиль таким образом, чтобы приспособить его внешний вид и назначение к стоящим перед ним целям. В приложении Styles мы разбили создаваемые окна на два набора. Окна первого располагаются на экране независимо от положения основного окна — все они имеют стиль WS_POPUP, а для окон из второго набора установлен стиль WS_CHILD и, следовательно, они не могут покинуть пределы рабочей области основного окна.

Для того чтобы понять принципы построения и работу рассматриваемой программы, вернемся к обработчику сообщения WM_CREATE. Повторим ту часть программы, на которую следует обратить внимание:

```
extern char szNameWnd[128];

// Создание пяти дочерних окон
  for(int i = 0; i < 5; i++)
  {
    apChildWnd[i] = new CChildWnd;
    apChildWnd[i]->Create(NULL, NULL, WS_VISIBLE,
      CRect(0, 0, 0, 0), this, (UINT)i);
  }

  // Установка нового курсора для класса дочерних окон
  ::SetClassLong(apChildWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_IBEAM));

  // Создание четырех всплывающих окон
  for(i = 0; i < 4; i++)
  {
    apPopupWnd[i] = new CPopupWnd;
    apPopupWnd[i]->Create(NULL, NULL, 0,
      rectDefault, this);
```

---

[1] Существуют и другие способы решения этой задачи. Их мы опишем ниже.

[2] Поскольку любое порожденное окно программисты называют "дочерним", то часто возникает некоторая путаница. Вы должны совершенно четко отличать технический термин WS_CHILD от более широкого разговорного понятия "дочерний".

```
    // Указание номера окна в поле данных ассоциированных с окном
    ::SetWindowLong(apPopupWnd[i]->GetSafeHwnd(),
      GWL_USERDATA, i);

    // Формирование нового заголовка окна
    sprintf(szNameWnd, "Окно %d", i);
    apPopupWnd[i]->SetWindowText(szNameWnd);
  }

  // Создание перекрывающегося окна
  apPopupWnd[4] = new CPopupWnd;
  apPopupWnd[4]->Create(NULL, NULL, 0, rectDefault);
  ::SetWindowLon2g(apPopupWnd[i]->GetSafeHwnd(),
    GWL_USERDATA, 4);
  sprintf(szNameWnd, "Окно %d", i);
  apPopupWnd[4]->SetWindowText(szNameWnd);

  // Установка нового курсора для класса всплывающих окон
  ::SetClassLong(apPopupWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_UPARROW));

  // Создание и установка нового цвета фона для всплывающих окон
  HBRUSH hNewBrush = ::CreateSolidBrush(RGB(255, 255, 220));
  ::SetClassLong(apPopupWnd[3]->GetSafeHwnd(),
    GCL_HBRBACKGROUND, (LONG)hNewBrush);

  // Отображение всплывающих окон
  for(i = 0; i < 5; i++)
  {
    apPopupWnd[i]->ShowWindow(SW_SHOWNORMAL);
    apPopupWnd[i]->UpdateWindow();
  }
```

При ближайшем рассмотрении это не более, чем создание десяти окон, принадлежащих двум новым классам. Объекты одного класса позволяют создавать дочерние окна:

```
class CChildWnd : public CWnd
{
  ...
};
```

а объекты второго — всплывающие и/или перекрывающиеся окна (хотя с тем же успехом можно создавать и дочерние):

```
class CPopupWnd : public CFrameWnd
{
  ...
};
```

Такое разбиение не продиктовано необходимостью и произведено лишь для наглядности. Дело в том, что, как мы упоминали, на базе класса *CWnd* можно создавать только дочерние окна (имеющие стиль WS_CHILD). Его мы и решили использовать для этих целей. На базе же класса *CFrameWnd* могут быть созданы окна любого из трех имеющихся типов. Поэтому для создания всплывающих окон именно он был использован в качестве базового.

В основном процесс создания окон обоих типов стандартен: создаются объекты классов, после чего вызывается некоторая функция для создания окна Windows. С целью рассмотрения как можно большего числа предоставляемых библиотекой возможностей мы включили в текст программы дополнительные строки. Но обо всем по порядку. Начнем с дочерних окон, имеющих установленный бит стиля WS_CHILD (процесс создания объекта класса ничем не отличается от уже рассмотренных, и мы больше не будем останавливаться на этом вопросе). Итак, вызов функции *CWnd::Create*:

```
apChildWnd[i]->Create(NULL, NULL, WS_VISIBLE,
  CRect(0, 0, 0, 0), this, (UINT)i);
```

Подчеркнем некоторые моменты. Во-первых, не определен класс Windows, к которому относится окно, — эту задачу мы переложили на плечи библиотеки. Во-вторых, установив бит стиля WS_VISIBLE, мы, тем самым, задали, чтобы окно сразу же после создания было выведено на экран. В-третьих, окно мы превратили в точку и поместили его в левый верхний угол экрана. В-четвертых, окно имеет "родителя" — пятый параметр определен как *this*. И, наконец, в качестве последнего параметра мы передаем номер создаваемого окна.

Как и при создании главного окна мы переопределяем функцию

```
BOOL CChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= adwStyles[nCount];
  sprintf(szNameWnd, "Окно %d", nCount);
  cs.lpszName = szNameWnd;
  return CWnd::PreCreateWindow(cs);
}
```

и обработчик сообщения WM_CREATE

```
int CChildWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  MoveWindow(nCount++ * 118, 24, 116, 96);
  return 0;
}
```

Первая заполняет поля *style* и *lpszName* структуры *CREATESTRUCT* соответственно значениями стиля и имени конкретного окна, а вторая — устанавливает размер и положение вновь созданного окна (напомним, что мы задали окну нулевые размеры). Со структурой *CREATESTRUCT* мы уже познакомились, поэтому рассмотрим изящную функцию *CWnd::MoveWindow*, которая реализована в двух вариантах:

```
void CWnd::MoveWindow(
  int x,
  int y,
  int nWidth,
  int nHeight,
  BOOL bRepaint = TRUE);
```

и

```
void CWnd::MoveWindow(
  LPCRECT lpRect,
  BOOL bRepaint = TRUE)
```

С помощью любой из этих функций можно одновременно изменить размер и положение окна, задаваемые в параметрах *x*, *y*, *nWidth*, *nHeight* или *lpRect*.[1] Последний параметр этой функции определяет, будет ли окно перерисовываться после перемещения. Если этот параметр равен TRUE (по умолчанию), то объект "окно" получает сообщение Windows WM_PAINT. Кроме того, эта функция посылает еще и сообщение WM_GETMINMAXINFO, которое вы также можете обработать.

Как мы уже отметили, задача регистрации оконного класса была поручена библиотеке. Нам лишь остается "подправить" некоторые поля в соответствии с собственными нуждами. Один из возможных путей при этом состоит в использовании функций API *SetClassLong* и *SetWindowLong*. Возможно, вы захотите их использовать в своих программах, поэтому рассмотрим их подробнее:

```
DWORD SetClassLong(
  HWND hWnd,
  int nIndex,
  LONG dwNewLong) –
```

эта функция устанавливает в структуре *WNDCLASS*, описывающей оконный класс для *hWnd*, новое значение *dwNewLong* в поле, определяемом параметром *nIndex*, который может принимать одно из следующих значений:

GCL_CBCLSEXTRA — устанавливает размер (в байтах) дополнительной памяти, предназначенной для использования всеми окнами, создаваемыми на базе этого класса; размер памяти, уже ассоциированной с классом, при этом не изменяется

[1] Отметим только, что позиция дочернего окна определяется координатами его владельца, в то время как для всплывающего окна положение задается относительно экрана.

GCL_CBWNDEXTRA	устанавливает размер (в байтах) дополнительной памяти, предназначенной для использования отдельно каждым окном, создаваемыми на базе этого класса; размер памяти, уже ассоциированной с окном, при этом не изменяется
GCL_HBRBACKGROUND	устанавливает дескриптор кисти для фона, ассоциированного с этим классом
GCL_HCURSOR	устанавливает дескриптор курсора, ассоциированного с этим классом
GCL_HICON	устанавливает дескриптор пиктограммы, ассоциированной с этим классом; это значение используется только в Windows 95 и не определено в Windows NT
GCL_HMODULE	устанавливает дескриптор модуля, который зарегистрировал этот класс
GCL_MENUNAME	устанавливает адрес строки меню, которая идентифицирует ресурс меню, ассоциированного с этим классом
GCL_STYLE	устанавливает биты стиля оконного класса
GCL_WNDPROC	устанавливает адрес оконной процедуры, ассоциированной с классом

При успешном выполнении функция возвращает значение указанного слова из структуры класса окна или нулевое значение при ошибке. Дополнительную информацию об ошибке можно получить, вызвав функцию API *GetLastError*.

Мы воспользовались этой функцией для того, чтобы установить новые значения для курсора и цвета фона.

Обратную задачу — получение информации из структуры класса окна — выполняет функция

```
DWORD GetClassLong(
  HWND hWnd,
  int nIndex)
```

Для работы с дополнительной памятью в структуре окна API предоставляет также две функции:

```
DWORD SetWindowLong(
  HWND hWnd,
  int nIndex,
  LONG dwNewLong)
```

и

```
DWORD GetWindowLong(
  HWND hWnd,
  int nIndex)
```

По своему принципу действия они аналогичны только что рассмотренным, но работают с дополнительной памятью в структуре окна. Параметр *nIndex* при этом может принимать следующие значения:

GWL_EXSTYLE	устанавливает новый расширенный стиль окна
GWL_STYLE	устанавливает новый стиль окна
GWL_WNDPROC	устанавливает новый адрес оконной процедуры
GWL_HINSTANCE	устанавливает новый дескриптор приложения
GWL_ID	устанавливает новый идентификатор окна
GWL_USERDATA	устанавливает 32-разрядное значение, ассоциированное с конкретным окном и предназначенное для использования приложением, которым это окно создано

Если дескриптор *hWnd* определяет блок диалога, то доступными являются следующие значения:

DWL_DLGPROC	устанавливает новый адрес процедуры блока диалога
DWL_MSGRESULT	устанавливает значение, возвращаемое при обработке сообщения в процедуре блока диалога
DWL_USER	устанавливает новую дополнительную информацию для приложения, такую как идентификатор или указатель

Для функции *GetWindowLong* кроме перечисленных определено еще одно доступное значение:

GWL_HWNDPARENT	возвращает дескриптор родительского окна или NULL, если его нет

Мы использовали эти функции для записи в дополнительную память номера созданного окна. Другими словами, у первого окна в этом поле будет нуль, у второго — 1, у третьего — 2 и т. д. Позже, во время изображения окон на экране, программа воспользуется этими значениями для идентификации окон.

Перед тем как закончить обсуждение вопроса об изменении характеристик оконных классов и собственно окон, осталось отметить способ получения дескриптора окна Windows. Для этой цели в классе *CWnd* предусмотрена специальная функция

HWND CWnd::**GetSafeWnd**(),

которая позволяет получить дескриптор окна Windows, хранящийся в члене класса *CWnd::m_hWnd*, или NULL, если к объекту не присоединено никакое окно.

Но вернемся к нашей программе. Нам осталось разобраться в создании всплывающих окон. Обратите внимание на два несколько отличающихся вызова функции *CFrameWnd::Create*:

```
apPopupWnd[i]->Create(NULL, NULL, 0, rectDefault, this);
```

и

```
apPopupWnd[4]->Create(NULL, NULL, 0, rectDefault);
```

и связанное с этим отличие в поведении всплывающих окон 3 и 4. Для четвертого окна мы не указали родителя, и поэтому оно ведет себя достаточно независимо от нашего главного окна.

Второй момент, на который хотелось бы обратить ваше внимание, связан с еще одним способом изменения стилей созданных окон. Обратимся к тексту:

```
int CPopupWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  ModifyStyle((DWORD)-1, adwStyles[nCount]);
...
  return 0;
}
```

Как видите, для установки стиля окна мы воспользовались функцией класса *CWnd — ModifyStyle*

```
BOOL CWnd::ModifyStyle(
 DWORD dwRemove,
 DWORD dwAdd,
 UINT  nFlags = 0)
```

Эта функция позволяет удалить (*dwRemove*) или установить (*dwAdd*) определенные атрибуты стиля окна. Если установлен параметр *nFlags*, то вызывается функция API *::SetWindowPos* с этим параметром, являющимся комбинацией одного или нескольких значений: SWP_NOMOVE, SWP_NOACTIVATE, SWP_NOSIZE и SWP_NOZORDER.

Для изменения флагов расширенного стиля можно использовать функцию:

```
BOOL CWnd::ModifyStyleEx(
 DWORD dwRemove,
 DWORD dwAdd,
 UINT nFlags = 0)
```

Попутно упомянем две функции, которые позволяют узнать текущие обычный и расширенный стили окна:

```
DWORD CWnd::GetStyle()
```

и

```
DWORD CWnd::GetExStyle()
```

При рассмотрении вопроса о создании главного окна приложения мы обошли молчанием конструктор и деструктор объекта этого окна. Вернемся теперь к их содержимому:

```
CMainFrame::CMainFrame()
{
  for(int i = 0; i < 5; i++)
```

```
  {
    apChildWnd[i] = NULL;
  }
}
CMainFrame::~CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    if(apChildWnd[i] != NULL)
    {
      delete apChildWnd[i];
    }
  }
}
```

Как видите, мы явно освобождаем только ту память, которую занимали объекты, образованные из класса, производного от *CWnd*. При использовании в качестве базовых других классов (в нашем случае *CFrameWnd*) явно освобождать память не надо, т. к. объекты выполняют это сами.

Ограничение размеров окна

Мы создали главное окно приложения Styles таким образом, что пользователь не может изменить его размеры. Более того, окно не изменит своих размеров даже тогда, когда вы попытаетесь максимизировать его. Такое поведение окна достигается введением соответствующей обработки сообщения Windows WM_GETMINMAXINFO, которая выполняется обработчиком:

```
void CMainFrame::OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI)
{
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  lpMMI->ptMinTrackSize.x = 6*cx;
  lpMMI->ptMinTrackSize.y = 2*cy;
  lpMMI->ptMaxTrackSize.x = 6*cx;
  lpMMI->ptMaxTrackSize.y = 2*cy;
  lpMMI->ptMaxPosition.x = cx;
  lpMMI->ptMaxPosition.y = cy;
  CFrameWnd::OnGetMinMaxInfo(lpMMI);
}
```

Обработчику в качестве параметра передается указатель на структуру типа *MINMAXINFO*, которая содержит информацию о максимально возможных размерах и позиции окна, а также о минимальных и максимальных изменяемых размерах:

```
typedef struct tagMINMAXINFO {
  POINT ptReserved;
  POINT ptMaxSize;
  POINT ptMaxPosition;
  POINT ptMinTrackSize;
  POINT ptMaxTrackSize;
} MINMAXINFO;
```

Поля этой структуры имеют следующее назначение: *ptReserved* — зарезервировано для внутреннего использования; *ptMaxSize* — устанавливает размеры окна, которые оно может иметь в развернутом виде; *ptMaxPosition* — устанавливает значение, которое должен иметь левый верхний угол развернутого окна; *ptMinTrackSize* — устанавливает минимальные размеры, до которых пользователь может сжать границы окна; *ptMaxTrackSize* — устанавливает максимальные размеры, до которых пользователь может растянуть границы окна.

Как видите все очень просто. Последнее, что нам осталось рассмотреть — каким образом можно получить информацию о системных параметрах.

Функция GetSystemMetrics

Эта функция позволяет получить размеры (высоту и ширину) изображаемых элементов Windows, а также другую информацию. Все размеры даются в пикселях:

int **GetSystemMetrics**(int nIndex)

Значение *nIndex* определяет параметр, который вы хотите получить (возвращаемые функцией значения могут изменяться в зависимости от типа используемого дисплея и установок пользователя). Приведем небольшой список наиболее часто используемых параметров: [1]

SM_CMOUSEBUTTONS	число кнопок мыши; если мышь не установлена, то возвращается 0
SM_CXDORDER, SM_CYDORDER	ширина и высота рамки окна
SM_CXFULLSCREEN, SM_CYFULLSCREEN	ширина и высота рабочей области максимизированного окна
SM_CXICON, SM_CYICON	ширина и высота большой пиктограммы, заданные по умолчанию; обычно они равны 32, но могут изменяться в зависимости от установок пользователя [2]

[1] Полный список возможных параметров для получения с помощью функции *GetSystemMetrics* приведен в приложении 5.

[2] Функция *LoadIcon* в Windows 95 может загружать пиктограммы только этих размеров; если требуются другие, то следует использовать битовые массивы.

SM_CXMIN, SM_CYMIN	минимальные допустимые ширина и высота окна
SM_CXMINTRACK, SM_CYMINTRACK	минимальные величины, до которых пользователь может уменьшить ширину и высоту окна; для их изменения необходимо обработать сообщение Windows WM_GETMINMAXINFO
SM_CXSCREEN, SM_CYSCREEN	ширина и высота экрана
SM_CYCAPTION	высота области заголовка окон
SM_CYMENU	высота полосы меню
SM_SWAPBUTTON	не нуль, если левая и правая кнопки мыши обменялись своими функциями
SM_CXEDGE, SM_CYEDGE	горизонтальный и вертикальный размеры трехмерной рамки (только в Windows 95 и Windows NT 4.0)
SM_CXFIXEDFRAME, SM_CYFIXEDFRAME	толщина (по горизонтали и вертикали) рамки окна, которое имеет заголовок, но не может изменить свои размеры (только в Windows 95 и Windows NT 4.0)
SM_CXMAXIMIZED, SM_CYMAXIMIZED	предельные допустимые размеры развернутого окна (только в Windows 95 и Windows NT 4.0)
SM_CXMAXTRACK, SM_CYMAXTRACK	максимальные величины, до которых пользователь может увеличить ширину и высоту окна; для их изменения необходимо обработать сообщение Windows WM_GETMINMAXINFO (только в Windows 95 и Windows NT 4.0)
SM_CXMINIMIZED, SM_CYMINIMIZED	предельные допустимые размеры свернутого окна (только в Windows 95 и Windows NT 4.0)
SM_CXSIZEFRAME, SM_CYSIZEFRAME	толщина (по горизонтали и вертикали) рамки окна, которое может изменять свои размеры (только в Windows 95 и Windows NT 4.0)
SM_CXSMICON, SM_CYSMICON	рекомендуемые размеры маленькой пиктограммы, которая обычно используется в заголовке окна (только в Windows 95 и Windows NT 4.0)
SM_CYSMCAPTION	высота области уменьшенного заголовка (только в Windows 95 и Windows NT 4.0)
SM_NETWORK	в возвращаемом значении установлен самый младший бит, если сеть присутствует; остальные биты зарезервированы для будущего использования (только в Windows 95 и Windows NT 4.0)
SM_SECURE	функция возвращает ненулевое значение, если установлена защита, и 0 в противном случае (только в Windows 95 и Windows NT 4.0)

Ограничение доступа к окну

Большинство из стилей окна, использованных в приложении Styles, достаточно ясны. Поэтому мы остановимся только на одном из них — WS_DISABLED. Окна, имеющие такой стиль, абсолютно неподвижны, и ими нельзя манипулировать ни с помощью мыши, ни с помощью клавиатуры. Обратите также внимание, что у такого окна и курсор не соответствует заданному в оконном классе. Для доступа к этому свойству окон в классе *CWnd* реализована специальная функция

BOOL CWnd::**EnableWindow**(BOOL bEnable = TRUE)

которая разрешает (*bEnable* = TRUE) или запрещает (*bEnable* = FALSE) доступ к окну для мыши и клавиатуры. Если при вызове этой функции изменилось состояние окна, то до возврата из нее Windows посылает окну сообщение WM_ENABLE.

Для того чтобы узнать текущее состояние окна в смысле его доступности, можно использовать функцию

BOOL CWnd::**IsWindowEnable**()

которая возвращает значение TRUE, если окно доступно, и FALSE в противном случае.

Основная цель рассмотренного приложения и этой главы, заключается в знакомстве с основными классами библиотеки MFC, отвечающими за создание и работу окон Windows практически всех типов. Исключение составляют окна так называемых MDI-приложений — приложений многодокументного (или многооконного) интерфейса, к которым мы сейчас и переходим.

Глава 12

Создание многодокументных приложений

Мы надеемся, что нет необходимости подробно представлять *многодокументный интерфейс* (MDI, Multiple Document Interface), который позволяет в одном приложении работать с несколькими документами или с разными представлениями одного и того же документа. Этот интерфейс описан в руководстве по разработке интерфейса пользователя System Application Architecture Common User Access Advanced Interface Design Guide (SAA/CUA), созданном фирмой IBM. В Windows интерфейс MDI используется, начиная с версии 3.0, типичным примером приложения на базе MDI является хорошо всем знакомый текстовый процессор Microsoft Word для Windows.

Как и все приложения Windows, MDI-приложение имеет одно главное окно, которое управляет своими окнами "MDI child".[1] Поведение окон "MDI child" напоминает поведение как дочерних, так и перекрывающихся окон. Так же, как и первые, их невозможно переместить за пределы главного окна приложения. И в то же время окна "MDI child" можно сделать активными, при этом их заголовок выделяется цветом. Кроме того, каждое окно "MDI child" имеет, как правило, системное меню и кнопки изменения размера. Если свернуть такое окно, то его пиктограмма расположится в нижней части главного окна приложения. При увеличении размеров окна до максимальных, пиктограмма системного меню окна "MDI child" помещается в левую часть основного меню, а в правой части расположатся кнопки изменения размера и закрытия окна.

Одним словом, можно, конечно, создать аналогичное приложение с использованием обычных окон, но для этого придется затратить немало усилий. Более того, достаточно серьезную работу надо проделать, чтобы создать полноценное MDI-приложение, используя функции библиотеки SDK (Software Development Kit). Для того чтобы в этом убедиться, достаточно

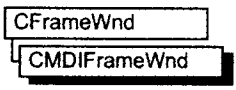
---

[1] Для названия окна документа мы будем придерживаться терминологии фирмы Microsoft, которая для дочерних окон MDI-приложений использует названия "MDI child frame" или "MDI child window".

рассмотреть исходные тексты приложения MULTIPAD, поставляемые в составе примеров приложений с системой разработки Microsoft Developer Studio Visual C++. Совсем другая картина предстанет перед вами, когда вы возьмете исходные тексты такого же приложения, но созданного с использованием библиотеки MFC. Мы попытаемся обойти эти две крайние позиции и рассмотреть интерфейс MDI одновременно с деталями его внутренних механизмов, знание которых позволит вам создавать приложения любой сложности наиболее эффективно и с наименьшими затратами на поиск ошибок, которые зачастую возникают из-за непонимания происходящего.

Начнем с рассмотрения двух основных классов библиотеки MFC, разработанных специально для создания многодокументных приложений, — *CMDIFrameWnd* и *CMDIChildWnd*.

Класс CMDIFrameWnd



CMDIFrameWnd — базовый класс для главного окна любого многодокументного приложения. Если быть более точными, то в данном случае следует говорить не об окне, а о фрейме, поскольку помимо него создается специальное окно, называемое окном MDICLIENT, которое управляет рабочей областью фрейма и является, собственно, родительским для окон "MDI child".

Уточним некоторые моменты, которые связаны с процессом создания MDI-окна. Как обычно, сначала создается объект нашего оконного класса. Далее, если вы сравните тексты функции *InitInstance* для создания главных окон SDI- и MDI-приложений, то не увидите никакой разницы. И в том и в другом случае вызываются одни и те же функции. Однако, если при создании SDI-окна вызывается функция *CFrameWnd::OnCreateClient*, то при создании MDI-окна вызывается функция *CMDIFrameWnd::OnCreateClient*. Первая отвечает за создание (если необходимо) присоединяемого к окну представления, а вторая — за создание специального окна MDICLIENT, которое, как мы уже говорили, полностью управляет рабочей областью главного окна MDI-приложения и отвечает за работу с окнами "MDI child". На рис. 23 показаны все перечисленные окна MDI-приложения.

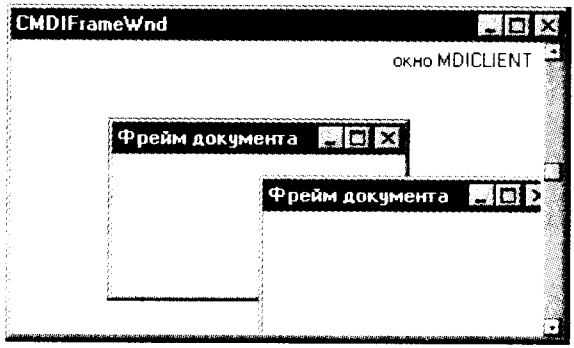


Рис. 23. Типы окон интерфейса MDI

Нам осталось рассмотреть функциональные возможности, которые класс *CMDIFrameWnd* предоставляет в наше распоряжение.

void CMDIFrameWnd::**MDIActivate**(CWnd *pWndActivate),
где pWndActivate — указатель на окно "MDI child", которое должно быть активизировано.

Эта функция посылает сообщение WM_MDIACTIVATE в два окна: сначала в то, которое активизируется, затем в то, которое теряет активность. То же самое сообщение посылается, если пользователь изменил фокус дочернего окна при помощи мыши или клавиатуры. Окна "MDI child" активизируются независимо от основного MDI-окна. Когда активизируется основное окно, то дочернее, которое было активно последним, посылает сообщение WM_NCACTIVATE, чтобы нарисовать его рамку и заголовок, но при этом не получает сообщения WM_MDIACTIVATE.

CMDIChildWnd* CMDIFrameWnd::**MDIGetActive**(
BOOL *pbMaximized = NULL) —
возвращает указатель на текущее активное окно "MDI child". Дополнительно, в параметре pbMaximized, возвращается информация о том, развернуто это окно или нет.

void CMDIFrameWnd::**MDIIconArrange**() —
упорядочивает все свернутые окна "MDI child", не затрагивая те, которые не свернуты.

void CMDIFrameWnd::**MDIMaximize**(CWnd *pWnd) —
разворачивает окно "MDI child". При этом Windows изменяет его размеры таким образом, чтобы оно заполнило всю рабочую область окна фрейма. Windows размещает системное меню дочернего окна в полосе основного меню, а его заголовок добавляется к заголовку окна фрейма. [1]

void CMDIFrameWnd::**MDINext**() — активизирует следующее окно "MDI child".

void CMDIFrameWnd::**MDIRestore**(CWnd *pWnd) —
восстанавливает исходные размеры окна "MDI child", заданного параметром pWnd, если оно было развернуто или свернуто.

CMenu* CMDIFrameWnd::**MDISetMenu**(
CMenu *pFrameMenu,
CMenu *pWindowMenu),
если pFrameMenu и/или pWindowMenu не равны NULL, то заменяет соответствующие текущие меню на переданные в качестве параметров. Возвращает временный указатель на замененное меню.

После вызова этой функции следует вызвать функцию *CWnd::DrawMenuBar* для обновления полосы меню. Если функция изменяет раскрывающееся меню, то элементы меню окна "MDI child" удаляются из предыдущего меню окна и добавляются в раскрывающееся меню. Если вы используете фрейм для управления окнами "MDI child", то не следует вызывать эту функцию.

---

[1] Если активное окно развернуто, и в этот момент активизируется другое окно "MDI child", то Windows его также разворачивает.

void CMDIFrameWnd::**MDITile**()
и
void CMDIFrameWnd::**MDITile**(int nType) —
располагают все окна "MDI child" без перекрытия. Первая версия функции располагает окна по вертикали, а вторая в качестве параметра *nType* может принимать одно из следующих значений:

MDITILE_HORIZONTAL	расположить окна по горизонтали
MDITILE_VERTICAL	расположить окна по вертикали
MDITILE_SKIPDISABLED	предотвращает отображение блокированных окон "MDI child"

void CMDIFrameWnd::**MDICascade**()
и
void CMDIFrameWnd::**MDICascade**(int nType) —
располагают все окна "MDI child" каскадом. Первая версия функции отображает и блокированные окна, а вторая, которая может быть вызвана только с параметром MDITILE_SKIPDISABLED, — все, кроме блокированных.

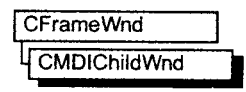
virtual BOOL CMDIFrameWnd::**CreateClient**(
LPCREATESTRUCT lpCreateStruct,
Cmenu *pWindowMenu) —
создает для текущего объекта CMDIFrameWnd окно Windows MDICLIENT, которое управляет объектами CMDIChildWnd. Эта функция должна быть вызвана, если вы переопределили функцию CWnd::OnCreate.[1]

virtual HMENU CMDIFrameWnd::**GetWindowMenuPopup**(
HMENU hMenuBar)
позволяет получить дескриптор раскрывающегося меню для элемента меню Window (Окно) основного меню, заданного параметром hMenuBar. Реализация заданная по умолчанию, ищет раскрывающееся меню, которое содержит стандартные команды элемента меню Window (Окно) основного меню типа ID_WINDOW_NEW и ID_WINDOW_TILE_HORZ. Если это раскрывающееся меню не содержит идентификаторов стандартных команд, то эту функцию следует переопределить.

Как видите, всю основную работу по поддержанию работы MDI-приложений выполняет Windows совместно с библиотекой классов MFC.

Прежде чем двигаться дальше, давайте так же внимательно рассмотрим класс, который отвечает за функционирование "MDI child" окон.

Класс CMDIChildWnd



Как вы уже поняли, основным назначением этого класса является обеспечение работоспособности дочерних окон MDI-приложений — окон "MDI child", которые, говоря упрощенно, позаимствовали свои свойства как у дочерних, так и у перекрывающихся окон.

---

[1] Если вы переопределяете эту функцию, то не забывайте вызывать функцию базового класса, т. к. фирма Microsoft не рекомендует производить какие-либо изменения, связанные с окном MDICLIENT.

Большую часть возможностей этот класс наследует от своего базового — *CFrameWnd,* но имеет и свойства, присущие только ему. К ним можно отнести автоматическое отображение меню окна "MDI child" вместо основного меню MDI-приложения, а также добавление заголовка дочернего окна к заголовку его родителя.

Для того чтобы произошла автоматическая замена основного меню на меню окна "MDI child", в классе определена единственная переменная

HMENU CMDIChildWnd::**m_hMenuShared**

Она содержит дескриптор меню, ассоциированного с окном "MDI child".

Сохранив в ней дескриптор меню дочернего окна, вы можете больше не заботиться о вопросах переключения между различными меню.

Кроме того, в классе определены несколько функций:

BOOL CMDIChildWnd::**Create**(
  LPCTSTR lpszClassName,
  LPCTSTR lpszWindowName,
  DWORD dwStyle = WS_CHILD|WS_VISIBLE|WS_OVERLAPPEDWINDOW,
  const RECT &rect = rectDefault,
  CMDIFrameWnd *pParentWnd = NULL,
  CCreateContext *pContext = NULL) —
эта функция абсолютно идентична функции CFrameWnd::Create, однако задачи, которые они решают, достаточно сильно различаются, и нельзя подменять вызов одной из них вызовом другой. Как уже упоминалось, активное текущее окно "MDI child" может определять заголовок своего родительского окна, если для него установить бит стиля FWS_ADDTOTITLE. Эта функция вызывается MDI-окном в ответ на команду пользователя создать дочернее окно. При этом параметр pContext позволяет правильно связать дочернее окно с приложением. Если такой необходимости нет, то для него можно использовать значение по умолчанию (NULL).

void CMDIChildWnd::**MDIDestroy**() —
удаляет заголовок окна "MDI child" из MDI-окна и разрушает само окно "MDI child".

void CMDIChildWnd::**MDIActivate**() —
активизирует текущее окно "MDI child". Когда MDI-окно становится активным, то активизируется и окно "MDI child", которое находилось в этом состоянии последним.

void CMDIChildWnd::**MDIMaximize**() —
разворачивает текущее окно "MDI child". Если у него не сброшен бит стиля FWS_ADDTOTITLE, то его заголовок добавляется в MDI-окно.

void CMDIChildWnd::**MDIRestore**() —
восстанавливает исходные размеры текущего окна "MDI child", если оно было развернуто или свернуто.

CMDIFrameWnd* CMDIChildWnd::**GetMDIFrame**() —
позволяет получить указатель на родительское MDI-окно объекта класса CMDIFrameWnd. Для того чтобы получить указатель на родительское окно типа MDICLIENT, которое управляет объектом класса CMDIChildWnd, необходимо вызвать функцию CWnd::GetParent.

Теперь, когда мы познакомились с обоими классами, позволяющими создавать приложения на базе интерфейса MDI, пришло время рассмотреть конкретный пример, который позволит закрепить полученные сведения и ознакомиться с дополнительными возможностями библиотеки MFC.

MDI — пример приложения

В этом разделе представлен текст приложения, на примере которого мы рассмотрим основные принципы построения MDI-приложений. Оно может послужить основой для разработки многодокументных приложений любой степени сложности. Кроме того, будут рассмотрены некоторые предварительные вопросы обработки команд меню и работы со шрифтами.

Как и в примере, приведенном в предыдущей главе, все англоязычные комментарии вставлены в текст приложения средством *AppWizard* или *ClassWizard*, и оставлены без изменения.

```
/////////////////////////////////////////////////////////////
// Файл MDI.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

// MDI.h : main header file for the MDI application
//

#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////
// CMDIApp:
// See MDI.cpp for the implementation of this class
//

class CMDIApp : public CWinApp
{
public:
  CMDIApp();

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMDIApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL

// Implementation
  //{{AFX_MSG(CMDIApp)
  afx_msg void OnAppAbout();
```

```
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл MDI.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// MDI.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "MDI.h"
#include "AppAbout.h"
#include "MainFrm.h"
/////////////////////////////////////////////////////////////
// CMDIApp
BEGIN_MESSAGE_MAP(CMDIApp, CWinApp)
  //{{AFX_MSG_MAP(CMDIApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////
// CMDIApp construction
CMDIApp::CMDIApp()
{
}
/////////////////////////////////////////////////////////////
// The one and only CMDIApp object
CMDIApp theApp;
/////////////////////////////////////////////////////////////
// CMDIApp initialization
BOOL CMDIApp::InitInstance()
{
// Standard initialization
  CString csCaption;
// Загружаем строковый ресурс с именем приложения
  csCaption.LoadString(IDR_MAINFRAME);
// Проверяем, не было ли приложение уже загружено
  CWnd *pWnd = CWnd::FindWindow(NULL, csCaption);
  if(pWnd != NULL){
  // Если да, то делаем его видимым и переводим на передний план
  if(pWnd->IsIconic())
    pWnd->ShowWindow(SW_RESTORE);
```

```
  pWnd->SetForegroundWindow();
  return FALSE;
  }

  Enable3dControls();   // Call this when using MFC in a shared DLL

  // create main MDI Frame window
  CMainFrame* pMainFrame = new CMainFrame;
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
  return FALSE;
  m_pMainWnd = pMainFrame;

  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();

  return TRUE;
}

// App command to run the dialog
void CMDIApp::OnAppAbout()
{
  CAppAbout *aboutDlg = new CAppAbout();
}

/////////////////////////////////////////////////////////////////
// Файл AppAbout.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// CAppAbout dialog

class CAppAbout : public CDialog
{
// Construction
public:
  CAppAbout(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
  //{{AFX_DATA(CAppAbout)
  enum { IDD = IDD_ABOUTBOX };
  // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAppAbout)
  //}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
  //{{AFX_MSG(CAppAbout)
  // NOTE: the ClassWizard will add member functions here
```

```
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////
// Файл AppAbout.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////
// AppAbout.cpp : implementation file
//
#include "stdafx.h"
#include "MDI.h"
#include "AppAbout.h"
/////////////////////////////////////////////////////////////////
// CAppAbout dialog
ppAbout::CAppAbout(CWnd* pParent /*=NULL*/)
{
  Create(CAppAbout::IDD, pParent);
  //{{AFX_DATA_INIT(CAppAbout)
  // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
}
BEGIN_MESSAGE_MAP(CAppAbout, CDialog)
  //{{AFX_MSG_MAP(CAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////
// MainFrm.cpp : implementation of the CMainFrame class
//
#include "stdafx.h"
#include "MDI.h"
#include "MDIChilds.h"
#include "MainFrm.h"
/////////////////////////////////////////////////////////////////
// CMainFrame
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_COMMAND(ID_FILE_TEXT, OnText)
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

LPCTSTR lpszTextClass;

//////////////////////////////////////////////////////////
// CMainFrame message handlers

// Обработчик команды ID_FILE_TEXT
void CMainFrame::OnText()
{
  // Создаем объект класса для дочернего окна "MDI child"
  CTextWnd *pTextWnd = new CTextWnd;

  // Создает окно Windows "MDI child"
  if (!pTextWnd->Create(lpszTextClass,
                        _TEXT("Дочернее окно \"MDI child\""),
                        0, rectDefault, this))
  return;
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
  return -1;

  // Регистрируем новый класс окна Windows для "MDI child"
  lpszTextClass = AfxRegisterWndClass(
        CS_HREDRAW | CS_VREDRAW,
        AfxGetApp()->LoadCursor(IDC_MDITEXT),
        (HBRUSH) (COLOR_WINDOW+1),
        AfxGetApp()->LoadIcon(IDI_MDITEXT));

  // Вызываем обработчик команды ID_FILE_TEXT
  OnText();

  return 0;
}

//////////////////////////////////////////////////////////
// Файл MDIChilds.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////

// MDIChilds.h : header file
//

//////////////////////////////////////////////////////////
// CTextWnd frame

class CTextWnd : public CMDIChildWnd
{
public:
  CTextWnd();
```

```
// Implementation
protected:
  static CMenu    menu;      // меню для всех дочерних окон
  UINT     nIDColor;
  COLORREF clrText;

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CTextWnd)
  protected:
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  //}}AFX_VIRTUAL

// Generated message map functions
  //{{AFX_MSG(CTextWnd)
  afx_msg void OnColor();
  afx_msg void OnPaint();
  afx_msg void OnUpdateColor(CCmdUI* pCmdUI);
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл MDIChilds.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

// MDIChilds.cpp : implementation file
//

#include "stdafx.h"
#include "MDI.h"
#include "MDIChilds.h"

CMenu CTextWnd::menu;      // меню для всех дочерних окон

COLORREF colorArray[] =
{
  RGB (0, 0, 0),
  RGB (255, 255, 255),
  RGB (255, 0, 0),
  RGB (0, 0, 255),
  RGB (0, 255, 0)
};

BEGIN_MESSAGE_MAP(CTextWnd, CMDIChildWnd)
  //{{AFX_MSG_MAP(CTextWnd)
  ON_COMMAND(IDM_BLACK, OnColor)
  ON_COMMAND(IDM_BLUE, OnColor)
  ON_COMMAND(IDM_GREEN, OnColor)
  ON_COMMAND(IDM_RED, OnColor)
```

```
  ON_COMMAND(IDM_WHITE, OnColor)
  ON_UPDATE_COMMAND_UI(IDM_BLACK, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_BLUE, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_GREEN, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_RED, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_WHITE, OnUpdateColor)
  ON_WM_PAINT()
  ON_WM_LBUTTONDOWN()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////
// CTextWnd
CTextWnd::CTextWnd()
{
  nIDColor = IDM_BLACK;
  clrText = RGB(0, 0, 0);
}
////////////////////////////////////////////////////////////
// CTextWnd message handlers
BOOL CTextWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= WS_CHILD | WS_VISIBLE |
              WS_OVERLAPPEDWINDOW;
  // Загружаем меню дочернего окна
  if (menu.m_hMenu == NULL)
  menu.LoadMenu(IDR_MDITEXT);
  m_hMenuShared = menu.m_hMenu;

  return CMDIChildWnd::PreCreateWindow(cs);
}
// Обработчик сообщения WM_PAINT
void CTextWnd::OnPaint()
{
  CPaintDC dc(this);
  CRect rect;
  GetClientRect(rect);
  // Устанавливаем текущие цвета текста и фона
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  // Отображаем текст в центре окна
  dc.DrawText(_TEXT("Обработка сообщения WM_PAINT"), -1, rect,
              DT_SINGLELINE | DT_CENTER | DT_VCENTER);
}
```

```
// Обработчик команд IDM_BLACK - IDM_WHITE (один на все)
void CTextWnd::OnColor()
{
  // Получаем идентификатор выбранной команды
  nIDColor = LOWORD(GetCurrentMessage()->wParam);

  // По идентификатору команды выбираем из массива
  // соответствующее значение цвета
  clrText = colorArray[nIDColor - IDM_BLACK];

  // Перерисовать окно в ответ на сообщение WM_PAINT
  Invalidate();
}

// Обработчик команд обновления IDM_BLACK - IDM_WHITE (один на все)
void CTextWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  // Устанавливаем или убираем галочку у соответствующего элемента меню
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}

// Обработчик сообщения WM_LBUTTONDOWN
void CTextWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
  TCHAR strText[] = "Обработка сообщения WM_LBUTTONDOWN";

  CClientDC dc(this);
  CRect rect;

  // Устанавливаем текущие цвета текста и фона
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));

  // Отображаем текст в окне, начиная с точки (10, 10)
  dc.TextOut(10, 10, strText, strlen(strText));

  // Не забываем передать управление родительскому
  // окну для дальнейшей обработки
  CMDIChildWnd::OnLButtonDown(nFlags, point);
}

//////////////////////////////////////////////////////////
// Файл resource.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by MDI.rc
#define IDD_ABOUTBOX                    100
#define IDM_BLACK                       101
#define IDM_WHITE                       102
#define IDM_RED                         103
```

```
#define IDM_BLUE                        104
#define IDM_GREEN                       105
#define IDR_MAINFRAME                   128
#define IDR_MDITYPE                     129
#define IDR_MDITEXT                     129
#define IDI_MDITEXT                     130
#define ID_FILE_TEXT                    32771

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_3D_CONTROLS                1
#define _APS_NEXT_RESOURCE_VALUE        131
#define _APS_NEXT_COMMAND_VALUE         32787
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           106
#endif
#endif
////////////////////////////////////////////////////////////
// Файл MDI.rc
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.

  #include "afxres.h"
#undef APSTUDIO_READONLY_SYMBOLS

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

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

#ifdef APSTUDIO_INVOKED

// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END
```

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

3 TEXTINCLUDE DISCARDABLE
BEGIN
  "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
  "#define _AFX_NO_OLE_RESOURCES\r\n"
  "#define _AFX_NO_TRACKER_RESOURCES\r\n"
  "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
  "\r\n"
  "#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)\r\n"
  "#ifdef _WIN32\r\n"
  "LANGUAGE 9, 1\r\n"
  "#pragma code_page(1252)\r\n"
  "#endif\r\n"
  "#include ""res\\MDI.rc2""  // non-Microsoft Visual C++ edited
                              resources\r\n"
    "#include ""afxres.rc""      // Standard components\r\n"
    "#endif\0"
END

#endif    // APSTUDIO_INVOKED

////////////////////////////////////////////////////////////
// Icon

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\MDI.ico"
IDI_MDITEXT             ICON    DISCARDABLE     "res\\mditext.ico"

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

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "Файл"
  BEGIN
    MENUITEM "Новое окно\t",                ID_FILE_TEXT
    MENUITEM SEPARATOR
    MENUITEM "Выход",                       ID_APP_EXIT
  END
    POPUP "?"
    BEGIN
    MENUITEM "О программе ...",             ID_APP_ABOUT
  END
END
```

```
IDR_MDITEXT MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "Файл"
    BEGIN
     MENUITEM "Новое окно",                    ID_FILE_TEXT
     MENUITEM SEPARATOR
     MENUITEM "Выход",                         ID_APP_EXIT
  END
  POPUP "Цвет"
  BEGIN
    MENUITEM "Черный",                         IDM_BLACK
    MENUITEM "Белый",                          IDM_WHITE
    MENUITEM "Красный",                        IDM_RED
    MENUITEM "Синий",                          IDM_BLUE
    MENUITEM "Зеленый",                        IDM_GREEN
  END
  POPUP "Окно"
  BEGIN
    MENUITEM "Каскадом",                       ID_WINDOW_CASCADE
    MENUITEM "По горизонтали",                 ID_WINDOW_TILE_HORZ
    MENUITEM "Упорядочить значки",             ID_WINDOW_ARRANGE
  END
  POPUP "?"
  BEGIN
    MENUITEM "&О программе ...",               ID_APP_ABOUT
  END
END

////////////////////////////////////////////////////////////
// Dialog

IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "About MDI"
FONT 8, "MS Sans Serif"
BEGIN
  ICON            IDR_MAINFRAME,IDC_STATIC,7,7,18,20
  CTEXT           "MDI-приложение Версия 1.0",IDC_STATIC,51,17,119,8,
                  SS_NOPREFIX
  CTEXT           "Copyright © 1997",IDC_STATIC,127,34,71,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
  LTEXT           "Мешков А., Тихомиров Ю.",IDC_STATIC,35,34,89,8
END

#ifndef _MAC

////////////////////////////////////////////////////////////
// Version

VS_VERSION_INFO VERSIONINFO
  FILEVERSION 1,0,0,1
```

```
  PRODUCTVERSION 1,0,0,1
  FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
  FILEFLAGS 0x1L
#else
  FILEFLAGS 0x0L
#endif
  FILEOS 0x4L
  FILETYPE 0x1L
  FILESUBTYPE 0x0L
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "041904e3"
    BEGIN
      VALUE "Comments", "Пример создания MDI-приложения\0"
      VALUE "CompanyName", "Pragma Design Ltd.\0"
      VALUE "FileDescription", "MDI-приложение на базе MFC\0"
      VALUE "FileVersion", "1, 0, 0, 1\0"
      VALUE "InternalName", "MDI\0"
      VALUE "LegalCopyright", "Copyright © 1997\0"
      VALUE "OriginalFilename", "MDI.EXE\0"
      VALUE "ProductName", "MDI-приложение\0"
      VALUE "ProductVersion", "1, 0, 0, 1\0"
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x419, 1251
  END
END

#endif    // !_MAC

/////////////////////////////////////////////////////////////
// DESIGNINFO

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
  IDD_ABOUTBOX, DIALOG
  BEGIN
    LEFTMARGIN, 7
    RIGHTMARGIN, 210
    TOPMARGIN, 7
    BOTTOMMARGIN, 48
  END
END
#endif    //
```

```
APSTUDIO_INVOKED//////////////////////////////////////////

// Cursor

IDC_MDITEXT             CURSOR  DISCARDABLE     "res\\text.cur"

//////////////////////////////////////////////////////////////
// String Table

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME         "Главное окно MDI-приложения"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_APP_ABOUT          "Вывод информации о программе и ее версии\nO"
  ID_APP_EXIT           "Завершение работы приложения"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_NEXT_PANE          "Переход к следующему окну"
  ID_PREV_PANE          "Переход к предыдущему окну"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_WINDOW_ARRANGE     "Упорядочить значки"
  ID_WINDOW_CASCADE     "Расположить окна с перекрытием"
  ID_WINDOW_TILE_HORZ   "Расположить окна без перекрытия"
END

STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_TEXT          "Открыть новое окно"
END

#endif    // Russian resources

#ifndef APSTUDIO_INVOKED

//////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.

#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE 9, 1
#pragma code_page(1252)
```

```
#endif
#include "res\MDI.rc2"  // non-Microsoft Visual C++ edited resources
#include "afxres.rc"         // Standard components
#endif
#endif    // not APSTUDIO_INVOKED
```

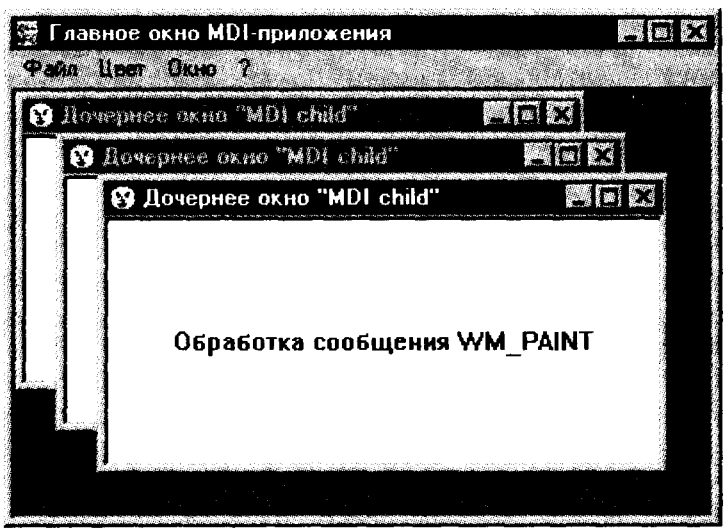


Рис. 24. Главное и дочерние окна MDI-приложения

Первое, что сразу бросается в глаза при рассмотрении программы, это практически полная идентичность интерфейса и реализации нашего класса "приложение" (*CMDIApp*) и *CFirstApp*, описанного выше. А если не считать появления специального обработчика команды, который мы рассмотрим дальше в этой главе, и вызова функции *CWnd::FindWindow*, то совпадение будет полным. На самом деле это совсем не случайно. Достаточно большая часть программ для Windows создается с использованием метода "клея и ножниц", т. е. взяв за основу некоторый шаблон (а такие заготовки имеются у всех, даже начинающих, программистов) вы выполняете несколько маленьких изменений и потом забываете об этом.[1]

После этого небольшого отступления рассмотрим фрагмент кода:

```
CString csCaption;
csCaption.LoadString(IDR_MAINFRAME);

CWnd *pWnd = CWnd::FindWindow(NULL, csCaption);
```

[1] В следующих наших примерах мы больше не будем приводить полные листинги программ, а ограничимся обсуждением только тех изменений, которые появились в программе по сравнению с имеющейся заготовкой.

```
if(pWnd != NULL)
{
  if(pWnd->IsIconic())
    pWnd->ShowWindow(SW_RESTORE);
  pWnd->SetForegroundWindow();
  return FALSE;
}
```

Не будем пока акцентировать внимание на новом для нас классе *CString*, а ограничимся только замечанием, что в объекте этого класса хранится символьная строка с именем окна, загруженная из файла ресурсов.

Мы вставили этот код для того, чтобы не загружать наше приложение еще раз, если оно уже было запущено. Действительно, MDI-приложения позволяют работать с несколькими документами одновременно (в отличие от SDI-приложений), и нет никакой необходимости расходовать дополнительную память. Как мы уже говорили, для проверки того, не было ли приложение уже загружено, мы не можем воспользоваться параметром *hInstance*, передаваемом в командной строке, т. к. для приложений Win32 он всегда равен NULL. Поэтому мы воспользовались функцией *CWnd::FindWindow*, которая позволяет найти окно верхнего уровня по имени оконного класса или по заголовку окна (ее нельзя использовать для поиска дочерних окон):

static CWnd* CWnd::**FindWindow**(
  LPCTSTR lpszClassName,
  LPCTSTR lpszWindowName) –
    через параметр lpszClassName передается указатель на символьную строку, которая определяет имя класса искомого окна. Если этот параметр равен NULL, как в нашем случае, то поиск будет осуществляться во всех классах.[1] Поскольку нам надо найти вполне определенное окно с заданным заголовком, то имя заголовка мы передаем через второй параметр lpszWindowName, который в общем случае также может быть равен NULL.

Если окно найдено, то функция *FindWindow* вернет временный указатель на этот объект класса *CWnd*, в противном случае – NULL.

После того как главное окно найдено, нужно проверить, не было ли оно свернуто. Для этого используется специальная функция:

BOOL CWnd::**IsIconic**() – возвращает TRUE, если окно свернуто.

В этом случае восстанавливаем его нормальный размер с помощью уже известной вам функции *CWnd::ShowWindow*.

---

[1] Если назначение имени класса мы отдаем библиотеке классов MFC, то параметром *lpszClassName* пользоваться просто невозможно, т.к. имя класса генерируется каждый раз новое и имеет примерно такой вид: "Afx:400000:8:1526:0:13F7" (при следующем запуске, даже второй копии приложения, класс будет иметь имя "Afx:400000:8:1526:0:251F" — почувствуйте разницу!).

Теперь осталось только поместить окно на передний план и активизировать.[1] Это легко сделать при помощи функции, которая предназначена именно для подобных целей:

BOOL CWnd::**SetForegroundWindow**() –
переводит поток, который создал окно, в приоритетный режим и активизирует его главное окно.

Ее дополняет функция

static CWnd* CWnd::**GetForegroundWindow**() –
возвращает указатель на временный объект приоритетного окна (окна, с которым пользователь в текущий момент работает).

Для того чтобы картина была полной, приведем еще одну функцию, которая позволяет изменить Z-порядок окон:

void CWnd::**BringWindowToTop**() –
позволяет вывести на передний план окна верхнего уровня, всплывающие или окна "MDI child". По своему действию она похожа на функцию CWnd::SetWindowpPos, однако при этом не позволяет изменять стили окна.

Возвращаясь к нашему примеру, по этому поводу осталось сказать, что если была найдена работающая копия нашего приложения, то нам остается только завершить работу текущей копии. Если же работающая копия не найдена, то переопределенная функция *InitInstance* приступает к инициализации приложения. Этот процесс уже был подробно описан нами, и мы не будем к нему возвращаться.

Займемся лучше используемыми классами окон. Прежде всего, обратим внимание на базовый класс нашего главного окна:

```
class CMainFrame : public CMDIFrameWnd
{
  ...
};
```

Как видите, он образован из специально спроектированного для окон MDI класса. При этом действия по созданию главного окна MDI-приложения точно такие же, как и при создании любого другого окна — создание объекта "окно", регистрация оконного класса и создание окна Windows. Чтобы не повторяться, коснемся только одного момента. Строгих правил, определяющих место, где следует регистрировать оконный класс, нет. Как правило, большие классы регистрируют в отдельном модуле, а маленькие — в той же самой функции, что и основное окно. Однако это дело вкуса, и вы вправе разрабатывать или перенимать любой стиль, который вас устраивает. Поскольку мы собираемся работать со многими окнами "MDI child" одновременно, и, в данном случае, все они принадлежат одному оконному

[1] Этот процесс мы описывали в части II книги, но только для функций Win32 API.

классу, то в качестве места регистрации их оконного класса мы выбрали обработчик сообщения WM_CREATE главного окна приложения:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct){
 if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
  return -1;
  lpszTextClass = AfxRegisterWndClass(
    CS_HREDRAW | CS_VREDRAW,
    AfxGetApp()->LoadCursor(IDC_MDITEXT),
    (HBRUSH) (COLOR_WINDOW+1),
    AfxGetApp()->LoadIcon(IDI_MDITEXT));
  OnText();
  return 0;
}
```

Помимо регистрации оконного класса для окон "MDI child", в этот обработчик мы включили явный вызов обработчика команды *OnText* для создания и вывода на экран первого дочернего окна MDI-приложения. Об этом чуть позже, а пока обратим ваше внимание на то, что нам не надо явно создавать специальное окно MDICLIENT, т. к. оно автоматически создается функциями библиотеки. Вы, конечно, можете вмешаться в этот процесс, переопределив функцию класса *CMDIFrameWnd — CreateClient*. Однако еще раз подчеркнем: чтобы не нарушать правильную работу составных частей вашего приложения, не забывайте в таких случаях вызывать переопределенную функцию базового класса.

В этом месте необходимо сделать небольшое отступление и рассмотреть вопросы создания и обработки команд меню.[1]

Обработка команд меню

Код, создающий основное меню для нашей программы достаточно прост:

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "Файл"
  BEGIN
    MENUITEM "Новое окно\t",                ID_FILE_TEXT
    MENUITEM SEPARATOR
    MENUITEM "Выход",                       ID_APP_EXIT
  END
  POPUP "?"
  BEGIN
    MENUITEM "О программе ...",             ID_APP_ABOUT
  END
END
```

[1] Более подробно меню и другие ресурсы мы рассмотрим в отдельной главе.

Прежде всего, мы определяем идентификатор меню — IDR_MAINFRAME, который передаем в функцию *LoadFrame* для регистрации оконного класса и создания окна Windows. Само тело меню заключено между операторными скобками BEGIN и END. Как видно из текста, элементы меню бывают двух видов: POPUP — это целое окно (например, раскрывающийся перечень, который появляется, когда вы открываете меню File (Файл) во многих приложениях); MENUITEM — это единичный элемент меню, после объявления которого в кавычках записана символьная строка, изображаемая в строке меню. И, наконец, объявляется константа, которая ассоциирована с данным пунктом меню. Обратите внимание, что *каждый* пункт меню ассоциируется с некоторой константой. Именно они и представляют для нас сейчас основной интерес. Дело в том, что когда пользователь выбирает конкретный пункт меню, соответствующая константа и пункт меню упаковываются в параметры сообщения WM_COMMAND и посылаются в соответствующую оконную процедуру. Связав с этой константой определенный обработчик сообщений, мы можем выполнить любые необходимые нам действия. Например, для того чтобы создать новое окно "MDI child" и вывести его на экран, мы создали обработчик сообщения *OnText* и связали его посредством карты сообщений с командой меню ID_FILE_TEXT, аналогично тому, как были связаны между собой сообщение WM_CREATE и обработчик *OnCreate*:

```
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg void OnText();
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()

...

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_COMMAND(ID_FILE_TEXT, OnText)
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

...

void CMainFrame::OnText()
{
  CTextWnd *pTextWnd = new CTextWnd;
  if (!pTextWnd->Create(lpszTextClass,
                        _TEXT("Дочернее окно \"MDI child\""),
                        0, rectDefault, this))
  return;
}
```

Непосредственный процесс создания окна вам уже хорошо знаком. Поэтому обратим ваше внимание только на следующую строку:

```
_TEXT("Дочернее окно \"MDI child\"").
```

Интерес здесь представляет макрос _TEXT, содержащийся во включаемом файле TCHAR.H. Чтобы ответить на вопрос "для чего он нужен?", сделаем еще одно небольшое отступление.

UNICODE

UNICODE — стандарт, разработанный в 1988 году фирмами Apple и Xerox. Его полное описание содержится в *The Unicode Standard: Worldwide Character Encoding, Version 1.0.* Нас же будет интересовать общее понимание того, что же такое Unicode.

Поскольку система Windows становится все популярнее, то разработчикам приложений для нее надо ориентироваться на международный рынок. Естественно, что при этом встает вопрос о поддержке самых разных языков. А ведь существуют и такие, что возможностей одного байта, позволяющего кодировать не более 256 символов, недостаточно для представления всех букв этого языка. Для поддержки подобных языков были созданы двухбайтовые наборы символов (double-byte character sets, DBCS). Однако работать с ними оказалось очень неудобно, т. к. для представления одних символов достаточно одного байта, а для других необходимо два.

В отличие от DBCS, всем символам в Unicode требуется ровно два байта, что позволяет кодировать около 65000 символов — более чем достаточно для работы с любым языком. В настоящее время *кодовые позиции* (code points) определены для арабского, китайского, русского (кириллица), греческого, еврейского, латинского (английского), японской каны и некоторых других алфавитов. Кроме того, в набор символов включено большое количество знаков препинания, математических, технических, диакритических и других символов.

И хотя пока Unicode поддерживается только в системе Windows NT и не поддерживается в Windows 95, для того чтобы упростить задачу локализации программного продукта в будущем, мы советуем вам уже сейчас включать поддержку Unicode в ваши программные проекты. Тем более, что это совсем несложно, ведь Microsoft разработала Win32 API для Unicode так, чтобы как можно меньше повлиять на ваш код. Для переключения на Unicode надо всего лишь определить два макроса и перекомпилировать исходный текст. Естественно, это возможно при условии, что вы вставили в вашу программу файл заголовков TCHAR.H, и используете макросы из этого файла вместо стандартных функций для работы со строками, такими, например, как *strlen*. Полный перечень макросов вы найдете в файле TCHAR.H. Но это еще не все. Для задания массива символов ANSI/Unicode следует использовать тип

данных TCHAR. При этом, если определен тип _UNICODE, то TCHAR объявляется как:

```
typedef wchar_t TCHAR;
```

а если не определен, то как:

```
typedef char TCHAR;
```

При использовании этого типа данных строка символов объявляется как:

```
TCHAR szString[128];
```

Кроме этого, для корректной работы со строками необходимо использовать макрос _TEXT (или _T), который определен как:

```
#define _TEXT(x) L##x      // для Unicode
```

и

```
#define _TEXT(x) x           // для ANSI
```

что мы и продемонстрировали в нашей программе.

В заключение просто перечислим шаги, которые необходимо проделать, чтобы подготовить приложение к использованию Unicode:

- задавайте текстовые строки массивами символов, а не байтов (char);
- используйте настраиваемые типы данных (вроде TCHAR или LPTSTR) для символов и текстовых строк;
- используйте макрос _TEXT для задания символьных или строковых литералов;
- для избежания каких-либо проблем в строковой арифметике используйте выражение *(sizeof(szBuffer) / (sizeof(TCHAR))* вместо *sizeof(szBuffer)*.

Вот, пожалуй, и все, что мы пока хотели сказать о Unicode и можно возвращаться к нашему примеру.

Как мы уже отмечали, для того чтобы подключить меню окна "MDI child" к основному окну MDI-приложения, мы определили член *menu* класса *CTextWnd*:

```
class CTextWnd : public CMDIChildWnd
{
...
protected:
  static CMenu menu;  // меню для всех окон "MDI child"
...
};
```

Теперь остается только присвоить идентификатор меню окна "MDI child" специальному члену класса:

```
BOOL CTextWnd::PreCreateWindow(CREATESTRUCT& cs)
{
...
  if (menu.m_hMenu == NULL)
    menu.LoadMenu(IDR_MDITEXT);
  m_hMenuShared = menu.m_hMenu;
...
}
```

Все остальные заботы по отслеживанию команд меню взяла на себя операционная система. Нам осталось только правильно их обработать. Если вы внимательно посмотрите на карту сообщений класса *CTextWnd*, например, для меню "Цвет":

```
BEGIN_MESSAGE_MAP(CTextWnd, CMDIChildWnd)
  //{{AFX_MSG_MAP(CTextWnd)
...
  ON_COMMAND(IDM_BLACK, OnColor)
  ON_COMMAND(IDM_BLUE, OnColor)
  ON_COMMAND(IDM_GREEN, OnColor)
  ON_COMMAND(IDM_RED, OnColor)
  ON_COMMAND(IDM_WHITE, OnColor)
  ON_UPDATE_COMMAND_UI(IDM_BLACK, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_BLUE, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_GREEN, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_RED, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_WHITE, OnUpdateColor)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

то увидите, что для всех пяти команд меню мы использовали один и тот же обработчик сообщений — *OnColor*, возложив на него задачу идентификации этих команд: [1]

```
void CTextWnd::OnColor()
{
  nIDColor = LOWORD(GetCurrentMessage()->wParam);
  clrText = colorArray[nIDColor - IDM_BLACK];
  Invalidate();
}
```

---

[1] Другой способ использования одного обработчика для нескольких команд, основанный на использовании макроса ON_COMMAND_RANGE, представлен в примере главы 15.

Использованная в этом коде функция *CWnd::GetCurrentMessage* возвращает указатель на структуру *MSG*, в слове *wParam* которой содержится идентификатор выбранного пункта меню (идентификатор команды). Из файла RESOURCE.H

```
...
#define IDM_BLACK                       101
#define IDM_WHITE                       102
#define IDM_RED                         103
#define IDM_BLUE                        104
#define IDM_GREEN                       105
...
```

видно, что используемые в данном фрагменте идентификаторы команд идут один за другим. Это не случайно и позволяет нам использовать их для определения индекса в массиве цветов (*nIDColor — IDM_BLACK*). Попробуйте изменить числовое значение у какого-либо идентификатора — программа сразу же перестанет адекватно реагировать на эту команду.

Последней в этом обработчике вызывается функция *CWnd::Invalidate*. Нам представляется важным познакомиться с ней поближе:

void CWnd::**Invalidate**(BOOL bErase = TRUE) —

эта функция определяет в качестве области модификации всю рабочую область окна как требующую перерисовки.[1] В результате приложению будет послано сообщение WM_PAINT. Это сообщение Windows посылает всякий раз, когда область модификации окна не пуста, и для этого окна нет других сообщений в очереди. Параметр, передаваемый в функцию, определяет необходимость стирания фона окна перед перерисовкой. Если он имеет значение TRUE, фон стирается, если FALSE — нет.

И хотя мы вплотную подошли к вопросу обработки сообщения WM_PAINT, задержимся еще для того, чтобы рассмотреть принцип действия специальной команды обновления и связанный с ней класс *CCmdUI*.

Вернемся к нашей программе. Нашей задачей было поставить галочку у того элемента меню, который является выбранным на текущий момент времени, что мы и сделали, воспользовавшись функцией *CCmdUI::SetCheck* в обработчике *OnUpdateColor*. Поскольку этот обработчик каждый раз вызывается для всех элементов раскрывающегося меню, то перед перерисовкой галочка будет установлена только у того, который был перед этим выбран (*pCmdUI->m_nID == nIDColor*), а у остальных — убрана:

```
void CTextWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}
```

---

[1] Для того чтобы отметить не всю рабочую область, а какую-либо ее часть, в классе *CWnd* реализованы специальные функции *InvalidateRect* и *InvalidateRgn*, первая из которых "помечает" прямоугольную область, а вторая - произвольную.

Обработка сообщения WM_PAINT

Сообщение, к рассмотрению которого мы приступаем, является одним из наиболее важных, если не самым важным, поскольку приложение, как правило, выполняет всю работу по рисованию именно во время обработки этого сообщения. Оно посылается в окно всякий раз, когда требуется изменить содержимое клиентской области, т. е. того пространства окна, о котором программист заботится самостоятельно, в то время как Windows обслуживает остальные его части. Например, если окно было "спрятано" под другим окном и выводится на передний план, то оно получает сообщение WM_PAINT, т. к. необходимо перерисовать область скрытую под другим окном.

Рассмотрим обработчик этого сообщения, реализованный в нашей программе:

```
void CTextWnd::OnPaint()
{
  CPaintDC dc(this);

  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));

  CRect rect;
  GetClientRect(rect);

  dc.DrawText(_TEXT("Обработка сообщения WM_PAINT"), -1, rect,
    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
}
```

Как вы помните, любой вывод в программах, написанных для Windows, осуществляется в контекст устройства, который позволяет абстрагироваться от конкретного типа внешнего устройства. Здесь еще не место и не время подробно рассматривать поддержку контекстов устройств, имеющуюся в библиотеке MFC, — этому вопросу будет посвящен отдельный большой раздел. Однако, ввиду важности сообщения WM_PAINT, и приняв в качестве постулата необходимость наличия специального контекста для окна, рассмотрим обработчик этого сообщения достаточно подробно.

Первая же строка обработчика содержит код для создания объекта класса *CPaintDC*, который предоставляет контекст дисплея для обработки сообщения WM_PAINT.

Конструктор этого объекта в качестве параметра получает указатель на оконный объект, для которого запрашивается контекст, автоматически вызывает функцию *CWnd::BeginPaint*, обеспечивающую доступ к нему, и сохраняет структуру *PAINTSTRUCT* в специальной переменной *m_ps*. После окончания работы с контекстом устройства его необходимо вернуть в систему, чтобы избежать истощения ресурсов. Для этого в деструкторе автоматически вызывается функция *CWnd::EndPaint*. Между этими двумя процедурами располагается сердцевина обработчика сообщения WM_PAINT.

Наш обработчик демонстрирует лишь один из основных способов вывода текста в Windows-программах. Он основан на использовании функции *DrawText*:

virtual int CDC::**DrawText**(
  LPCTSTR lpszStr,
  int nCount,
  LPRECT lpRect,
  UINT nFormat),

где *lpszStr* — указатель на выводимую строку, *nCount* — длина строки (может быть равна -1 для ASCIIZ-строк, т. е. строк заведомо оканчивающихся нуль-символом), *lpRect* — область, внутри которой выводится строка, *nFormat* — задает метод форматирования текста.

При успешном завершении функция возвращает высоту текста. В нашем примере для форматирования мы использовали центрирование по вертикали и горизонтали, а также вывод всего текста одной строкой. Эта функция использует контекст устройства, чтобы установить цвета для фона и текста. Для того чтобы задать свои значения этих параметров, мы воспользовались специальными функциями *SetTextColor* и *SetBkColor*, которые будут рассмотрены при подробном обсуждении графического интерфейса.

В нашем обработчике осталась еще одна функция. И поскольку она не имеет отношения к контексту устройств, мы опишем ее подробно:

void CWnd::**GetClientRect**(LPRECT lpRect) —

эта функция заполняет структуру (или объект) *RECT*, на которую указывает параметр *lpRect*, размерами рабочей области окна. При этом параметры *top* и *left* будут равны нулю, а *right* и *bottom* определяют ширину и высоту рабочей области.

Сама структура объявлена в файле WINDEF.H:

```
typedef struct tagRECT{
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT;
```

и определяет координаты левого верхнего (*left* — x-координата, *top* — y-координата) и правого нижнего (*right* — x-координата, *bottom* — y-координата) углов некоторого прямоугольника.

Ввиду того что Windows большей частью работает с прямоугольными областями, в библиотеке MFC определен специальный класс *CRect*, который похож на только что рассмотренную структуру *RECT*, но, кроме того, включает специальные функции для манипулирования объектами этого класса. Мы не будем рассматривать эти функции, т. к. их использование достаточно тривиально, а перечень и описание легко можно посмотреть в

Help. То же самое относится к удобным и часто используемым классам *CSize* и *CPoint*.

И последнее, что хотелось бы сказать по поводу сообщения WM_PAINT. Проделайте следующий эксперимент. Добавьте в программу обработчик нажатия левой кнопки мыши:

```
void CTextWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
  CClientDC dc(this);
  TCHAR strText[] = "Обработка сообщения WM_LBUTTONDOWN";
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  dc.TextOut(10, 10, strText, strlen(strText));
  CMDIChildWnd::OnLButtonDown(nFlags, point);
}
```

Еще раз подчеркнем — в Windows для проведения любых операций, связанных с выводом, *необходимо* сначала получить контекст устройства. Как видите, здесь для получения контекста при выводе текста мы использовали другой класс — *CClientDC*. Кроме того, для вывода текста мы воспользовались еще одной имеющейся в библиотеке функцией — *TextOut*.

```
virtual BOOL CDC::TextOut(
 int x,
 int y,
 LPCTSTR lpszString,
 int nCount) —
```

эта функция выводит в контекст устройства, начиная с точки (*x*,*y*), строку *lpszString* длиной *nCount*. Все остальные параметры, такие как шрифт, цвета фона и текста и т. д. берутся из контекста устройства.

Для того чтобы понять разницу в использовании *CClientDC* и *CPaintDC*, вам достаточно перетащить одно дочернее окно поверх другого, в котором предварительно был изображен текст. Если после этого вы вновь поместите первое окно на передний план, щелкнув по его заголовку, то увидите примерно то, что изображено на рис. 25.

Если вы отчетливо понимаете, что произошло, то вы действительно вникли в суть механизмов, задействованных в системе сообщений Windows. Сообщение WM_PAINT посылается в окно каждый раз, когда открывается его часть, временно закрытая другим окном. Таким образом, текст "Обработка сообщения WM_PAINT" будет виден всегда, когда видимо окно, просто потому, что Windows гарантирует посылку сообщения WM_PAINT в нужный момент. Однако, любой текст, выводимый обработчиком *OnLButtonDown*, не будет перерисовываться вновь. Причина заключается в том, что в этот обработчик посылается только сообщение

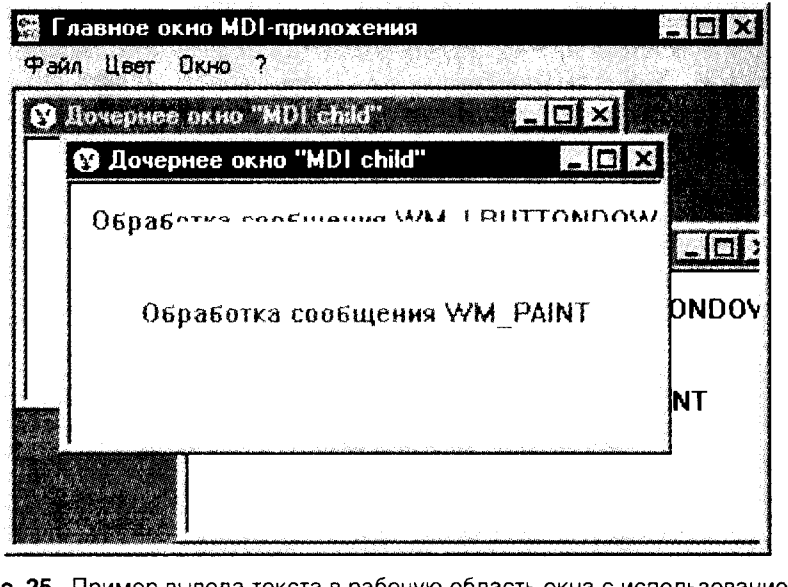


Рис. 25. Пример вывода текста в рабочую область окна с использованием и без использования обработчика сообщения WM_PAINT

WM_LBUTTONDOWN, и он "ничего не знает" о сообщении WM_PAINT и не обращает на него внимания.

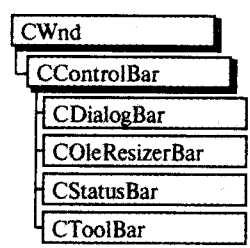
В заключение скажем несколько слов о сообщениях, обрабатываемых окнами "MDI child" по умолчанию. К ним относятся: WM_MOVE, WM_SETFOCUS, WM_CHILDACTIVATE, WM_GETMINMAXINFO, WM_MENUCHAR, WM_SIZE, WM_NEXTMENU и WM_SYSCOMMAND. И если вы перехватываете какое-либо из них, то не забывайте вызывать соответствующий обработчик базового класса.

Глава 13

Панели элементов управления

Любое современное профессионально написанное приложение имеет в своем главном окне как минимум два дополнительных инструментальных средства, значительно облегчающих работу, — это панель инструментов (Toolbar) и строка состояния (Statusbar). И хотя в библиотеке MFC для этих, а также некоторых других элементов давно реализованы соответствующие классы, только 32-разрядные операционные системы Windows 95 и Windows NT обеспечивают поддержку этих элементов интерфейса пользователя. На самом деле, создать панель элементов управления не сложнее, чем обыкновенное окно, тем более, используя классы MFC. Пришло время познакомиться с этими, очень удобными и полезными компонентами любого современного приложения. Начнем с рассмотрения общих свойств и возможностей, характерных для всех подобных элементов.

Класс CControlBar



CControlBar — базовый класс для всех классов панелей элементов управления, включающих *CStatusBar*, *CToolBar*, *CDialogBar* и *COleResizerBar*.[1] Панель элементов управления, сама являясь окном, может содержать дочерние элементы, которые могут быть либо окнами Windows, способными отправлять и получать сообщения, либо не являться таковыми, т. е. управляться кодом приложения или фрейма. В качестве примера элементов первого типа можно назвать комбинированный список (COMBO BOX) и элемент редактирования (EDIT), а второго типа — кнопки с рисунком (BITMAP BUTTONS) и области строки состояния (STATUS BAR PANES). Базовые функциональные возможности для всех типов панелей элементов управления похожи. Все объекты панелей элементов управ-

[1] Класс *COleResizerBar* будем рассматривать в разделе, посвященном OLE.

ления являются дочерними окнами некоторого родительского фрейма. Причем обычно они являются окнами одного уровня с окном MDICLIENT или представлением. Панель элементов управления может быть *привязанной* (docking) к любой стороне своего родительского фрейма или *плавающей* (floating), что определяется установкой соответствующего стиля. Помимо этого, объекты *CControlBar* самостоятельно позиционируются, используя информацию о рабочей области своего родительского окна, после чего они изменяют ее размеры таким образом, чтобы представление или окно MDICLIENT заполнили оставшуюся часть рабочей области.

Класс содержит только одну общедоступную переменную:

BOOL CControlBar::**m_bAutoDelete** —
служит для индикации того, когда будет удаляться объект этого или производного класса — при разрушении объекта Windows (TRUE) или при разрушении объекта фрейма, владеющего этой панелью. Если объект этого класса создается динамически, то рекомендуется установить значение переменной равной TRUE с тем, чтобы родительский фрейм сам позаботился о его удалении.

Из функций выделим следующие:

int CControlBar::**GetCount**() —
возвращает число неоконных элементов объекта CControlBar, тип которых зависит от производного объекта — областей для объектов CStatusBar и кнопок или разделителей для объектов CToolBar. Для объекта CDialogBar функция возвращает 0.

virtual CSize CControlBar::**CalcFixedLayout**(
BOOL bStretch,
BOOL bHorz) —
определяет горизонтальный размер панели элементов управления. Параметр bHorz задает ориентацию панели: TRUE — горизонтальная, FALSE — вертикальная. Если bStretch равно TRUE, то панель элементов управления предварительно растягивается до размеров родительского фрейма и не может менять свое местоположение, а если FALSE, то изменения размеров не происходит, и она может быть как привязанной к фрейму, так и быть плавающей.

DWORD CControlBar::**GetBarStyle**() —
возвращает текущий стиль, установленный для панели элементов управления.

void CControlBar::**SetBarStyle**(DWORD dwStyle) —
устанавливает требуемый стиль панели элементов управления. При этом доступными являются нижеперечисленные стили, имеющие префикс CBRS_:

CBRS_ALIGN_TOP	привязка к верхней стороне рабочей области фрейма
CBRS_ALIGN_BOTTOM	привязка к нижней стороне рабочей области фрейма
CBRS_ALIGN_LEFT	привязка к левой стороне рабочей области фрейма
CBRS_ALIGN_RIGHT	привязка к правой стороне рабочей области фрейма

CBRS_ALIGN_ANY	привязка к какой-либо стороне рабочей области фрейма
CBRS_BORDER_TOP	рамка очерчивает верхний край панели элементов управления, когда она видима
CBRS_ BORDER _BOTTOM	рамка очерчивает нижний край панели элементов управления, когда она видима
CBRS_ BORDER _LEFT	рамка очерчивает правый край панели элементов управления, когда она видима
CBRS_ BORDER _RIGHT	рамка очерчивает левый край панели элементов управления, когда она видима
CBRS_FLOAT_MULTI	в единственном окне мини-фрейма могут располагаться несколько плавающих панелей элементов управления
CBRS_TOOLTIPS	для панели элементов управления выводятся всплывающие подсказки (tool tips)
CBRS_FLYBY	текст сообщения обновляется одновременно со всплывающими подсказками

Кроме перечисленных можно использовать специальные стили общих элементов управления:

CCS_ADJUSTABLE	пользователь может изменять конфигурацию элемента управления
CCS_BOTTOM	элемент управления должен быть расположен в нижней части рабочей области окна
CCS_TOP	элемент управления должен быть расположен в верхней части рабочей области окна
CCS_NODIVIDER	в верхней части элемента управления не надо рисовать разделительную линию шириной 2 пикселя
CCS_NOHILITE	в верхней части элемента управления не надо рисовать разделительную линию шириной 1 пиксель
CCS_NOMOVEY	в ответ на сообщение WM_SIZE элемент управления может изменять свои размеры и перемещаться только горизонтально, а вертикальное положение меняться не будет. Не используйте этот стиль совместно со стилем CCS_NORESIZE
CCS_NOPARENTALIGN	элемент управления не будет автоматически перемещаться в верхнюю или нижнюю часть родительского окна
CCS_NORESIZE	приложение должно задать размеры элемента управления явным образом, т. к. размеры, заданные по умолчанию, не будут использоваться

Как видите, достаточно много стилей панелей элементов управления связано со способом ее размещения в родительском окне. Для управления этим процессом в классе реализованы специальные функции:

CFrameWnd* CControlBar::**GetDockingFrame**() —
служит для получения указателя на текущий фрейм, к которому привязана ваша панель элементов управления.

BOOL CControlBar::**IsFloating**() —
возвращаемое значение позволяет определить состояние панели элементов управления: плавающая (TRUE) или привязанная (FALSE).

void CControlBar::**EnableDocking**(DWORD dwStyle) —
разрешает привязать панель элементов управления к одной из сторон родительского фрейма, которая должна быть доступна для привязки. Если сторона, задаваемая параметром dwStyle, и сторона, разрешенная для привязки у фрейма, не совпадают, то панель элементов управления не может быть привязана к родительскому фрейму. Параметр dwStyle может принимать следующие значения: CBRS_ALIGN_TOP, CBRS_ALIGN_RIGHT, CBRS_ALIGN_LEFT, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_ANY и CBRS_ALIGN_MULTI.

Поскольку возможность привязки к родительскому фрейму зависит "от желания" не только самой панели элементов управления, но и ее родителя, нам представляется необходимым здесь же рассмотреть функции класса *CFrameWnd*, которые отвечают за работу с панелями элементов управления:

void CFrameWnd::**SaveBarState**(LPCTSTR lpszProfileName)
и
void CFrameWnd::**LoadBarState**(LPCTSTR lpszProfileName) —
эти функции позволяют сохранять информацию о каждой панели элементов управления текущего фрейма в INI-файле или реестре и восстанавливать ее. При этом сохраняется и восстанавливается информация следующего рода: видимость, ориентация (горизонтальная или вертикальная), состояние привязки и положение.

void CFrameWnd::**ShowControlBar**(
CControlBar *pBar,
BOOL bShow,
BOOL bDelay) —
параметр bShow определяет видимость панели на экране: TRUE — показать, FALSE — скрыть панель элементов управления pBar. Значение параметра bDelay, равное TRUE, определяет, что перед выводом на экран нужна задержка.

void CFrameWnd::**EnableDocking**(DWORD dwDockStyle) —
разрешает привязку панели элементов управления к стороне(ам) фрейма, задаваемой(ым) в параметре dwDockStyle: CBRS_ALIGN_TOP, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_RIGHT, CBRS_ALIGN_LEFT, CBRS_ALIGN_ANY. По умолчанию панели элементов управления привязываются к сторонам фрейма в следующем порядке: верхняя, нижняя, левая и правая.

void CFrameWnd::**DockControlBar**(
CControlBar *pBar,
UINT nDockBarID = 0,
LPCRECT lpRect = NULL) —
привязывает панель элементов управления pBar к одной из сторон текущего фрейма, которая задается параметром nDockBarID и может быть нулем или комбинацией следующих значений:

AFX_IDW_DOCKBAR_TOP	привязка к верхней стороне фрейма
AFX_IDW_DOCKBAR_BOTTOM	привязка к нижней стороне фрейма
AFX_IDW_DOCKBAR_LEFT	привязка к левой стороне фрейма
AFX_IDW_DOCKBAR_RIGHT	привязка к правой стороне фрейма

Если этот параметр равен нулю, то сторона может быть любой из числа предварительно определенных при помощи двух функций: CControlBar::EnableDocking и CFrameWnd:: EnableDocking. Параметр lpRect определяет, где (в координатах экрана) вне рабочей области фрейма будет привязано окно, содержащее плавающую панель элементов управления.

CFrameWnd* CFrameWnd::**FloatControlBar**(
CControlBar *pBar,
CPoint point,
DWORD dwStyle = CBRS_ALIGN_TOP) —

переводит панель элементов управления pBar в плавающее состояние. Обычно вызывается при запуске приложения для восстановления предыдущих установок. Кроме того, эта функция вызывается фреймом, когда пользователь выполняет операцию отпускания (drop) после перемещения панели, не достигнув места, доступного для привязывания. Параметр point задает координаты точки, в которой будет находиться левый верхний угол панели элементов управления, а dwStyle определяет стиль ее выравнивания и ориентации внутри нового фрейма: CBRS_ALIGN_BOTTOM и CBRS_ALIGN_TOP — вертикальная ориентация, а CBRS_ALIGN_RIGHT и CBRS_ALIGN_ LEFT — горизонтальная. Если указаны оба типа ориентации, то панель будет ориентирована горизонтально.

virtual void CFrameWnd::**RecalcLayout**(BOOL bNotify = TRUE) —

перепозиционирует панели элементов управления текущего объекта. Параметр bNotify определяет, получит ли активизированный по месту элемент извещение (notify) при изменении размещения. Эта функция вызывается фреймом, когда переключаются стандартные панели элементов управления, или когда изменяется размер фрейма. Для того чтобы иметь возможность управлять отображением и поведением панелей управления, после того, как изменилась компоновка фрейма, необходимо переопределить эту функцию в своем классе. Реализация этой функции, определенная по умолчанию, вызывает функцию CWnd::RepositionBars, чтобы перепозиционировать все панели элементов управления во фрейме.

Как мы уже сказали, этот класс реализует основные функциональные возможности для всех панелей элементов управления. Более специфические и интересные возможности реализованы в его производных классах. Но прежде чем переходить к их рассмотрению, отметим еще один важный момент: в том случае, если кнопка панели элементов управления не имеет обработчиков для UPDATE_COMMAND_UI или COMMAND, то она автоматически блокируется фреймом.

Класс CToolBar

Этот класс определяет поведение *панели инструментов* — панели элементов управления, отображаемой полосой кнопок с рисунками (все они должны быть одного размера — стандартно 23×22 пикселя) и разделителей. Кнопки активизируют команды, т. к. нажатие кнопки панели инструментов подобно выбору элемента меню. Так же, как и элементы меню, кнопки панели инструментов могут действовать аналогично обычным кнопкам, переключателям и флажкам. Как и любая другая панель элементов управления, панель инструментов может быть привязана к какой-либо стороне родительского фрейма или быть плавающей. В последнем случае пользователь может изменять ее размер. Класс *CToolBar* поддерживает работу со всплывающей подсказкой, крошечным всплывающим окном, содержащим текстовую строку.

Для изображения всех кнопок панели инструментов создается один битовый массив, который должен содержать один и только один битовый рисунок (образ) для каждой кнопки. Обычно порядок расположения образов тот же, в котором они отображаются на экране. Все образы должны иметь один и тот же размер (стандартно 16×15 пикселей) и располагаться рядом друг с другом без промежутков. Изменение изображения кнопок панели инструментов в зависимости от их состояния и стиля (нажатая, заблокированная, неопределенная, отжатая и т. п.) формируется автоматически благодаря механизму обработки команды ON_UPDATE_COMMAND_UI: во время цикла простоя (idle time) панель инструментов вызывает обработчик команды ON_UPDATE_COMMAND_UI с идентификаторами команд для всех своих кнопок (исключая разделители). Для панелей инструментов класс *CCmdUI*, обеспечивающий правильное функционирование обработчика, поддерживает следующие функции: *CCmdUI::Enable*, *CCmdUI::SetCheck* и *CCmdUI::SetRadio*.[1]

Теперь, когда у вас сложилось общее представление о панели инструментов, мы, как обычно, рассмотрим основные функциональные возможности, предоставляемые классом *CToolBar*.

Поскольку панель инструментов — это окно, то для ее создания необходимо проделать стандартные шаги: создать объект класса и затем создать окно Windows на базе предопределенного оконного класса TOOLBARCLASSNAME. Конструктор класса

CToolBar::**CToolBar()** –
: создает объект класса и устанавливает размеры по умолчанию: размер кнопок – 23×22, размер образа – 16×15, высота верхней и нижней рамки – 3 пикселя.

---

[1] Подробно о классе *CCmdUI* было рассказано в главе 10.

Для создания окна Windows и присоединения его к объекту используется функция

BOOL CToolBar::**Create**(
CWnd *pParentWnd,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP,
UINT nID = AFX_IDW_TOOLBAR) —
которой в качестве параметров передаются следующие значения: pParentWnd — указатель на родительское окно панели инструментов, nID — идентификатор дочернего окна и dwStyle — стиль панели инструментов. Помимо общих стилей окон Windows и панелей элементов управления, она поддерживает дополнительные:

CBRS_TOP	предписывает расположить панель вверху рабочей области родительского окна
CBRS_BOTTOM	предписывает расположить панель внизу рабочей области родительского окна
CBRS_NOALIGN	разрешает не переопределять размеры панели, если изменились родительские
CBRS_TOOLTIPS	задает вывод подсказок для кнопок
CBRS_SIZE_DYNAMIC	разрешает пользователю менять размер панели инструментов при помощи мыши, когда первая находится в плавающем состоянии
CBRS_SIZE_FIXED	запрещает менять размер панели инструментов
CBRS_FLOATING	задает плавающую панель инструментов
CBRS_FLYBY	предписывает отображать информацию о кнопке в строке состояния
CBRS_HIDE_INPLACE	предписывает сделать панель инструментов невидимой для пользователя

Теперь осталось только загрузить образы кнопок из ресурсов. Для этого в классе предусмотрены две возможности: вызов функции *LoadToolBar* или загрузка образов кнопок при помощи функции *LoadBitmap* и привязка их к соответствующим командам функцией *SetButtons*. Рассмотрим эти функции более подробно:

BOOL CToolBar::**LoadToolBar**(LPCTSTR lpszResourceName)

и

BOOL CToolBar::**LoadToolBar**(UINT nIDResource) —
загружает изображение панели инструментов и ресурса, заданного либо именем (параметр lpszResourceName), либо идентификатором (параметр nIDResource). При этом сопоставление кнопок с образами и командами осуществляется автоматически. Visual C++ 4.x поддерживает ресурс типа TOOLBAR, где не только задается вид кнопок, но также осуществляется привязка команд к ним.[1]

[1] Там же можно задавать разделители и общие размеры кнопок, которые могут отличаться от стандартных.

BOOL CToolBar::**LoadBitmap**(LPCTSTR lpszResourceName)

и

BOOL CToolBar::**LoadBitmap**(UINT nIDResource) —

загружает битовый массив, определяемый параметром lpszResourceName или nIDResource, который должен содержать один образ для каждой кнопки панели инструментов. Если образы имеют нестандартный размер, то необходимо вызвать функцию SetSizes для установки правильных размеров кнопок и их образов.

BOOL CToolBar::**SetButtons**(
const UINT *lpIDArray,
int nIDCount) —

устанавливает идентификатор команды каждой кнопки панели инструментов в значение, определяемое соответствующим элементом массива lpIDArray, число элементов которого задается параметром nIDCount. Если элемент массива имеет предопределенное значение ID_SEPARATOR, то в соответствующей позиции панели инструментов создается разделитель. Если значение lpIDArray равно NULL, то функция просто отводит место для nIDCount элементов, а для установки атрибутов каждого элемента следует вызвать функцию SetButtonInfo. Эта функция устанавливает также для каждой кнопки стиль TBBS_BUTTON, а для каждого разделителя — стиль TBBS_SEPARATOR, и назначает каждой кнопке индекс образа. *Индекс образа* определяет его позицию внутри битового массива.

После того как панель инструментов создана, с ней можно начинать "общаться", и нам осталось рассмотреть предназначенные для этого функции:

void CToolBar::**SetSizes**(
SIZE sizeButton,
SIZE sizeImage) —

устанавливает размер (в пикселях) кнопок панели инструментов, задаваемый параметром sizeButton. При этом параметр sizeImage должен содержать размер образов, зафиксированный в соответствующем битовом массиве). Размеры кнопок должны задаваться исходя из размеров образов, плюс 7 пикселей для ширины и 6 — для высоты. Размеры самой панели инструментов функция устанавливает автоматически. Microsoft настаивает (это отражено в документе An Application Design Guide) на том, чтобы приложения содержали кнопки стандартных размеров, но любезно предоставляет возможность их изменять. [1]

void CToolBar::**SetHeight**(int cyHeight) —

устанавливает новое значение высоты панели инструментов, определяемое параметром cyHeight. Если высота окажется слишком мала, то кнопки будут обрезаны снизу.

BOOL CToolBar::**SetBitmap**(HBITMAP hbmImageWell) —

задает для панели инструментов новый битовый массив, определяемый параметром hbmImageWell.

---

[1] Требования к пользовательскому интерфейсу не последние в ряду требований Microsoft к программным продуктам, претендующим на получение логотипа "сделано для Windows". См. часть I.

int CToolBar::**CommandToIndex**(UINT nIDFind) —
возвращает индекс первой кнопки (начиная с нуля), чей идентификатор команды совпадает с заданным в параметре nIDFind. Если такой идентификатор не найден, то возвращается –1.

UINT CToolBar::**GetItemID**(int nIndex) —
возвращает идентификатор команды кнопки или разделителя из позиции, задаваемой параметром nIndex. Если найден разделитель, функция возвращает значение ID_SEPARATOR.

virtual void CToolBar::**GetItemRect**(
int nIndex,
LPRECT lpRect) —
заполняет структуру lpRect координатами кнопки или разделителя, определяемой параметром nIndex. Координаты отсчитываются относительно левого верхнего угла панели инструментов. Эту функцию можно использовать для определения координат разделителя, который вы хотите заменить на комбинированный список или другой элемент управления.

UINT CToolBar::**GetButtonStyle**(int nIndex) —
возвращает стиль кнопки или разделителя, заданной номером позиции на панели инструментов nIndex.

void CToolBar::**SetButtonStyle**(
int nIndex,
UINT nStyle) —
позволяет установить стиль кнопки, заданной номером позиции nIndex. Стиль кнопки определяет ее реакцию на действия пользователя. Доступными являются следующие стили:

TBBS_BUTTON	стандартная кнопка
TBBS_SEPARATOR	разделитель между группами кнопок
TBBS_CHECKBOX	кнопка с фиксацией; эта кнопка "залипает" при нажатии, для возврата в исходное состояние необходимо нажать ее еще раз
TBBS_GROUP	отмечает начало группы стандартных кнопок
TBBS_CHECKGROUP	отмечает начало группы кнопок с фиксацией, остается нажатой до тех пор, пока не нажата другая кнопка из этой же группы

UINT CToolBar::**GetButtonInfo**(
int nIndex,
UINT nID,
UINT nStyle,
int iImage) —
позволяет получить идентификатор команды, стиль и индекс образа кнопки (начиная с 0), позиция которой задается параметром nIndex. Эти значения присваиваются переменным, на которые ссылаются соответственно nID, nStyle и iImage. Если запрашивается информация о разделителе, то в iImage заносится значение его ширины в пикселях.

void CToolBar::**SetButtonInfo**(
  int nIndex,
  UINT nID,
  UINT nStyle,
  int iImage) –
позволяет установить идентификатор команды *nID*, стиль nStyle и индекс образа iImage кнопки, позиция которой задается параметром nIndex. Для разделителя эта функция устанавливает его ширину в пикселях, значение которой передается в параметре iImage.

Помимо образа, каждой кнопке панели инструментов может быть поставлена в соответствие текстовая строка, которая появляется на ней.[1] Для определения текста этой строки можно воспользоваться следующими функциями:

void CToolBar::**GetButtonText**(
  int nIndex,
  CString &rString)

и

CString CToolBar::**GetButtonText**(int nIndex)

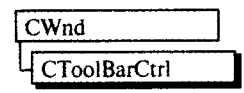
и

BOOL CToolBar::**SetButtonText**(
  int nIndex,
  LPCTSRT lpszText)

Последняя функция класса *CToolBar* позволяет получить ссылку на объект *CToolBarCtrl*:

CToolBar& CToolBar::**GetToolBarCtrl**() –
позволяет воспользоваться функциональными возможностями общего элемента управления TOOLBAR, появившегося в Win32. Использование этого элемента управления позволяет, в частности, осуществлять более тонкую настройку панели инструментов.

Класс CToolBarCtrl



Для использования преимуществ общего элемента управления TOOLBAR можно пойти двумя путями: либо непосредственно создать окно *CToolBarCtrl*, либо создать панель инструментов на основе класса *CToolBar*, а затем просто получить доступ к элементам объекта *CToolBarCtrl*, вызвав функцию этого класса *CToolBar::CToolBarCtrl*. Мы остановимся на втором способе и не будем рассматривать создание элемента управления TOOLBAR.

---

[1] После дополнительных действий и выполнения многочисленных условий. См. пример далее в этой главе.

Любая кнопка панели инструментов имеет ассоциированную с ней структуру *TBBUTTON*, которая содержит полную информацию об этой кнопке. Описание структуры находится в файле COMMCTRL.H:

```
typedef struct _TBBUTTON {
  int iBitmap;
  int idCommand;
  BYTE fsState;
  BYTE fsStyle;
#ifdef _WIN32
  BYTE bReserved[2];
#endif
  DWORD dwData;
  int iString;
} TBBUTTON, NEAR* PTBBUTTON, FAR* LPTBBUTTON;
```

В поле *iBitmap* содержится номер кнопки (нумерация начинается с нуля). Для разделителя здесь должно быть записано нулевое значение.

В поле *idCommand* находится идентификатор команды, который будет передаваться окну-владельцу вместе с сообщением WM_COMMAND, когда пользователь выберет эту кнопку. Для разделителя здесь должно быть записано нулевое значение.

Поле *fsState* содержит комбинацию одного или нескольких флагов, определяющих состояние кнопки:

TBSTATE_CHECKED	кнопка имеет стиль TBSTYLE_CHECKED — кнопка с фиксацией, и изображается в нажатом состоянии
TBSTATE_ENABLED	кнопка доступна для действий пользователя; если это состояние не установлено, то кнопка заблокирована и изображается серым цветом
TBSTATE_HIDDEN	кнопка не видима
TBSTATE_INDETERMINATE	кнопка отображается серым цветом
TBSTATE_PRESSED	кнопка изображается в нажатом состоянии
TBSTATE_WRAP	кнопка, находящаяся в этом состоянии, сообщает о том, что все следующие кнопки будут отображаться на новой строке; использование этого флага позволяет создавать многострочные панели инструментов; этот флаг можно указывать только вместе с флагом TBSTATE_ENABLED

В поле *fsStyle* записывается стиль кнопки. Можно использовать комбинацию следующих стилей:

TBSTYLE_BUTTON	стандартная кнопка
TBSTYLE_SEP	разделитель, представляющий собой маленький промежуток между группами кнопок; пользователь не может взаимодействовать с кнопкой, имеющей такой стиль

TBSTYLE_CHECK	кнопка с фиксацией; эта кнопка "залипает" при нажатии, для возврата в исходное состояние необходимо нажать ее еще раз
TBSTYLE_GROUP	отмечает начало группы стандартных кнопок
TBSTYLE_CHECKGROUP	отмечает начало группы кнопок с фиксацией; кнопка, имеющая такой стиль, остается нажатой до тех пор, пока не нажата другая кнопка из этой же группы

В поле *dwData* можно хранить дополнительные данные, связанные с конкретной кнопкой. Если это поле не используется, то лучше записать туда нулевое значение.

Если вы хотите написать на поверхности кнопки какой-либо текст, то в поле *iString* следует записать ее номер во внутреннем списке элемента управления. В противном случае здесь должно быть записано нулевое значение.

Поле *bReserved* зарезервировано для внутреннего использования.

Полную информацию о любой кнопке панели инструментов можно получить при помощи функции

```
BOOL CToolBarCtrl::GetButton(
  int nIndex,
  LPTBBUTTON lpButton) --
      кнопка, информация о которой записывается в структуру lpButton, задается номером nIndex.
```

Достаточно большая группа функций позволяет установить новое или узнать текущее состояние определенной кнопки этого элемента управления. Их действие настолько очевидно, что мы ограничимся только перечислением: *IsButtonEnabled*, *IsButtonChecked*, *IsButtonPressed*, *IsButtonHidden*, *IsButtonIndeterminate*, *GetState*, *SetState*, *EnableButton*, *CheckButton*, *PressButton*, *HideButton*, *Indeterminate*.

Если необходимо узнать общее число кнопок панели инструментов, то можно воспользоваться функцией

```
int CToolBarCtrl::GetButtonCount()
```

Для смены окна, которое будет получать извещения от панели инструментов, необходимо сменить владельца этой панели. Это можно сделать при помощи функции

```
void CToolBarCtrl::SetOwner(CWnd *pWnd)
```

Как мы уже говорили, кнопки панели инструментов могут располагаться в несколько рядов. Для того чтобы узнать или изменить их число, в классе реализованы две функции

```
void CToolBarCtrl::SetRows(
   int nRows,
   BOOL bLarger,
   LPRECT lpRect)
```

и

```
int CToolBarCtrl::GetRows()
```

которые имеют следующие параметры: nRows — требуемое число строк; bLarger — определяет, будет ли увеличено (TRUE) или уменьшено (FALSE) число строк. Если панель инструментов не может изменить свой размер в соответствии с заданным числом строк, то она будет изменена на максимально возможное количество строк, естественно, меньшее, чем заданное в параметре nRow; lpRect — указатель на объект CRect или структуру RECT, в которую будут записаны новые размеры прямоугольника, ограничивающего панель инструментов. Число строк должно выбираться так, чтобы в каждой строке (может быть, за исключением последней) было одинаковое количество элементов. Обе эти функции могут быть вызваны только для панелей инструментов, которые созданы со стилем TBSTYLE_WRAPABLE.

Чтобы проверить, поддерживает ли дисплей "большие" битовые массивы предназначена функция

```
UINT CToolBArCtrl::GetBitmapFlags()
```

которая возвращает TBBF_LARGE, если "большие" битовые массивы можно использовать, и 0 в противном случае.

Следующая группа функций позволяет добавить новые битовые массивы или текстовые строки для кнопок:

```
int CToolBarCtrl::AddStrings(LPCTSTR lpszStrings)
```

и

```
int CToolBarCtrl::AddString(UINT nStringID)

int CToolBarCtrl::AddBitmap(
   int nNumButtons,
   UINT nBitmapID)
```

и

```
int CToolBarCtrl::AddBitmap(
   int nNumButtons,
   CBitmap *pBitmap)
```

При работе с панелью инструментов можно изменять ее внешний вид.

```
void CTollBatCtrl::Customize() —
```

выводит на экран специальное диалоговое окно для настройки панелей инструментов (Customize Toolbar). При этом сама панель инструментов должна иметь установленный флаг стиля CCS_ADJUSTABLE и обрабатывать специальные извещения настройки. То, как это делается, описано ниже.

Параметры настройки можно сохранять в реестре и восстанавливать их из него:

void CToolBarCtrl::**SaveState**(
  HKEY hKeyRoot,
  LPCTSTR lpszSubKey,
  LPCTSTR lpszValueName)

и

void CToolBarCtrl::**RestoreState**(
  HKEY hKeyRoot,
  LPCTSTR lpszSubKey,
  LPCTSTR lpszValueName) —
  в этих функциях используются следующие параметры: hKeyRoot — идентификатор текущего открытого ключа в реестре или один из предопределенных дескрипторов HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE и HKEY_USERS; lpszSubKey — указатель на символьную строку, которая содержит имя подключа, с которым ассоциируется значение; lpszValueName — указатель на символьную строку, содержащую имя сохраняемого или восстанавливаемого значения.

Если изменились размеры родительского окна или панели инструментов, то можно вызвать функцию

void CToolBarCtrl::**AutoSize**()
  для автоматической установки новых размеров элемента управления.

Оставшиеся функции позволяют добавлять и удалять кнопки панели инструментов:

BOOL CToolBarCtrl::**AddButtons**(
  int nNumButtons,
  LPTBBUTTON lpButtons) —
  добавляет nNumButtons кнопок к панели инструментов. Предварительно необходимо заполнить массив структур lpButtons полной информацией о кнопках. Этот массив должен иметь такое же число элементов, какое указано в параметре nNumButtons.

BOOL CToolBarCtrl::**InsertButton**(
  int nIndex,
  LPTBBUTTON lpButton) —
  вставляет новую кнопку в панель инструментов слева от указанной в параметре nIndex. Перед вызовом функции структура lpButton должна быть заполнена. Рисунок (текст) кнопки должен быть предварительно добавлен к списку панели инструментов при помощи функций AddBitmap, AddString или AddStrings.

BOOL CTollBarCtrl::**DeleteButton**(int nIndex) —
  удаляет кнопку, заданную параметром nIndex, из панели инструментов.

Последнее, что нам необходимо рассмотреть для полноценной работы с этим элементом управления, — это настройка панели инструментов, по сути — обработка извещений, которые посылает система на каждом шаге процесса настройки. Причем отсутствие обработчика исключает возможность, на первый взгляд, самых простых действий по настройке, например, смены положения кнопок в панели.

Если пользователь начинает перемещать какую-либо кнопку панели инструментов, удерживая нажатой клавишу <Shift>, то элемент управления автоматически обрабатывает операцию перемещения, посылая своему родительскому окну извещение TBN_QUERYDELETE, чтобы определить, можно ли удалить кнопку. Операция перемещения заканчивается, если родительское окно возвращает FALSE, в противном случае панель инструментов захватывает ввод от мыши и ждет, когда пользователь отпустит ее кнопку. Как только кнопка отпущена, панель инструментов определяет местонахождение курсора мыши. Если он находится вне панели, то кнопка удаляется. Если на другой кнопке, то в родительское окно посылается извещение TBN_QUERYINSERT, чтобы выяснить, можно ли вставить кнопку слева от данной. Кнопка вставляется, если родительское окно возвращает TRUE и не вставляется в противном случае. При завершении операции перемещения панель инструментов посылает извещение TBN_TOOLBARCHANGE. Если манипуляции с кнопками панели инструментов осуществляются без нажатой клавиши <Shift>, то родительскому окну дополнительно посылаются два извещения — TBN_BEGINDRAG в начале выполнения операции перемещения и TBN_ENDDRAG в конце.

Когда пользователь настраивает панель инструментов, используя блок диалога для настройки (Customize Toolbar), она посылает определенную последовательность извещений. Прежде всего, это извещение TBN_BEGINADJUST, которое посылается после того, как пользователь дважды щелкнет левой кнопкой мыши на панели инструментов, но до создания блока диалога. Затем панель инструментов начинает посылать ряд извещений, чтобы определить, можно ли вставлять кнопки. Извещения посылаются до тех пор, пока родительское окно не вернет TRUE (если для всех кнопок возвращается FALSE, то панель инструментов уничтожает блок диалога). И, наконец, она проверяет, можно ли удалять кнопки, посылая для этого по одному сообщению TBN_QUERYDELETE для каждой. Если кнопку можно удалить, то родительское окно возвращает TRUE, в противном случае — FALSE. После этого панель инструментов добавляет все свои кнопки в блок диалога, блокируя те, которые не могут быть удалены.

Всякий раз, когда панель инструментов нуждается в информации о какой-либо кнопке блока диалога, она посылает извещение TBN_GETBUTTONINFO, задавая индекс кнопки и адрес структуры *TBNOTIFY*, куда родительское окно должно записать соответствующую информацию.

Блок диалога настройки панели инструментов включает также две кнопки: Help (Справка) и Reset (Сброс). Когда пользователь нажимает кнопку Help (Справка), панель инструментов посылает извещение TBN_CUSTHELP, в ответ на которое родительское окно должно отобразить справочную информацию. При нажатии кнопки Reset (Сброс) блок диалога посылает извещение TBN_RESET, которое говорит о том, что панель инструментов собирается переинициализировать блок диалога.

Для того чтобы подключить обработчик соответствующего извещения, необходимо в карту сообщений окна, владеющего панелью инструментов, добавить запись следующего вида:[1]

```
ON_NOTIFY(wNotifyCode, idControl, memberFxn)
```

где *wNotifyCode* — код идентификатора извещения, например, TBN_ENDDRAG; *idControl* — идентификатор элемента управления, посылающего сообщение; *memberFxn* — функция-обработчик сообщения, которая должна иметь следующий прототип:

```
afx_msg void memberFxn(
  NMHDR *pNotifyStruct,
  LRESULT *result)
```

Если обработчик должен возвратить какое-либо значение, то его следует поместить в параметр *result*. Параметр *pNotifyStruct* указывает на одну из структур *NMHDR* или *TBNOTIFY*:

```
typedef struct tagNMHDR{
  HWND hwndFrom;
  UINT idFrom;
  UINT code;
} NMHDR;
```

В этой структуре поле *hwndFrom* содержит идентификатор окна, приславшего сообщение, поле *idFrom* — идентификатор элемента управления, приславшего сообщение, а поле *code* — код извещения, который может быть специфическим для определенного типа элементов управления или одним из перечисленных ниже, общих для всех элементов управления:

NM_CLICK	пользователь нажал левую кнопку мыши внутри элемента управления
NM_DBLCLICK	пользователь сделал двойной щелчок левой кнопкой мыши внутри элемента управления
NM_RCLICK	пользователь нажал правую кнопку мыши внутри элемента управления
NM_RDBLCLICK	пользователь сделал двойной щелчок правой кнопкой мыши внутри элемента управления
NM_KILLFOCUS	элемент управления потерял фокус ввода
NM_SETFOCUS	элемент управления получил фокус ввода
NM_RETURN	когда элемент управления имел фокус ввода, пользователь нажал клавишу <Enter>

[1] Более подробно об этом рассказано в части IV.

NM_OUTOFMEMORY элемент управления не может завершить операцию из-за нехватки доступной памяти

```
typedef struct tagTBNOTIFY{
  NMHDR hdr;
  int iItem;
  TBBUTTON tbButton;
  int cchText;
  LPSTR pszText;
}TBNOTIFY, FAR* LPTBNOTIFY;
```

Поле *hdr* содержит информацию, общую для всех сообщений WM_NOTIFY; поле *iItem* содержит индекс кнопки, ассоциированной с сообщением; *tbButton* — структура, содержащая информацию о ней; *cchText* — число символов в тексте кнопки; *lpszText* — указатель на этот текст.

Помимо перечисленных, общих для всех элементов управления кодов извещений, для панели инструментов определены дополнительные, большинство из которых мы упоминали при обсуждении процесса настройки:

TBN_BEGINADJUST посылается, когда пользователь приступил к настройке панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *NMHDR;* обработчик может ничего не возвращать

TBN_ENDADJUST посылается, когда пользователь закончил настройку панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *NMHDR*; обработчик может ничего не возвращать

TBN_BEGINDRAG посылается, когда пользователь начал перемещать кнопку по панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *TBNOTIFY*, поле iItem которой содержит индекс перемещаемой кнопки; обработчик может ничего не возвращать

TBN_ENDDRAG посылается, когда пользователь закончил перемещать кнопку по панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *TBNOTIFY*, поле iItem которой содержит индекс перемещаемой кнопки; обработчик может ничего не возвращать

TBN_CUSTHELP посылается, когда пользователь нажимает кнопку Help (Справка) в блоке диалога настройки панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *NMHDR*; обработчик должен восстановить исходный вид панели инструментов и может ничего не возвращать

TBN_RESET посылается, когда пользователь нажимает кнопку Reset (Сброс) в блоке диалога настройки панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *NMHDR;* обработчик может ничего не возвращать

TBN_GETBUTTONINFO посылается, когда пользователь настраивает панель инструментов, которая использует это извещение, чтобы получить информацию, необходимую блоку диалога настройки панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *TBNOTIFY*, поле iItem которой содержит индекс кнопки; обработчик должен заполнить поля структуры и в переменной *result* возвратить TRUE; если поля структуры по каким-либо причинам не заполнены, то возвращается FALSE

TBN_QUERYDELETE посылается, когда пользователь настраивает панель инструментов, чтобы определить, может ли кнопка быть удалена с панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *TBNOTIFY*, поле iItem которой содержит индекс удаляемой кнопки; обработчик должен возвратить TRUE в переменной result, чтобы удалить кнопку, или FALSE, чтобы предотвратить ее удаление

TBN_QUERYINSERT посылается, когда пользователь настраивает панель инструментов, чтобы определить, может ли кнопка быть вставлена в панель инструментов слева от указанной; указатель pNotifyStruct обработчика указывает на структуру *TBNOTIFY*, поле iItem которой содержит индекс кнопки, которую требуется вставить; обработчик должен возвратить TRUE в переменной result, чтобы вставить кнопку, или FALSE, чтобы предотвратить ее вставку

TBN_TOOLBARCHANGE посылается, когда пользователь закончил настройку панели инструментов; указатель pNotifyStruct обработчика указывает на структуру *NMHDR*; обработчик может ничего не возвращать

В заключение отметим, что настройка панели элементов управления — достаточно трудоемкий процесс, необходимо написать много функций, пусть и не больших по размеру. Кроме того, настраивать объект, созданный на базе класса *CToolBar*, несколько сложнее, чем объект класса *CToolBarCtrl*. Связано это с тем, что последний класс сам по себе не поддерживает списков описаний кнопок, и программист вынужден это делать самостоятельно, что существенно упрощает обработку такого уведомляющего сообщения, как TBN_GETBUTTONINFO. С другой стороны, объекты класса *CToolBarCtrl* не обеспечены поддержкой со стороны библиотеки классов MFC возможности привязки и управления типом расположения, в отличие от объектов класса *CToolBar*.

Класс CStatusBar

Рассмотрев, достаточно подробно, все аспекты работы с панелью инструментов, переходим к обсуждению не менее полезного элемента управления — *строки состояния*. Для него библиотека MFC также содержит два класса — *CStatusBar* и *CStatusBarCtrl*.

Класс *CStatusBar* определяет поведение строки состояния — панели элементов управления, отображаемой полосой, которую можно разделить на несколько областей для раздельного вывода в них текста или графической информации. Наиболее часто области используются для вывода сообщений (например, расширенной информации об элементах меню или о кнопках панели инструментов) и для отображения индикаторов состояния (например, клавиш <Caps Lock>, <Num Lock> и <Ins>). В качестве стандартного шрифта для строки состояния используется шрифт MS Sans Serif высотой 10 пунктов. Изменение изображения строки состояния осуществляется автоматически благодаря механизму ON_UPDATE_COMMAND_UI: во время цикла простоя (idle time) строка состояния вызывает обработчик команды ON_UPDATE_COMMAND_UI с идентификаторами строк областей индикации. Для строки состояния класс *CCmdUI*, обеспечивающий правильное функционирование обработчика, поддерживает следующие функции: *CCmdUI::Enable* и *CCmdUI::SetText*.

Строка состояния может работать в двух режимах — *упрощенном* и *стандартном*. В первом режиме разделение этого окна на несколько областей невозможно и выводить в него можно только текстовую информацию. В стандартном режиме приложение может разбить окно строки состояния на несколько областей для раздельного вывода в них текстовой или графической информации. По умолчанию принимается, что самая левая область — "эластичная": ее длина определяется длиной области строки состояния, не используемой всеми другими областями, которые, к тому же, выровнены по правому краю. Переход из одного режима в другой осуществляется при помощи сообщения SB_SIMPLE. Все настройки обычно проводятся перед первым отображением строки состояния на экране.

Перейдем теперь к рассмотрению основных членов этого класса.

CStatusBar::**CStatusBar**() –

конструктор объекта, который определяет шрифт строки состояния и устанавливает его характеристики в значения, используемые по умолчанию.

BOOL CStatusBar::**Create**(
CWnd *pParentWnd,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,
UINT nID = AFX_IDW_STATUS_BAR) –

создает дочернее окно "строка состояния", ассоциирует его с объектом *CStatusBar*, инициализирует шрифт и устанавливает его размер в значение, заданное по умолчанию. Для строки состояния помимо стандартных стилей окна доступны еще следующие: CBRS_TOP, CBRS_BOTTOM, CBRS_NOALIGN и SBRS_SIZEGRIP. Обратите внимание на последний стиль, при котором правый нижний угол строки состояния принимает вид, показанный на рис. 26.



Рис. 26. Строка состояния, созданная со стилем SBRS_SIZEGRIP

По умолчанию, строка состояния располагается в нижней части родительского окна. Однако, при желании, вы можете расположить ее и в верхней части, использовав стиль CBRS_TOP. Очевидно, что этот стиль нельзя комбинировать со стилем SBRS_SIZEGRIP, т. к., во-первых, такой элемент управления будет выглядеть очень странно, а во-вторых, он не будет работать так, как ожидает пользователь.

BOOL CStatusBar::**SetIndicators**(
const UINT *lpIDArray,
int nIDCount) —
устанавливает идентификатор каждой области в значение, заданное соответствующим элементом массива lpIDArray, загружает для них строки ресурса и записывает соответствующие текстовые строки в эти области. Параметр nIDCount определяет число элементов в массиве идентификаторов.

void CStatusBar::**SetPaneInfo**(
int nIndex,
UINT nID,
UINT nStyle,
int cxWidth) —
устанавливает для области, заданной номером nIndex, новые значения идентификатора (nID), стиля (nStyle) и ширины (cxWidth). Поддерживаются следующие дополнительные стили:

SBPS_NOBORDERS	вокруг области нет трехмерной рамки
SBPS_POPOUT	текст рисуется с рамкой, которая выглядит приподнятой над поверхностью строки состояния
SBPS_DISABLED	в область запрещен вывод текста
SBPS_STRETCH	"эластичная" (stretch) область, занимающая ту часть строки состояния, которая не используется под другие области; этот стиль может иметь только одна область строки состояния
SBPS_NORMAL	обычная область — без рамок и без возможности растяжения

void CStatusBar::**GetPaneInfo**(
int nIndex,
UINT &nID,
UINT &nStyle,
int &cxWidth) —
позволяет получить информацию об области строки состояния, заданной номером nIndex, включающую идентификатор (nID), стиль (nStyle) и ширину (cxWidth).

UINT CStatusBar::**SetPaneStyle**(
int nIndex,
UINT nStyle) —
устанавливает стиль (nStyle) области (nIndex) строки состояния.

UINT CStatusBar::**GetPaneStyle**(int nIndex) —
возвращает стиль области строки состояния, который определяет ее внешний вид. Номер области задается параметром nIndex.

```
BOOL CStatusBar::SetPaneText(
   int nIndex,
   LPCTSTR lpszNewText,
   BOOL bUpdate = TRUE) —
```

записывает текст, на который указывает *lpszNewText*, в область строки состояния, заданную номером *nIndex*. Если параметр *bUpdate* равен TRUE, то после записи текста область помечается как поврежденная для ее немедленного обновления.

```
void CStatusBar::GetPaneText(
   int nIndex,
   CString &rString)
```

и

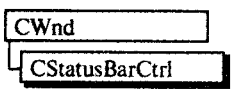
```
CString CStatusBar::GetPaneText(int nIndex) —
```

позволяют получить текст, который находится в области с номером *nIndex*.

```
CStatusBarCtrl& CStatusBar::GetStatusBarCtrl() —
```

возвращает ссылку на объект общего элемента управления STATUSBAR, для прямого доступа к ее функциональным возможностям, включающим, в частности, настройку строки состояния.

Класс CStatusBarCtrl



Для полного использования функциональных возможностей общего элемента управления STATUSBAR можно пойти двумя путями: либо непосредственно создать окно *CStatusBarCtrl*, либо создать строку состояния на основе класса *CStatusBar*, а затем просто получить доступ к элементам объекта *CStatusBarCtrl*, вызвав функцию этого класса *CStatusBar::CStatusBarCtrl*. Остановимся на втором способе и не будем рассматривать непосредственное создание общего элемента управления STATUSBAR.

Для того чтобы прочитать текст из какой-либо области элемента управления, в классе реализована функция *GetText*:

```
int CStatusBarCtrl::GetText(
   LPCTSTR lpszText,
   int nPane,
   int *pType)
```

где *nPane* — индекс области, из которой читается текст; *lpszText* — указатель на буфер, в который этот текст будет записан; *pType* — указатель на целое значение, в котором будет возвращена информация о типе отображения текста.

Сама функция возвращает длину текста, которую можно также получить при помощи функции

```
int CStatusBarCtrl::GetTextLength(
   int nPane,
   int *pType)
```

В любую область строки состояния можно записать произвольную текстовую или, как ни странно, графическую информацию, воспользовавшись для этого функцией

BOOL CStatusBarCtrl::**SetText**(
LPCTSTR lpszText,
int nPane,
int nType)

где nPane — индекс области, в которую записывается текст из lpszText. Если nPane = 255, то строка состояния содержит единственную область, в которую можно выводить только текст. Параметр nType задает тип отображения текста и может принимать одно из значений:

0	текст выводится с рамкой, которая выглядит утопленной
SBT_NOBORDERS	текст выводится без рамки
SBT_OWNERDRAW	текст (или графическое изображение) выводится родительским окном во время обработки сообщения WM_DRAWITEM
SBT_POPOUT	текст выводится с рамкой, которая выглядит приподнятой

Если *nType* принимает значение SBT_OWNERDRAW, то в параметре *lpszText* передаются 32 бита данных. При выполнении этой функции область, в которую выводится текст, становится областью модификации, чтобы отобразить новый текст при получении сообщения WM_PAINT.

Для того чтобы разбить элемент управления на области определенных размеров, можно воспользоваться функцией

BOOL CStatusBarCtrl::**SetParts**(
int nParts,
int *pWidths)

которая устанавливает число областей и координату правого края каждой из них. Число областей (nParts) не может быть больше 255. Параметр pWidths содержит адрес целочисленного массива, каждый элемент которого определяет позицию правого края соответствующей области. Если элемент равен –1, то правый край этой области совпадает с правым краем элемента управления.

Обратную задачу позволяет решить функция

int CStatusBarCtrl::**GetParts**(
int nParts,
int *pParts)

которая возвращает число областей, на которое разбит элемент управления, а также координаты правых краев областей, число которых запрашивается в параметре nParts. Если это число больше, чем имеется у элемента управления, то информация возвращается только для существующих областей. Координаты записываются в целочисленный массив pParts, число элементов которого равно nParts. Если для какой-либо области в качестве координаты правого края области в массив записывается –1, то это означает, что он совпадает с правым краем элемента управления.

Если требуется информация о размерах рамок элемента управления и его областей, то ее можно получить при помощи функции *GetBorders*, которая реализована в двух вариантах.

BOOL CStatusBarCtrl::**GetBorders**(
int &nHorz,
int &nVert,
int &nSpacing)

и

BOOL CStatusBarCtrl::**GetBorders**(int *pBorders) –

Первая возвращает ширину горизонтальной (nHorz) и вертикальной (nVert) рамки, а также рамки между различными областями (nSpacing). Вторая записывает всю эту информацию в целочисленный трехэлементный массив. Эти расстояния определяют интервал между внешним краем элемента управления и прямоугольником, внутри которого находится текстовая информация.

Помимо изменения горизонтальных размеров различных областей строки состояния, можно установить ее минимальную высоту, необходимую для отображения текстовой или графической информации:

void CStatusBarCtrl::**SetMinHeight**(int nMin) –

минимальная высота определяется суммой значения параметра nMin и двойной ширины рамки.

Для переключения между упрощенным и стандартным режимами работы строки состояния используется функция

BOOL CStatusBarCtrl::**SetSimple**(BOOL bSimple = TRUE) –

если bSimple равен TRUE, то элемент управления переводится в упрощенный режим, в противном случае – в стандартный. При переходе из одного режима в другой происходит немедленная перерисовка строки состояния. Содержимое областей при переключении не теряется, т. к. данные соответствующих режимов хранятся в различных местах.

BOOL CStatusBarCtrl::**GetRect**(
int nPane,
LPRECT lpRect) –

эта функция служит для получения размеров и координат прямоугольника, ограничивающего область nPane элемента управления. Информация о прямоугольнике записывается в структуру *RECT*, на которую указывает lpRect.

Как уже упоминалось, в строке состояния может находиться не только текстовая, но и графическая информация. Для реализации этой возможности необходимо сделать две вещи: при вызове функции *CStatusBarCtrl::SetText* указать константу SBT_OWNERDRAW и переопределить функцию

virtual void CStatusBarCtrl::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct) –

которая вызывается родительским окном, когда область элемента управления должна быть перерисована. Реализация этой функции по умолчанию не делает ничего. Указатель на структуру *DRAWITEMSTRUCT*, передаваемый в качестве параметра функции, содержит всю информацию необходимую для рисования.

Рассмотрим поля структуры *DRAWITEMSTRUCT*, применительно к элементу управления "строка состояния":

```
typedef struct tagDRAWITEMSTRUCT{
  UINT CtlType;       // не используется
  UINT CtlID;
  UINT itemID;
  UINT itemAction;    // не используется
  UINT itemState;     // не используется
  HWND hwndItem;
  HDC hDC;
  RECT rcItem;
  DWORD itemData;
} DRAWITEMSTRUCT;
```

Поле *hwndItem* содержит дескриптор окна строки состояния, поле *CtlId* — ее идентификатор, а поле *itemID* — номер области, которая будет перерисована. Контекст устройства, необходимый для выполнения операций рисования, находится в поле *hDC*, а координаты области, в которой будет осуществляться рисование, передаются через поле *rcItem*.

Кроме того, можно воспользоваться содержимым поля *itemData*, которое равно значению параметра *lpszText* функции *CStatusBarCtrl::SetText*. Например, через этот параметр можно передать идентификатор изображения, которое должно быть нарисовано в данной области строки состояния, или идентификатор шрифта, которым будет написан текст, или какую-либо другую информацию.

В заключение нам осталось описать, как строка состояния обрабатывает команду ON_UPDATE_COMMAND_UI. Как обычно, обработчик этой команды вызывается во время цикла простоя последовательно для всех областей строки состояния. Доступными функциями класса *CCmdUI*, отвечающего за обработку, являются *Enable* и *SetText*. Первая из них блокирует или разблокирует соответствующую область. При этом надо иметь в виду, что у заблокированных областей выключается отображение текста. Вторая функция позволяет изменить текст, который выводится в текущей области. Следует помнить, что при выводе текста не происходит автоматического изменения размеров области.

На этом мы заканчиваем рассмотрение наиболее часто используемых панелей элементов управления — панели инструментов и строки состояния — и переходим к описанию не менее интересного класса *CDialogBar*.

Класс CDialogBar

Панель диалога — панель элементов управления с функциональными возможностями немодального блока диалога, которая содержит стандартные элементы управления Windows. Она действует подобно блоку диалога,

поддерживая, в частности, перемещение между элементами при помощи клавиши <Tab>. Кроме того, для нее, как и для блока диалога, используется шаблон диалога. Это с одной стороны. А с другой, ее можно рассматривать как более мощную панель инструментов. Извещения от элементов управления панели диалога посылаются ее родительскому окну точно так же, как и от кнопок панели инструментов.[1] Основное их различие заключается в том, что панель диалога может содержать произвольные элементы управления, в то время как панель инструментов, стандартно, — только кнопки. Панель диалога наиболее часто используется в качестве дополнения к основному окну, и любые извещения, такие как BN_CLICKED или EN_CHANGE будут посланы ее родителю — главному окну приложения.

Класс *CDialogBar* имеет всего две основные функции: конструктор

```
CDialogBar::CDialogBar()
```

который создает объект этого класса, и функцию *Create*, как обычно, отвечающую за создание окна Windows и присоединение его к текущему объекту:

```
BOOL CDialogBar::Create(
  CWnd *pParentWnd,
  LPCTSTR lpszTemplateName,
  UINT nStyle,
  UINT nID)
```

и

```
BOOL CDialogBar::Create(
  CWnd *pParentWnd,
  UINT nIDTemplate,
  UINT nStyle,
  UINT nID)
```

Функция имеет следующие параметры: *pParentWnd* — указатель на родительское окно (не может быть равен NULL); *lpszTemplateName* — указатель на строку, содержащую имя ресурса шаблона панели диалога; второй вариант функции вместо имени использует идентификатор ресурса *nIDTemplate*; *nID* — идентификатор панели диалога; *nStyle* — стиль выравнивания панели диалога:

CBRS_TOP	панель диалога располагается у верхней стороны фрейма
CBRS_BOTTOM	панель диалога располагается у нижней стороны фрейма
CBRS_LEFT	панель диалога располагается у левой стороны фрейма
CBRS_RIGHT	панель диалога располагается у правой стороны фрейма

[1] Однако *ClassWizard* ничем помочь, увы, не может — он ничего "не знает" об элементах управления, указанных в шаблоне.

CBRS_NOALIGN панель диалога не меняет своего расположения, когда у родительского окна изменяется размер

Эта функция (в любом варианте) загружает шаблон диалога из ресурса, заданного параметром *lpszTemplateName* или *nIDTemplate*, создает окно панели диалога, устанавливает его стиль и ассоциирует с объектом *CDialogBar*. Если определен стиль CBRS_TOP или CBRS_BOTTOM, то ширина панели определяется шириной фрейма, а высота берется из шаблона *nIDTemplate*. Если же определен стиль CBRS_LEFT или CBRS_RIGHT, то высота панели определяется высотой фрейма, а ширина берется из шаблона *nIDTemplate*.

Все остальные функциональные возможности этот класс получает от своего базового *CControlBar*.

ControlBars — пример приложения

Как известно, лучше один раз увидеть, чем сто раз услышать. Перейдем к примеру использования, конечно не всех, но хотя бы части заявленных возможностей. Сначала несколько слов о составе примера. Весь код приложения, окно которого представлено на рис. 27, занимает более 800 строк, не считая добавленных нами комментариев, плюс еще три файла для хранения битовых массивов панелей инструментов и файл пиктограммы приложения. Мы ограничились лишь двумя файлами исходного текста на C++ — этого больше чем достаточно.

Все приложение состоит из описания двух классов: неизменного класса приложения (*CControlBarsApp*) и его главного окна (*CMainFrame*).

```
////////////////////////////////////////////////////////////
// Файл ControlBars.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

...

////////////////////////////////////////////////////////////
// Инициализация CControlBarsApp

BOOL CControlBarsApp::InitInstance()
{
  ...
  // создание нового SDI-окна
  CFrameWnd* pMainFrame = new CMainFrame;
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;

  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();
```

```
  m_pMainWnd = pMainFrame;
  ...
}
...
////////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////////

#ifndef __AFXEXT_H__
#include <afxext.h>          // для включения объявлений классов
#endif                       // CToolBar и CStatusBar

class CMainFrame : public CFrameWnd
{
// конструктор
public:
  CMainFrame();

// операции
public:
  void SetColumns(UINT nColumns); // изменение кол-ва столбцов
                                  // панели инструментов рисования

public:
  virtual ~CMainFrame();
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

  // названия функций говорят сами за себя
  BOOL CreateToolBar();
  BOOL CreatePaneBar();
  BOOL CreateStatusBar();

protected:

  // встроенные объекты "панель элементов управления"
  CStatusBar m_wndStatusBar;  // строка состояния
  CToolBar   m_wndToolBar;    // панель инструментов (команд)
  CToolBar   m_wndPaneBar;    // панель инструментов рисования
  CDialogBar m_wndDlgBar;     // панель диалога

  // дополнительные члены класса
  UINT       m_nPaneCol;      // кол-во столбцов панели инструментов
                              // рисования
  CComboBox  m_comboBox;      // комбинированный список для панели
                              // инструментов (команд)
  CFont      m_font;          // шрифт для комбинированного списка
```

```
// обработчики сообщений
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct); // создание
                                                       // окна
  afx_msg void OnViewPane(); // команда отображения панели
                             // инструментов рисования
  afx_msg void OnUpdateViewPane(CCmdUI* pCmdUI);
  afx_msg void OnSelChangePane(); // изменение выбора
                                  // инструмента рисования
  afx_msg void OnPane2column(); // команды смены количества
  afx_msg void OnPane3column(); // столбцов панели инструментов
                                // рисования
  afx_msg void OnUpdatePane2column(CCmdUI* pCmdUI);
  afx_msg void OnUpdatePane3column(CCmdUI* pCmdUI);
  afx_msg void OnUpdateIndicatorIdle(CCmdUI* pCmdUI);  // обновление
                                                  // области строки состояния
  afx_msg void OnTimer(UINT nIDEvent); // сообщения от таймера
  //}}AFX_MSG

// смена выбора для комбинированных списков
  afx_msg void OnStyle(UINT nID); // список стилей выравнивания
  afx_msg void OnUpdateStyle(CCmdUI* pCmdUI);
  afx_msg void OnPane(UINT nID); // список инструментов рисования
  afx_msg void OnUpdatePane(CCmdUI* pCmdUI);

  DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "ControlBars.h"

#include "MainFrm.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////
// CMainFrame

// связываем функции с командами и сообщениями
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_WM_CREATE()
```

```
  ON_COMMAND(IDM_VIEWPANE, OnViewPane)
  ON_UPDATE_COMMAND_UI(IDM_VIEWPANE, OnUpdateViewPane)
  ON_CBN_SELCHANGE(IDC_PANE, OnSelChangePane)
  ON_COMMAND(IDM_PANE_2COLUMN, OnPane2column)
  ON_COMMAND(IDM_PANE_3COLUMN, OnPane3column)
  ON_UPDATE_COMMAND_UI(IDM_PANE_3COLUMN, OnUpdatePane3column)
  ON_UPDATE_COMMAND_UI(IDM_PANE_2COLUMN, OnUpdatePane2column)
  ON_UPDATE_COMMAND_UI(ID_INDICATOR_IDLE, OnUpdateIndicatorIdle)
  ON_WM_TIMER()
  //}}AFX_MSG_MAP
  ON_COMMAND_RANGE(ID_STYLE_LEFT, ID_STYLE_JUSTIFIED, OnStyle)
  ON_UPDATE_COMMAND_UI_RANGE(ID_STYLE_LEFT, ID_STYLE_JUSTIFIED,
OnUpdateStyle)
  ON_COMMAND_RANGE(ID_PANEERASE, ID_PANEOVAL, OnPane)
  ON_UPDATE_COMMAND_UI_RANGE(ID_PANEERASE, ID_PANEOVAL,
                             OnUpdatePane)
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////
// массивы идентификаторов для инициализации панелей элементов
// управления
static UINT BASED_CODE toolbar[] =
{
  // тот же порядок, что и в файле toolbar.bmp
  ID_FILE_NEW,
  ID_FILE_OPEN,
  ID_FILE_SAVE,
    ID_SEPARATOR,
  ID_EDIT_CUT,
  ID_EDIT_COPY,
  ID_EDIT_PASTE,
    ID_SEPARATOR,
  ID_SEPARATOR,           // резервируем для элементов COMBO BOX
    ID_SEPARATOR,
  ID_FILE_PRINT,
  ID_APP_ABOUT,
  ID_CONTEXT_HELP
};
static UINT BASED_CODE pane[] =
{
  // тот же порядок, что и в файле pane.bmp
  ID_PANEERASE,
  ID_PANEPEN,
  ID_PANESELECT,
  ID_PANEBRUSH,
  ID_PANESPRAY,
  ID_PANEPAINT,
```

```
  ID_PANELINE,
  ID_PANEEYEDROP,
  ID_PANEMAG,
  ID_PANERECT,
  ID_PANEROUND,
  ID_PANEOVAL
};

static UINT BASED_CODE indicators[] =
{
  ID_SEPARATOR,           // индикатор выбираемой команды
  ID_INDICATOR_NUM,
  ID_INDICATOR_IDLE,
  ID_INDICATOR_TIME
};

/////////////////////////////////////////////////////////////////
// CMainFrame

CMainFrame::CMainFrame()
{
  m_nPaneCol = 3;
}

CMainFrame::~CMainFrame()
{
}

// перегружаем функцию PreCreateWindow для задания своего класса
// окна, задавая иной фон и отказываясь от стилей
// CS_VREDRAW/CS_HREDRAW, чтобы исключить перерисовку панелей
// элементов управления при изменении размеров главного окна
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszClass = AfxRegisterWndClass(CS_DBLCLKS,
    AfxGetApp()->LoadStandardCursor(IDC_ARROW),
                                    (HBRUSH)(COLOR_WINDOW+1),
    AfxGetApp()->LoadIcon(IDR_MAINFRAME));

  return TRUE;
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  if (!CreateToolBar())
    return -1;
  if (!CreatePaneBar())
    return -1;
```

```
  if (!CreateStatusBar())
    return -1;

  if (!m_wndDlgBar.Create(this, IDD_VIEWSELECT,
    CBRS_LEFT|CBRS_TOOLTIPS|CBRS_FLYBY, IDD_VIEWSELECT))
  {
    TRACE0("Ошибка при создании панели \n");
    return -1;
  }

  // разрешаем привязку панели инструментов
  EnableDocking(CBRS_ALIGN_ANY);

  m_wndPaneBar.EnableDocking(CBRS_ALIGN_ANY);
  DockControlBar(&m_wndPaneBar, AFX_IDW_DOCKBAR_TOP);

  // размещаем панель инструментов в задаваемом месте экрана
  CPoint pt(::GetSystemMetrics(SM_CXSCREEN) - 200,
    ::GetSystemMetrics(SM_CYSCREEN) / 3);

  FloatControlBar(&m_wndPaneBar, pt);

  // создаем таймер для смены содержимого области строки состояния
  SetTimer(0, 1000, NULL);

  return 0;
}

// разбивка панели на столбцы - находим
// конечные кнопки каждой строки
void CMainFrame::SetColumns(UINT nColumns)
{
  int nCount = m_wndPaneBar.GetToolBarCtrl().GetButtonCount();

  for (int i = 0; i < nCount; i++)
  {
    UINT nStyle = m_wndPaneBar.GetButtonStyle(i);
    BOOL bWrap = (((i + 1) % nColumns) == 0);
    if (bWrap)
      nStyle |= TBBS_WRAPPED;
    else
      nStyle &= ~TBBS_WRAPPED;
    m_wndPaneBar.SetButtonStyle(i, nStyle);
  }

  m_wndPaneBar.Invalidate();
  m_wndPaneBar.GetParentFrame()->RecalcLayout();
}

BOOL CMainFrame::CreateToolBar()
{
  const int nDropHeight = 100;
```

```
// вставлять комбинированный список легче, если панель инструментов
// или любой другой элемент управления формируется на основе
// массива команд, когда место для элемента заранее зарезервировано
if (!m_wndToolBar.Create(this, WS_CHILD|WS_VISIBLE|CBRS_TOP|
    CBRS_TOOLTIPS|CBRS_FLYBY) ||
  !m_wndToolBar.LoadBitmap(IDB_TOOLBAR) ||
  !m_wndToolBar.SetButtons(toolbar, sizeof(toolbar)/sizeof(UINT)))
{
  TRACE0("Ошибка при создании панели\n");
  return FALSE;       // fail to create
}

//
// создание комбинированного списка
//

// задаем ширину для списка и связываем его
// идентификатор с местом в панели
m_wndToolBar.SetButtonInfo(8, IDW_COMBO, TBBS_SEPARATOR, 100);

// Microsoft требует (и не следует этому требованию сама!), чтобы
// расстояние между списками и кнопками было 12 пикселей в ширину
m_wndToolBar.SetButtonInfo(7, ID_SEPARATOR, TBBS_SEPARATOR, 12);
m_wndToolBar.SetButtonInfo(9, ID_SEPARATOR, TBBS_SEPARATOR, 12);

// собственно о создании различных элементов управления
// читайте 2-ой том
CRect rect;
m_wndToolBar.GetItemRect(8, &rect);
rect.top = 2;
rect.bottom = rect.top + nDropHeight;
if (!m_comboBox.Create(CBS_NOINTEGRALHEIGHT|
    CBS_DROPDOWNLIST|WS_VISIBLE|WS_TABSTOP,
    rect, &m_wndToolBar, IDW_COMBO))
{
  TRACE0("Ошибка при создании комбинированного списка\n");
  return FALSE;
}

m_comboBox.AddString(_T("Влево"));
m_comboBox.AddString(_T("По ценру"));
m_comboBox.AddString(_T("Вправо"));
m_comboBox.AddString(_T("По ширине"));

// Про работу со шрифтами тоже говорить будем не скоро,
// пока поверьте на слово
LOGFONT logFont;
memset(&logFont, 0, sizeof(logFont));

if (!::GetSystemMetrics(SM_DBCSENABLED))
{
  logFont.lfHeight = -12;
```

```
    logFont.lfWeight = FW_BOLD;
    logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
    lstrcpy(logFont.lfFaceName, _T("MS Sans Serif"));
    if (!m_font.CreateFontIndirect(&logFont))
      TRACE0("Невозможно создание шрифта для списка\n");
    else
      m_comboBox.SetFont(&m_font);
  }
  else
  {
    m_font.Attach(::GetStockObject(SYSTEM_FONT));
    m_comboBox.SetFont(&m_font);
  }
return TRUE;
}
BOOL CMainFrame::CreatePaneBar()
{
  if (!m_wndPaneBar.Create(this, WS_CHILD |
      WS_VISIBLE | CBRS_SIZE_FIXED |
      CBRS_TOP | CBRS_TOOLTIPS, ID_PANEBAR) ||
      !m_wndPaneBar.LoadBitmap(IDB_PANE) ||
      !m_wndPaneBar.SetButtons(pane,
      sizeof(pane)/sizeof(UINT)))
  {
    TRACE0("Ошибка при создании панели\n");
    return FALSE;
  }
  //
  // добавляем к рисункам на кнопках текстовые строки
  //
  // узнаем и запоминаем размеры рисунков
  CRect imageRect;
  m_wndPaneBar.GetItemRect(0, &imageRect);
  // добавляем по строке на каждую кнопку,
  // если строк будет меньше, то все оставшиеся кнопки
  // будут "осчастливлены" ПЕРВОЙ ЗАДАННОЙ СТРОКОЙ !!!
  for (int index = ID_PANEERASE; index <= ID_PANEOVAL; index++)
  {
    CString szPane;
    szPane.LoadString(index);

    // строки хранятся в следующем виде:
    //   - сообщение для строки состояния о возможном выборе,
    //   - символ '\n',
    //   - краткое содержание всплывающей подсказки,
    // именно последнее мы и используем
```

```
    m_wndPaneBar.SetButtonText(index - ID_PANEERASE,
      szPane.Mid(szPane.Find('\n')+1));
  }

  // после каждой вставки текста автоматически происходило изменение
  // размеров панели, но Windows об этом "знает", а объект класса
  // CToolBar нет; сообщим ему об этом, памятуя, что размеры
  // рисунков остались без изменений;
  // константы 7 и 6 - требования Microsoft
  CRect buttonRect;
  m_wndPaneBar.GetItemRect(0, &buttonRect);
  m_wndPaneBar.SetSizes(CSize(buttonRect.Width(), button-
Rect.Height()),
    CSize(imageRect.Width()-7, imageRect.Height()-6));

  // заголовок будущего окна для панели в плавающем режиме
m_wndPaneBar.SetWindowText(_T("Инструменты"));

  // сразу настраиваем количество столбцов
  SetColumns(m_nPaneCol);

  return TRUE;
}

BOOL CMainFrame::CreateStatusBar()
{
  if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
  {
    TRACE0("Ошибка при создании панели \n");
    return FALSE;
  }

  return TRUE;
}

...

////////////////////////////////////////////////////////////
// Обработчики сообщений класса CMainFrame

// управление отображением панели инструментов рисования
void CMainFrame::OnViewPane()
{
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);

  ShowControlBar(&m_wndPaneBar, !bVisible, FALSE);
  RecalcLayout();
}

void CMainFrame::OnUpdateViewPane(CCmdUI* pCmdUI)
```

```
{
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);
  pCmdUI->SetCheck(bVisible);
}
// обработчик изменения выбора инструмента рисования
void CMainFrame::OnSelChangePane()
{
  CString strText;
  CString strItem;
  CComboBox* pCBox = (CComboBox*)m_wndDlgBar.GetDlgItem(IDC_PANE);
  int nIndex = pCBox->GetCurSel();
  if (nIndex == CB_ERR)
    return;
  pCBox->GetLBText(nIndex, strItem);
  AfxFormatString1(strText, IDS_SELECTED_PROMPT, (LPCTSTR)strItem);
  SetMessageText(strText);
}
// обработчики команд смены количества
// столбцов панели инструментов рисования
void CMainFrame::OnPane2column()
{
  SetColumns(m_nPaneCol = 2);
}
void CMainFrame::OnPane3column()
{
  SetColumns(m_nPaneCol = 3);
}
void CMainFrame::OnUpdatePane2column(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_nPaneCol == 2);
}
void CMainFrame::OnUpdatePane3column(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_nPaneCol == 3);
}
// каждый раз меняем символ в области строки состояния,
// кстати, эта функция при отсутствии каких-либо действий
// вызывается примерно раз в секунду, а при манипуляциях с мышью -
// практически постоянно
void CMainFrame::OnUpdateIndicatorIdle(CCmdUI* pCmdUI)
{
  static BOOL bIdle = FALSE;
  pCmdUI->SetText(bIdle ? " " : "\x95");
  bIdle = !bIdle;
}
```

```
// обработчик сообщений от таймера
void CMainFrame::OnTimer(UINT nIDEvent)
{
  // класс CTime будет рассмотрен во втором томе
  m_wndStatusBar.SetPaneText(m_wndStatusBar.CommandToIndex(
                              ID_INDICATOR_TIME),
    CTime::GetCurrentTime().Format("%H:%M:%S"));

  CFrameWnd::OnTimer(nIDEvent);
}

// обработчик смены выбора стилей выравнивания
void CMainFrame::OnStyle(UINT nID)
{
  m_comboBox.SetCurSel((int)(nID-ID_STYLE_LEFT));
}

void CMainFrame::OnUpdateStyle(CCmdUI* pCmdUI)
{
  UINT nIndex = m_comboBox.GetCurSel();
  pCmdUI->SetCheck((UINT)(ID_STYLE_LEFT+nIndex) == pCmdUI->m_nID);
}

// обработчик смены выбора инструментов рисования
void CMainFrame::OnPane(UINT nID)
{
  CComboBox* pCBox = (CComboBox*)m_wndDlgBar.GetDlgItem(IDC_PANE);
  pCBox->SetCurSel((int)(nID - ID_PANEERASE));
  OnSelChangePane();
}

void CMainFrame::OnUpdatePane(CCmdUI* pCmdUI)
{
  CComboBox* pCBox = (CComboBox*)m_wndDlgBar.GetDlgItem(IDC_PANE);
  UINT nIndex = pCBox->GetCurSel();
  pCmdUI->SetCheck((UINT)(ID_PANEERASE + nIndex) == pCmdUI->m_nID);
}

////////////////////////////////////////////////////////////
// Файл ControlBars.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////

...

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

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
```

```
BEGIN
  POPUP "Файл"
  BEGIN
    MENUITEM "Выход",                       ID_APP_EXIT
  END
  POPUP "Вид"
  BEGIN
    MENUITEM "Панель инструментов",         ID_VIEW_TOOLBAR
    MENUITEM "Инструменты рисования",       IDM_VIEWPANE, CHECKED
    MENUITEM "Строка состояния",            ID_VIEW_STATUS_BAR
  END
  POPUP "Стиль"
  BEGIN
    MENUITEM "Влево",                       ID_STYLE_LEFT
    MENUITEM "По центру",                   ID_STYLE_CENTERED
    MENUITEM "Вправо",                      ID_STYLE_RIGHT
    MENUITEM "По ширине",                   ID_STYLE_JUSTIFIED
  END
  POPUP "Инструмент"
  BEGIN
    MENUITEM "2 столбца",                   IDM_PANE_2COLUMN
    MENUITEM "3 столбца",                   IDM_PANE_3COLUMN
  END
  POPUP "?"
  BEGIN
    MENUITEM "О программе...",              ID_APP_ABOUT
  END
END

////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VIEWSELECT DIALOG DISCARDABLE  0, 0, 69, 84
STYLE WS_CHILD
FONT 8, "MS Sans Serif"
BEGIN
  GROUPBOX       "",IDC_STATIC,1,14,66,34
  CONTROL        "Команды",ID_VIEW_TOOLBAR,"Button",BS_AUTOCHECKBOX |
                  WS_TABSTOP,5,25,59,10
  CONTROL
"Инструменты",IDM_VIEWPANE,"Button",BS_AUTOCHECKBOX |
                  WS_TABSTOP,5,35,60,10
  COMBOBOX        IDC_PANE,2,55,65,50,CBS_DROPDOWNLIST | WS_VSCROLL
END

...
```

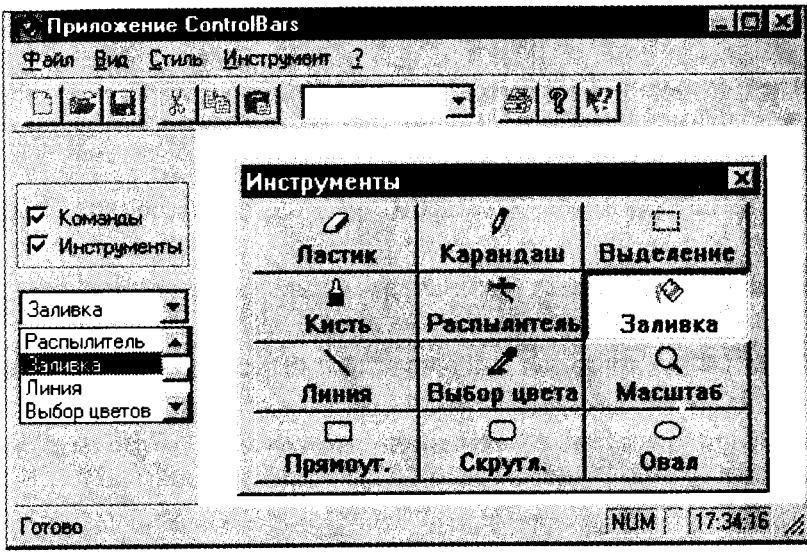


Рис. 27. Главное окно приложения ControlBars

Все функции с префиксом *OnUpdate* используются во время простоя, т. е. в момент, когда очередь сообщений потока пуста. Все такие функции расположены в объявлении класса за функциями обработки команд или событий, и позволяют воздействовать на соответствующие им элементы управления.

В главное окно приложения помещены следующие панели элементов управления:

- панель инструментов (команд) — привязанная к верхней части окна без возможности перемещения. В нее вставлен комбинированный список для задания стиля выравнивания (обработчик его сообщений — функция *CreateToolBar*), выбор элемента которого отслеживается функцией *OnStyle*;
- панель инструментов рисования — плавающая с возможностью привязки. Все кнопки имеют кроме рисунка еще и текст (функция *CreatePaneBar*). Выбор одного из инструментов сопровождается вызовом функции *OnPane*;
- панель диалога — привязанная к левой части окна без возможности перемещения. Содержит элементы управления: статическая рамка, два флажка для отображения/скрытия панели инструментов (команд)[1] и панели инструментов рисования (функция *OnViewPane*), и, наконец, комбинированный список отображения/выбора инструмента рисования, выбор элемента которого отслеживается функцией *OnSelChangePane*;

---

[1] Идентификатор команды для управления режимом показа панели инструментов (команд) равен ID_VIEW_TOOLBAR, который интерпретируется как команда и управляет панелями с идентификатором AFX_IDW_TOOLBAR, так что специального кода создавать не надо — все делает библиотека классов MFC.

- строка состояния — состоящая из четырех областей: для подсказки, отображения режима NUMLOCK, для демонстрации обработки во время простоя (функция *OnUpdateIndicatorIdle*) и для отображения текущего времени примерно раз в секунду по сигналу от таймера (функция *OnTimer*).

Осталось рассмотреть меню, в котором имеются, кроме общих, следующие пункты:

- меню "Вид" — управляет отображением/скрытием панелей управления (кроме панели диалога) и строкой состояния; [1]
- меню "Стиль" — управляет выбором стиля выравнивания. Используется, как и в случае панели инструментов (команд), функция *OnStyle*;
- меню "Инструмент" — обеспечивает задание количества столбцов для отображения панели инструментов рисования (функции *CPane2colomn* и *CPane3colomn*).

Часть VI

Блоки диалога

14. Модальные и немодальные блоки диалога

15. Стандартные блоки диалога

Глава 14

Модальные и немодальные блоки диалога

Блоки диалога (dialog boxes) — специальные окна, поддерживающие гибкие средства, используя которые пользователь может взаимодействовать с программой. Как правило, они включают *дочерние* (child) окна, которые в контексте этого типа окон называются *элементами управления.*

Блоки диалога бывают двух типов — *модальные* (modal) и *немодальные* (modeless). Разница между ними заключается в способе управления потоками сообщений. Модальные блоки диалога блокируют все остальные окна приложения так, что пользователь не может с ними ничего сделать, пока не закроет его. Другими словами, они отсекают поток сообщений, идущих от мыши и клавиатуры к его родительским окнам или окнам того же уровня (sibling), делая их недоступными. Если пользователь все же пытается взаимодействовать с недоступным окном, то система предупреждает об этом звуковым сигналом. Блоки диалога этого типа обычно не препятствуют переключению с одного приложения на другое. В редких случаях можно использовать системные модальные блоки диалога, которые позволяют пользователю работать внутри лишь одного блока диалога и не позволяют переключаться ни на какие другие приложения.

Немодальные блоки диалога больше похожи на обычные окна, т. к. они дают пользователю возможность доступа к остальным окнам приложения. Это связано с тем, что блоки диалога этого типа не прерывают потоков сообщений, идущих в любую часть вашей программы.

Оба типа блоков диалога являются важнейшей частью пользовательского интерфейса Windows, и уже давно сложился стандартный набор элементов управления, широко используемых в блоках диалога.[1]

---

[1] Наряду с уже привычными элементами управления Windows: кнопками (PUSHBUTTON), флажками (CHECK BOX), переключателями (RADIO BUTTON), списками (LIST BOX, элементами редактирования (EDIT), комбинированными списками (COMBO BOX), полосами прокрутки (SCROLLBAR) и статическими элементами (STATIC) в Win32 добавлено достаточно большое число новых, которые перечислены в первой части книги и будут подробно рассмотрены ниже.

На первый взгляд немодальные блоки диалога более привлекательны, поскольку дают пользователю свободу выбора способа действия. Однако, большинство блоков диалога все-таки модальные. Это связано с тем, что модальный блок диалога идеально концентрирует внимание пользователя на некотором ограниченном наборе действий, которые он может в данный момент выполнить.

Создание и использование блока диалога, модального и немодального, требует одновременного наличия трех компонентов:

- шаблона блока диалога,
- программного кода, создающего и отображающего его на экране,
- диалоговой процедуры, которая обслуживает взаимодействие пользователя с блоком диалога.

Начнем с шаблона, т. к. без него не будет и самого блока диалога.

Шаблон блока диалога представляет собой объект данных, определенный в файле ресурсов (.rc), аналогичный приведенному ниже:

```
// Блок диалога About
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "О программе MDI"
FONT 8, "MS Sans Serif"
BEGIN
  ICON  IDR_MAINFRAME,IDC_STATIC,7,7,21,20
  CTEXT "MDI-приложение Версия 1.0",
    IDC_STATIC,51,17,119,8, SS_NOPREFIX
  CTEXT  "Copyright © 1997",IDC_STATIC,127,34,71,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
  LTEXT  "Мешков А., Тихомиров Ю.",IDC_STATIC,35,34,89,8
END
```

Один из способов создания шаблона блока диалога — посредством редактирования файла ресурсов. Другой способ заключается в использовании какого-либо визуального средства, такого как App Studio или ему подобного, которое позволяет изобразить предварительный набросок блока диалога (рис. 28).

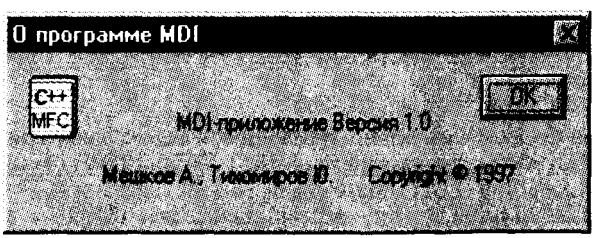


Рис. 28. Эскиз блока диалога

Но какой бы способ вы не выбрали, помните, что блок диалога это всего-навсего еще один ресурс. Внимательно посмотрев на приведенный текст, вы заметите, что первая строчка содержит идентификатор ресурса блока диалога (IDD_ABOUTBOX), за которым следует объявление типа ресурса (DIALOG) и размеров блока диалога. Следующая строчка определяет его стиль, объединяя четыре константы, в основном уже знакомые вам. После того как объявлен стиль, можно указать заголовок блока диалога (CAPTION), который появляется в его полосе заголовка. Если не определять этот оператор, то, естественно, полоса останется пустой.[1] Следующая строчка описания ресурса (FONT) определяет размер и имя шрифта, который будет использоваться в этом блоке диалога.

Описание блока диалога заканчивается несколькими строчками, заключенными между BEGIN и END. Они определяют пиктограмму, две строки текста и кнопку. Как мы говорили, все они являются элементами управления, т. е. окнами. Другими словами, наш блок диалога имеет четыре дочерних окна, размещенных внутри него: кнопку, два статических элемента управления и пиктограмму.

Строчка, описывающая кнопку, начинается с оператора, определяющего тип элемента управления (DEFPUSHBUTTON) и имеет 7 параметров. Первый — текст, ассоциирующийся с элементом управления, за тем идут идентификатор, сектор размещения и стиль. Сектор размещения задается четырьмя числами: координатами левого верхнего угла, шириной и высотой.

Поскольку и пиктограмма (ICON) и статические элементы (LTEXT и CTEXT) являются окнами, то неудивительно, что они имеют такие же параметры, что и кнопка. Некоторые стили элементов управления указываются непосредственно в названии. Например, DEFPUSHBUTTON говорит о том, что у кнопки установлен стиль BS_DEFPUSHBUTTON, а LTEXT — о том, что элемент управления имеет установленный флаг стиля ES_LEFT. Вместо конкретных операторов можно указывать также оператор CONTROL, который является общим для различных элементов управления и имеет 8 параметров:

```
CONTROL <текст>, nID, <класс>, x, y, cx, cy, <стили>
```

где к уже перечисленным параметрам добавляется символьная строка, определяющая имя оконного класса, с помощью которого должен быть создан элемент управления блока диалога.

Итак, формирование шаблона блока диалога включает создание объекта ресурсов, который описывает форму, размер и расположение на экране блока диалога и каждого из его внутренних элементов управления. Данный объект помещается в файл ресурсов приложения вместе с другими объекта-

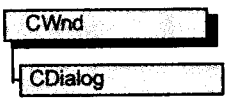
---

[1] Если при определении блока диалога вы не используете стиль WS_CAPTION, то не будет полосы заголовка.

ми (пиктограммами, меню, битовыми образами и т. д.). После того как шаблон сформирован, можно приступать к созданию программного кода, способного вызывать появление блока диалога на экране и его последующее удаление.

Как обычно, библиотека MFC предоставляет специальный класс для создания и работы с блоками диалогов — *CDialog*. Следуя принятому нами стилю изложения рассмотрим этот класс подробнее.

Класс CDialog



Основное назначение этого класса состоит в предоставлении интерфейса программирования для управления блоками диалога. Использование класса *CDialog* позволяет создавать объекты обоих типов блоков диалога — модальные и немодальные. Повторим еще раз: блок диалога подобен любому другому окну Windows, но в отличие от других типов окон, объект этого класса является комбинацией класса *CDialog* или производного от него и шаблона диалога. Два способа создания шаблона мы уже рассмотрели, а к третьему — создание шаблона в памяти, вернемся несколько позже. Теперь же познакомимся непосредственно с самим классом *CDialog*.

Для создания объекта модального блока диалога на основе шаблона, хранящегося в ресурсах, реализованы два варианта конструктора:

```
CDialog::CDialog(
  LPCTSTR lpszTemplateName,
  CWnd *pParentWnd = NULL)
```

и

```
CDialog::CDialog(
  UINT nIDTemplate,
  CWnd *pParentWnd = NULL) —
```

первый параметр определяет шаблон: lpszTemplateName — указатель на символьную строку, содержащую имя шаблона, и nIDTemplate — номером идентификатора ресурса. Параметр pParentWnd указывает на объект родителя или владельца, которому принадлежит объект блока диалога. Если он равен NULL, то родительским окном является основное окно приложения.

Следующим шагом является создание блока диалога Windows. Для этого в классе реализована специальная функция

virtual int CDialog::**DoModal**() —

она создает и активизирует модальный блок диалога Windows, осуществляет полное взаимодействие с пользователем и удаляет блок диалога после завершения работы, возвращая целое значение типа IDOK или IDCANCEL, позволяющее определить конечный результат его работы. Если функция не смогла создать блок диалога, то она возвращает –1.

Помимо использования шаблона, хранящегося в ресурсе, в системе предусмотрена возможность создания шаблона в памяти непосредственно во

время выполнения программы. Это необходимо при косвенном создании блока диалога, как модального, так и немодального. Первым шагом на этом пути является определение и заполнение структуры *DLGTEMPLATE*, которая позволяет задать размеры и стиль блока диалога. Познакомимся с ней поближе:

```
typedef struct {
  DWORD style;
  DWORD dwExtendedStyle;
  WORD cdit;
  short x;
  short y;
  short cx;
  short cy;
} DLGTEMPLATE; —
```

поле *style* определяет стиль блока диалога. Наряду с уже известными стилями окон (такими как WS_CHILD и WS_CAPTION), для блоков диалога применяются также дополнительные:

DS_3DLOOK — вывод трехмерной рамки вокруг элементов управления в блоке диалога; этот стиль применим только для Windows 95

DS_ABSALIGN — координаты блока диалога задаются относительно экрана; если этот стиль не определен, то они задаются относительно рабочей области

DS_CENTER — блок диалога располагается по центру рабочей области; этот стиль применим только для Windows 95

DS_CENTERMOUSE — курсор мыши располагается в центре блока диалога; этот стиль применим только для Windows 95

DS_CONTEXTHELP — в полосу заголовка блока диалога добавляется знак вопроса, который подключает контекстно-зависимую справку; на самом деле, после создания блока диалога система проверяет наличие этого стиля, и если он присутствует, то добавляет флаг WS_EX_CONTEXTHELP к расширенному стилю блока диалога

DS_CONTROL — блок диалога может использоваться как элемент управления в другом блоке диалога; наличие этого стиля позволяет пользователю перемещаться по элементам управления дочернего блока диалога, использовать акселераторы клавиатуры и т. д.

DS_FIXEDSYS — блок диалога использует SYSTEM_FIXED_FONT вместо SYSTEM_FONT; этот стиль применим только для Windows 95

DS_MODALFRAME — модальный блок диалога; этот стиль часто комбинируют с WS_CAPTION и WS_SYSMENU

DS_NOFAILCREATE — создает блок диалога, даже в том случае, если произошли ошибки; например, если не может быть создан какой-либо элемент управления

DS_NOIDLEMSG	блокирует сообщение WM_ENTERIDLE, которое Windows посылает владельцу блока диалога, пока он отображается на экране
DS_SETFONT	показывает, что шаблон блока диалога (структура DLGTEMPLATE) содержит дополнительный элемент, определяющий имя и размер шрифта; этот шрифт используется для текста в рабочей области блока диалога и для текста внутри элементов управления; при этом сообщение Windows WM_SETFONT посылается в блок диалога и в каждый элемент управления
DS_SETFOREGROUND	переводит блок диалога на передний план, для чего Windows вызывает функцию SetForegroundWindow
DS_SYSMODAL	создает системный модальный блок диалога; при этом он обязательно должен иметь установленным расширенный стиль WS_EX_TOPMOST, иначе стиль будет проигнорирован

Поле *dwExtendedStyle* служит для установки расширенных стилей окна. Для создания блоков диалога это поле не используется, за исключением вышеупомянутых случаев. Оно используется, когда вы создаете другие типы окон, используя шаблон блока диалога.

В поле *cdit* задается число элементов блока диалога. Другими словами — число структур *DLGITEMTEMPLATE*, которые описывают элементы управления блока диалога. Эта структура содержит следующие поля:

```
typedef struct {
  DWORD style;
  DWORD dwExtendedStyle;
  short x;
  short y;
  short cx;
  short cy;
  WORD id;
} DLGITEMTEMPLATE;
```

Все поля этой структуры аналогичны только что описанным, кроме поля *id*, которое определяет идентификатор элемента управления.

При ее использовании, необходимо дополнительно описать еще три массива, определяющие класс, заголовок и данные элемента управления. Каждый массив состоит из одного или более 16-битных элементов:

- Первым за структурой следует *массив класса*. Если его первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет значение атома, идентифицирующее зарегистрированный оконный класс. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая имя зарегистрированного класса окна.
- После массива класса расположен *массив заголовка*. Если его первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент,

который определяет идентификатор ресурса для этого элемента управления. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая начальный текст элемента управления.

- *Массив данных* может содержать произвольное число элементов, которые оконная процедура элемента управления должна "понимать" и уметь обрабатывать. Эти данные передаются элементу управления, когда Windows вызывает функцию *CreateWindowEx*.

Поля *x*, *y*, *cx* и *cy* определяют соответственно координаты левого верхнего угла блока диалога, его ширину и высоту. Все размеры задаются в единицах блока диалога, которые можно преобразовать в значения экранных координат при помощи функции *MapDialogRect*.

При использовании структуры *DLGTEMPLATE* необходимо дополнительно описать еще три массива переменной длины, которые определяют меню, класс и заголовок блока диалога. Каждый массив содержит один или более 16-битных элементов:

- Первым располагается *массив меню*. Если его первый элемент равен 0x0000, то массив не имеет дополнительных элементов, а блок диалога не имеет меню. Если первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет идентификатор ресурса для меню. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая имя ресурса меню в исполняемом файле.
- За массивом меню следует *массив класса*. Если его первый элемент равен 0x0000, то Windows использует предопределенный оконный класс для блока диалога. Если первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет значение атома, идентифицирующее зарегистрированный класс окна блока диалога. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая имя зарегистрированного оконного класса.
- После массива класса расположен *массив заголовка*. Если его первый элемент равен 0x0000, то блок диалога не имеет заголовка. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая текст заголовка.

Если для блока диалога установлен стиль DS_SETFONT, то за массивом заголовка располагается 16-битный элемент, определяющий размер шрифта, за которым следует массив, задающий его вид. Массив вида шрифта — строка Unicode, заканчивающаяся нулевым символом и определяющая имя вида. Если эти параметры заданы, то Windows посылает сообщения WM_SETFONT оконным процедурам блока диалога и элементов управления.

После создания и заполнения необходимых структур и массивов можно создавать окно блока диалога. Как обычно, сначала нужно создать объект класса, используя для этого пустой конструктор, а затем вызвать функцию

```
BOOL CDialog::InitModalIndirect(
  LPCDLGTEMPLATE lpDialogTemplate,
  CWnd *pParentWnd = NULL)
```

или

```
BOOL CDialog::InitModalIndirect(
  HGLOBAL hDialogTemplate,
  CWnd *pParentWnd = NULL) —
```

оба варианта этой функции инициализируют объект модального блока диалога, используя шаблон, предварительно созданный в памяти, на который указывает либо lpDialogTemplate — указатель на структуру *DLGTEMPLATE*, либо hDialogTemplate — дескриптор глобальной памяти, в которой определены структура *DLGTEMPLATE* и данные для каждого элемента управления.

Примечание

Обратите особое внимание на то, что после выполнения функции *CDialog::InitModalIndirect* Windows создает блок диалога, но не отображает его. Он появляется на экране только после непосредственного вызова функции *CDialog::DoModal*.

Рассмотрим теперь создание немодального блока диалога. Первый способ предполагает наличие ресурса шаблона. Сначала вызывается пустой конструктор для создания объекта (так же, как при создании модального блока диалога на основе шаблона в памяти). Затем, для создания немодального блока диалога и присоединения его к объекту *CDialog*, вызывается функция

```
BOOL CDialog::Create(
  LPCTSTR lpszTemplateName,
  CWnd *pParentWnd = NULL)
```

или

```
BOOL CDialog::Create(
  UINT nIDTemplate,
  CWnd *pParentWnd = NULL)
```

Параметры, используемые этой функцией, аналогичны параметрам конструктора для модального блока диалога. Эту функцию можно вызвать прямо внутри конструктора класса, производного от CDialog.

Второй способ создания немодального блока диалога опирается на шаблон, предварительно созданный в памяти. Этот способ по своей сути ничем не отличается от аналогичного для модального блока диалога, за исключением того, что вместо *CDialog::InitModalIndirect* вызывается функция

```
BOOL CDialog::CreateIndirect(
  LPCDLGTEMPLATE lpDialogTemplate,
  CWnd *pParentWnd = NULL)
```

или

```
BOOL CDialog::CreateIndirect(
  HGLOBAL hDialogTemplate,
  CWnd *pParentWnd = NULL)
```

Еще одним отличием является то, что если при создании модального блока диалога не было необходимости устанавливать стиль WS_VISIBLE, т. к. такой блок диалога становится видимым автоматически, то для немодального блока диалога наличие или отсутствие этого стиля определяет, будет ли он отображаться на экране после своего создания. При отсутствии этого стиля необходим явный вызов функции *CWnd::ShowWindow* для вывода блока диалога на экран.

В отличие от функции *CDialog::DoModal*, функции *CDialog::Create* и *CDialog::CreateIndirect* завершаются сразу же после создания блока диалога Windows.

После того как работа с немодальным блоком диалога закончена, необходимо вызвать функцию *CWnd::DestroyWindow* для того, чтобы уничтожить его.

Несмотря на то, что блок диалога представляет собой стандартное окно Windows, для него имеются свои специфические сообщения. Одним из основных является WM_INITDIALOG. Это сообщение посылается процедуре блока диалога во время вызова функций *Create*, *CreateIndirect* или *DoModal* после того как были созданы окна всех его элементов управления, но до того как они стали видимыми. В ответ на это сообщение процедура блока диалога инициализирует каждый элемент управления. Например, она может заполнить список элементами, с которыми потом будет работать пользователь. В ответ на сообщение WM_INITDIALOG Windows вызывает специальный обработчик

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

через стандартную глобальную процедуру блока диалога, общую для всех блоков диалога библиотеки MFC.

Для того чтобы выполнить какую-либо специальную обработку, необходимо переопределить эту функцию в своем классе. При этом не забывайте вызывать функцию *OnInitDialog* базового класса для корректной инициализации. Функция должна возвращать нулевое значение только в том случае, когда явно устанавливает фокус ввода на какой-либо элемент управления, и ненулевое, когда фокус ввода устанавливается на первый элемент.

После того как работа с модальным блоком диалога закончена, обязательно должна вызываться функция

```
void CDialog::EndDialog(int nResult) –
```

она сама не закрывает блок диалога, а только устанавливает флажок, чтобы закрыть его как только закончится обработка текущего сообщения. Параметр nResult передается в функцию DoModal, которая использует его в качестве возвращаемого значения. Вызывать эту функцию можно в любой момент, даже в

функции OnInitDialog. В этом случае блок диалога необходимо закрыть до того как он будет выведен на экран или до установки фокуса ввода на какой-либо элемент управления.

Чаще всего пользователь ожидает увидеть в блоке диалога две кнопки — OK и Cancel (Отмена). Поэтому для вас не должно быть неожиданностью, что класс *CDialog* имеет встроенную поддержку обработки команд от этих кнопок. Начнем с кнопки OK. Обработчик

virtual void CDialog::**OnOK**()
фиксирует данные блока диалога и обновляет соответствующие переменные вашего приложения. Если вам нужны какие-либо дополнительные действия, то можно переопределить эту функцию для модального блока диалога. В случае немодального блока диалога ее необходимо переопределить. Дело в том, что функция EndDialog, которая вызывается в функции базового класса, делает блок диалога невидимым, но не уничтожает его. Поэтому внутри переопределенной функции необходимо вызвать функцию DestroyWindow, для уничтожения соответствующего окна Windows.

Для кнопки Cancel (Отмена) в классе реализован свой обработчик

virtual void CDialog::**OnCancel**()
который просто завершает работу с модальным блоком диалога, вызывая EndDialog, и возвращает IDCANCEL. Для немодального блока диалога эту функцию необходимо переопределить и вызвать внутри нее функцию DestroyWindow.

Последней переопределяемой функцией, реализованной в классе, является

virtual void CDialog::**OnSetFont**(CFont *pFont) —
вызывается в ответ на сообщение WM_SETFONT и устанавливает шрифт, определенный в редакторе блока диалога. Если вам нужно изменить этот шрифт, то функцию следует переопределить. Установленный шрифт используется по умолчанию всеми элементами управления блока диалога.

При обсуждении способов создания блока диалога мы упомянули о том, что они имеют свою систему координат, которые называются *координатами блока диалога* и позволяют определять размеры блока диалога независимо от конкретного устройства, поскольку масштаб элементов координатной сетки определяется относительно размеров текущего шрифта. Элементы масштаба оси X координат блока диалога соответствуют примерно 1/4 средней ширины используемого шрифта, а элементы оси Y — 1/8 высоты. Для преобразования координат блока диалога в экранные координаты в классе реализована функция

void CDialog::MapDialogRect(LPRECT lpRect) —
на входе получает размеры и координаты блока диалога через указатель lpRect (в системе координат блока диалога) и заменяет значения в соответствующих полях на экранные координаты.

Для перемещения по элементам управления блока диалога можно использовать клавишу <Tab> или мышь. Однако возникают ситуации, когда такого рода переходы необходимо выполнить программно. Для реализации программных перемещений можно использовать следующие функции:

void CDialog::**NextDlgCtrl**()

и

void CDialog::**PrevDlgCtrl**()

которые устанавливают фокус на следующий или, соответственно, на предыдущий элемент управления. Все переходы осуществляются циклически, т. е. с последнего элемента управления фокус устанавливается на первый (функция NextDlgCtrl), а с первого — на последний элемент управления (функция PrevDlgCtrl).

Для перемещения фокуса на какой-либо определенный элемент управления можно использовать функцию

void CDialog::**GotoDlgCtrl**(CWnd *pWndCtrl)

параметр *pWndCtrl* которой определяет указатель на объект "элемент управления" (дочернее окно), на который перемещается фокус ввода. Для получения этого указателя можно воспользоваться функцией

CWnd* CWnd::**GetDlgItem**(int nID)

или

void CWnd::**GetDlgItem**(
int nID,
HWND *phWnd)

которые по идентификатору элемента управления блока диалога возвращают либо указатель на соответствующий оконный объект, либо указатель на дескриптор окна этого объекта.

В заключение обзора членов класса рассмотрим две функции, которые позволяют узнать или переустановить "кнопку, действующую по умолчанию". При открытии блока диалога одна из кнопок имеет фокус по умолчанию, т. е. действие, приписанное ей, будет выполнено при нажатии пользователем клавиши <Enter> (для нее в шаблоне установлен стиль DEFPUSHBUTTON):

DWORD CDialog::**GetDefID**() —

если для кнопки определен идентификатор, то функция возвращает его в младшем слове, старшее слово при этом содержит значение DC_HASDEFID. Обычно возвращается IDOK (кнопка OK).

void CDialog::**SetDefID**(UINT nID) —

изменяет кнопку блока диалога, действующую по умолчанию. Параметр nID определяет идентификатор новой кнопки.

Теперь, после того как мы познакомились с возможностями класса *CDialog*, рассмотрим пример простейшего блока диалога. Он настолько прост, что даже нет необходимости создавать свой класс для его поддержки. Однако, мы все-таки сформируем класс *CAppAbout*, чтобы можно было наглядно продемонстрировать разницу между модальным и немодальным блоками диалога.

Такой, или аналогичный ему, блок диалога фактически стал обязательным элементом любого современного приложения, которое выводит его на

экран, чтобы сообщить пользователю некоторые сведения о программе, ее авторе, номере версии и т. д. Он обычно появляется при выборе соответствующего элемента в меню справки, как это показано на рис. 29. Сам блок диалога имеет вид, показанный на рис. 30. Для того чтобы не дублировать код, мы ограничимся только теми добавлениями, которые внесены в приложение MDI.



Рис. 29. Элемент "О программе" меню "?"

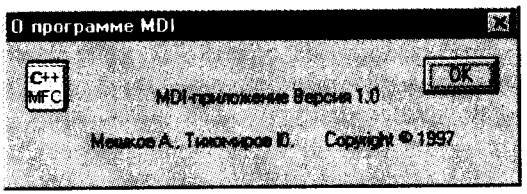


Рис. 30. Блок диалога

```
//////////////////////////////////////////////////////
// Файл CAppAbout.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////
class CAppAbout : public CDialog
{
// Construction
public:
  CAppAbout(CWnd* pParent=NULL);  // standard constructor

// Dialog Data
  //{{AFX_DATA(CAppAbout)
  enum { IDD = IDD_ABOUTBOX };
  // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAppAbout)
  protected:
  // DDX/DDV support
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL

// Implementation
protected:
// Generated message map functions
  //{{AFX_MSG(CAppAbout)
  // NOTE: the ClassWizard will add member functions here
  //}}AFX_MSG
```

```
DECLARE_MESSAGE_MAP()
};
```

Добавления, внесенные в реализацию членов этого класса:

```
/////////////////////////////////////////////////////////////
// Файл CAppAbout.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
// Блок диалога CAppAbout: реализация класса

CAppAbout::CAppAbout(CWnd* pParent /*=NULL*/) :
  Dialog(CAppAbout::IDD, pParent) // этого можно и не делать (см.
пример главы 12)
{
  Create(CAppAbout::IDD, pParent);

  //{{AFX_DATA_INIT(CAppAbout)
  //}}AFX_DATA_INIT
}

void CAppAbout::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);

  //{{AFX_DATA_MAP(CAppAbout)
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAppAbout, CDialog)
  //{{AFX_MSG_MAP(CAppAbout)
  // No message handlers
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Изменения в классе CMDIApp:

```
/////////////////////////////////////////////////////////////
// Файл MDI.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

// Объект приложения CMDIApp: интерфейс класса

class CMDIApp : public CWinApp
{

...

// Implementation
  //{{AFX_MSG(CMDIApp)
  afx_msg void OnAppAbout();
```

```
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл MDI.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
BEGIN_MESSAGE_MAP(CMDIApp, CWinApp)
  //{{AFX_MSG_MAP(CMDIApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
// App command to run the dialog
void CMDIApp::OnAppAbout()
{
  CAppAbout aboutDlg;
  aboutDlg.DoModal();
}
```

Во включаемый файл добавляется идентификатор:

```
/////////////////////////////////////////////////////////////
// Файл resource.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

...
#define IDD_ABOUTBOX                    100
...
```

В файл ресурса добавляется шаблон блока диалога:

```
/////////////////////////////////////////////////////////////
// Файл MDI.rc
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////
// Dialog
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "О программе MDI"
FONT 8, "MS Sans Serif"
BEGIN
  ICON  IDR_MAINFRAME,IDC_STATIC,7,7,21,20
  CTEXT "MDI-приложение Версия 1.1",
    IDC_STATIC,51,17,119,8, SS_NOPREFIX
  CTEXT  "Copyright © 1997",IDC_STATIC,127,34,71,8
```

```
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
  LTEXT  "Мешков А., Тихомиров Ю.",IDC_STATIC,35,34,89,8
END
```

В шаблон меню необходимо добавить следующий пункт:

```
POPUP "&?"
  BEGIN
    MENUITEM "&О программе ...",                ID_APP_ABOUT
  END
```

Как видите, все достаточно просто. Блок диалога появляется на экране при выборе пользователем пункта "О программе..." из меню справки. Заметим, что такой путь является стандартным для вызова блока диалога "О программе...". После того как данный пункт меню выбран, вызывается обработчик *OnAppAbout* команды ID_APP_ABOUT, в котором создается модальный блок диалога.

Для того чтобы вместо модального блока диалога создать немодальный, достаточно сделать следующее: изменить конструктор и обработчик команды ID_APP_ABOUT, как показано ниже:

```
CAppAbout::CAppAbout(CWnd* pParent)
{
...
 Create(CAppAbout::IDD, pParent);
...
}
void CMDIApp::OnAppAbout()
{
  CAppAbout *aboutDlg = new CAppAbout();
}
```

Теперь, если вы выведете на экран блок диалога "О программе...", можно продолжать работу с любым окном приложения без каких-либо ограничений.

Приведенный пример полностью иллюстрирует общий подход к созданию блоков диалога на основе шаблона, определенного в файле ресурсов. Теперь, опираясь на эти знания, рассмотрим пример приложения, который поможет нам изучить еще несколько новых интересных моментов:

- создание приложения, в котором в качестве главного окна будет выступать блок диалога,
- обмен данными с блоком диалога,
- создание блока диалога на основе шаблона в памяти.

Template — пример приложения

```
/////////////////////////////////////////////////////////////
// Файл Template.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"  // main symbols

/////////////////////////////////////////////////////////////
// CTemplateApp:

class CTemplateApp : public CWinApp
{
public:
  CTemplateApp();

// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CTemplateApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL

// Implementation

  //{{AFX_MSG(CTemplateApp)
  //}}AFX_MSG

DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////
// Файл Template.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Template.h"
#include "TemplateDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// CTemplateApp
```

```
BEGIN_MESSAGE_MAP(CTemplateApp, CWinApp)
  //{{AFX_MSG_MAP(CTemplateApp)
  //}}AFX_MSG
END_MESSAGE_MAP()

// Конструктор CTemplateApp

CTemplateApp::CTemplateApp()
{
}

// The one and only CTemplateApp object

CTemplateApp theApp;

// CTemplateApp initialization

BOOL CTemplateApp::InitInstance()
{
#ifdef _AFXDLL
  Enable3dControls();       // shared DLL
#else
  Enable3dControlsStatic(); // statically
#endif

  // Создаем объект "блок диалога"
  CTemplateDlg dlg;

  // Поскольку блок диалога является главным ресурсом приложения,
  // то указатель на этот объект присваивается переменной m_pMainWnd
  m_pMainWnd = &dlg;

  // Отображаем его на экране
  int nResponse = dlg.DoModal();

  return FALSE;
}

////////////////////////////////////////////////////////
// Файл TemplateDlg.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

// Класс CItem для создания элементов управления

class CItem
{
public:
  DLGITEMTEMPLATE  m_ctrItem;

  enum ctrtype {EDITTEXT = 0x0081, STATICTEXT};
  ctrtype m_ctrtype;
  CString m_strTitle;
```

```
public:
  CItem(enum ctrtype cType);
  CItem() {};
  void Initialize(enum ctrtype cType, UINT nID,
    CRect *prect = NULL,
    LPCTSTR pszCaption = NULL);
};
// Класс CMemory шаблона блока диалога
class CMemory
{
public:
  DLGTEMPLATE m_dlgMemory;
  CItem m_dlgItem[2]; // в шаблоне будет 2 элемента
  CMemory();
  void Show();
};
// Класс CTemplateDlg главного окна приложения
class CTemplateDlg : public CDialog
{
public:
  CTemplateDlg(CWnd* pParent = NULL);
// Dialog Data
  //{{AFX_DATA(CTemplateDlg)
  enum { IDD = IDD_TEMPLATE_DIALOG };
  CString m_strTitle;
  UINT m_x;
  UINT m_y;
  UINT m_nWidth;
  UINT m_nHeight;
  int m_nItem;
  CString m_strItem;
  //}}AFX_DATA
  int m_nActiveItem;
  CMemory m_dlgTemplate;
// ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CTemplateDlg)
  protected:
// поддержка обмена данными с блоком диалога (DDX/DDV)
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL
// Implementation
protected:
  HICON m_hIcon;
  void InfoToDialog(int nItem, LPCTSTR lpszItem);
```

```
// Generated message map functions
  //{{AFX_MSG(CTemplateDlg)
  virtual BOOL OnInitDialog();
  afx_msg void OnShow();
  afx_msg void OnEdit();
  afx_msg void OnStatic();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
////////////////////////////////////////////////////////////
// Файл TemplateDlg.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "Template.h"
#include "TemplateDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CItem - элементы управления шаблона в памяти
CItem::CItem(enum ctrtype cType)
{
  m_ctrtype = cType;
}
void CItem::Initialize(enum ctrtype cType, UINT nID,
  CRect* prect, LPCTSTR pszCaption)
{
  m_ctrtype = cType;
  // заполняем поля структуры DLGITEMTEMPLATE
  if (prect != NULL)
  {
    m_ctrItem.x = (short)prect->left;
    m_ctrItem.y = (short)prect->top;
    m_ctrItem.cx = (short)prect->Width();
    m_ctrItem.cy = (short)prect->Height();
  }
  else
  {
    m_ctrItem.x = 0;
    m_ctrItem.y = 0;
    m_ctrItem.cx = 20;
    m_ctrItem.cy = 20;
  }
```

```
  m_ctrItem.dwExtendedStyle = 0;
  m_ctrItem.id = nID;

  switch (m_ctrtype)
  {
    case STATICTEXT:
      m_ctrItem.style = WS_CHILD | WS_VISIBLE | SS_CENTER;
      break;
    case EDITTEXT:
      m_ctrItem.style = WS_CHILD | WS_VISIBLE |
        ES_MULTILINE | ES_LEFT |
        WS_EX_CLIENTEDGE | WS_BORDER;
      break;
    default:
      ASSERT(FALSE);
  }

  m_strTitle = (pszCaption != NULL) ? pszCaption : _T("");
}

// CMemory - шаблон в памяти

#define SETRECT(rect, l, t, r, b) \
  rect.left = l; \
  rect.top = t; \
  rect.right = r; \
  rect.bottom = b;

CMemory::CMemory()
{
  // инициализация элементов шаблона блока диалога
  CRect rect(40, 10, 200, 35);
  m_dlgItem[0].Initialize(CItem::STATICTEXT,
    IDC_TEXTSTATIC, &rect, _T("Статический текст"));

  SETRECT(rect, 40, 32, 200, 96);
  m_dlgItem[1].Initialize(CItem::EDITTEXT,
    IDC_TEXTEDIT, &rect, _T("Редактируемый текст"));

  // инициализация структуры DLGTEMPLATE
  m_dlgMemory.cx = 240;
  m_dlgMemory.cy = 104;
  m_dlgMemory.style = WS_CAPTION  | WS_VISIBLE |
    WS_DLGFRAME | WS_POPUP | DS_MODALFRAME | DS_SETFONT;
  m_dlgMemory.dwExtendedStyle = 0;
  m_dlgMemory.x = 0;
  m_dlgMemory.y = 0;
  m_dlgMemory.cdit = 2;
}

void CMemory::Show()
```

```
{
  // Прежде всего необходимо распределить память под шаблон
  WCHAR szBoxTitle[] =
    L"Шаблон блока диалога в памяти: Выход - <ESC>";
  WCHAR szFontName[] = L"Courier New Cyr";

  // Определяем размер структуры DLGTEMPLATE и заголовка
  // блока диалога
  int nBufferSize = sizeof(DLGTEMPLATE) + // структура
    (2*sizeof(WORD)) +                    // меню и класс
    sizeof(szBoxTitle) +                  // заголовок
    sizeof(WORD)+sizeof(szFontName);      // шрифт

  nBufferSize = (nBufferSize + 3) & ~3;

  // Добавляем размеры структуры DLGITEMTEMPLATE
  // и заголовков элементов
  for (int i = 0; i < 2; i++)
  {
    int nItemLength = (sizeof(DLGITEMTEMPLATE) +
      3*sizeof(WORD) +
      (m_dlgItem[i].m_strTitle.GetLength()+1)*sizeof(WCHAR));

    if (i != 1)
      nItemLength = (nItemLength + 3) & ~3;
    nBufferSize += nItemLength;
  }

  // Выделяем память
  HLOCAL hLocal = LocalAlloc(LHND, nBufferSize);
  if (hLocal == NULL)
  {
    // Обработка ошибки распределения памяти
  }

  BYTE *pBuffer = (BYTE*)LocalLock(hLocal);
  if (pBuffer == NULL)
  {
    LocalFree(hLocal);
  }

  BYTE *pDest = pBuffer;

  // Переписываем структуру DLGTEMPLATE в буфер
  memcpy(pDest, &m_dlgMemory, sizeof(DLGTEMPLATE));
  pDest += sizeof(DLGTEMPLATE);
  *(WORD*)pDest = 0; // меню нет
  *(WORD*)pDest = 0; // используем класс по умолчанию

  pDest += 2*sizeof(WORD);
  memcpy(pDest, szBoxTitle, sizeof(szBoxTitle));
```

```
  pDest += sizeof(szBoxTitle);
  *(WORD*)pDest = 10; // размер шрифта
  pDest += sizeof(WORD);
  memcpy(pDest, szFontName, sizeof(szFontName));
  pDest += sizeof(szFontName);

  // Переписываем информацию о каждом элементе шаблона в буфер
  for (i = 0; i < 2; i++)
  {
    pDest = (BYTE*)(((DWORD)pDest + 3) & ~3);
    memcpy(pDest, (void*)&m_dlgItem[i].m_ctrItem,
      sizeof(DLGITEMTEMPLATE));
    pDest += sizeof(DLGITEMTEMPLATE);

    *(WORD*)pDest = 0xFFFF; // есть идентификатор ресурса
    pDest += sizeof(WORD);
    *(WORD*)pDest = m_dlgItem[i].m_ctrtype;
    pDest += sizeof(WORD);

    // Переписываем заголовок, предварительно преобразовав
    // его в строку Unicode
    WCHAR *pchCaption;
    int nChar;

    nChar = m_dlgItem[i].m_strTitle.GetLength() + 1;
    pchCaption = new WCHAR[nChar];
    nChar = MultiByteToWideChar(CP_ACP, 0,
      m_dlgItem[i].m_strTitle, -1, pchCaption, nChar);

    ASSERT(nChar > 0);
    memcpy(pDest, pchCaption, nChar*sizeof(WCHAR));
    pDest += nChar*sizeof(WCHAR);
    delete pchCaption;

    *(WORD*)pDest = 0;
    pDest += sizeof(WORD);
  }

  ASSERT(pDest - pBuffer == nBufferSize);

  CDialog dlg;
  dlg.InitModalIndirect((DLGTEMPLATE*)pBuffer);
  dlg.DoModal();

  LocalUnlock(hLocal);
  LocalFree(hLocal);
}

// Блок диалога CTemplateDlg

CTemplateDlg::CTemplateDlg(CWnd* pParent) :
  CDialog(CTemplateDlg::IDD, pParent)
```

```
{
  m_nActiveItem = -1;
  //{{AFX_DATA_INIT(CTemplateDlg)
  m_strTitle = _T("");
  m_x = 0;
  m_y = 0;
  m_nWidth = 0;
  m_nHeight = 0;
  m_nItem = -1;
  m_strItem = _T("");
  //}}AFX_DATA_INIT
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CTemplateDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CTemplateDlg)
  DDX_Text(pDX, IDC_TITLE, m_strTitle);
  DDX_Text(pDX, IDC_XPOS, m_x);
  DDX_Text(pDX, IDC_YPOS, m_y);
  DDX_Text(pDX, IDC_WIDTH, m_nWidth);
  DDX_Text(pDX, IDC_HEIGHT, m_nHeight);
  DDX_Radio(pDX, IDC_STATICTEXT, m_nItem);
  DDX_Text(pDX, IDC_CAPTION, m_strItem);
  //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CTemplateDlg, CDialog)
  //{{AFX_MSG_MAP(CTemplateDlg)
  ON_BN_CLICKED(ID_SHOW, OnShow)
  ON_BN_CLICKED(IDC_EDITTEXT, OnEdit)
  ON_BN_CLICKED(IDC_STATICTEXT, OnStatic)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////
// Обработчики сообщений класса CTemplateDlg

BOOL CTemplateDlg::OnInitDialog()
{
 CDialog::OnInitDialog();

  SetIcon(m_hIcon, TRUE);

  m_nItem = 0;
  OnStatic();

  return TRUE;
}
```

```
void CTemplateDlg::OnShow()
{
  InfoToDialog(m_nActiveItem, NULL);
  m_dlgTemplate.Show();
}

void CTemplateDlg::OnEdit()
{
  InfoToDialog(1, _T("Поле ввода текста"));
}

void CTemplateDlg::OnStatic()
{
  InfoToDialog(0, _T("Статический текст"));
}

void CTemplateDlg::InfoToDialog(int nItem,
  LPCTSTR lpszItem)
{
  if (m_nActiveItem != -1)
  {
    UpdateData(TRUE);
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.x =
      m_x;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.y =
      m_y;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.cx =
      m_nWidth;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.cy =
      m_nHeight;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_strTitle =
      m_strTitle;
  }

  if (m_nActiveItem != nItem)
  {
    m_x = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.x;
    m_y = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.y;
    m_nWidth = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.cx;
    m_nHeight = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.cy;
    m_strTitle = m_dlgTemplate.m_dlgItem[nItem].m_strTitle;

    ASSERT(lpszItem != NULL);
    m_strItem = lpszItem;
    m_nItem = nItem;
    m_nActiveItem = nItem;
    UpdateData(FALSE);
  }
}
```

```
//////////////////////////////////////////////////////////
// Файл Template.rc
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////

//Microsoft Developer Studio generated resource script.

#include "resource.h"

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

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

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

#ifdef APSTUDIO_INVOKED

//////////////////////////////////////////////////////////
// TEXTINCLUDE

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

3 TEXTINCLUDE DISCARDABLE
BEGIN
"#define _AFX_NO_SPLITTER_RESOURCES\r\n"
"#define _AFX_NO_OLE_RESOURCES\r\n"
"#define _AFX_NO_TRACKER_RESOURCES\r\n"
"#define _AFX_NO_PROPERTY_RESOURCES\r\n"
"\r\n"
"#if!defined(AFX_RESOURCE_DLL)||defined(AFX_TARG_ENU)\r\n"
"#ifdef _WIN32\r\n"
"LANGUAGE 9, 1\r\n"
"#pragma code_page(1252)\r\n"
"#endif\r\n"
```

```
// non-Microsoft Visual C++ edited resources\r\n"
"#include ""res\\Template.rc2""
"#include ""afxres.rc"" // Standard components\r\n"
"#endif\0"
END

#endif    // APSTUDIO_INVOKED

////////////////////////////////////////////////////////
// Icon

IDR_MAINFRAME ICON DISCARDABLE  "res\\Template.ico"

////////////////////////////////////////////////////////
// Dialog

IDD_TEMPLATE DIALOGEX 0, 0, 275, 150
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Основное окно приложения"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON   "Показать",ID_SHOW,218,7,50,14
  DEFPUSHBUTTON   "Выход",IDCANCEL,218,23,50,14
  GROUPBOX        "Тип элемента управления",
                  IDC_STATIC,17,49,102,56
  CONTROL         "Статический текст",
                  IDC_STATICTEXT,"Button",
                  BS_AUTORADIOBUTTON | WS_GROUP |
                  WS_TABSTOP,25,66,90,10
  CONTROL         "Редактируемый текст",
                  IDC_EDITTEXT,"Button",
                  BS_AUTORADIOBUTTON |
                  WS_TABSTOP,25,86,90,10
  ICON            IDR_MAINFRAME,
                  IDC_STATIC,17,7,18,20,SS_SUNKEN |
                  WS_GROUP
  GROUPBOX        "Параметры элемента управления",
                  IDC_CAPTION,132,49,128,82
  EDITTEXT        IDC_XPOS,156,62,30,14,ES_AUTOHSCROLL
  EDITTEXT        IDC_YPOS,156,83,30,14,ES_AUTOHSCROLL
  EDITTEXT        IDC_WIDTH,220,62,30,14,ES_AUTOHSCROLL |
                  ES_NUMBER
  EDITTEXT        IDC_HEIGHT, 220,83,30,14,ES_AUTOHSCROLL
  EDITTEXT        IDC_TITLE,162,111,88,14,ES_AUTOHSCROLL
  LTEXT           "Ширина:",IDC_STATIC,189,65,29,8
  LTEXT           "X:",IDC_STATIC,143,64,8,8
  LTEXT           "Y:",IDC_STATIC,143,87,8,8
  LTEXT           "Высота:",IDC_STATIC,189,86,28,8
```

```
  LTEXT           "Текст:",IDC_STATIC,138,113,23,8
  CTEXT           "Выберите тип элемента управления и
                  задайте его размеры и выводимый текст",
                  IDC_STATIC,71,7,111,24
END

/////////////////////////////////////////////////////////////
// DESIGNINFO

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
  IDD_TEMPLATE, DIALOG
  BEGIN
   LEFTMARGIN, 7
   RIGHTMARGIN, 268
   TOPMARGIN, 7
   BOTTOMMARGIN, 142
  END
END
#endif    // APSTUDIO_INVOKED

#endif    // Russian resources

#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//
#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE 9, 1
#pragma code_page(1252)
#endif
// non-Microsoft Visual C++ edited resources
#include "res\Template.rc2"
#include "afxres.rc"         // Standard components
#endif
/////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////
// Файл resource.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
```

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Template.rc
//
#define IDD_TEMPLATE_DIALOG             102
#define IDD_TEMPLATE                    102
#define IDR_MAINFRAME                   128
#define IDC_STATICTEXT                  1000
#define IDC_EDITTEXT                    1001
#define IDC_XPOS                        1003
#define IDC_YPOS                        1004
#define IDC_WIDTH                       1005
#define IDC_HEIGHT                      1006
#define IDC_TITLE                       1007
#define ID_SHOW                         1008
#define IDC_CAPTION                     1009
#define IDC_TEXTSTATIC                  1010
#define IDC_TEXTEDIT                    1011
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1010
#define _APS_NEXT_SYMED_VALUE           103
#endif
#endif
```

```
////////////////////////////////////////////////////////
// Файл stdafx.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////

// stdafx.h : include file for standard system
// include files,or project specific include files
// that are used frequently, but are changed infrequently
//

// Exclude rarely-used stuff from Windows headers
#define VC_EXTRALEAN

#include <afxwin.h>    // MFC core and standard components
#include <afxext.h>    // MFC extensions

// для подключения функции MultiByteToWideChar
#include <winnls.h>
```

Рассмотрим основные положения этой программы.

Блок диалога в качестве главного окна приложения

Для реализации этой возможности нужно проделать всего три шага. Прежде всего, в ресурсах определить шаблон блока диалога. Способы, которыми это можно сделать мы подробно рассмотрели выше. На рис. 31 показан блок диалога, созданный для нашего приложения.

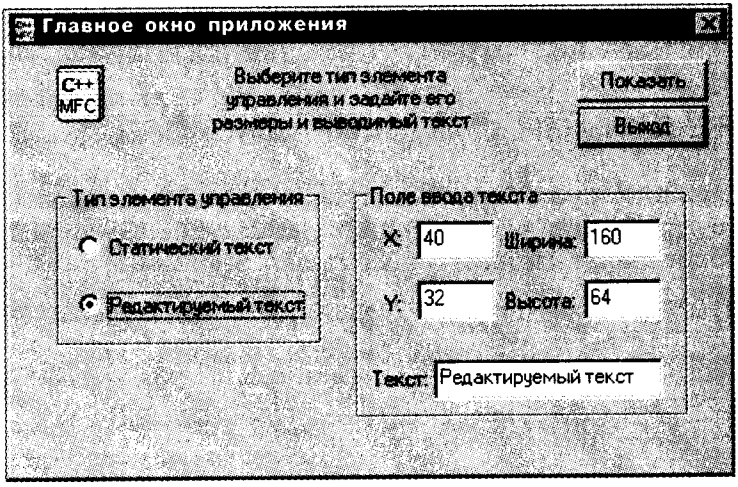


Рис. 31. Главное окно приложения является блоком диалога

Вторым шагом является создание класса на основе *CDialog*:[1]

```
class CTemplateDlg : public CDialog
{
public:
  CTemplateDlg(CWnd* pParent = NULL);
...
};
```

И, наконец, необходимо создать объект этого класса, что мы и делаем в переопределенной функции *InitInstance*:

```
BOOL CTemplateApp::InitInstance()
{
...
  CTemplateDlg dlg;
```

[1] Мы больше не говорим о необходимости создания класса приложения, т. к. ничего нового там нет.

```
  m_pMainWnd = &dlg;
  int nResponse = dlg.DoModal();
...
}
```

После создания окно сразу выводится на экран. Как видите, все настолько просто, что даже нет смысла останавливаться на этом более подробно. Поэтому мы переходим к новой теме, связанной с взаимодействием между пользователем и блоком диалога.

Обмен данными с блоком диалога

Для того чтобы можно было обмениваться данными с блоком диалога[1], необходимо прежде всего определить в вашем классе соответствующие члены, которые смогут принимать и передавать эти данные:

```
class CTemplateDlg : public CDialog
{
...
// Dialog Data
  // Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA(CTemplateDlg)
  // Здесь располагаются переменные для обмена данными
  ...
  //}}AFX_DATA
  ...
  //{{AFX_VIRTUAL(CTemplateDlg)
  protected:
  // поддержка обмена данными с блоком диалога (DDX/DDV)
  virtual void DoDataExchange(CDataExchange* pDX);
 //}}AFX_VIRTUAL
...
DECLARE_MESSAGE_MAP()
};
```

[1] Рассматриваемый ниже механизм справедлив для любого класса, производного от *CWnd*, а не только для блоков диалога. Однако, поскольку чаще всего он применяется именно при работе с блоками диалога, мы позволили себе использовать термин "обмен данными с блоком диалога".

Будет уместно привести список элементов управления и соответствующих им типов переменных:

Элемент редактирования (EDITBOX)	CString, int, UINT, long, DWORD, float, double, short, BOOL
Обычный флажок (CHECKBOX)	BOOL
Флажок с тремя состояниями (CHECKBOX)	int
Первый в группе переключатель (RADIOBUTTON)	int
Неотсортированный список (LISTBOX)	CString, int
Комбинированный список (COMBOBOX)	CString, int
Все остальные виды списков	CString

Начальные значения данных устанавливаются в переопределенной функции *OnInitDialog* или в конструкторе вашего класса блока диалога:

```
CTemplateDlg::CTemplateDlg(CWnd* pParent) :
  CDialog(CTemplateDlg::IDD, pParent)
{
...
// Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA_INIT(CTemplateDlg)
...
  // Инициализация данных для обмена с блоком диалога
...
  //}}AFX_DATA_INIT
...
}
```

После того как определены источник и приемник данных, а также их типы, пришло время ознакомиться с тем, каким образом осуществляется обмен. Для организации обмена нужны всего две функции:

BOOL CWnd::**UpdateData**(BOOL bSaveAndValidate = TRUE) —
единственный параметр этой функции bSaveAndValidate определяет направленность действий. Если он равен TRUE, то данные, введенные пользователем в блоке диалога, записываются в соответствующие переменные класса, в противном случае данные инициализируют элементы управления блока диалога. Она вызывается в функции CDialog::OnInitDialog с параметром, равным FALSE, для инициализации данных при создании модального блока диалога, и в функции CDialog::OnOK с параметром, равным TRUE, для получения и проверки на допустимость данных из блока диалога, если он закрыт при помощи кнопки OK. Основным ее назначением является проведение инициализации и обработки ошибок при обращении к функции

virtual void CWnd::**DoDataExchange**(CDataExchange *pDX)

которая, собственно, через ее параметр pDX (указатель на объект класса CDataExchange) и "занимается" обменом данных и проверкой их корректности. Эту функцию необходимо переопределить для каждого класса блока диалога, который должен обеспечивать взаимодействие с пользователем. Переопределенная функция имеет следующую стандартную форму, аналогичную приведенной в рассмотренном приложении:

```
void CTemplateDlg::DoDataExchange(CDataExchange* pDX)
{
// Обязательный вызов функции базового класса
  CDialog::DoDataExchange(pDX);

// Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA_MAP(CTemplateDlg)
  // вызовы функций для обмена данными

   ...

  // вызовы функций для проверки корректности данных

   ...

  //}}AFX_DATA_MAP
}
```

Общая схема обмена данными наглядно представлена на рис. 32.

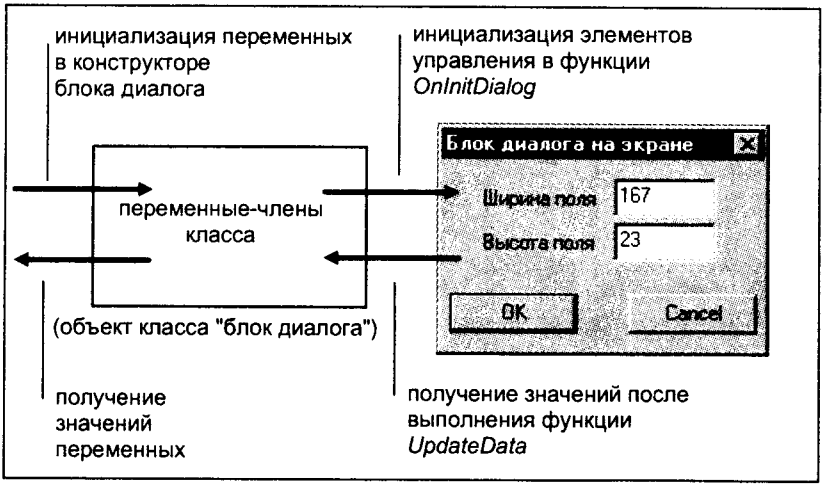


Рис. 32. Общая схема обмена данными в приложении

Функция *DoDataExchange* во многом похожа на функцию *Serialize*, а ее параметр *pDX*, очень напоминает параметр *CArchive* функции *CObject::Serialize*, например, объект *pDX* класса *CDataExchange*, так же как объект класса *CArchive*, имеет флаг направления

BOOL CDataExchange::**m_bSaveAndValidate** —
этот флаг указывает направление при операциях обмена данными (Dynamic Data Exchange, DDX). Значение TRUE задает передачу данных от элементов управления блока диалога к членам соответствующего класса, а значение FALSE — в обратном направлении. При операциях проверки корректности данных (Dynamic Data Validation, DDV) всегда установлен в TRUE. Значение флага определяется параметром, передаваемым функцией CWnd::UpdateData.

Кроме этого, в классе *CDataExchange* имеются еще четыре члена:

BOOL CDataExchange::**m_pDlgWnd** —
указатель на объект класса CWnd (обычно CDialog), для которого осуществляются обмен и проверка корректности введенных данных;

HWND CDataExchange::**PrepareCtrl**(int nIDC)
и
HWND CDataExchange::**PrepareEditCtrl**(int nIDC) —
эти функции подготавливают элементы управления к обмену данными. Подготовка заключается в сохранении дескриптора (HWND) элемента управления в классе CDataExchange. В случае сбоя они позволяют восстановить фокус ввода у последнего элемента управления, имевшего фокус. Параметр nIDC задает идентификатор элемента управления, подготавливаемый для операций DDX\DDV. Первая функция применяется для всех элементов управления, за исключением элементов редактирования, вторая работает исключительно с элементами редактирования;

void CDataExchange::**Fail**()
вызывается в случае сбоя при проведении операции проверки корректности данных блока диалога. Она восстанавливает фокус у элемента управления, вызвавшего сбой, и вызывает программу обработки исключения типа CUserException для остановки операции DDV. Исключение выводит окно сообщения с объяснением причины сбоя, после чего пользователь может повторно вводить данные в этот элемент управления.

После того как вы определите свою функцию *DoDataExchange*, вам необходимо вставить в нее требуемые функции DDX, по одной на каждый элемент управления, участвующий в обмене. Библиотека MFC предоставляет достаточно большое число таких функций для организации автоматического обмена и проверки корректности данных. Полный их список можно посмотреть в файле afxdd_.h. То же самое можно сказать и про функции DDV. Правда, тут есть один маленький нюанс. Вы не можете вызвать функцию проверки корректности введенных пользователем данных (DDV) без вызова соответствующей функции обмена (DDX). Более того, функция DDV должна следовать непосредственно за функцией DDX этого элемента управления. Функции имеют следующий формат:

DDX_XXX(pDX, nIDC, member) — для обмена данными
и
DDV_XXX(pDX, member, ...) — для проверки корректности введенных данных

В этих функциях задействованы следующие параметры: XXX — имя соответствующей функции (такой как *DDX_Check* или *DDV_MinMaxInt*), *pDX* — указатель на объект *CDataExchange*, *nIDC* — идентификатор элемента управления, с которым производится обмен данными, и *member* — член класса, участвующий в обмене.

Этот шаг, связанный с работой с блоками диалога, также достаточно прост, и мы переходим к третьей теме.

Создание блока диалога на основе шаблона в памяти

Мы уже говорили, что в этом случае объект класса создается при помощи функций *CDialog::InitModalIndirect* или *CDialog::CreateIndirect*. Однако, перед их использованием необходимо проделать некоторую подготовительную работу. Обратимся к рассматриваемому примеру. Как вы помните, для создания шаблона блока диалога в памяти необходимо определить и заполнить структуру *DLGTEMPLATE* для блока диалога и некоторое количество структур *DLGITEMTEMPLATE* (по числу используемых элементов управления). Задачу определения структур у нас решают два класса: *CMemory* и *CItem*, которые позволяют создать объект "блок диалога" и необходимое число объектов его элементов управления (в данном случае — "статическое поле" и "элемент редактирования"). После того как объекты созданы и проинициализированы, можно создавать соответствующие окна Windows. Эту задачу у нас выполняет функция *CMemory::Show*:

```
void CMemory::Show()
{
  // При работе с шаблонами блоков диалога в памяти все
  // текстовые строки должны быть представлены в формате Unicode

  WCHAR szBoxTitle[] =
    L"Шаблон блока диалога в памяти: Выход — <ESC>";
  WCHAR szFontName[] = L"Courier New Cyr";

  // Прежде всего, определяем размер памяти, необходимой
  // для размещения шаблона блока диалога, и дополнительной
  // памяти под четыре массива: для меню, класса,
  // заголовка и шрифта

  int nBufferSize = sizeof(DLGTEMPLATE) + // структура
    (2*sizeof(WORD)) +                    // меню и класс
    sizeof(szBoxTitle) +                  // заголовок
    sizeof(WORD)+sizeof(szFontName);      // шрифт
  // Выравнивание по DWORD
  nBufferSize = (nBufferSize + 3) & ~3;
```

```
// Добавляем размеры структуры DLGITEMTEMPLATE
// и заголовков элементов
for (int i = 0; i < 2; i++)
{
  int nItemLength=(sizeof(DLGITEMTEMPLATE) + // структура
    // 3 массива для класса, заголовка и данных
    // элемента управления
    3*sizeof(WORD) +
    // размер строки Unicode для заголовка
    (m_dlgItem[i].m_strTitle.GetLength()+1)*sizeof(WCHAR));
    // Выравнивание по DWORD
    if (i != 1)
      nItemLength = (nItemLength + 3) & ~3;
    nBufferSize += nItemLength;
}
// Выделяем память под шаблон
HLOCAL hLocal = LocalAlloc(LHND, nBufferSize);
if (hLocal == NULL)
{
// Обработка ошибки распределения памяти
}
BYTE *pBuffer = (BYTE*)LocalLock(hLocal);

if (pBuffer == NULL)
{
  LocalFree(hLocal);
  // Обработка ошибки блокировки памяти
}

BYTE *pDest = pBuffer;
// Переписываем структуру DLGTEMPLATE в буфер
memcpy(pDest, &m_dlgMemory, sizeof(DLGTEMPLATE));
pDest += sizeof(DLGTEMPLATE);
*(WORD*)pDest = 0; // меню нет
*(WORD*)pDest = 0; // используем класс по умолчанию
pDest += 2*sizeof(WORD);

// Переписываем заголовок
memcpy(pDest, szBoxTitle, sizeof(szBoxTitle));
pDest += sizeof(szBoxTitle);
*(WORD*)pDest = 10; // размер шрифта
pDest += sizeof(WORD);

// Переписываем имя шрифта
memcpy(pDest, szFontName, sizeof(szFontName));
pDest += sizeof(szFontName);

// Переписываем информацию о каждом элементе шаблона
for (i = 0; i < 2; i++)
```

```
{
  // Выравнивание буфера по DWORD
  pDest = (BYTE*)(((DWORD)pDest + 3) & ~3);
  memcpy(pDest, (void*)&m_dlgItem[i].m_ctrItem,
    sizeof(DLGITEMTEMPLATE));
  pDest += sizeof(DLGITEMTEMPLATE);

  *(WORD*)pDest = 0xFFFF; // есть идентификатор ресурса
  pDest += sizeof(WORD);
  *(WORD*)pDest = m_dlgItem[i].m_ctrtype;
  pDest += sizeof(WORD);

  // Преобразуем заголовок в Unicode строку
  WCHAR *pchCaption;
  int nChar = m_dlgItem[i].m_strTitle.GetLength() + 1;
  pchCaption = new WCHAR[nChar];

  // Преобразуем строку ANSI в строку Unicode
  nChar = MultiByteToWideChar(
    CP_ACP,        // кодовая страница ANSI
    0, // предварительно сформированная строка
    m_dlgItem[i].m_strTitle, // адрес строки
    -1, // автоматическое определение длины
    pchCaption, // адрес буфера строки Unicode
    nChar); // размер буфера строки Unicode
  ASSERT(nChar > 0);

  // Переписываем заголовок элемента управления
  memcpy(pDest, pchCaption, nChar*sizeof(WCHAR));
  pDest += nChar*sizeof(WCHAR);
  delete pchCaption;
  *(WORD*)pDest = 0;
  pDest += sizeof(WORD);
}

// Проверка на полноту заполнения буфера
ASSERT(pDest - pBuffer == nBufferSize);

// После заполнения буфера, информация в нем хранится
// в следующем виде:
// Структура DLGTEMPLATE
// 0x0000 (WORD) - индикатор наличия меню
// 0x0000 (WORD) - класс блока диалога по умолчанию
// Заголовок - строка Unicode
// 0x0000 - размер используемого шрифта
// "Courier New Cyr" - имя используемого шрифта
// Структура DLGITEMTEMPLATE для статического текста
// 0xFFFF - есть идентификатор элемента управления
// 0x0081 - идентификатор класса Static
// Заголовок - строка Unicode
```

```
  // 0x0000 -- дополнительных данных нет
  // Структура DLGITEMTEMPLATE для редактируемого текста
  // 0xFFFF - есть идентификатор элемента управления
  // 0x0082 - идентификатор класса Edit
  // Заголовок - строка Unicode
  // 0x0000 - дополнительных данных нет

  // Создание объекта "блок диалога"
  CDialog dlg;

  // Создание окна Windows блока диалога
  dlg.InitModalIndirect((DLGTEMPLATE*)pBuffer);

  // Отображение модального блока диалога на экране
  dlg.DoModal();

  // Освобождение использованной памяти
  LocalUnlock(hLocal);
  LocalFree(hLocal);
}
```

Функция вызывается всякий раз, когда пользователь нажимает кнопку "Показать" в главном окне приложения. Параметры, которые пользователь вводит в полях "X:", "Y:", "Ширина", "Высота" и "Текст", передаются в соответствующие элементы управления, которые используют их при инициализации. В результате выполнения функции на экран выводится блок диалога, представленный на рис. 33.

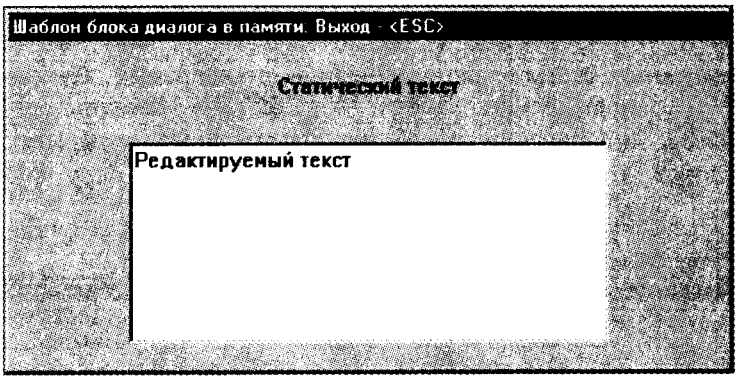
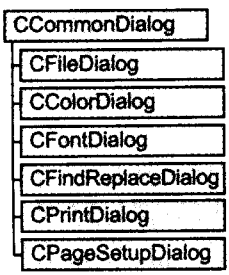


Рис. 33. Блок диалога, созданный функцией CDialog::InitModalIndirect

Понятно, что приведенный пример не претендует на изящество, однако свою роль в иллюстрации рассматриваемых положений он успешно выполняет.

Глава 15

Стандартные блоки диалога



В свое время фирма Microsoft ввела понятие *стандартных блоков диалога* (Common Dialog Box), применяемых для выполнения тех или иных стандартных действий, таких, например, как открытие файла.[1] Теперь уже для всех очевидны достоинства такого подхода, ведь использование стандартных блоков диалога означает целостность интерфейса для пользователя и меньший объем работы для программиста. Эти блоки диалога давно уже стали составной частью Windows, и с первых версий библиотеки MFC в нее, естественно, были включены классы, поддерживающие создание стандартных блоков диалога. В настоящее время библиотека поддерживает следующие классы: *CFileDialog*, *CColorDialog*, *CFontDialog*, *CFindReplaceDialog*, *CPrintDialog* и *CPageSetupDialog*.

Ввиду достаточно частого использования стандартных блоков диалога рассмотрим их основные возможности. Все они базируются на одном классе *CCommonDialog*, инкапсулирующего функциональные возможности стандартных блоков диалога Windows и имеющего всего одну функцию — конструктор:

CCommonDialog::**CCommonDialog**(CWnd *pParentWnd) —
здесь, как и для класса CDialog, если параметр pParentWnd равен NULL, то родительским окном блока диалога будет главное окно приложения.

Значительно больший интерес представляют производные от него классы. Знакомство с ними начнем с класса *CFileDialog*, приведя краткое описание членов и функций классов. Для некоторых классов будут рассмотрены примеры, демонстрирующие основные принципы настройки внешнего вида

---

[1] Как отмечает Адриан Кинг в своей книге "Windows 95 изнутри": "В течение весьма длительного времени одно из известных причитаний Билла Гейтса звучало так: "Ну почему же все должны писать код для открытия файла?"

соответствующего блока диалога. В дальнейшем, при описании работы с документами или объектами GDI, мы продемонстрируем остальные свойства этих классов.

Класс CFileDialog

Этот класс отвечает за создание и работоспособность двух блоков диалога: один позволяет выбрать файл для его последующего открытия, а другой предназначен для выбора места расположения и имени файла при его сохранении.

Перед тем как начать рассматривать члены этого класса, скажем несколько слов об общих принципах создания этих блоков диалога на базе рассматриваемого класса.

Прежде всего, если вам не нужны никакие дополнительные настройки, вы совершенно спокойно можете использовать прямо этот класс, не создавая своего. Если же ваши запросы несколько шире (надо сказать, что процесс создания "идеального" блока диалога File Open (Открытие файла), давно превратился в своего рода искусство), то необходимо создать свой класс на базе *CFileDialog* и спроектировать конструктор таким образом, чтобы удовлетворить все ваши потребности. После того как блок диалога создан, можно выполнять различные настройки его элементов управления, заполняя поля структуры типа *OPENFILENAME*. И, наконец, вызывается функция *DoModal*, для отображения блока диалога и работы с ним. Если после закрытия блока диалога функция возвращает значение IDOK, можно использовать специальные функции для получения информации о сделанном выборе.

В классе представлены следующие члены:

OPENFILENAME CFileDialog::**m_ofn** –

структура типа *OPENFILENAME*:

```
typedef struct tagOFN{
  DWORD lStructSize;
  HWND hwndOwner;
  HINSTANCE hInstance;
  LPCTSTR lpstrFilter;
  LPTSTR lpstrCustomFilter;
  DWORD nMaxCustFilter;
  DWORD nFilterIndex;
  LPTSTR lpstrFile;
  DWORD nMaxFile;
  LPTSTR lpstrFileTitle;
  DWORD nMaxFileTitle;
  LPCTSTR lpstrInitialDir;
```

```
  LPCTSTR lpstrTitle;
  DWORD Flags;
  WORD nFileOffset;
  WORD nFileExtension;
  LPCTSTR lpstrDefExt;
  DWORD lCustData;
  LPOFNHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
} OPENFILENAME; —
```

используется для инициализации внешнего вида блоков диалога File Open (Открытие файла) и File Save As (Сохранить файл) после создания объекта класса, но до вызова функции *DoModal*. Рассмотрим поля этой структуры, чтобы иметь ясное представление о тех возможностях, которыми мы располагаем:

lStructSize — размер структуры в байтах.

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL.

hInstance — определяет идентификатор блока памяти (установлен флаг OFN_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг OFN_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется. В Windows NT и Windows 95 (если не установлен флаг OFN_EXPLORER) это поле определяет шаблон, который заменяет шаблон, заданный по умолчанию. Если же в Windows 95 указанный флаг установлен, то система использует в качестве шаблона стиль блока диалога Explorer (Проводник), в котором изображается дерево, а также список файлов и каталогов в виде маленьких пиктограмм.

lpstrFilter — определяет указатель на буфер, содержащий одну или больше пар текстовых строк, задающих фильтры для выбора имен файлов (шаблоны имен файлов). Первая строка в каждой паре описывает название фильтра (например, "Исполняемые файлы"), а вторая — образец фильтра (например, "*.EXE; *.COM"). Если в единственной строке фильтра необходимо указать несколько образцов, то они должны отделяться друг от друга нулевым символом. Последняя строка в буфере должна заканчиваться двумя нулевыми символами. Порядок расположения пар строк в буфере задает порядок следования фильтров. Если этот параметр равен NULL, то в блоке диалога не определен никакой фильтр.

lpstrCustomFilter — определяет указатель на статический буфер, содержащий пару текстовых строк, для хранения образца выбранного пользователем фильтра. Первая строка описывает отображаемую строку, а вторая — образец фильтра, выбранный пользователем. При первом создании блока диалога вы определяете первую строку, которая может быть любой непустой строкой. Когда пользователь выбирает какой-либо файл, блок диалога копирует текущий образец фильтра во вторую строку. Им может быть как

один из образцов, определенных в буфере *lpstrFilter*, так и введенный пользователем. Система использует строки для инициализации определенного пользователем фильтра при следующем создании блока диалога. Если поле *nFilterIndex* содержит нуль, то блок диалога использует этот настраиваемый фильтр. Блок диалога не сохраняет введенный пользователем образец фильтра, если это поле содержит NULL. В противном случае размер этого буфера должен быть указан в поле *nMaxCustFilter*.

nMaxCustFilter — определяет размер буфера, на который указывает *lpstrCustomFilter*. При работе со строками ANSI размер указывается в байтах, а для строк Unicode — в символах. Размер буфера должен быть не меньше 40 символов. Значение этого поля игнорируется, если в *lpstrCustomFilter* записан NULL.

nFilterIndex — определяет индекс (начиная с единицы) текущего выбранного фильтра в элементе управления File Types (Тип файлов). Нулевой индекс говорит о том, что используется настраиваемый фильтр *lpstrCustomFilter*. Можно определить значение индекса при открытии блока диалога. При выборе файла в этом поле сохраняется значение индекса текущего фильтра. Если *nFilterIndex* равен нулю и *lpstrCustomFilter* равен NULL, то используется первый фильтр из буфера *lpstrFilter*. Если при этом также *lpstrFilter* равен NULL, то фильтр не используется вообще.

lpstrFile — указывает на буфер, который содержит имя файла (или путь), используемое по умолчанию при инициализации элемента управления File Name (Имя Файла), или NULL, если инициализация не требуется. При успешном возврате из функций *GetOpenFileName* или *GetSaveFileName* этот буфер будет содержать полный путь к выбранному файлу. Если размера буфера не достаточно для хранения строки, то указанные функции возвращают FALSE, и первые два байта в этом поле содержат значение требуемого размера буфера.

nMaxFile — определяет размер в байтах (строки ANSI) или символах (строки Unicode) буфера, на который указывает *lpstrFile*. Минимальный размер буфера — 256 символов.

lpstrFileTitle — указывает на буфер, в который после выбора будет записано имя файла с расширением (без информации о пути).

nMaxFileTitle — определяет размер в байтах (строки ANSI) или символах (строки Unicode) буфера, на который указывает *lpstrFileTitle*. Значение в этом поле игнорируется, если *lpstrFileTitle* равен NULL.

lpstrInitialDir — указывает на строку, которая определяет начальный директорий. Чтобы начать работу в текущем директории, в это поле следует записать NULL.

lpstrTitle — указывает на строку, которая определяет заголовок блока диалога. Если это поле содержит NULL, будут использованы стандартные заголовки "Open" и "Save As...".

Flags — это поле позволяет задать различные режимы выбора файла, влияющие на внешний вид блока диалога. Допустимыми являются комбинации следующих флагов:

OFN_ALLOWMULTISELECT — Разрешает выбор нескольких файлов одновременно. При выборе пользователем более одного файла буфер, на который указывает lpstrFile, будет содержать путь к текущему директорию, за которым через пробел следуют имена выбранных файлов с расширениями, разделенные между собой пробелами. Если блок диалога создан на основе шаблона, определенного пользователем, то элемент управления LIST BOX должен иметь стиль LBS_EXTENDEDSEL.

OFN_CREATEPROMPT — Если пользователь определяет несуществующий файл, то выводится блок диалога, в котором предлагается создать файл. Этот флаг устанавливается автоматически при использовании флагов OFN_PATHMUSTEXIST и OFN_FILEMUSTEXIST.

OFN_ENABLEHOOK — Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook.

OFN_ENABLETEMPLATE — Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога в модуле, определяемом hInstance.

OFN_ENABLETEMPLATEHANDLE — Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpTemplateName.

OFN_EXPLORER — При установленном флаге функции Windows NT GetOpenFileName и GetSaveFileName возвращают ошибку. Этот флаг может применяться только в Windows 95 и указывает на то, что при создании блоков диалога Open (Открытие) и Save As (Сохранить) используется новый стиль EXPLORER. Этот стиль используется по умолчанию, независимо от того, установлен ли этот флаг. Его применение необходимо только в том случае, если используется функция фильтра или настраиваемый шаблон. Для того чтобы не использовать новый стиль блоков диалога, необходимо сбросить этот флаг и обеспечить возврат значения FALSE функцией фильтра.

OFN_EXTENSIONDIFFERENT — Устанавливается после возвращения из функции и указывает, что расширение выбранного файла отличается от того, которое было определено в поле lpstrDefExt. Этот флаг не устанавливается, если перед вызовом функции в поле lpstrDefExt было записано значение NULL или файл не имеет расширения.

OFN_FILEMUSTEXIST	Определяет, что в поле File Name (Имя Файла) пользователь может вводить только имена существующих файлов. В противном случае на экран выводится предупреждающее сообщение. Если этот флаг определен, то автоматически устанавливается флаг OFN_PATHMUSTEXIST.
OFN_HIDEREADONLY	Предписывает убрать из блока диалога флажок Read Only (Только чтение).
OFN_LONGNAMES	Предписывает блокам диалога Open (Открытие) и Save As (Сохранить) отображать длинные имена файлов. В противном случае они выводятся в формате 8.3. Если для Windows 95 установлен флаг OFN_EXPLORER, то рассматриваемый флаг игнорируется, т. к. в этом случае всегда отображаются длинные имена файлов.
OFN_NOCHANGEDIR	Устанавливает первоначальный текущий директорий, если пользователь изменил его при поиске файлов.
OFN_NODEREFERENCELINKS	Предписывает блоку диалога возвратить путь и имя выбранного сокращателя (shortcut) — файла с расширением .LNK. В противном случае блок диалога возвращает путь и имя файла, на который ссылается сокращатель.
OFN_NOLONGNAMES	Определяет, что в списке File Name (Имя Файла) не отображаются длинные имена файлов. Этот флаг игнорируется для Windows 95, если установлен флаг OFN_EXPLORER, т. к. в этом случае всегда отображаются длинные имена файлов.
OFN_NONETWORKBUTTON	Скрывает и блокирует кнопку Network (Сеть).
OFN_NOREADONLYRETURN	Определяет, что выбранный файл не имеет атрибута Read Only (Только чтение) и не располагается в защищенном от записи директории.
OFN_NOTESTFILECREATE	Определяет, что файл не создается перед закрытием блока диалога, а также отсутствие проверки на защиту от записи, переполнение диска или сетевую защиту.
OFN_NOVALIDATE	Определяет, что стандартные блоки диалога допускают наличие неразрешенных символов в именах возвращаемых файлов.
OFN_OVERWRITEPROMPT	Предписывает блоку диалога Save As (Сохранить) выводить окно запроса на перезапись, если выбранный файл уже существует.
OFN_PATHMUSTEXIST	Определяет, что пользователь может использовать только имеющиеся путь и имена файлов. В противном случае на экран выводится предупреждающее сообщение.

OFN_READONLY	При отображении блока диалога флажок Read Only (Только чтение) будет находиться во включенном состоянии, а при закрытии блока диалога указывает состояние флажка.
OFN_SHAREWARE	Устанавливается после возвращения из функции OpenFile и указывает, что произошла ошибка при совместном доступе к файлу в сети.
OFN_SHOWHELP	Предписывает блоку диалога отображать кнопку Help (?)

nFileOffset — определяет смещение первого символа имени файла относительно начала буфера, на который указывает *lpstrFile*.

nFileExtension — определяет смещение первого символа расширения имени файла относительно начала буфера, на который указывает *lpstrFile*. Если пользователь не задал расширение, и *lpstrDefExt* содержит NULL, то в этом поле указывается смещение до завершающего нулевого символа. Если в качестве последнего символа имени файла пользователь ввел ".", то в это поле записывается нуль.

lpstrDefExt — указатель на буфер, который содержит расширение имени файла, используемое по умолчанию. Эта строка добавляется к имени выбранного файла, если расширение не было указано. Она может иметь произвольную длину, но добавляются только первые три символа, среди которых не должно быть точки.

lCustData — определяет данные, которые система посылает функции фильтра *lpfnHook*. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр *lParam* этого сообщения указывает на структуру *OPENFILENAME*, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю *lCustData*.

lpfnHook — указатель на функцию фильтра. Это поле игнорируется, если не установлен флаг OFN_ENABLEHOOK.

Для Windows NT и Windows 95 (флаг OFN_EXPLORER не установлен) это поле содержит указатель на функцию фильтра *OFNHookProcOldStyle*, которая получает сообщения, предназначенные для блока диалога. Она возвращает FALSE, чтобы передать сообщение процедуре блока диалога, выполняемой по умолчанию, и TRUE, чтобы игнорировать его.

Для Windows 95, в случае установленного флага OFN_EXPLORER, это поле содержит указатель на функцию фильтра *OFNHookProc*, которая получает извещения от блока диалога. Кроме того, она получает сообщения для всех дополнительных элементов управления, которые вы определили в шаблоне блока диалога. Для стандартных элементов управления эта функция сообщений не получает.

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг OFN_ENABLETEMPLATE.

Ниже приведено краткое описание основных функций класса.

CFileDialog::**CFileDialog**(
BOOL bOpenFileDialog,
LPCTSTR lpszDefExt = NULL,
LPCTSTR lpszFileName = NULL,
DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
LPCTSTR lpszFilter = NULL,
CWnd *pParentWnd = NULL) —

конструктор класса, создает объект "стандартный блок диалога".

Если параметр *bOpenDialog* равен TRUE, создается объект "блок диалога File Open" (Открытие файла), и объект "блок диалога File Save As" (Сохранить файл), если *bOpenDialog* равен FALSE. Остальные параметры, передаваемые в функцию *CFileDialog*, вам уже известны из описания структуры *OPENFILENAME*. Исключение составляет лишь параметр *lpszFilter*. Здесь, в отличие от соответствующего поля структуры, в качестве разделителя между названиями и образцами фильтров используется символ "|", вместо пробела. Кроме того, строка должна заканчиваться двумя символами "|", за которыми следует завершающий NULL.

virtual int CFileDialog::**DoModal**() —

вызов этой функции выводит на экран модальный стандартный блок диалога для открытия или сохранения файла. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_ofn, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от того, при помощи какой кнопки пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки.

Для того чтобы получить имя файла, определенное пользователем в блоке диалога, в классе реализованы несколько функций:

virtual CString CFileDialog::**GetPathName**() —

возвращает полное имя файла, включая путь.

virtual CString CFileDialog::**GetFileName**() —

возвращает имя файла с расширением.

virtual CString CFileDialog::**GetFileExt**() —

возвращает расширение имени файла.

virtual CString CFileDialog::**GetFileTitle**() —

возвращает имя файла без расширения.

Если при использовании этих функций для блока диалога установлен флаг OFN_ALLOWMULTISELECT, то возвращаемая строка содержит последовательность строк, первая из которых является полным путем к этой группе файлов, а последующие определяют имена выбранных файлов. В этом случае для извлечения имен файлов следует использовать функции:

POSITION CFileDialog::**GetStartPosition**() –
возвращает позицию первого имени файла в списке, если для блока диалога разрешен множественный выбор, или NULL, если список пуст.

CString CFileDialog::**GetNextPathName**(POSITION &pos) –
возвращает следующее полное имя файла из группы файлов, выбранных в блоке диалога. В параметре pos возвращается позиция текущего имени файла, либо NULL, если достигнут конец списка.

BOOL CFileDialog::**GetReadOnlyPref**() –
позволяет определить, был ли установлен (возвращается TRUE) флажок Read Only (Только чтение) в блоке диалога.

Оставшиеся обработчики сообщений переопределяются достаточно редко, и мы приводим их для того, чтобы иметь полное представление обо всех возможностях класса:

void CFileDialog::**OnShareViolation**() –
вызывается, когда происходит общее нарушение.

void CFileDialog::**OnFileNameOK**() –
вызывается, чтобы зафиксировать имя файла, введенное в блоке диалога.

void CFileDialog::**OnLBSelChangedNotify**() –
вызывается, когда изменился выбор в поле списка.

void CFileDialog::**OnInitDone**() –
вызывается для обработки извещения CDN_INITDONE.

void CFileDialog::**OnFileNameChange**() –
вызывается для обработки извещения CDN_SELCHANGE.

void CFileDialog::**OnFolderChange**() –
вызывается для обработки извещения CDN_FOLDERCHANGE.

void CFileDialog::**OnTypeChange**() –
вызывается для обработки извещения CDN_TYPECHANGE.

В заключение, как и обещали, приведем небольшой пример (приложение DlgSmpl), показывающий принципы настройки блока диалога этого класса. За основу мы взяли уже рассмотренный пример MDI-приложения. Поэтому здесь приведены только те добавления, которые иллюстрируют рассматриваемые положения.

```
////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////
class CMainFrame : public CMDIFrameWnd
{
...
// Generated message map functions
```

```
protected:
  //{{AFX_MSG(CMainFrame)

...

  afx_msg void OnOpen();
  afx_msg void OnClose();
  //}}AFX_MSG

DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////

// Настройка и вывод на экран стандартного блока диалога
// File Open (Открытие файла)

void CMainFrame::OnOpen()
{
  // Заголовок блока диалога
  TCHAR title[] = "Пример настройки блока диалога \"Открытие
файла\"";

  // Создаем объект "блок диалога"
  CFileDialog fileDlg(TRUE);

  // Формируем шаблоны имен отображаемых файлов
  CString str("Все файлы (*.*)); str += (TCHAR)NULL;
  str += "*.*"; str += (TCHAR)NULL;
  str += ("Файлы исходных текстов(cpp, h)); str += (TCHAR)NULL;
  str += "*.cpp; *.h"; str += (TCHAR)NULL;
  str += "Исполняемые файлы (*.exe); str += (TCHAR)NULL;
  str += "*.exe"; str += (TCHAR)NULL;

  // Устанавливаем фильтр
  fileDlg.m_ofn.lpstrFilter = str;

  // При открытии блока диалога отображается список
  // файлов, имеющих расширение .cpp или .h
  fileDlg.m_ofn.nFilterIndex = 2;

  // Готовим буфер для полного имени выбранного файла
  TCHAR strName[_MAX_PATH];

  // Этот буфер должен быть пуст
  strName[0] = (TCHAR)NULL;
  fileDlg.m_ofn.lpstrFile = strName;

  // Передаем заголовок блока диалога
  fileDlg.m_ofn.lpstrTitle = title;
```

```
  // К флагам, устанавливаемым по умолчанию, добавляем возможность
  // выбора нескольких файлов одновременно
  fileDlg.m_ofn.Flags |= OFN_ALLOWMULTISELECT;

  // Выводим блок диалога на экран
  if (fileDlg.DoModal() == IDOK)
  {
    // Если блок диалога закрыт нажатием кнопки "Открыть",
    // в заголовок текущего активного дочернего окна
    // выводим имя файла, выбранного первым
    CChildWnd *pWnd = (CChildWnd *)GetActiveFrame();
    pWnd->SetWindowText(fileDlg.m_ofn.lpstrFile);
  }
}
```

При использовании приведенных установок блок диалога будет иметь внешний вид показанный на рис. 34.

А после закрытия блока диалога кнопкой "Открыть" получим общий вид главного окна приложения, показанный на рис. 35.



Рис. 34. Блок диалога типа File Open (Открытие файла)

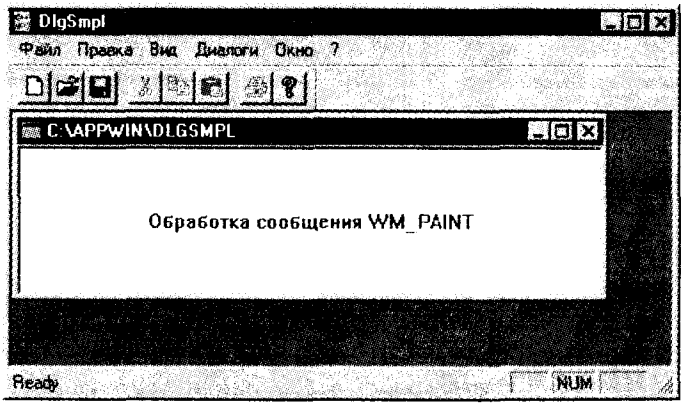


Рис. 35. Главное окно приложения DlgSmpl с окном открытого файла

```
// Настройка и вывод на экран стандартного блока диалога
// File Save As (Сохранить файл)
void CMainFrame::OnClose()
{
  // Создаем объект "блок диалога"
  CFileDialog fileDlg(FALSE);

  // Определяем расширение, которое будет присваиваться
  // имени файла по умолчанию
  CString ext("hhp");
  fileDlg.m_ofn.lpstrDefExt = ext;

  // Заголовок блока диалога
  TCHAR title[] = "Пример настройки блока диалога \"Сохранить файл\"";
  fileDlg.m_ofn.lpstrTitle = title;

  // Задаем вывод флажка Read Only (Только чтение) и
  // снимаем режим Проводник (Explorer)
  fileDlg.m_ofn.Flags &= ~(OFN_HIDEREADONLY  | OFN_EXPLORER);

  // Выводим блок диалога на экран
  if(fileDlg.DoModal() == IDOK)
  {
    // Если блок диалога закрыт нажатием кнопки OK,
    // в заголовок текущего активного дочернего окна
    // выводим имя файла
    CChildWnd *pWnd = (CChildWnd *)GetActiveFrame();
    pWnd->SetWindowText(fileDlg.GetFileName());
  }
}
```

При использовании приведенных установок блок диалога будет иметь внешний вид показанный на рис. 36.

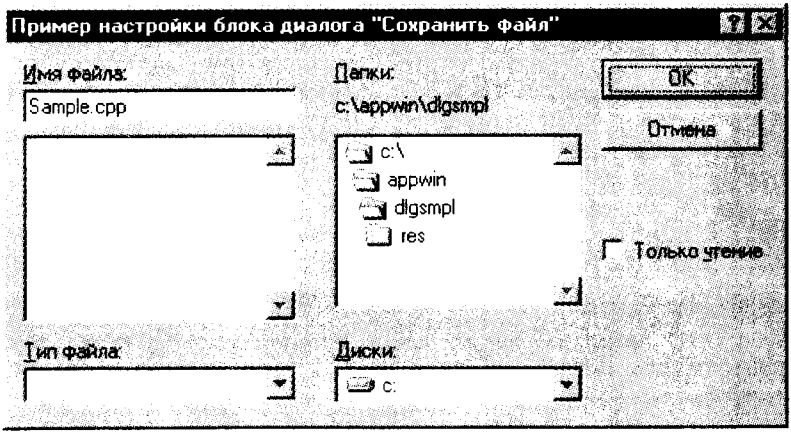


Рис. 36. Блок диалога типа File Save As (Сохранить файл)

Дальнейшие эксперименты по настройке внешнего вида представленных блоков диалога можно провести самостоятельно, наблюдая за теми изменениями, которые возникают при использовании того или иного параметра. Нам же осталось привести шаблон меню из файла ресурсов, поскольку оно изменилось по сравнению с исходным из приложения MDI:

```
//////////////////////////////////////////////////////////////
// Файл MDI.rc
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////////
...
IDR_CHILD MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "Файл"
  BEGIN
    MENUITEM "Создать\tCtrl+N",          ID_NEW
    MENUITEM "Открыть...\tCtrl+O",       ID_OPEN
    MENUITEM "Закрыть",                  ID_FILE_CLOSE
    MENUITEM "Сохранить\tCtrl+S",        ID_FILE_SAVE
    MENUITEM "Сохранить как...",         ID_CLOSE
    MENUITEM SEPARATOR
    MENUITEM "Список файлов",            ID_FILE_MRU_FILE1, GRAYED
    MENUITEM SEPARATOR
    MENUITEM "Выход",                    ID_APP_EXIT
  END
  POPUP "Правка"
  BEGIN
    MENUITEM "Отменить\tCtrl+Z",         ID_EDIT_UNDO
    MENUITEM SEPARATOR
    MENUITEM "Вырезать\tCtrl+X",         ID_EDIT_CUT
    MENUITEM "Копировать\tCtrl+C",       ID_EDIT_COPY
    MENUITEM "Вставить\tCtrl+V",         ID_EDIT_PASTE
END
  POPUP "Вид"
  BEGIN
    MENUITEM "Набор инструментов",       ID_VIEW_TOOLBAR
    MENUITEM "Строка состояния",         ID_VIEW_STATUS_BAR
  END
  POPUP "Диалоги"
  BEGIN
    POPUP "Цвет"
    BEGIN
      MENUITEM "Черный",                 IDM_BLACK
      MENUITEM "Красный",                IDM_RED
      MENUITEM "Синий",                  IDM_BLUE
      MENUITEM "Зеленый",                IDM_GREEN
```

```
      MENUITEM "Белый",                    IDM_WHITE
      MENUITEM "Изменить палитру ...",     IDM_CUSTOM
    END
    MENUITEM "Шрифт ...",                  IDM_FONT
  END
  POPUP "Окно"
  BEGIN
    MENUITEM "Новое",                      ID_WINDOW_NEW
    MENUITEM "Каскадом",                   ID_WINDOW_CASCADE
    MENUITEM "Без перекрытия",             ID_WINDOW_TILE_HORZ
    MENUITEM "Упорядочить пиктограммы",    ID_WINDOW_ARRANGE
  END
  POPUP "?"
  BEGIN
    MENUITEM "О программе",                ID_APP_ABOUT
  END
END
...
```

Этот же шаблон используется в двух оставшихся примерах, рассматриваемых в этой главе.

Класс CColorDialog

Этот класс отвечает за создание модального блока диалога для выбора и/или создания цветов, которые определены для вашего дисплея.

Основным членом класса, позволяющим осуществлять настройку объекта этого класса, является структура типа *CHOOSECOLOR*:

CHOOSECOLOR CColorDialog::**m_cc**

```
typedef struct {
  DWORD lStructSize;
  HWND hwndOwner;
  HWND hInstance;
  COLORREF rgbResult;
  COLORREF *lpCustColor;
  DWORD Flags;
  LPARAM lCustData;
  LPCCHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
} CHOOSECOLOR; —
```

служит для настройки внешнего вида блока диалога. Структура имеет следующие поля:

lStructSize — размер структуры в байтах.

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL.

hInstance — определяет идентификатор блока памяти (установлен флаг CC_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг CC_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется.

rgbResult — при создании блока диалога определяет первоначально выбранный цвет, если при этом в параметре *Flags* установлен флаг CC_RGBINIT. В противном случае выбирается черный цвет. При завершении работы блока диалога в это поле записывается цвет, выбранный пользователем.

lpCustColors — указатель на массив, содержащий 16 значений COLORREF для настраиваемых цветов блока диалога. Если пользователь изменяет значения этих цветов, то система сама модифицирует содержимое массива. Для того чтобы иметь возможность в дальнейшем использовать эти новые значения, массив должен быть объявлен как статический.

lCustData — определяет данные, которые система посылает функции фильтра *lpfnHook*. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр *lParam* этого сообщения указывает на структуру *CHOOSECOLOR*, определенную при создании блока диалога и может использоваться для доступа к полю *lCustData*.

lpfnHook — указатель на функцию фильтра *CCHookProc*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг CC_ENABLEHOOK.

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*, подставляемого вместо стандартного шаблона блока диалога. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг CC_ENABLETEMPLATE.

Flags — это поле определяет множество битовых флагов, которые можно использовать при инициализации блока диалога. Допустимыми являются комбинации следующих флагов:

CC_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook. Используется только при инициализации блока диалога.
CC_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога в модуле, определяемом полем hInstance. Используется только при инициализации блока диалога.
CC_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpTemplateName. Используется только при инициализации блока диалога.

CC_FULLOPEN	Предписывает блоку диалога отображать дополнительные элементы управления, необходимые для создания произвольных цветов. Если этот флаг не установлен, то для отображения дополнительных элементов управления пользователь должен использовать кнопку Define Custom Color (Определить цвет).
CC_PREVENTFULLOPEN	Блокирует кнопку Define Custom Color (Определить цвет), т. е. пользователь не может определить произвольный цвет.
CC_RGBINIT	Устанавливает цвет, заданный в поле rgbResult, в качестве используемого по умолчанию.
FR_SHOWHELP	В блоке диалога отображается кнопка Help (?)

Ниже приведено описание основных функций класса.

CColorDialog::**CColorDialog**(
 COLORREF clrInit = 0,
 DWORD dwFlags = 0,
 CWnd *pParentWnd = NULL) —
конструктор объекта "блок диалога". Параметр clrInit определяет первоначально выбранный цвет в формате RGB (Red — красный, Green — зеленый, Blue — синий).

virtual int CColorDialog::**DoModal**() —
вызов этой функции выводит на экран модальный стандартный блок диалога Windows для выбора цвета. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_cc, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки.

COLORREF CColorDialog::**GetColor**() —
позволяет получить информацию о цвете, выбранном пользователем. Вызывается после того, как завершилась функция CColorDialog::DoModal.

static COLORREF* CColorDialog::**GetSavedCustomColors**() —
возвращает указатель на массив, содержащий 16 дополнительных цветов, созданных пользователем. Вызывается после завершения функции CColorDialog::DoModal. Те цвета из 16, которые пользователь не настраивал, определяют белый цвет.

void CColorDialog::**SetCurrentColor**(COLORREF clr) —
вызывается из функции CColorDialog::OnColorOK для принудительного выбора цвета, заданного в параметре clr.

virtual BOOL CColorDialog::**OnColorOK**() —
переопределение этой функции практически не требуется, т. к. реализация по умолчанию обеспечивает полную проверку правильности введенного цвета. Если вы переопределяете эту функцию, то для нормального завершения работы блока диалога она должна возвращать FALSE.

В заключение обещанный пример.

```
////////////////////////////////////////////////////////////////
// Файл ChildWnd.h
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////////

class CChildWnd : public CMDIChildWnd
{

...

// Implementation
protected:
  static CMenu menu;
  static COLORREF aCustom[16];
  UINT nIDColor;
  COLORREF clrText;

 ...

// Generated message map functions
  //{{AFX_MSG(CChildWnd)

...

  afx_msg void OnCustom();
  afx_msg void OnColor(UINT nID);
  afx_msg void OnUpdateColor(CCmdUI *pCmdUI);
  afx_msg void OnFont();
  //}}AFX_MSG

DECLARE_MESSAGE_MAP()
};


////////////////////////////////////////////////////////////////
// Файл ChildWnd.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////////

// В качестве дополнения к примеру здесь используются
// макросы ON_COMMAND_RANGE и ON_UPDATE_COMMAND_UI_RANGE
// для обработки ряда идентификаторов команд

BEGIN_MESSAGE_MAP(CChildWnd, CMDIChildWnd)
  //{{AFX_MSG_MAP(CChildWnd)
  ON_WM_PAINT()
  ON_COMMAND(IDM_CUSTOM, OnCustom)
  ON_COMMAND(IDM_FONT, OnFont)
  //}}AFX_MSG_MAP
  ON_COMMAND_RANGE(IDM_BLACK, IDM_WHITE, OnColor)
```

```
  ON_UPDATE_COMMAND_UI_RANGE(IDM_BLACK, IDM_WHITE, OnUpdateColor)
END_MESSAGE_MAP()

...

// Соответствующие обработчики
void CChildWnd::OnColor(UINT nID)
{
  nIDColor = nID;
  clrText = colorArray[nIDColor - IDM_BLACK];

  Invalidate();
}

void CChildWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}

// Обработчик для настройки цветов
void CChildWnd::OnCustom()
{
  // Создаем объект "блок диалога"
  CColorDialog clrDlg;

  clrDlg.m_cc.lStructSize = sizeof(clrDlg.m_cc);
  clrDlg.m_cc.rgbResult = colorArray[nIDColor - IDM_BLACK];
  clrDlg.m_cc.lpCustColors = aCustom;

  // Если не используются установки по умолчанию, не забывайте
  // устанавливать флаг CC_ENABLEHOOK, чтобы подключить необходимую
  // для обработки функцию обратного вызова
  clrDlg.m_cc.Flags = CC_ENABLEHOOK; // CC_FULLOPEN | CC_RGBINIT

  // Выводим блок диалога на экран
  if(clrDlg.DoModal() == IDOK)
  {
    // Изменяем цвет выводимого текста при закрытии
    // блока диалога кнопкой OK
    nIDColor = IDM_CUSTOM;
    colorArray[nIDColor - IDM_BLACK] = clrDlg.GetColor();
    OnColor(nIDColor);
  }
}
```

На рис. 37 представлен внешний вид рассматриваемого блока диалога.

При используемых параметрах по умолчанию выбирается черный цвет. Если же при создании использовать флаги CC_FULLOPEN и CC_RGBINIT, то внешний вид существенно изменится (рис. 38).

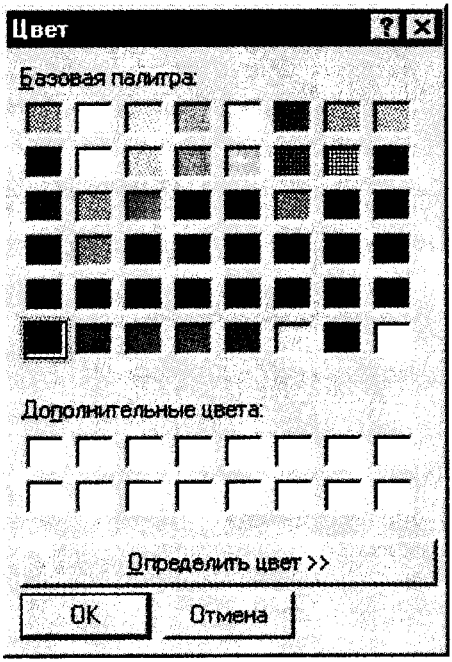


Рис. 37. Блок диалога для выбора цвета

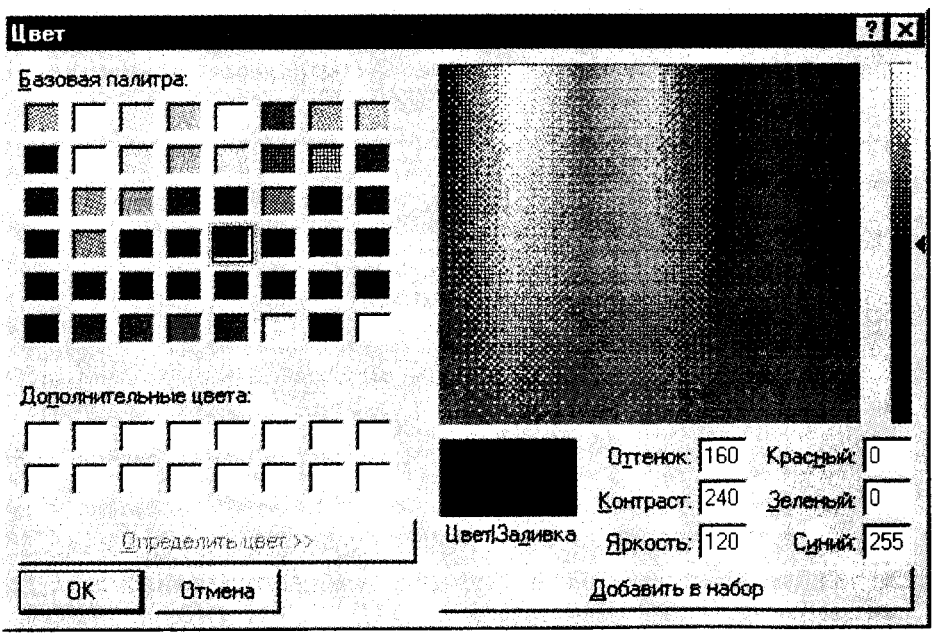


Рис. 38. Расширенный вариант блока диалога для выбора цвета

Класс CFontDialog

Этот класс позволяет создать стандартный модальный блок диалога, предназначенный для выбора одного из зарегистрированных в системе шрифтов, а также его стиля, размера, цвета и т. д.

За внешний вид стандартного блока диалога, создаваемого на основе этого класса, отвечает структура типа *CHOOSEFONT*:

CHOOSEFONT CFontDialog::**m_cf**;

```
typedef struct {
  DWORD lStructSize;
  HWND hwndOwner;
  HDC hDC;
  LPLOGFONT lpLogFont;
  int iPointSize;
  DWORD Flags;
  DWORD rgbColors;
  LPARAM lCustData;
  LPCFHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
  HINSTANCE hInstance;
  LPTSTR lpszStyle;
  WORD nFontType;
  WORD __MISSING ALIGNMENT__;
  int nSizeMin;
  int nSizeMax;
} CHOOSEFONT; —
```

служит для настройки внешнего вида блока диалога. Структура имеет следующие поля:

lStructSize — размер структуры в байтах.

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL.

hDC — дескриптор контекста устройства (или информационного контекста) принтера, шрифты которого должны быть отражены в блоке диалога. Это поле используется только в случае, если в поле *Flags* установлен один из флагов CF_BOTH или CF_PRINTERFONTS.

lpLogFont — указатель на структуру *LOGFONT*, которая содержит значения, определяющие параметры шрифта.[1] Если при создании блока диалога установлен флаг CF_INITTOLOGFONTSTRUCT, то параметры этой структуры будут использоваться в качестве начальных. При завершении работы с блоком диалога поля этой структуры будут заполнены параметрами шрифта, выбранного пользователем.

---

[1] Сама структура будет нами описана во втором томе при описании работы со шрифтами.

iPointSize — определяет размер выбранного шрифта в десятых долях пункта. Поле заполняется после закрытия блока диалога.

rgbColors — цвет символов шрифта, который будет выбран при отображении блока диалога. При этом должен быть установлен флаг CF_EFFECTS.

lCustData — определяет данные, которые приложение может передать функции фильтра *lpfnHook*. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр *lParam* этого сообщения указывает на структуру *CHOOSEFONT*, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю *lCustData*.

lpfnHook — указатель на функцию фильтра *CFHookProc*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг CF_ENABLEHOOK.

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг CF_ENABLETEMPLATE.

hInstance — определяет идентификатор блока памяти (установлен флаг CF_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг CF_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется.

lpszStyle — указатель на буфер, содержащий данные стиля. Если установлен флаг CF_USESTYLE, эта строка используется для инициализации стиля шрифта.

__MISSING ALIGNMENT__ — используется для выравнивания.

nSizeMin — определяет минимальный размер шрифта, который пользователь может выбрать. При этом должен быть установлен флаг CF_LIMITSIZE.

nSizeMax — определяет максимальный размер шрифта, который пользователь может выбрать. При этом должен быть установлен флаг CF_LIMITSIZE.

Flags — флаги инициализации, которые также определяют выбор пользователя после закрытия блока диалога. Можно использовать комбинации следующих значений:

CF_APPLY — Предписывает блоку диалога отобразить кнопку Apply (Применить). Для обработки действия этой кнопки необходимо определить функцию фильтра, которая может обработать сообщение WM_CHOOSEFONT_GETLOGFONT для получения адреса структуры *LOGFONT*, содержащей текущие параметры шрифта.

CF_ANSIONLY — Для Windows 95 практически не используется. В списке доступных шрифтов отображаются только шрифты в кодировке ANSI. Для Windows NT определяет, что в списке отображаются все шрифты, которые используют наборы символов Windows и Unicode.

CF_BOTH	В списке шрифтов отображаются и экранные, и принтерные шрифты.
CF_TTONLY	В списке шрифтов отображаются только масштабируемые шрифты TrueType.
CF_EFFECTS	Предписывает блоку диалога отображать элементы управления, с помощью которых можно задавать цвет букв, создавать подчеркнутые и перечеркнутые шрифты. В этом случае необходимо проинициализировать поле rgbColor этой структуры, и поля lfStrikeOut и lfUnderLine структуры *LOGFONT*.
CF_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook.
CF_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога Font (Шрифт) в модуле, определяемом hInstance.
CF_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога Font (Шрифт). В этом случае игнорируется содержимое поля lpTemplateName.
CF_FIXEDPITCHONLY	Разрешает выбирать только шрифты с фиксированной шириной символов.
CF_FORCEFONTEXIST	Определяет, что если пользователь пытается выбрать несуществующий шрифт, то выдается сообщение об ошибке.
CF_INITTOLOGFONTSTRUCT	При инициализации блока диалога используется содержимое структуры *LOGFONT*, адрес которой передается через поле lpLogFont.
CF_LIMITSIZE	При выборе шрифта учитывается содержимое полей nSizeMin и nSizeMax.
CF_NOOEMFONT	Запрещает выбирать векторные шрифты. Этот флаг аналогичен флагу CF_VEKTORFONTS.
CF_NOFACESEL	При использовании структуры *LOGFONT* для инициализации блока диалога не отображается имя шрифта.
CF_NOSCRIPTSEL	Блокирует раскрывающийся список SCRIPT COMBO BOX (Подчеркивание). Используется только в Windows 95 и при инициализации блока диалога.
CF_NOSTYLESEL	При использовании структуры *LOGFONT* для инициализации блока диалога не отображается стиль шрифта.
CF_NOSIZESEL	Интерфейсу GDI запрещается эмуляция шрифтов.
CF_VECTORFONTS	Запрещает выбирать векторные шрифты.

CF_NOVERTFONTS	Используется только в Windows 95 и предписывает блоку диалога отображать только горизонтально ориентированные шрифты.
CF_PRINTERFONTS	Предписывает блоку диалога отображать только шрифты, поддерживаемые принтером, контекст которого задан в поле hDC.
CF_SCALABLEONLY	Разрешает выбор только масштабируемых шрифтов, которые включают векторные, масштабируемые шрифты принтера, шрифты TrueType.
CF_SCREENFONTS	Предписывает блоку диалога отображать только экранные шрифты, поддерживаемые системой.
CF_SCRIPTSONLY	Используется только в Windows 95 и разрешает выбор всех шрифтов, кроме OEM и Symbol. Используется вместо CF_ANSIONLY.
CF_SELECTSCRIPT	Используется только в Windows 95 и разрешает отображать только шрифты, характеристики которых определены в поле lfCharSet структуры *LOGFONT*.
CF_SHOWHELP	В блоке диалога отображается кнопка Help (?)
CF_USESTYLE	Определяет, что поле lpszStyle указывает на буфер, содержащий данные стиля, который отображается при инициализации блока диалога.
CF_WYSIWYG	Разрешает выбирать только те шрифты, которые доступны и для отображения на экране, и для печати на принтере. При этом должны быть также установлены флаги CF_BOTH и CF_SCALABLEONLY.

nFontType — определяет тип выбираемого шрифта. Можно использовать одно из следующих значений:

BOLD_FONTTYPE	Полужирное начертание. Эта информация продублирована в поле lfWeight структуры *LOGFONT* и эквивалентна FW_BOLD.
ITALIC_FONTTYPE	Курсивное начертание. Эта информация продублирована в поле lfWeight структуры *LOGFONT* и эквивалентна FW_ITALIC.
REGULAR_FONTTYPE	Обычное начертание (не полужирное и не курсивное). Эта информация продублирована в поле lfWeight структуры *LOGFONT* и эквивалентна FW_REGULAR.
PRINTER_FONTTYPE	Принтерный шрифт.
SCREEN_FONTTYPE	Экранный шрифт.
SIMULATED_FONTTYPE	GDI может эмулировать этот шрифт.

Ниже приведено описание основных функций этого класса.

CFontDialog::**CFontDialog**(
LPLOGFONT lplfInitial = NULL,
DWORD dwFlags = CF_EFFECTS | CF_SCREENFONTS,
CDC *pdcPrinter = NULL,
CWnd *pParentWnd = NULL) —
конструктор объекта для стандартного модального блока диалога Windows Font (Шрифт). Он автоматически записывает значения, принятые по умолчанию, в структуру *CHOOSEFONT*.

virtual int CFontDialog::**DoModal**() —
выводит на экран модальный стандартный блок диалога для выбора шрифта. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_cf, то делать это следует после создания объекта "блок диалога", но до вызова функции DoModal. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки. Если функция возвратила IDOK, то можно вызывать другие функции класса для получения информации о выборе пользователя.

void CFontDialog::**GetCurrentFont**(LPLOGFONT lplf) —
заполняет поля структуры *LOGFONT* характеристиками текущего выбранного шрифта.

CString CFontDialog::**GetFaceName**() —
позволяет получить имя выбранного шрифта.

CString CFontDialog::**GetStyleName**() —
позволяет получить стиль выбранного шрифта.

int CFontDialog::**GetSize**() —
позволяет получить размер (в десятых долях пункта) выбранного шрифта.

COLORREF CFontDialog::**GetColor**() —
позволяет получить цвет выбранного шрифта.

int CFontDialog::**GetWeight**() —
позволяет получить вес[1] выбранного шрифта.

BOOL CFontDialog::**IsStrikeOut**() —
определяет, будет ли выбранный шрифт зачеркнутым.

BOOL CFontDialog::**IsUnderLine**() —
определяет, будет ли выбранный шрифт подчеркнутым.

BOOL CFontDialog::**IsItalic**() —
определяет, будет ли выбранный шрифт наклонным.

Приведенный ниже пример иллюстрирует основные принципы настройки блока диалога этого класса. Описание класса *CChildWnd* приведено в разделе "Класс CColorDialog".

---

[1] Подробно об этом и других параметрах шрифта читайте во втором томе.

```
////////////////////////////////////////////////////////////////
// Файл ChildWnd.cpp
// Copyright (c) 1997 Мешков А., Тихомиров Ю.
////////////////////////////////////////////////////////////////

// Обработчик для настройки блока диалога выбора шрифта
void CChildWnd::OnFont()
{
  CFontDialog fontDlg;

  fontDlg.m_cf.Flags = CF_BOTH | CF_APPLY | CF_ENABLEHOOK;

  // Инициализация начального цвета
  fontDlg.m_cf.rgbColors = RGB(255, 255, 255);

  // Эти установки становятся доступны только при
  // использовании флага CF_EFFECTS
  fontDlg.m_cf.Flags |= CF_EFFECTS;

  // Выводим блок диалога на экран
  if (fontDlg.DoModal() == IDOK)
  {
    // Вывод имени выбранного шрифта в текущее
    // активное дочернее окно
    CClientDC dc(this);

    // Устанавливаем цвет для выводимого текста
    // в соответствии со сделанным выбором
    dc.SetTextColor(fontDlg.GetColor());
    dc.SetBkColor(::GetSysColor(COLOR_WINDOW));

    // Выводим в дочерние окна текст
    TCHAR strText[256];
    strcpy(strText, "Выбран шрифт ");
    strcat(strText, fontDlg.GetFaceName());
    TCHAR str[32];
    sprintf(str, " размером %d пунктов", fontDlg.GetSize()/10);
    strcat(strText, str);
    dc.TextOut(10, 10, strText, strlen(strText));
  }
}
```

Внешний вид блока диалога изображен на рис. 39. При выборе какого-либо шрифта его имя и размер отображаются в текущем активном дочернем окне, как показано на рис. 40.

Дальнейшие эксперименты по настройке внешнего вида представленных блоков диалога можно провести самостоятельно, наблюдая за изменениями, которые возникают при использовании того или иного параметра.

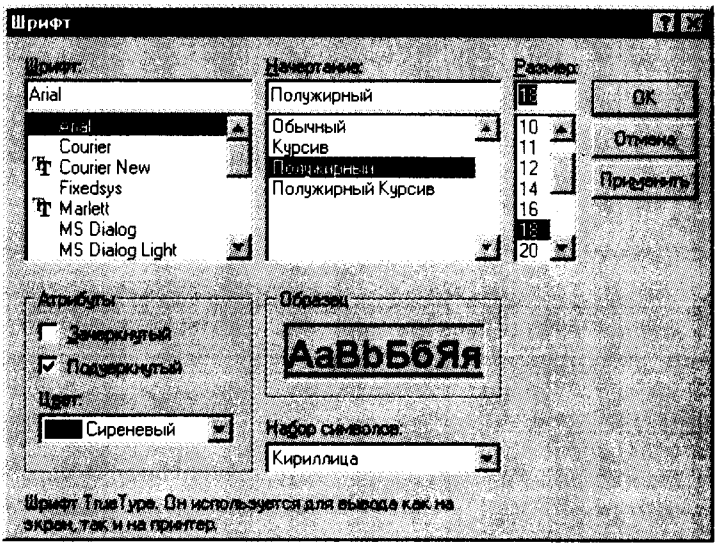


Рис. 39. Блок диалога для настройки шрифта

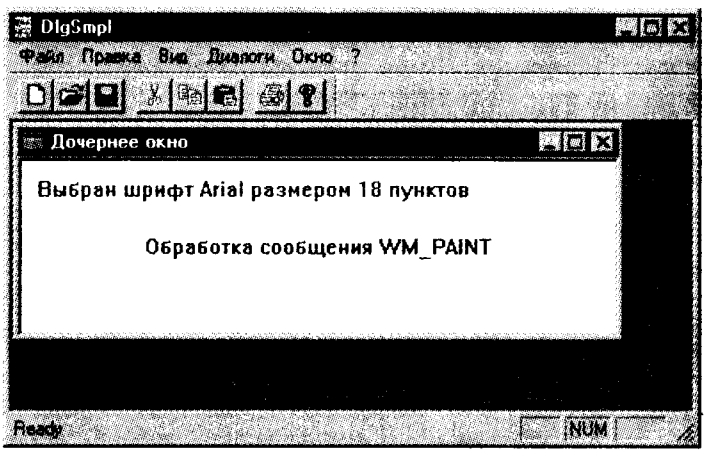


Рис. 40. Результат настройки шрифта

Класс CFindReplaceDialog

Этот класс отвечает за создание двух немодальных блоков диалога: Find (Найти) для поиска строк и Find/Replace (Найти/Заменить) для поиска и/или замены. Собственно функции поиска и замены вы должны добавить сами.

Как мы уже отмечали, для немодальных блоков диалога лучше использовать динамическое создание объекта при помощи оператора *new*, вызывая затем

функцию *Create*. Поскольку этот процесс нами уже многократно рассматривался, мы переходим непосредственно к описанию членов этого класса.

FINDREPLACE CFindReplaceDialog::**m_fr** —

структура типа *FINDREPLACE*, которая содержит характеристики объекта блока диалога:

```
typedef struct {
  DWORD lStructSize;
  HWND hwndOwner;
  HINSTANCE hInstance;
  DWORD Flags;
  LPTSTR lpstrFindWhat;
  LPTSTR lpstrReplaceWith;
  WORD wFindWhatLen;
  WORD wReplaceWithLen;
  LPARAM lCustData;
  LPFRHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
} FINDREPLACE; —
```

служит для настройки внешнего вида блока диалога. Структура имеет следующие поля:

lStructSize — размер структуры в байтах.

hwndOwner — дескриптор окна, которое владеет блоком диалога. Этот параметр не может быть равен NULL.

hInstance — определяет идентификатор блока памяти (установлен флаг FR_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг FR_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется.

lpstrFindWhat — указывает на буфер, который содержит строку, введенную пользователем в поле Find What (Что). Место под буфер (не менее 80 символов) должно быть выделено либо динамически, либо в глобальном или статическом массиве. Если буфер содержит строку при инициализации блока диалога, то она будет отображена в поле Find What (Что).

lpstrReplaceWith — указывает на буфер, который содержит строку, введенную пользователем в поле Replace With (Заменить на). Место под буфер должно быть выделено либо динамически, либо в глобальном или статическом массиве. Если буфер содержит строку при инициализации блока диалога, то она будет отображена в поле Replace With (Заменить на).

wFindWhatLen — определяет длину буфера (в байтах), на который указывает поле *lpstrFindWhat*.

wReplaceWithLen — определяет длину буфера (в байтах), на который указывает поле *lpstrReplaceWith*.

lCustData — определяет данные, которые система посылает функции фильтра *lpfnHook*. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр *lParam* этого сообщения указывает на структуру *FINDREPLACE*, определенную при создании блока диалога и может использоваться для доступа к полю *lCustData*.

lpfnHook — указатель на функцию фильтра *FRHookProc*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг FR_ENABLEHOOK.

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг FR_ENABLETEMPLATE.

Flags — это поле определяет множество битовых флагов, которые можно использовать при инициализации блока диалога. Допустимыми являются комбинации следующих флагов:

FR_DIALOGTERM	Сообщает о закрытии блока диалога.
FR_DOWN	При инициализации блока диалога переключатель направления поиска устанавливается в положение Down (Вниз), определяя, что поиск будет осуществляться от положения курсора до конца документа. В противном случае переключатель устанавливается в положение Up (Вверх) – поиск осуществляется от позиции курсора к началу документа.
FR_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook. Используется только при инициализации блока диалога.
FR_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога в модуле, определяемом hInstance.
FR_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpTemplateName.
FR_FINDNEXT	Указывает, что пользователь нажал кнопку Next (Следующий) в блоке диалога. При этом поле lpstrFindWhat содержит строку, которую нужно найти.
FR_HIDEUPDOWN	Предписывает убрать из блока диалога переключатель направления поиска.
FR_HIDEMATCHCASE	Предписывает убрать из блока диалога флажок Match Case (Учитывать регистр).
FR_HIDEWHOLEWORD	Предписывает убрать из блока диалога флажок Match Whole Word Only (Только слово целиком).

FR_MATCHCASE	При инициализации устанавливает в блоке диалога флажок Match Case (Учитывать регистр), указывая, что при поиске необходимо учитывать регистр.
FR_NOMATCHCASE	При инициализации сбрасывает в блоке диалога флажок Match Case (Учитывать регистр), указывая, что при поиске не надо учитывать регистр.
FR_NOUPDOWN	При инициализации блокирует в блоке диалога переключатель направления поиска.
FR_NOWHOLEWORD	При инициализации блокирует флажок Match Whole Word Only (Только слово целиком) в блоке диалога.
FR_REPLACE	Указывает, что пользователь нажал кнопку Replace (Заменить) в блоке диалога. При этом поле lpstrFindWhat указывает на искомую строку, а поле lpstrReplaceWith определяет строку, на которую будет произведена замена.
FR_ REPLACEALL	Указывает, что пользователь нажал кнопку Replace All (Заменить Все) в блоке диалога. При этом поле lpstrFindWhat указывает на искомую строку, а поле lpstrReplaceWith определяет строку, на которую будет произведена замена.
FR_SHOWHELP	В блоке диалога отображается кнопка Help (?)
FR_WHOLEWORD	При инициализации блока диалога устанавливает флажок Match Whole Word Only (Только слово целиком), указывая, что будет производиться поиск только целых слов. В противном случае можно искать отдельные фрагменты слов.

Ниже приведено описание основных функций класса.

CFindReplaceDialog::**CFindReplaceDialog**() —
конструктор класса, который, как и другие немодальные блоки диалога, необходимо создавать динамически.

```
BOOL CFindReplaceDialog::Create(
 BOOL bFindDialogOnly,
 LPCTSTR lpszFindWhat,
 LPCTSTR lpszReplaceWith = NULL,
 DWORD dwFlags = FR_DOWN,
 CWnd *pParentWnd = NULL) —
```

создает объект "стандартный блок диалога". Если параметр bFindDialogOnly равен TRUE, создается объект "блок диалога Find" (Найти). В противном случае — объект "блок диалога Find/Replace" (Найти/Заменить). Остальные параметры вам уже известны из описания полей структуры *FINDREPLACE*. Если параметр pParentWnd равен NULL, то функция в качестве родительского окна устанавливает главное окно приложения.

BOOL CFindReplace::**FindNext**() —
вызывается из вашей функции обратного вызова (callback), если требуется продолжать поиск.

static CFindReplaceDialog* CFindReplaceDialog::**GetNotifier**(
LPARAM lParam) –
позволяет из вашей функции обратного вызова получить указатель на текущий блок диалога Find/Replace (Найти/Заменить) для получения возможности доступа к его функциям и структуре m_fr.

CString CFindReplaceDialog::**GetFindString**() –
возвращает заданную по умолчанию строку для поиска.

CString CFindReplaceDialog::**GetReplaceString**() –
возвращает заданную по умолчанию строку, на которую будут заменяться найденные строки.

BOOL CFindReplaceDialog::**IsTerminating**() –
при вызове из вашей функции обратного вызова позволяет определить, решил ли пользователь завершить работу с блоком диалога. Если функция возвращает TRUE, то необходимо вызвать DestroyWindow и установить указатель на объект "блок диалога" в NULL. Перед завершением можно сохранить текущие установки блока диалога.

BOOL CFindReplaceDialog::**MatchCase**() –
позволяет определить, хочет ли пользователь осуществлять поиск точно, или можно не учитывать регистр.

BOOL CFindReplaceDialog::**MatchWholeWord**() –
позволяет определить, хочет ли пользователь осуществлять поиск слова целиком, или только его части.

BOOL CFindReplaceDialog::**ReplaceAll**() –
позволяет определить, хочет ли пользователь осуществлять автоматическую замену всех найденных строк.

BOOL CFindReplaceDialog::**RepiaceCurrent**() –
позволяет определить, хочет ли пользователь заменить найденную строку.

BOOL CFindReplaceDialog::**SearchDown**() –
позволяет определить направление поиска. Если возвращается TRUE, поиск осуществляется к концу документа; в противном случае — к началу.

Подробно создание и работа с блоком диалога этого класса будет рассмотрена при описании архитектуры документ/представление библиотеки классов MFC.

Класс CPrintDialog

Этот класс отвечает за создание двух модальных блоков диалога: Print (Печать) и Print Setup (Настройка печати), с помощью которых пользователь может напечатать документ, выбрать нужный принтер или изменить его параметры. В большинстве случаев достаточно тех функциональных возможностей, которые заложены в этом классе. Однако, если вас не устраивает внешний вид этих стандартных блоков диалога, вы можете образовать из него свой класс для использования собственных шаблонов и функции фильтра для обеспечения дополнительной обработки сообщений.

Настройка объекта этого класса осуществляется с помощью структуры типа *PRINTDLG*:

PRINTDLG CPrintDialog::**m_pd**;

```
typedef struct tagPD{
  DWORD lStructSize;
  HWND hwndOwner;
  HANDLE hDevMode;
  HANDLE hDevNames;
  HDC hDC;
  DWORD Flags;
  WORD nFromPage;
  WORD nToPage;
  WORD nMinPage;
  WORD nMaxPage;
  WORD nCopies;
  HINSTANCE hInstance;
  DWORD lCustData;
  LPPRINTHOOKPROC lpfnPrintHook;
  LPSETUPHOOKPROC lpfnSetupHook;
  LPCTSTR lpPrintTemplateName;
  LPCTSTR lpSetupTemplateName;
  HANDLE hPrintTemplate;
  HANDLE hSetupTemplate;
} PRINTDLG; —
```

служит для настройки внешнего вида блока диалога. Рассмотрим назначение ее полей:

lStructSize — размер структуры в байтах.

hwndOwner — дескриптор окна, которое владеет блоком диалога или NULL.

hDevMode — определяет идентификатор глобального перемещаемого блока памяти, который содержит структуру типа *DEVMODE*, используемую для инициализации параметров принтера. Если содержимое этого поля указать как NULL, то функции библиотеки сами запросят глобальную память под эту структуру и, после завершения работы с блоком диалога, запишут туда выбранные параметры принтера. Поля структуры *DEVMODE* мы опишем позже.

hDevNames — определяет идентификатор глобального перемещаемого блока памяти, который содержит структуру типа *DEVNAMES*, определяющую имя драйвера принтера, его имя и порт вывода, к которому подключен принтер. Если содержимое этого поля указать как NULL, то функции библиотеки сами запросят глобальную память под эту структуру и, после завершения работы с блоком диалога, запишут туда строки, соответствующие выбранному принтеру. Эти значения приложение использует для создания контекста устройства или информационного контекста. Поля структуры *DEVNAMES* мы опишем позже.

hDC — определяет идентификатор контекста устройства или информационного контекста, в зависимости от установленного флага PD_RETURNDC или PD_RETURNIC. В первом случае в это поле записывается контекст принтера, который можно использовать для печати, а во втором — информационный контекст, который используется для получения разнообразной информации о принтере.

nFromPage — определяет начальное значение для инициализации элемента управления From (От). Используется только в том случае, если установлен флаг PD_PAGENUMS. После завершения работы с блоком диалога это поле содержит номер страницы, с которой должна начинаться печать.

nToPage — определяет начальное значение для инициализации элемента управления To (До). Используется только в том случае, если установлен флаг PD_PAGENUMS. После завершения работы с блоком диалога это поле содержит номер страницы, до которой должна выполняться печать.

nMinPage — определяет минимальное количество страниц, которое можно задать при помощи элементов управления From (От) и To (До).

nMaxPage — определяет максимальное количество страниц, которое можно задать при помощи элементов управления From (От) и To (До).

nCopies — определяет начальное число копий, записываемое в элемент управления Copies (Число копий), если поле *hDevMode* содержит NULL. В противном случае это число задается в поле *dmCopies* структуры *DEVMODE*. При завершении работы с блоком диалога здесь содержится фактическое число копий, которые нужно напечатать.

hInstance — дескриптор модуля, содержащего шаблон блока диалога. Это поле используется только в том случае, если установлен хотя бы один из флагов PD_ENABLEPRINTTEMPLATE или PD_ENABLESETUPTEMPLATE.

lCustData — определяет данные, которые приложение может передать функции фильтра *lpfnPrintHook* или *lpfnSetupHook*. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр *lParam* этого сообщения указывает на структуру *PRINTDLG*, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю *lCustData*.

lpfnPrintHook — указатель на функцию фильтра *PrintHookProc*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг PD_ENABLEPRINTHOOK.

lpfnSetupHook — указатель на функцию фильтра *SetupHookProc*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг PD_ENABLESETUPHOOK.

lpPrintTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*. Это поле игнорируется, если не установлен флаг PD_ENABLEPRINTTEMPLATE.

lpSetupTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*. Это поле игнорируется, если не установлен флаг PD_ENABLESETUPTEMPLATE.

hPrintTemplate — определяет дескриптор блока памяти, содержащий предварительно загруженный шаблон блока диалога Print (Печать). Для использования этого поля необходимо установить флаг PD_ENABLEPRINTTEMPLATEHANDLE.

hSetupTemplate — определяет дескриптор блока памяти, содержащий предварительно загруженный шаблон блока диалога Print Setup (Настройка печати). Для использования этого поля необходимо установить флаг PD_ENABLESETUPTEMPLATEHANDLE.

Flags — определяет множество битовых флагов, используемых для инициализации блока диалога Print (Печать). При завершении работы с блоком диалога это поле отражает выбор пользователя. Допустимыми являются комбинации следующих флагов:

PD_ALLPAGES — Этот флаг используется по умолчанию и устанавливает переключатель All (Все) во включенное состояние. При этом предполагается, что надо распечатать весь текст, а не отдельные страницы или выделенный фрагмент.

PD_COLLATE — При инициализации блока диалога устанавливает переключатель Collate (Разобрать по копиям) во включенное состояние. При завершении работы установленный флаг сообщает о том, что драйвер принтера не поддерживает этот режим (значение поля dmCollate структуры *DEVMODE* не определено), и приложение должно обеспечить его само.

PD_DISABLEPRINTTOFILE — Блокирует флажок Print to File (Печать в файл).

PD_ENABLEPRINTHOOK — Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnPrintHook, для блока диалога Print (Печать).

PD_ENABLEPRINTTEMPLATE — Указывает на то, что поле lpPrintTemplateName содержит указатель на имя ресурса шаблона блока диалога Print (Печать) в модуле, определяемом hInstance.

PD_ENABLEPRINTTEMPLATEHANDLE — Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога Print (Печать). В этом случае игнорируется содержимое поля lpPrintTemplateName.

PD_ENABLESETUPHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnSetupHook, для блока диалога Print Setup (Настройка печати).
PD_ENABLESETUPTEMPLATE	Указывает на то, что поле lpSetupTemplateName содержит указатель на имя ресурса шаблона блока диалога Print Setup (Настройка печати) в модуле определяемом hInstance.
PD_ENABLESETUPTEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога Print Setup (Настройка печати). В этом случае игнорируется содержимое поля lpSetupTemplateName.
PD_HIDEPRINTTOFILE	Предписывает убрать из блока диалога флажок Print to File (Печать в файл).
PD_NOPAGENUMS	Блокирует переключатель Pages (Страницы) и связанный с ним элемент управления.
PD_NOSELECTION	Блокирует переключатель Selection (Выделенный фрагмент).
PD_NOWARNING	Отменяет вывод сообщения о том, что в системе не установлен принтер по умолчанию.
PD_PAGENUMS	Устанавливает переключатель Страницы (Pages) во включенное состояние. При этом предполагается, что надо распечатать отдельные страницы текста, а не выделенный фрагмент или весь текст.
PD_PRINTSETUP	Определяет, что на экран будет выведен блок диалога Print Setup (Настройка печати), а не блок диалога Print (Печать).
PD_PRINTTOFILE	При инициализации блока диалога устанавливает флажок Print to File (Печать в файл) во включенное состояние. При завершении работы с блоком диалога установленный флаг сообщает о том, что поле wOutputOffset структуры *DEVNAMES* содержит строку "FILE:". Это предписывает подсистеме печати запросить у пользователя имя выходного файла.
PD_RETURNDC	Определяет, что в поле hDC будет записан идентификатор контекста принтера.
PD_RETURNIC	В поле hDC будет записан идентификатор информационного контекста, который можно использовать для получения информации о принтере.

PD_RETURNDEFAULT	В поля hDevNames и hDevMode записываются идентификаторы структур *DEVNAMES* и *DEVMODE*, которые заполнены параметрами принтера, установленного по умолчанию. Исходно эти поля должны содержать NULL, в противном случае будет возвращен признак ошибки.
PD_SELECTION	Устанавливает переключатель Selection (Выделенный фрагмент) во включенное состояние. При этом предполагается, что надо распечатать выделенный фрагмент, а не весь текст или отдельные страницы.
PD_SHOWHELP	В блоке диалога отображается кнопка Help (?).
PD_USEDEVMODECOPIESANDCOLLATE	Блокирует поле Copies (Число копий), если драйвер принтера не поддерживает печать нескольких копий, и сбрасывает флажок Collate (Разобрать по копиям), если драйвер не поддерживает этот режим.

Чтобы иметь полное представление об установках, необходимыми для настройки принтера, приведем краткое описание полей структур *DEVMODE* и *DEVNAMES*.

Структура *DEVMODE* содержит разнообразную информацию о конфигурации и параметрах принтера:

```
typedef struct devicemode{
  BCHAR dmDeviceName[CCHDEVICENAME];
  WORD dmSpecVersion;
  WORD dmDriverVersion;
  WORD dmSize;
  WORD dmDriverExtra;
  DWORD dmFields;
  short dmOrientation;
  short dmPaperSize;
  short dmPaperLength;
  short dmPaperWidth;
  short dmScale;
  short dmCopies;
  short dmDefaultSource;
  short dmPrintQuality;
  short dmColor;
  short dmDuplex;
  short dmYResolution;
  short dmTTOption;
  short dmCollate;
  BCHAR dmFormName[CCHFORMNAME];
  WORD dmLogPixels;
```

```
  DWORD dmBitsPerPel;
  DWORD dmPelsWidth;
  DWORD dmPelsHeight;
  DWORD dmDisplayFlags;
  DWORD dmDisplayFrequency;
 #if (WINVER >= 0x0400)
  DWORD dmICMMetod; // только для Windows 95
  DWORD dmICMIntent; // только для Windows 95
  DWORD dmMediaType; // только для Windows 95
  DWORD dmDitherType; // только для Windows 95
  DWORD dmReserved1; // только для Windows 95
  DWORD dmReserved2; // только для Windows 95
 #endif /* WINVER >=0x0400 */
} DEVMODE; —
```

ниже перечислены поля этой структуры:

dmDeviceName — имя драйвера принтера.

dmSpecVersion — номер версии структуры *DEVMODE*.

dmDriverVersion — номер версии драйвера принтера.

dmSize — размер структуры в байтах.

dmDriverExtra — размер в байтах дополнительной структуры данных, которая может находиться в памяти сразу за структурой *DEVMODE*. Если дополнительные данные не используются, сюда должен быть записан нуль.

dmFields — набор флагов, каждый из которых отвечает за определенное поле структуры *DEVMODE*. Если флаг установлен, инициализируется соответствующее поле: DM_ORIENTATION, DM_PAPERSIZE, DM_PAPERLENGTH, DM_PAPERWIDTH, DM_COPIES, DM_SCALE, DM_DEFAULTSOURCE, DM_PRINTQUALITY, DM_COLOR, DM_DUPLEX, DM_YRESOLUTION, DM_TTOPTION, DM_COLLATE, DM_FORMNAME, DM_LOGPIXELS, DM_BITSPERPEL, DM_PELSWIDTH, DM_PELSHEIGHT, DM_DISPLAYFLAGS, DM_DISPLAYFREQUENCY, DM_ICMMETHOD, DM_ICMINTENT, DM_MEDIATYPE, DM_DITHERTYPE. Четыре последние флага используются только в Windows 95.

dmOrientation — определяет ориентацию бумаги. Может принимать одно из следующих значений: DMORIENT_PORTRAIT или DMORIENT_LANDSCAPE.

dmPaperSize — определяет размер бумаги. Например, для бумаги формата A4 применяется константа DMPAPER_A4. В это поле можно записать нуль, если определены значения обоих полей: *dmPaperLength* и *dmPaperWidth*.

dmPaperLength — переопределяет длину бумаги, заданную в поле *dmPaperSize*. Длина задается в десятых долях миллиметра.

dmPaperWidth — переопределяет ширину бумаги, заданную в поле *dmPaperSize*. Ширина задается в десятых долях миллиметра.

dmScale — определяет коэффициент (в процентах) масштабирования при печати.

dmCopies — определяет число печатаемых копий.

dmDefaultSource — зарезервировано. Должно быть установлено в нуль.

dmPrintQuality — определяет код разрешения принтера: DMRES_HIGH, DMRES_MEDIUM, DMRES_LOW, DMRES_DRAFT или положительное число, равное количеству точек на дюйм.

dmColor — определяет режим печати для цветного принтера: DMCOLOR_COLOR — цветная печать, DMCOLOR_MONOCHROME — черно-белая печать.

dmDuplex — определяет возможность печати с двух сторон листа: DMDUP_SIMPLEX — односторонняя печать, DMDUP_HORIZONTAL — двусторонняя печать по горизонтали, DMDUP_VERTICAL — двусторонняя печать по вертикали.

dmYResolution — определяет вертикальное разрешение принтера в точках на дюйм. Если установлено значение в этом поле, то в поле *dmPrintQuality* указывается горизонтальное разрешение в точках на дюйм.

dmTTOption — определяет способ печати шрифтов TrueType: DMTT_BITMAP — печать в графическом режиме (используется по умолчанию для матричных принтеров), DMTT_DOWNLOAD — загрузка шрифтов в память принтера (используется по умолчанию для лазерных принтеров), DMTT_SUBDEV — замена шрифтов на принтерные (используется по умолчанию для PostScript-принтеров).

dmCollate — определяет, следует ли упорядочивать копии при печати. Возможные значения: DMCOLLATE_TRUE — сначала печатаются все копии одной страницы; DMCOLLATE_FALSE — сначала печатается весь документ, а потом его копии. Первый режим обеспечивает более быстрый способ печати, т. к. в этом случае данные в принтер посылаются только один раз.

dmFormName — для Windows NT определяет имя используемой формы, например, "Letter" или "Legal". Полный набор допустимых имен можно получить при помощи функции *EnumForm*. В Windows 95 это поле не используется.

dmLogPixels — определяет число пикселей на логический дюйм. Драйверы принтера не используют это значение.

dmBitsPerPel — определяет цветную разрешающую способность, в битах на пиксель. Драйверы принтера не используют это значение.

dmPelsWidth — определяет ширину (в пикселях) видимой поверхности устройства. Драйверы принтера не используют это значение.

dmPelsHeight — определяет высоту (в пикселях) видимой поверхности устройства. Драйверы принтера не используют это значение.

dmDisplayFlags — определяет режим работы дисплея. Драйверы принтера не используют это значение.

dmDisplayFrequency — определяет частоту работы дисплея (в герцах). Драйверы принтера не используют это значение.

Последняя группа полей используется только в Windows 95.

dmICMMethod — определяет режим обработки ICM (Image Color Matching). Может принимать одно из следующих предопределенных значений: DMICMMETHOD_NONE — режим ICM заблокирован, DMICMMETHOD_SYSTEM — режим ICM поддерживается системой, DMICMMETHOD_DRIVER — режим ICM поддерживается драйвером устройства, DMICMMETHOD_DEVICE — режим ICM поддерживается адресуемым устройством. Большинство драйверов принтера поддерживают только значения DMICMMETHOD_SYSTEM или DMICMMETHOD_NONE.

dmICMIntent — определяет, какой из трех возможных методов отображения цветов используется по умолчанию. Может принимать одно из следующих предопределенных значений: DMICM_SATURATE — соответствие цвета оптимизируется для насыщенности (деловая графика), DMICM_CONTRAST — соответствие цвета оптимизируется для контраста (сканеры, фотографии), DMICM_COLORMETRIC — соответствие цвета оптимизируется точно по запрошенному цвету.

dmMediaType — определяет тип носителя, на который будет осуществляться печать: DMMEDIA_STANDARD — простая бумага, DMMEDIA_GLOSSY — глянцевая бумага, DMMEDIA_TRANSPARENCY — прозрачная пленка.

dmDitherType — определяет как выполняется случайная добавка: DMDITHER_NONE — добавка отсутствует, DMDITHER_LINEART — специальный метод добавки, при котором проводится четкая граница между черным, белым и серым масштабированием, DMDITHER_COARSE — грубая добавка, DMDITHER_FINE — тонкая добавка, DMDITHER_GRAYSCALE — добавка градаций серого.

Структура *DEVNAMES* содержит имена драйвера, принтера и порта вывода, к которому подключен принтер:

```
typedef struct tagDEVNAMES{
  WORD wDriverOffset;
  WORD wDeviceOffset;
  WORD wOutputOffset;
  WORD wDefault;
} DEVNAMES; —
```

первые три поля этой структуры содержат смещения текстовых строк с именами соответственно драйвера, принтера и порта вывода. Строки располагаются в памяти непосредственно за этой структурой. Поле *wDefault* определяет, описывают ли предыдущие поля принтер, установленный по умолчанию. В этом случае оно должно содержать флаг DN_DEFAULTPRN.

Ниже приведено описание основных функций класса *CPrintDialog*.

CPrintDialog::**CPrintDialog**(
BOOL bPrintSetupOnly,
DWORD dwFlags = PD_ALLPAGES | PD_USEDEVMODECOPIES |
PD_NOPAGENUM | PD_HIDEPRINTTOFILE |
PD_NOSELECTION,
CWnd *pParentWnd = NULL) –

конструктор объекта для стандартных модальных блоков диалога Windows: Print (Печать) – bPrintSetupOnly = FALSE; Print Setup (Настройка печати) – bPrintSetupOnly = TRUE. В первом случае в блоке диалога отображается кнопка Print Setup (Настройка печати), и к заданным флагам автоматически добавляется флаг PD_RETURNDC.

virtual int CPrintDialog::**DoModal**() –

вызов этой функции выводит на экран модальный стандартный блок диалога печати. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_pd, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки. После вызова этой функции можно вызывать другие функции класса для получения информации о выборе пользователя.

HDC CPrintDialog::**CreatePrinterDC**() –

создает и делает текущим контекст устройства принтера на базе структур DEVMODE и DEVNAMES. Использование этой функции не связано с открытием блока диалога.

HDC CPrintDialog::**GetPrinterDC**() –

если при создании объекта класса конструктор вызывался с параметром CPrintDialog::bPrintSetupOnly = FALSE, функция возвращает дескриптор контекста устройства принтера. После использования вы должны удалить его, вызвав функцию DeleteDC.

BOOL CPrintDialog::**GetDefaults**() –

позволяет получить параметры установленного по умолчанию принтера без открытия блока диалога. Эти параметры записываются в структуру m_pd.

CString CPrintDialog::**GetDeviceName**() –

вызывается после DoModal и возвращает имя текущего выбранного принтера.

CString CPrintDialog::**GetDriverName**() –

вызывается после DoModal и возвращает имя драйвера текущего выбранного принтера.

CString CPrintDialog::**GetPortName**() –

вызывается после DoModal и возвращает имя порта вывода текущего выбранного принтера.

LPDEVMODE CPrintDialog::**GetDevMode**() –

вызывается после DoModal и возвращает указатель на структуру типа DEVMODE, которая содержит информацию о конфигурации и параметрах принтера.

int CPrintDialog::**GetCopies**() –
вызывается после DoModal и возвращает затребованное число копий, которые нужно напечатать.

int CPrintDialog::**GetFromPage**() –
вызывается после DoModal и возвращает номер первой печатаемой страницы.

int CPrintDialog::**GetToPage**() –
вызывается после DoModal и возвращает номер последней печатаемой страницы.

BOOL CPrintDialog::**PrintAll**() –
вызывается после DoModal и возвращает информацию о том, будут ли печататься все страницы документа (TRUE).

BOOL CPrintDialog::**PrintCollate**() –
вызывается после DoModal и возвращает информацию о том, выбрал ли пользователь опцию для печати документа по копиям (TRUE).

BOOL CPrintDialog::**PrintRange**() –
вызывается после DoModal и возвращает информацию о том, выбрал ли пользователь опцию для печати определенных страниц документа (TRUE).

BOOL CPrintDialog::**PrintSelection**() –
вызывается после DoModal и возвращает информацию о том, выбрал ли пользователь опцию для печати только выделенного фрагмента документа (TRUE).

Подробно создание и работа с блоком диалога этого класса будет рассмотрена при описании вывода документов на печать.

Класс CPageSetupDialog

Этот класс отвечает за создание и функционирование стандартного модального блока диалога Windows Page Setup (Параметры страницы) с дополнительной поддержкой установки и модификации полей печати.

Принципы работы с классом такие же, как и для рассмотренных выше стандартных блоков диалога. Но есть одно отличие. После того как пользователь закончил работу с блоком диалога, все изменения, которые он сделал, не будут сохранены. Программист должен позаботиться о сохранении настроек пользователя либо в документе, либо в классе приложения.

Настройка объекта этого класса осуществляется с помощью структуры типа *PAGESETUPDLG*:

PAGESETUPDLG CPageSetupDialog::**m_psd**;

```
typedef struct tagPSD{
 DWORD lStructSize;
 HWND hwndOwner;
 HGLOBAL hDevMode;
 HGLOBAL hDevNames;
 DWORD Flags;
```

```
 POINT ptPaperSize;
 RECT rtMinMargin;
 RECT rtMargin;
 HINSTANCE hInstance;
 LPARAM lCustData;
 LPPAGESETUPHOOK lpfnPageSetupHook;
 LPPAGEPAINTHOOK lpfnPagePaintHook;
 LPCTSTR lpPageSetupTemplateName;
 HGLOBAL hPageSetupTemplate;
} PAGESETUPDLG, *LP PAGESETUPDLG; —
```

служит для настройки внешнего вида блока диалога. Рассмотрим назначение ее полей:

lStructSize — размер структуры в байтах.

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL.

hDevMode — определяет идентификатор глобального блока памяти, который содержит структуру типа *DEVMODE*,[1] используемую для инициализации элементов управления блока диалога. После завершения работы с блоком диалога, в нее будут записаны значения, выбранные пользователем. В случае, когда выбор пользователя недоступен, в это поле записывается NULL.

hDevNames — определяет идентификатор глобального блока памяти, который содержит структуру типа *DEVNAMES*, определяющую имя драйвера принтера, имя принтера и порт вывода, к которому подключен принтер. Эта структура используется для инициализации элементов управления блока диалога. После завершения работы с блоком диалога в этом поле будет указан дескриптор глобального блока памяти для структуры *DEVNAMES*, куда записаны строки, определяющие выбор пользователя. В случае, когда выбор пользователя недоступен, в это поле записывается NULL.

ptPaperSize — определяет размеры страницы, установленные пользователем. Флаги PSD_INTHOUSANDTHSOFINCHES и PSD_INHUNDREDTHSOFMILLIMETERS задают единицы измерения.

rtMinMargin — определяет минимально допустимую ширину левого, верхнего, правого и нижнего полей. Значения в этом поле игнорируются, если не установлен флаг PSD_MINMARGINS. Они не должны превышать значения, установленные в поле *rtMargin*. Единицы измерения задаются флагами PSD_INHUNDREDTHSOFMILLIMETERS и PSD_INTHOUSANDTHSOFINCHES.

rtMargin — определяет ширину левого, верхнего, правого и нижнего полей. Если установлен флаг PSD_MARGINS, то эти значения задают начальные параметры при инициализации блока диалога. При завершении работы

---

[1] Описание полей структуры см. в описании класса CPrintDialog.

с ним в это поле записываются настройки пользователя. Единицы измерения задаются флагами PSD_INHUNDREDTHSOFMILLIMETERS и PSD_INTHOUSANDTHSOFINCHES.

hInstance — дескриптор модуля, содержащего шаблон блока диалога, определенный в поле *lpPageSetupTemplateName*. Это поле используется только в том случае, если установлен флаг PSD_ENABLEPAGESETUPTEMPLATE.

lCustData — определяет данные, которые приложение может передать функции фильтра *lpfnPageSetupHook*. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр *lParam* этого сообщения указывает на структуру *PAGESETUPDLG*, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю *lCustData*.

lpfnPageSetupHook — указатель на функцию фильтра *PageSetupHook*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг PSD_ENABLEPAGESETUPHOOK.

lpfnPagePaintHook — указатель на функцию фильтра *PagePaintHook*, которая получает сообщения WM_PSD_* из блока диалога, когда нужно перерисовать образец страницы. В процессе обработки функция фильтра может настраивать вид образца. Это поле игнорируется, если не установлен флаг PSD_ENABLEPAGEPAINTHOOK.

lpPageSetupTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле *hInstance*. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг PSD_ENABLEPAGESETUPTEMPLATE.

hPageSetupTemplate — определяет дескриптор блока памяти, содержащий шаблон блока диалога Page Setup (Параметры страницы). Для использования этого поля необходимо установить флаг PSD_ENABLEPAGESETUPTEMPLATEHANDLE.

Flags — определяет множество битовых флагов, используемых для инициализации блока диалога Page Setup (Параметры страницы). При завершении работы с блоком диалога это поле отражает выбор пользователя. Допустимыми являются комбинации следующих флагов:

PSD_DEFAULTMINMARGINS	Устанавливает минимальные значения, используемые по умолчанию, которые пользователь может определять для полей страницы при печати. Этот флаг игнорируется, если определены флаги PSD_MINMARGINS или PSD_MARGINS.
PSD_DISABLEMARGINS	Блокирует элементы управления в блоке диалога, отвечающие за установку полей.
PSD_DISABLEORIENTATION	Блокирует элементы управления в блоке диалога, отвечающие за ориентацию страницы.

PSD_DISABLEPAGEPAINTING	Блокирует вывод образца страницы в блоке диалога.
PSD_DISABLEPAPER	Блокирует элементы управления в блоке диалога, отвечающие за параметры страницы, такие как ее размер и источник.
PSD_DISABLEPRINTER	Блокирует кнопку Printer (Принтер) в блоке диалога.
PSD_ENABLEPAGEPAINTHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnPagePaintHook.
PSD_ENABLEPAGESETUPHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnPageSetupHook.
PSD_ENABLEPAGESETUPTEMPLATE	Указывает на то, что поле lpPageSetupTemplateName содержит указатель на имя ресурса шаблона блока диалога Page Setup (Параметры страницы) в модуле, определяемом hInstance.
PSD_ENABLEPAGESETUPTEMPLATEHANDLE	Указывает на то, что поле hPageSetupTemplate используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpPageSetupTemplateName.
PSD_INHUNDREDTHSOFMILLIMETERS	Указывает, что единицей измерения размеров бумаги и полей являются сотые доли миллиметра (значения в полях rtMargin, rtMinMargin и ptPaperSize).
PSD_INTHOUSANDTHSOFINCHES	Указывает, что единицей измерения размера бумаги и полей являются тысячные доли дюйма (значения в полях rtMargin, rtMinMargin и ptPaperSize).
PSD_MARGINS	Предписывает использовать значения, определенные в поле rtMargin, в качестве начальных для левого, верхнего, правого и нижнего полей. Если этот флаг не установлен, по умолчанию эти размеры инициализируются один дюйм
PSD_MINMARGINS	Предписывает использовать значения, определенные в поле rtMinMargin, в качестве минимально допустимых для левого, верхнего, правого и нижнего полей. Если этот флаг не установлен, минимальные размеры определяются параметрами принтера.

PSD_NOWARNING	Отменяет вывод сообщения о том, что в системе не установлен принтер по умолчанию.
PSD_RETURNDEFAULT	Определяет, что структуры, на которые указывают поля hDevNames и hDevMode инициализируются параметрами принтера, установленного по умолчанию. Блок диалога на экран не выводится.
PSD_SHOWHELP	В блоке диалога отображается кнопка Help (?).

Ниже приведено описание основных функций этого класса.

CPageSetupDialog::**CPageSetupDialog**(
DWORD dwFlags = PSD_MARGINS | PSD_INWININIINTLMEASURE,
CWnd *pParentWnd = NULL) —
конструктор объекта "блок диалога". Используемый по умолчанию флаг PSD_INWININIINTLMEASURE в настоящее время не реализован, и мы рекомендуем явно задавать единицы измерения, указав один из допустимых флагов: PSD_INHUNDREDTHSOFMILLIMETERS или PSD_INTHOUSANDTHSOFINCHES.

virtual int CPageSetupDialog::**DoModal**() —
вызов этой функции выводит на экран модальный стандартный блок диалога для настройки полей, размеров и ориентации страницы, а также параметров принтера. Если вы хотите самостоятельно задать параметры инициализации блока диалога, устанавливая соответствующие значения в поля структуры m_psd, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки. Если функция возвратила IDOK, то можно вызывать другие функции класса для получения информации о выборе пользователя.

Для того чтобы зафиксировать текущие установки пользователя, необходимо вызвать функцию

void CWinApp::**SelectPrinter**(
HANDLE hDevNames,
HANDLE hDevMode,
BOOL bFreeOld = TRUE)
которая заменяет (bFreeOld = TRUE) текущий принтер на новый, указанный в параметрах hDevNames и hDevMode. Если оба эти параметра равны NULL, то используется принтер по умолчанию. После вызова этой функции приложение получает новый контекст устройства для принтера с соответствующими атрибутами.

HDC CPageSetupDialog::**CreatePrinterDC**() —
создает и делает текущим контекст устройства принтера на базе структур *DEVMODE* и *DEVNAMES*.

CString CPageSetupDialog::**GetDeviceName**() —
вызывается после DoModal и возвращает имя текущего выбранного принтера.

LPDEVMODE CPageSetupDialog::**GetDevMode**() —
вызывается после DoModal и возвращает указатель на структуру типа *DEVMODE*, которая содержит информацию о конфигурации и параметрах принтера.

CString CPageSetupDialog::**GetDriverName**() —
вызывается после DoModal и возвращает имя драйвера текущего выбранного принтера.

CString CPageSetupDialog::**GetPortName**() —
вызывается после DoModal и возвращает имя порта вывода текущего выбранного принтера.

void CPageSetupDialog::**GetMargins**(
LPRECT lpRectMargins,
LPRECT lpRectMinMargins) —
вызывается после DoModal и возвращает значения размеров полей печати текущего выбранного принтера. Если какой-либо параметр вам не нужен, вместо указателя следует передавать NULL. Размеры возвращаются в сотых долях миллиметра или тысячных долях дюйма, в зависимости от установленных единиц измерения.

CSize CPageSetupDialog::**GetPaperSize**() —
вызывается после DoModal и возвращает размер бумаги, выбранной для текущего принтера. Размеры возвращаются в сотых долях миллиметра или тысячных долях дюйма, в зависимости от установленных единиц измерения.

virtual UINT CPageSetupDialog::**OnDrawPage**(
CDC *pDC,
UINT nMessage,
LPRECT lpRect) —
вызывается фреймом и рисует экранный образ страницы печати, который затем отображается в блоке диалога Page Setup (Параметры страницы); по умолчанию выводит изображение страницы текста. В качестве параметров в функцию передаются: pDC — указатель на контекст устройства принтера, lpRect — указатель на объект CRect или RECT, содержащий координаты перерисовываемой области, nMessage — сообщение, определяющее область выводимой страницы. Параметр nMessage может принимать одно из следующих значений:

WM_PSD_FULLPAGERECT	страница целиком
WM_PSD_MINMARGINRECT	область, ограниченная текущими минимальными полями
WM_PSD_MARGINRECT	область, ограниченная текущими полями
WM_PSD_GREEKTEXTRECT	содержание страницы
WM_PSD_ENVSTAMPRECT	область, зарезервированная для штампа
WM_PSD_YAFULLPAGERECT	область, простирающаяся до границ образца страницы

Эту функцию следует переопределить, если нужно организовать рисование своего изображения страницы. Не забывайте вызывать функцию базового класса для необрабатываемых вами сообщений.

virtual UINT CPageSetupDialog::**PreDrawPage**(
WORD wPaper,
WORD wFlags,
LPPAGESETUPDLG pPSD) –

вызывается фреймом перед рисованием экранного образа страницы печати. Переопределите эту функцию, если вам нужно организовать рисование собственного изображения страницы. Если переопределенная функция возвращает TRUE, то вы должны сами организовать вывод изображения, а если FALSE, то будет выведено изображение по умолчанию. В качестве параметров в функцию передаются: wPaper – определяет размер бумаги, может принимать одно из значений DM_PAPER, перечисленных в описании структуры DEVMODE, pPSD – указатель на структуру PAGESETUPDLG; wFlags – определяет ориентацию бумаги и тип принтера (матричный или Hewlett Packard – HPPCL) и может принимать одно из следующих значений:

0x001	матричный принтер, горизонтальная ориентация бумаги
0x003	принтер HPPCL, горизонтальная ориентация бумаги
0x005	матричный принтер, вертикальная ориентация бумаги
0x007	принтер HPPCL, вертикальная ориентация бумаги
0x00d	матричный принтер, горизонтальная ориентация конверта
0x00b	принтер HPPCL, горизонтальная ориентация конверта
0x01f	матричный принтер, вертикальная ориентация конверта
0x019	принтер HPPCL, вертикальная ориентация конверта

Подробно создание и работа с блоком диалога этого класса будет рассмотрена при описании архитектуры документ/представление библиотеки классов MFC.

Заключение

Самое время подвести предварительные итоги. Рассматриваемая версия библиотеки классов MFC (4.1) состоит из 197 (!) классов и структур, которые используют еще несколько десятков структур Win32 API. Мы же рассмотрели (с той или иной степенью полноты) 28 из них. Много это или мало? Ответ — для начала достаточно. Нами рассмотрена основа, самые главные классы, без которых не обходится ни одно Windows-приложение, написанное с использованием библиотеки классов MFC. Кроме того, мы подробно изучили целый ряд механизмов как самой системы Windows, так и используемых в классах и классами библиотеки MFC.

Теперь вы уже можете без мучительного поиска по исходным текстам кода библиотеки и примеров понять, что нужно сделать, чтобы реализовать ту или иную возможность, и, надеемся, полностью готовы подробнее ознакомиться с оставшимися классами и возможностями.

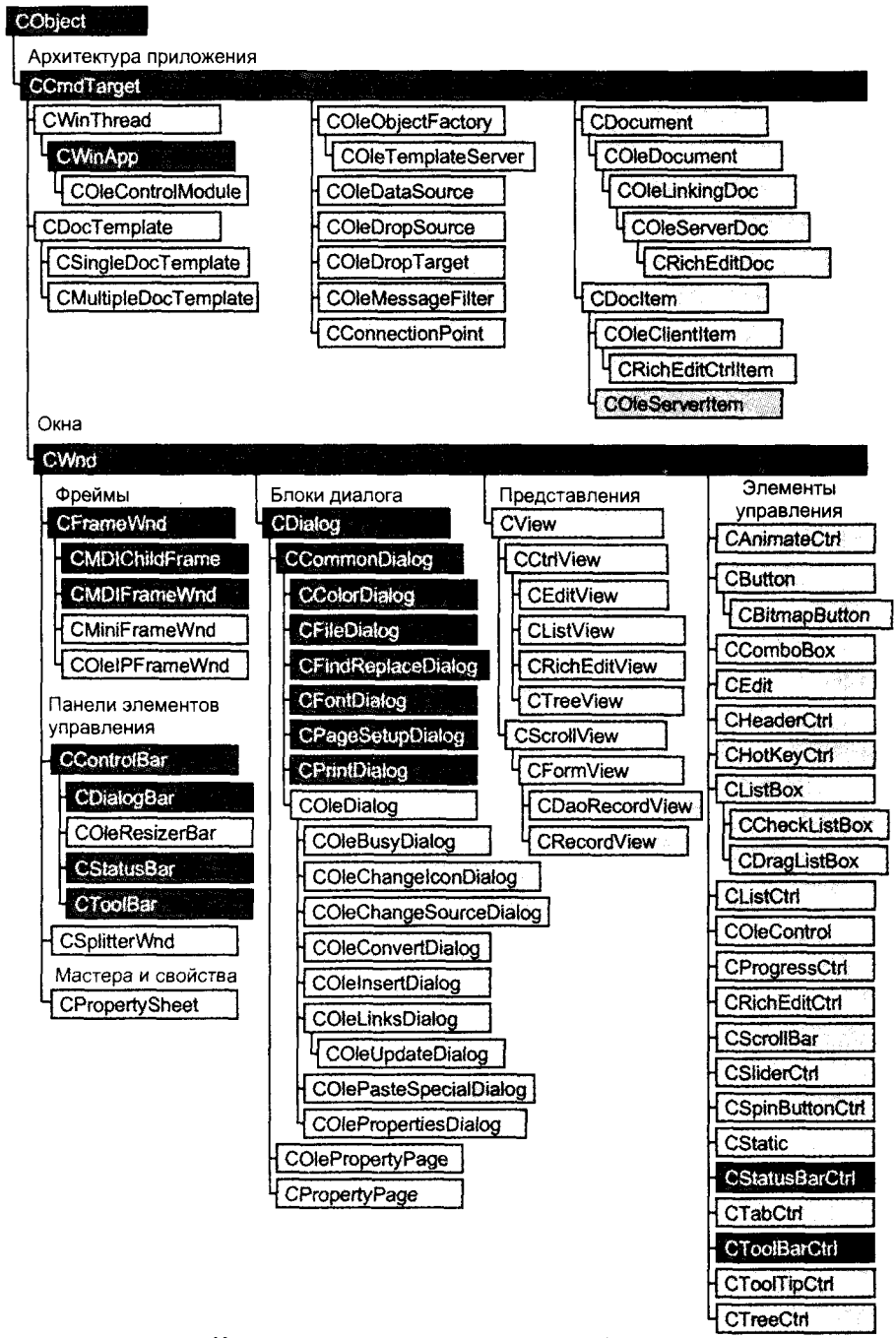
Еще раз кратко напомним темы, которые обсуждаются в следующих двух томах книги:

- графический вывод;
- элементы управления;
- архитектура документ/представление;
- ресурсы;
- работа с объектами OLE;
- синхронизация потоков и процессов;
- структурная обработка исключений;
- введение в OpenGL;
- создание справочной системы и дистрибутива приложения.

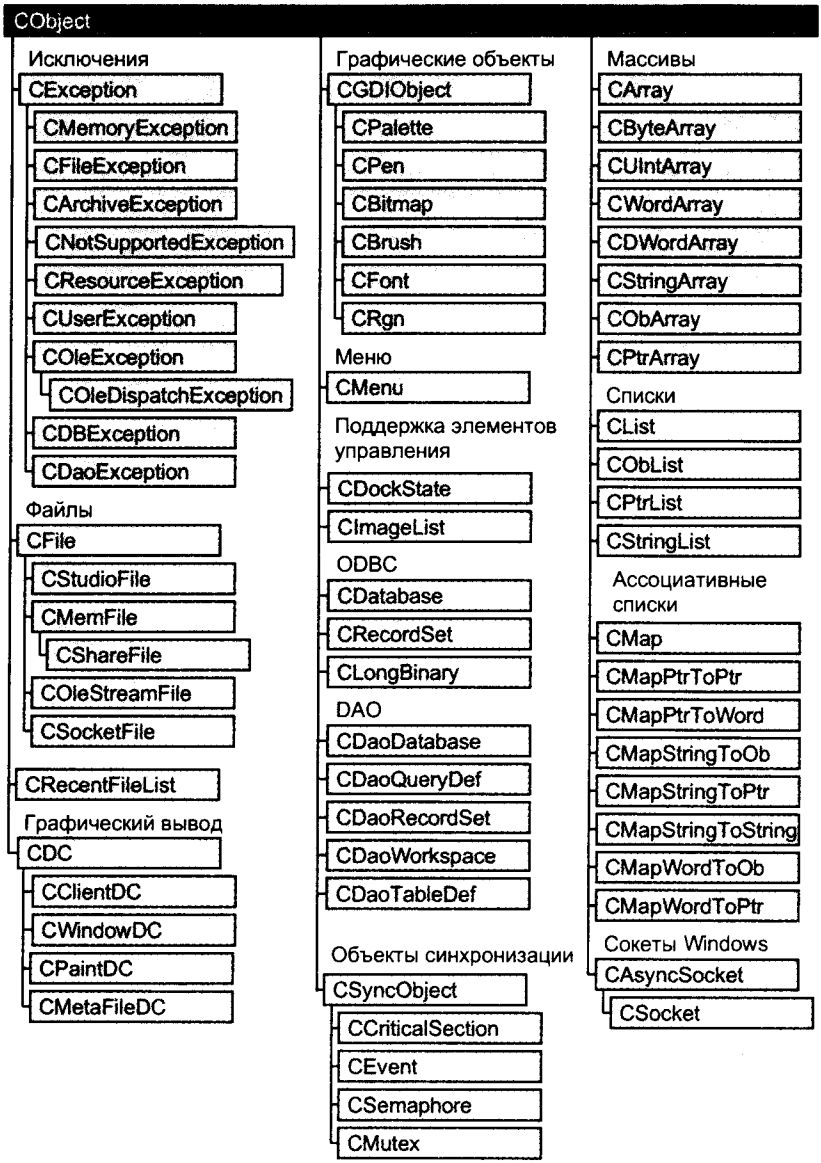
Приложения

1. Иерархия классов MFC 4.1
2. Список функций класса CWnd
3. Типы проектов и мастера
4. Настройки проекта
5. Значения параметра функции GetSystemMetrics

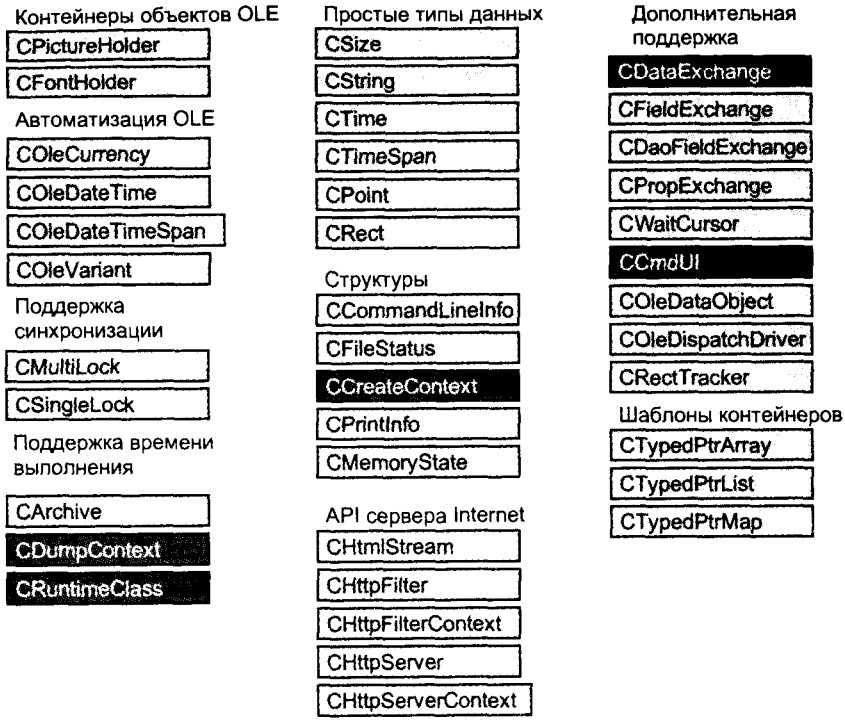
Приложение 1. Иерархия классов MFC 4.1



Классы приложения и оконных объектов



Классы неоконных объектов



Вспомогательные классы

Примечание. Черным цветом выделены классы, описанные в этом томе книги.

Приложение 2
Список функций класса CWnd

Атрибуты

HWND **GetSafeHwnd**() const	возвращает m_hWnd или NULL, если указатель this равен NULL
CWnd* **GetOwner**() const	возвращает указатель на владельца объекта CWnd
Void **SetOwner**(CWnd* pOwnerWnd)	изменяет владельца объекта CWnd

Конструкторы и создание окон

CWnd()	конструктор
static CWnd* PASCAL **FromHandle**(HWND hWnd)	возвращает указатель на объект CWnd по дескриптору окна; если объект CWnd не связан с дескриптором, создается и присоединяется временный объект CWnd
static CWnd* PASCAL **FromHandlePermanent**(HWND hWnd)	возвращает указатель на объект CWnd по дескриптору окна; если объект CWnd не присоединен к дескриптору, возвращается NULL
static void PASCAL **DeleteTempMap**()	вызывается автоматически дескриптором времени ожидания объекта CWinApp и удаляет любые временные объекты CWnd, созданные функцией FromHandle
BOOL **Attach**(HWND hWndNew)	присоединяет дескриптор Windows к объекту CWnd
HWND **Detach**()	отсоединяет дескриптор Windows от объекта CWnd и возвращает дескриптор
virtual void **PreSubclassWindow**()	позволяет выполнить предварительные настройки перед вызовом функции SubclassWindow
BOOL **SubclassWindow**(HWND hWnd)	присоединяет окно к объекту CWnd и направляет сообщения через карту сообщений этого объекта
BOOL **SubclassDlgItem**(UINT nID, CWnd* pParent)	присоединяет элемент управления Windows к объекту CWnd и направляет сообщения через карту сообщений CWnd

HWND **UnsubclassWindow**()	отсоединяет окно от объекта CWnd
virtual BOOL Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID, CCreateContext* pContext = NULL)	создает и инициализирует дочернее окно, связанное с объектом CWnd
BOOL **ExecuteDlgInit**(LPCTSTR lpszResourceName) и BOOL **ExecuteDlgInit**(LPVOID lpResource)	инициализирует ресурсы блока диалога
BOOL **CreateEx**(DWORD dwExStyle, LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD dwStyle, int x, int y, int nWidth, int nHeight, HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam = NULL)	создает перекрывающееся, всплывающее или дочернее окно и присоединяет его к объекту CWnd
BOOL **CreateControl**(REFCLSID clsid, LPCTSTR pszWindowName, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID, CFile* pPersist=NULL, BOOL Storage=FALSE, BSTR bstrLicKey=NULL) и BOOL **CreateControl**(LPCTSTR pszClass, LPCTSTR pszWindowName, DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID, CFile* pPersist=NULL, BOOL bStorage=FALSE, BSTR bstrLicKey=NULL)	создают элемент управления OLE, который будет представлен в MFC-программе объектом CWnd
LPUNKNOWN **GetControlUnknown**()	возвращает указатель на элемент управления OLE
virtual BOOL **DestroyWindow**()	удаляет все присоединенные окна Windows

virtual BOOL **PreCreateWindow**(CREATESTRUCT& cs)	вызывается перед созданием окна Windows, присоединенного к данному объекту CWnd
virtual void **CalcWindowRect**(LPRECT lpClientRect, UINT nAdjustType = adjustBorder)	вызывается для вычисления прямоугольника окна из прямоугольника пользователя

Доступ к дереву окон

int **GetDlgCtrlID**() const	если CWnd — дочернее окно, вызов этой функции возвращает значение идентификатора ID
Int **SetDlgCtrlID**(int nID)	устанавливает идентификатор ID окна или элемента управления для окна (которое может быть дочерним окном, а не только элементом управления в диалоговом окне)
CWnd* **GetDlgItem**(int nID) const и Void **GetDlgItem**(int nID, HWND* pWnd) const	возвращает указатель на элемент управления диалогового окна по его заданному идентификатору ID
CWnd* **GetDescendantWindow**(int nID, BOOL bOnlyPerm = FALSE) const	ищет все окна-потомки и возвращает окно с указанным идентификатором
void **SendMessageToDescendants**(UINT message, WPARAM wParam = 0, LPARAM lParam = 0, BOOL bDeep = TRUE, BOOL bOnlyPerm = FALSE)	посылает сообщение во все окна-потомки
CFrameWnd* **GetParentFrame**() const	возвращает указатель на родительское окно объекта CWnd
CWnd* **GetTopLevelParent**() const	возвращает указатель на родительское окно высшего уровня
CWnd* **GetTopLevelOwner**() const	возвращает указатель на окно высшего уровня
CWnd* **GetParentOwner**() const	возвращает указатель на родительское окно
CFrameWnd* **GetTopLevelFrame**() const	возвращает указатель на окно высшего уровня фрейма окна
static CWnd* PASCAL **GetSafeOwner**(CWnd* pParent = NULL, HWND* pWndTop = NULL)	возвращает указатель на владельца данного окна

Общие функции работы с сообщениями

LRESULT **SendMessage**(UINT message, WPARAM wParam = 0, LPARAM lParam = 0)	посылает сообщение объекту CWnd и ожидает его обработки
BOOL **PostMessage**(UINT message, WPARAM wParam = 0, LPARAM lParam = 0)	помещает сообщение в очередь приложения без ожидания его обработки
BOOL **SendNotifyMessage**(UINT message, WPARAM wParam, LPARAM lParam)	посылает определенное сообщение окну и завершается как можно скорее, возвращаемое значение зависит от того, создало ли окно поток вызова
BOOL **SendChildNotifyLastMsg**(LRESULT* pResult = NULL)	обеспечивает передачу сообщения из родительского окна дочернему окну так, что дочернее окно может обрабатывать задачу

Обработка сообщений модального диалога

BOOL **IsDialogMessage**(LPMSG lpMsg)	определяет, предназначено ли данное сообщение для немодального диалогового окна, и, если это так, обрабатывает его

Функции работы с текстом окна

void **SetWindowText**(LPCTSTR lpszString)	устанавливает текст или заголовок окна (если он есть) в соответствии с заданным в параметре
int **GetWindowText**(LPTSTR lpszStringBuf, int nMaxCount) const и void **GetWindowText**(CString& rString) const	возвращает текст или заголовок окна (если он есть)
int GetWindowTextLength() const	возвращает длину текста или заголовка окна
void **SetFont**(CFont* pFont, BOOL bRedraw = TRUE)	устанавливает текущий шрифт
CFont* **GetFont**() const	возвращает указатель на объект, содержащий текущий шрифт

Функции меню (кроме дочерних окон)

CMenu* **GetMenu**() const	возвращает указатель на заданное меню
BOOL **SetMenu**(CMenu* pMenu)	устанавливает элемент меню
void **DrawMenuBar**()	перерисовывает полосу меню
CMenu* **GetSystemMenu**(BOOL bRevert) const	позволяет приложению обращаться к системному меню для копирования и модификации
BOOL **HighlighteMenuItem**(CMenu* pMenu, UINT nIDHighlighteItem, UINT nHighlighte)	высвечивает или затеняет элемент меню верхнего уровня (полосы меню)

Функции изменения размеров и расположения окна

BOOL **IsIconic**() const	определяет, свернуто ли окно в пиктограмму
BOOL **IsZoomed**() const	определяет, развернуто ли окно
void **MoveWindow**(int x, int y, int nWidth, int nHeight, BOOL bRepaint = TRUE) и void **MoveWindow**(LPCRECT lpRect, BOOL bRepaint = TRUE)	изменяет позицию и размеры объекта окна CWnd
BOOL **SetWindowPos**(const CWnd* pWndInsertAfter, int x, int y, int cx, int cy, UINT nFlags)	изменяет размер, позицию и порядок дочерних, всплывающих окон и окон верхнего уровня
UINT **ArrangeIconicWindows**()	упорядочивает все свернутые дочерние окна (пиктограммы)
void **BringWindowToTop**()	переносит объект CWnd на вершину стека перекрывающихся окон
Void **GetWindowRect**(LPRECT lpRect) const	получает координаты окна объекта CWnd
Void **GetClientRect**(LPRECT lpRect) const	получает размеры клиентской области CWnd

BOOL **GetWindowPlacement**(WINDOWPLACEMENT* lpwndpl) const	возвращает видимость и состояние (нормальное, свернутое или развернутое) окна
BOOL **SetWindowPlacement**(const WINDOWPLACEMENT* lpwndpl)	устанавливает видимость и состояние окна (нормальное, свернутое или развернутое)

Функции преобразования координат

void **ClientToScreen**(LPPOINT lpPoint) const и void **ClientToScreen**(LPRECT lpRect) const	преобразует пользовательские координаты данной точки или прямоугольника на экране дисплея в координаты экрана
Void **ScreenToClient**(LPPOINT lpPoint) const и void **ScreenToClient**(LPRECT lpRect) const	преобразует экранные координаты данной точки или прямоугольника на экране дисплея в пользовательские координаты
Void **MapWindowPoints**(CWnd* pwndTo, LPPOINT lpPoint, UINT nCount) const и void **MapWindowPoints**(CWnd* pwndTo, LPRECT lpRect) const	преобразует набор точек из координатной области окна CWnd в координатную область другого окна

Функции обновления и рисования

CDC* **BeginPaint**(LPPAINTSTRUCT lpPaint)	готовит окно CWnd к перерисовке
Void **EndPaint**(LPPAINTSTRUCT lpPaint)	отмечает окончание перерисовки
CDC* **GetDC**()	возвращает указатель на контекст устройства для рабочей области окна
CDC* **GetWindowDC**()	возвращает указатель на контекст устройства для целого окна, включая полосу заголовка, меню и полосы прокрутки
Int **ReleaseDC**(CDC* pDC)	освобождает контекст устройства, связанный с окном, для его использования другими приложениями
Void **Print**(CDC* pDC, DWORD dwFlags) const	выводит текущее окно в определенном контексте устройства

void **PrintClient**(CDC* pDC, DWORD dwFlags) const	выводит любое окно в определенном контексте устройства (обычно принтер)
void **UpdateWindow**()	обновляет рабочую область
void **SetRedraw**(BOOL bRedraw = TRUE)	разрешает повторный вывод изменений в окно CWnd или предотвращает повторный вывод изменений
BOOL **GetUpdateRect**(LPRECT lpRect, BOOL bErase = FALSE)	возвращает указатель на координаты самого маленького прямоугольника, который полностью включает область обновления окна CWnd
int **GetUpdateRgn**(CRgn* pRgn, BOOL bErase = FALSE)	возвращает указатель на область перерисовки окна CWnd
void **Invalidate**(BOOL bErase = TRUE)	делает недействительной всю рабочую область
void **InvalidateRect**(LPCRECT lpRect, BOOL bErase = TRUE)	делает недействительной рабочую область внутри данного прямоугольника, добавляя прямоугольник к текущей области обновления
void **InvalidateRgn**(CRgn* pRgn, BOOL bErase = TRUE)	делает недействительной рабочую область внутри данного региона
void **ValidateRect**(LPCRECT lpRect)	делает действительной рабочую область внутри данного прямоугольника
void **ValidateRgn**(CRgn* pRgn)	делает действительной рабочую область внутри данного региона
BOOL **ShowWindow**(int nCmdShow)	показывает или скрывает окно
BOOL **IsWindowVisible**() const	определяет, является ли окно видимым
void **ShowOwnedPopups**(BOOL bShow = TRUE)	показывает или скрывает все всплывающие окна, принадлежащие окну
CDC* **GetDCEx**(CRgn* prgnClip, DWORD flags)	возвращает указатель на контекст устройства для рабочей области и допускает отсечение при рисовании
BOOL **LockWindowUpdate**()	блокирует вывод в данное окно
void **UnlockWindowUpdate**()	разблокирует окно, которое было блокировано с помощью функции LockWindowUpdate
BOOL **RedrawWindow**(LPCRECT lpRectUpdate = NULL, CRgn* prgnUpdate = NULL, UINT flags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE)	модифицирует определенный прямоугольник или область в рабочей области

BOOL **EnableScrollBar**(int nSBFlags, UINT nArrowFlags = ESB_ENABLE_BOTH)	включает или отключает одну или обе стрелки полосы прокрутки

Функции использования таймера

UINT **SetTimer**(UINT nIDEvent, UINT nElapse, void (CALLBACK* lpfnTimer)(HWND, UINT, UINT, DWORD))	устанавливает системный таймер, который при вызове посылает сообщение WM_TIMER
BOOL **KillTimer**(int nIDEvent)	уничтожает системный таймер

Функции для работы со всплывающими подсказками

BOOL **EnableToolTips**(BOOL bEnable = TRUE)	включает элемент управления всплывающей подсказки
static void PASCAL **CancelToolTips**(BOOL bKeys = FALSE)	отключает элемент управления всплывающей подсказки
void **FilterToolTipMessage**(MSG* pMsg)	возвращает заголовок или текст, связанный с элементом управления в диалоговом окне
Virtual int **OnToolHitTest**(CPoint point, TOOLINFO* pTI) const	определяет, находится ли точка в прямоугольнике, ограничивающем определенный инструмент, и возвращает информацию об инструменте

Функции состояния окна

BOOL **IsWindowEnabled**() const	определяет, допускается ли ввод с клавиатуры и мыши в это окно
BOOL **EnableWindow**(BOOL bEnable = TRUE)	включает или отключает ввод с клавиатуры и мыши
static CWnd* PASCAL **GetActiveWindow**()	возвращает указатель на активное окно
CWnd* **SetActiveWindow**()	активизирует окно
BOOL **SetForegroundWindow**()	устанавливает для потока, который создал окно, приоритетный режим и активизирует это окно

static CWnd* PASCAL **GetForegroundWindow**()	возвращает указатель на приоритетное окно (окно высшего уровня, с которым пользователь в настоящее время работает)
static CWnd* PASCAL **GetCapture**()	возвращает указатель на объект CWnd, для которого установлен захват мыши
CWnd* **SetCapture**()	посылает весь последующий ввод с мыши в объект CWnd
static CWnd* PASCAL **GetFocus**()	возвращает указатель на объект CWnd, который в настоящее время имеет фокус ввода
CWnd* **SetFocus**()	устанавливает фокус ввода
static CWnd* PASCAL **GetDesktopWindow**()	возвращает указатель на окно рабочего стола Windows
void **CloseWindow**()	закрывает окно

Функции для работы с блоками диалога

void **CheckDlgButton**(int nIDButton, UINT nCheck)	помещает или удаляет метку флажка
void **CheckRadioButton**(int nIDFirstButton, int nIDLastButton, int nIDCheckButton)	помечает переключатель и удаляет метку из всех других переключателей в определенной группе
int **GetCheckedRadioButton**(int nIDFirstButton, int nIDLastButton)	возвращает идентификатор ID текущего переключателя группы
int **DlgDirList**(LPTSTR lpPathSpec, int nIDListBox, int nIDStaticPath, UINT nFileType)	заполняет список перечнем каталогов или файлов
int **DlgDirListComboBox**(LPTSTR lpPathSpec, int nIDComboBox, int nIDStaticPath, UINT nFileType)	заполняет раскрывающийся список перечнем каталогов или файлов
BOOL **DlgDirSelect**(LPTSTR lpString, int nIDListBox)	возвращает указатель на текущий выбор из списка
BOOL **DlgDirSelectComboBox**(LPTSTR lpString, int nIDComboBox)	возвращает указатель на текущий выбор из раскрывающегося списка

UINT **GetDlgItemInt**(int nID, BOOL* lpTrans = NULL, BOOL bSigned = TRUE) const	преобразует текст элемента управления в данном диалоговом окне в целочисленное значение
int **GetDlgItemText**(int nID, LPTSTR lpStr, int nMaxCount) const и int **GetDlgItemText**(int nID, CString& rString) const	возвращает указатель на заголовок или текст, связанный с элементом управления
CWnd* **GetNextDlgGroupItem**(CWnd* pWndCtl, BOOL bPrevious = FALSE) const	ищет следующий (или предыдущий) элемент управления внутри группы
CWnd* **GetNextDlgTabItem**(CWnd* pWndCtl, BOOL bPrevious = FALSE) const	возвращает указатель на первый элемент управления со стилем WS_TABSTOP, который следует (или предшествует) за определенным элементом управления
UINT **IsDlgButtonChecked**(int nIDButton) const	определяет, выбрана ли кнопка
LRESULT **SendDlgItemMessage**(int nID, UINT message, WPARAM wParam = 0, LPARAM lParam = 0)	посылает сообщение определенному элементу управления
void **SetDlgItemInt**(int nID, UINT nValue, BOOL bSigned = TRUE)	устанавливает текст элемента управления, заданный целочисленным значением
void **SetDlgItemText**(int nID, LPCTSTR lpszString)	устанавливает заголовок или текст элемента управления в определенном диалоговом окне

Функции прокрутки

int **GetScrollPos**(int nBar) const	возвращает указатель на текущую позицию полосы прокрутки
void **GetScrollRange**(int nBar, LPINT lpMinPos, LPINT lpMaxPos) const	копирует текущие минимальные и максимальные позиции для данной полосы прокрутки
void **ScrollWindow**(int xAmount, int yAmount, LPCRECT lpRect = NULL, LPCRECT lpClipRect = NULL)	прокручивает содержимое рабочей области окна

int **SetScrollPos**(int nBar, int nPos, BOOL bRedraw = TRUE)	устанавливает текущую позицию полосы прокрутки и повторно выводит полосу прокрутки, чтобы отразить новую позицию (если она определена)
void **SetScrollRange**(int nBar, int nMinPos, int nMaxPos, BOOL bRedraw = TRUE)	устанавливает минимальные и максимальные значения для данной полосы прокрутки
void **ShowScrollBar**(UINT nBar, BOOL bShow = TRUE)	отображает или скрывает полосу прокрутки
void **EnableScrollBarCtrl**(int nBar, BOOL bEnable = TRUE)	включает или отключает элемент управления "полоса прокрутки"
virtual CScrollBar* **GetScrollBarCtrl**(int nBar) const	возвращает указатель на элемент управления "полоса прокрутки"
int **ScrollWindowEx**(int dx, int dy, LPCRECT lpRectScroll, LPCRECT lpRectClip, CRgn* prgnUpdate, LPRECT lpRectUpdate, UINT flags)	пролистывает содержимое рабочей области, подобно функции ScrollWindow, но с дополнительными возможностями
BOOL **SetScrollInfo**(int nBar, LPSCROLLINFO lpScrollInfo, BOOL bRedraw = TRUE)	устанавливает информацию о полосе прокрутки в структуре *SCROLLINFO*
BOOL **GetScrollInfo**(int nBar, LPSCROLLINFO lpScrollInfo, UINT nMask = SIF_ALL)	возвращает указатель на структуру *SCROLLINFO*, которая содержит информацию о полосе прокрутки
int **GetScrollLimit**(int nBar)	возвращает предельное значение полосы прокрутки

Функции доступа к окну

CWnd* **ChildWindowFromPoint**(POINT point) const и CWnd* **ChildWindowFromPoint**(POINT point, UINT nFlags) const	определяет, какому из дочерних окон принадлежит указанная точка
static CWnd* PASCAL **FindWindow**(LPCTSTR lpszClassName, LPCTSTR lpszWindowName)	возвращает дескриптор окна по его имени и классу

CWnd* **GetNextWindow**(UINT nFlag = GW_HWNDNEXT) const	возвращает указатель на следующее (или предыдущее) окно в списке окон
CWnd* **GetTopWindow**() const	возвращает верхнее дочернее окно, которое принадлежит окну CWnd
CWnd* **GetWindow**(UINT nCmd) const	возвращает указатель на окно, связанное с данным окном
CWnd* **GetLastActivePopup**() const	определяет, какое последнее всплывающее окно, принадлежащее классу CWnd, было активным
BOOL **IsChild**(const CWnd* pWnd) const	указывает, является ли окно CWnd дочерним окном или другим прямым потомком определенного окна
CWnd* **GetParent**() const	возвращает указатель на родительское окно CWnd (если оно есть)
CWnd* **SetParent**(CWnd* pWndNewParent)	меняет родительское окно
static CWnd* PASCAL **WindowFromPoint**(POINT point)	идентифицирует окно, которое содержит данный указатель

Функции обслуживания

BOOL **FlashWindow**(BOOL bInvert)	подсвечивает окно один раз
int **MessageBox**(LPCTSTR lpszText, LPCTSTR lpszCaption = NULL, UINT nType = MB_OK)	создает и отображает окно, которое содержит заголовок, сообщение и кнопку OK

Функции работы с буфером обмена (Clipboard)

BOOL **ChangeClipboardChain**(HWND hWndNext)	удаляет CWnd из списка программ, просматривающих буфер обмена
HWND **SetClipboardViewer**()	добавляет CWnd к списку окон, которые выводятся всякий раз, если изменено содержимое буфера обмена
BOOL **OpenClipboard**()	открывает буфер обмена; другие приложения не смогут изменить буфер обмена, пока не вызвана функция Windows CloseClipboard
static CWnd* PASCAL **GetClipboardOwner**()	возвращает указатель на текущего владельца буфера обмена

static CWnd* PASCAL **GetClipboardViewer**()	возвращает указатель на первое окно в списке программ просмотра буфера обмена (Clipboard viewers)
static CWnd* PASCAL **GetOpenClipboardWindow**()	возвращает указатель на окно, которое в настоящее время имеет открытый буфер обмена

Функции каретки

void **CreateCaret**(CBitmap* pBitmap)	создает новую форму системной каретки (маркера ввода с клавиатуры) и получает право на ее монопольное использование
void **CreateSolidCaret**(int nWidth, int nHeight)	создает курсор системной каретки и получает право на ее монопольное использование
void **CreateGrayCaret**(int nWidth, int nHeight)	создает серый курсор системной каретки и получает право на ее монопольное использование
static CPoint PASCAL **GetCaretPos**()	возвращает координаты текущей позиции каретки
static void PASCAL **SetCaretPos**(POINT point)	перемещает каретку в определенную позицию
void **HideCaret**()	скрывает каретку, удаляя ее с экрана дисплея
void **ShowCaret**()	показывает каретку на экране дисплея в текущей позиции; показанная однажды, каретка начинает высвечиваться автоматически

Функция взаимодействия

void **DragAcceptFiles**(BOOL bAccept = TRUE)	указывает, что окно будет поддерживать работу с перемещаемыми файлами (dragged files)

Функции работы с пиктограммами

HICON **SetIcon**(HICON hIcon, BOOL bBigIcon)	устанавливает дескриптор пиктограммы
HICON **GetIcon**(BOOL bBigIcon) const	возвращает дескриптор пиктограммы

Функции контекстно-зависимой справки

BOOL **SetWindowContextHelpId**(DWORD dwContextHelpId)	устанавливает идентификатор контекстной подсказки
DWORD **GetWindowContextHelpId**() const	возвращает идентификатор контекстной подсказки

Функция поддержки диалога

BOOL **UpdateData**(BOOL bSaveAndValidate = TRUE)	инициализирует или возвращает данные из блока диалога

Функции расположения и другие

void **RepositionBars**(UINT nIDFirst, UINT nIDLast, UINT nIDLeftOver, UINT nFlag = reposDefault, LPRECT lpRectParam = NULL, LPCRECT lpRectClient = NULL, BOOL bStretch = TRUE)	изменяет расположение панелей управления в рабочей области
void **UpdateDialogControls**(CCmdTarget* pTarget, BOOL bDisableIfNoHndler)	вызывается для модификации состояния кнопок и других элементов управления блока диалога
void **CenterWindow**(CWnd* pAlternateOwner = NULL)	центрирует окно относительно родительского
int **RunModalLoop**(DWORD dwFlags = 0)	возвращает, транслирует или посылает сообщения для окна, которое находится в модальном состоянии
virtual BOOL **ContinueModal**()	продолжает модальное состояние окна
virtual void **EndModalLoop**(int nResult)	заканчивает модальное состояние окна

Функции управления OLE

void AFX_CDECL **InvokeHelper**(DISPID dwDispID, WORD wFlags, VARTYPE vtRet, void* pvRet, const BYTE* pbParamInfo, ...)	вызывает метод или свойство элемента управления OLE

Void AFX_CDECL **SetProperty**(DISPID dwDispID, VARTYPE vtProp, ...)	устанавливает свойство элемента управления OLE
void **GetProperty**(DISPID dwDispID, VARTYPE vtProp, void* pvProp) const	возвращает свойство элемента управления OLE

Функции обработки сообщений управления окнами

virtual BOOL **OnCommand**(WPARAM wParam, LPARAM lParam)	вызывается, когда пользователь выбирает команду
virtual BOOL **OnNotify**(WPARAM wParam, LPARAM lParam, LRESULT* pResult)	вызывается фреймом, чтобы сообщить родительскому окну, что в одном из элементов управления произошло некоторое событие или что элемент управления нуждается в информации
afx_msg void **OnActivate**(UINT nState, CWnd* pWndOther, BOOL bMinimized)	вызывается, когда окно CWnd должно быть активным или неактивным
afx_msg void **OnActivateApp**(BOOL bActive, HTASK hTask)	вызывается, когда приложение должно быть активным или неактивным
afx_msg void **OnCancelMode**()	вызывается, чтобы позволить окну CWnd отменить любые внутренние режимы (типа захвата мыши)
afx_msg void **OnChildActivate**()	вызывается для дочерних окон многодокументного интерфейса (MDI), когда изменяется размер или позиция окна CWnd или оно активизировано
afx_msg void **OnClose**()	сообщает, что окно CWnd должно быть закрыто
afx_msg int **OnCreatse**(LPCREATESTRUCT lpCreateStruct)	вызывается как часть процесса создания окна
afx_msg HBRUSH **OnCtlColor**(CDC* pDC, CWnd* pWnd, UINT nCtlColor)	вызывается, если окно CWnd — родитель элемента управления (перед выводом элемента управления)
afx_msg void **OnDestroy**()	вызывается, когда разрушается окно CWnd
afx_msg void **OnEnable**(BOOL bEnable)	вызывается, когда окно CWnd становится доступным или недоступным

afx_msg void **OnEndSession**(BOOL bEnding)	вызывается, когда сеанс закончен
afx_msg void **OnEnterIdle**(UINT nWhy, CWnd* pWho)	вызывается, чтобы сообщить процедуре главного окна приложения, что модальное диалоговое окно или меню переходит в неактивное состояние (idle state)
afx_msg BOOL **OnEraseBkgnd**(CDC* pDC)	вызывается, когда нужно стереть фон окна
afx_msg void **OnGetMinMaxInfo**(MINMAXINFO* lpMMI)	вызывается всякий раз, когда Windows требуется знать предельные координаты или минимальный и максимальный размеры (tracking size) окна
afx_msg void **OnIconEraseBkgnd**(CDC* pDC)	вызывается, когда объект CWnd свернут, и фон пиктограммы должен быть заполнен перед рисованием
afx_msg void **OnKillFocus**(CWnd* pNewWnd)	вызывается перед тем как объект CWnd теряет фокус ввода
afx_msg LRESULT **OnMenuChar**(UINT nChar, UINT nFlags, CMenu* pMenu)	вызывается, когда пользователь нажимает символ мнемонического меню, который не соответствует любой предопределенной мнемонике в текущем меню
afx_msg void **OnMenuSelect**(UINT nItemID, UINT nFlags, HMENU hSysMenu)	вызывается, когда пользователь выбирает элемент меню
afx_msg void **OnMove**(int x, int y)	вызывается после того как позиция окна CWnd была изменена
afx_msg void **OnPaint**()	вызывается, чтобы перерисовать часть окна
afx_msg void **OnParentNotify**(UINT message, LPARAM lParam)	вызывается, когда дочернее окно создано или разрушено, или когда пользователь нажимает кнопку мыши, в то время как курсор находится над дочерним окном
afx_msg HCURSOR **OnQueryDragIcon**()	вызывается, когда свернутое окно CWnd должно быть перемещено пользователем
afx_msg BOOL **OnQueryEndSession**()	вызывается, когда пользователь заканчивает сеанс работы в Windows
afx_msg BOOL **OnQueryNewPalette**()	сообщает окну CWnd, что оно собирается получать входной фокус
afx_msg BOOL **OnQueryOpen**()	вызывается, когда окно CWnd свернуто (пиктограмма), и пользователь запрашивает его открытие

afx_msg void **OnSetFocus**(CWnd* pOldWnd)	вызывается после того как окно CWnd получает фокус ввода
afx_msg void **OnShowWindow**(BOOL bShow, UINT nStatus)	вызывается, когда окно CWnd должно быть скрыто или показано
afx_msg void **OnSize**(UINT nType, int cx, int cy)	вызывается после того как размер окна CWnd изменился
afx_msg void **OnWindowPosChanging**(WINDOWPOS* lpwndpos)	вызывается, когда размер, позиция или Z-порядок собирается измениться в результате обращения к SetWindowPos или другой функции управления окна
afx_msg void **OnWindowPosChanged**(WINDOWPOS* lpwndpos)	вызывается, когда размер, позиция или Z-порядок изменились в результате обращения к SetWindowPos или другой функции управления окна

Функции обработки сообщений нерабочей области окна

afx_msg BOOL **OnNcActivate**(BOOL bActive)	вызывается, когда нерабочая область должна быть изменена, чтобы указать активное или неактивное состояние
afx_msg void **OnNcCalcSize**(BOOL bCalcValidRects, NCCALCSIZE_PARAMS* lpncsp)	вызывается, когда должен быть вычислен размер и позиция рабочей области
afx_msg BOOL **OnNcCreate**(LPCREATESTRUCT lpCreateStruct)	вызывается до OnCreate, когда создается нерабочая область
afx_msg void **OnNcDestroy**()	вызывается, когда нерабочая область разрушается
afx_msg UINT **OnNcHitTest**(CPoint point)	вызывается системой Windows каждый раз при перемещении мыши, если окно CWnd содержит курсор или захватило ввод мыши с помощью функции SetCapture
afx_msg void **OnNcLButtonDblClk**(UINT nHitTest, CPoint point)	вызывается при двойном щелчке левой кнопки мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcLButtonDown**(UINT nHitTest, CPoint point)	вызывается, когда пользователь нажимает левую кнопку мыши, в то время как курсор находится внутри нерабочей области окна CWnd

afx_msg void **OnNcLButtonUp**(UINT nHitTest, CPoint point)	вызывается, когда пользователь отпускает левую кнопку мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcMButtonDblClk**(UINT nHitTest, CPoint point)	вызывается при двойном щелчке средней кнопки мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcMButtonDown**(UINT nHitTest, CPoint point)	вызывается, когда пользователь нажимает среднюю кнопку мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcMButtonUp**(UINT nHitTest, CPoint point)	вызывается, когда пользователь отпускает среднюю кнопку мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcMouseMove**(UINT nHitTest, CPoint point)	вызывается при перемещении курсора внутри нерабочей области окна CWnd
afx_msg void **OnNcPaint**()	вызывается для перерисовки нерабочей области окна CWnd
afx_msg void **OnNcRButtonDblClk**(UINT nHitTest, CPoint point)	вызывается при двойном щелчке правой кнопки мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcRButtonDown**(UINT nHitTest, CPoint point)	вызывается, когда пользователь нажимает правую кнопку мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnNcRButtonUp**(UINT nHitTest, CPoint point)	вызывается, когда пользователь отпускает правую кнопку мыши, в то время как курсор находится внутри нерабочей области окна CWnd
afx_msg void **OnDropFiles**(HDROP hDropInfo)	вызывается, когда пользователь отпускает левую кнопку мыши над окном, которое зарегистрировало себя как получатель перемещаемых файлов

Функции обработки системных сообщений

afx_msg void **OnPaletteIsChanging**(CWnd* pRealizeWnd)	сообщает другим приложениям, что приложение собирается изменять логическую палитру

afx_msg void **OnSysChar**(UINT nChar, UINT nRepCnt, UINT nFlags)	вызывается при нажатии системной клавиши
afx_msg void **OnSysCommand**(UINT nID, LPARAM lParam)	вызывается, когда пользователь выбирает команду из системного меню, кнопку Maximize (Развернуть) или Minimize (Свернуть)
afx_msg void **OnSysDeadChar**(UINT nChar, UINT nRepCnt, UINT nFlags)	вызывается при нажатии системной клавиши, передающей мертвый символ (типа символов диакритических знаков)
afx_msg void **OnSysKeyDown**(UINT nChar, UINT nRepCnt, UINT nFlags)	вызывается, когда пользователь удерживает клавишу <Alt> и при этом нажимает другую клавишу
afx_msg void **OnSysKeyUp**(UINT nChar, UINT nRepCnt, UINT nFlags)	вызывается, когда пользователь отпускает клавишу, которая была нажата одновременно с клавишей <Alt>
afx_msg void **OnCompacting**(UINT nCpuTime)	вызывается, когда Windows обнаруживает недостаток системной памяти
afx_msg void **OnDevModeChange**(LPTSTR lpDeviceName)	вызывается для всех окон высшего уровня, когда пользователь изменяет установки режима устройства (device-mode settings)
afx_msg void **OnFontChange**()	вызывается при изменении набора шрифтов
afx_msg void **OnPaletteChanged**(CWnd* pFocusWnd)	вызывается, чтобы позволить окнам, которые используют цветную палитру, использовать логические палитры, и модифицировать рабочие области этих окон
afx_msg void **OnSpoolerStatus**(UINT nStatus, NT nJobs)	вызывается из Print Manager (Диспетчер печати) всякий раз, когда задание добавлено или удалено из очереди
afx_msg void **OnSysColorChange**()	вызывается для всех окон верхнего уровня, когда сделано изменение в установке системных цветов
afx_msg void **OnTimeChange**()	вызывается для всех окон верхнего уровня после изменения системного времени
afx_msg void **OnWinIniChange**(LPCTSTR lpszSection)	вызывается для всех окон верхнего уровня после изменения файла инициализации Windows WIN.INI

Функции обработки входных сообщений

afx_msg void **OnChar**(NT nChar, INT nRepCnt, UINT nFlags)	вызывается, когда нажатие клавиши транслируется в несистемный символ
afx_msg void **OnDeadChar**(NT nChar, UINT nRepCnt, UINT nFlags)	вызывается, когда нажатие клавиши транслируется в мертвый символ (типа символов диакритических знаков)
afx_msg void **OnHScroll**(UINT nSBCode, UINT nPos, CscrollBar* pScrollBar)	вызывается, когда пользователь нажимает горизонтальную полосу прокрутки окна CWnd
afx_msg void **OnVScroll**(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)	вызывается, когда пользователь нажимает вертикальную полосу прокрутки окна CWnd
afx_msg void **OnKeyDown**(UINT nChar, UINT nRepCnt, UINT nFlags)	вызывается, когда нажата несистемная клавиша
afx_msg void **OnKeyUp**(UINT nChar, UINT nRepCnt, UINT nFlags)	вызывается, когда отпущена несистемная клавиша
afx_msg void **OnLButtonDblClk**(UINT nFlags, CPoint point)	вызывается при двойном щелчке левой кнопки мыши
afx_msg void **OnLButtonDown**(UINT nFlags, CPoint point)	вызывается при нажатии левой кнопки мыши
afx_msg void **OnLButtonUp**(UINT nFlags, CPoint point)	вызывается при отпускании левой кнопки мыши
afx_msg void **OnMButtonDblClk**(UINT nFlags, CPoint point)	вызывается при двойном щелчке средней кнопки мыши
afx_msg void **OnMButtonDown**(UINT nFlags, CPoint point)	вызывается при нажатии средней кнопки мыши
afx_msg void **OnMButtonUp**(UINT nFlags, CPoint point)	вызывается при отпускании средней кнопки мыши

afx_msg int **OnMouseActivate**(CWnd* pDesktopWnd, UINT nHitTest, UINT message)	вызывается, когда курсор находится в неактивном окне, и пользователь нажимает кнопку мыши
afx_msg void **OnMouseMove**(UINT nFlags, CPoint point)	вызывается при перемещении курсора мыши
afx_msg void **OnRButtonDblClk**(UINT nFlags, CPoint point)	вызывается при двойном щелчке правой кнопки мыши
afx_msg void **OnRButtonDown**(UINT nFlags, CPoint point)	вызывается при нажатии правой кнопки мыши
afx_msg void **OnRButtonUp**(UINT nFlags, CPoint point)	вызывается при отпускании правой кнопки мыши
afx_msg BOOL **OnSetCursor**(CWnd* pWnd, UINT nHitTest, UINT message)	вызывается, если ввод мыши не фиксируется (не захвачен), и мышь вызывает движение курсора внутри окна
afx_msg void **OnTimer**(UINT nIDEvent)	вызывается после каждого интервала времени, установленного функцией SetTimer

Функции обработки инициализирующих сообщений меню

afx_msg void **OnInitMenu**(CMenu* pMenu)	вызывается перед активизацией меню
afx_msg void **OnInitMenuPopup**(CMenu* pPopupMenu, UINT nIndex, BOOL bSysMenu)	вызывается перед активизацией всплывающего меню

Функции обработки сообщений буфера обмена

afx_msg void **OnAskCbFormatName**(UINT nMaxCount, LPTSTR lpszString)	вызывается приложением просмотра буфера обмена, когда владелец буфера обмена отображает его содержимое
afx_msg void **OnChangeCbChain**(HWND hWndRemove, HWND hWndAfter)	сообщает, что определенное окно удаляется из цепочки

afx_msg void **OnDestroyClipboard**()	вызывается, когда буфер обмена освобожден обращением к функции Windows EmptyClipboard
afx_msg void **OnDrawClipboard**()	вызывается, когда изменяется содержимое буфера обмена
afx_msg void **OnHScrollClipboard**(CWnd* pClipAppWnd, UINT nSBCode, UINT nPos)	вызывается, когда владелец буфера обмена должен листать содержимое буфера обмена; делает недействительным соответствующий раздел и модифицирует значение позиции полосы прокрутки
afx_msg void **OnPaintClipboard**(CWnd* pClipAppWnd, HGLOBAL hPaintStruct)	вызывается, когда необходимо перерисовать рабочую область просмотра буфера обмена
afx_msg void **OnRenderAllFormats**()	вызывается, когда приложение-владелец разрушается, и нужно пересмотреть все форматы
afx_msg void **OnRenderFormat**(UINT nFormat)	вызывается для владельца буфера обмена, когда должен быть представлен специфический формат с отсроченным представлением
afx_msg void **OnSizeClipboard**(CWnd* pClipAppWnd, HGLOBAL hRect)	вызывается, когда изменился размер рабочей области окна просмотра буфера обмена
afx_msg void **OnVScrollClipboard**(CWnd* pClipAppWnd, UINT nSBCode, UINT nPos)	вызывается, когда владелец должен листать содержимое буфера обмена, делает недействительным соответствующий раздел и модифицирует значение позиции полосы прокрутки

Функции обработки управляющих сообщений

afx_msg int **OnCompareItem**(int nIDCtl, LPCOMPAREITEMSTRUCT lpCompareItemStruct)	вызывается, чтобы определить относительную позицию нового элемента в дочернем отсортированном раскрывающемся списке или списке, у которого установлен стиль самоотображения (owner-draw)
afx_msg void **OnDeleteItem**(int nIDCtl, LPDELETEITEMSTRUCT lpDeleteItemStruct)	вызывается, когда разрушается список или раскрывающийся список, у которого установлен стиль самоотображения, или когда из него удалены элементы

afx_msg void **OnDrawItem**(int nIDCtl, LPDRAWITEMSTRUCT lpDrawItemStruct)	вызывается, когда должно быть выведено свое изображение на кнопке, в раскрывающемся списке, в списке или в меню
afx_msg UINT **OnGetDlgCode**()	вызывается для элемента управления, так чтобы он мог обрабатывать непосредственный ввод клавиш перемещения курсора и клавиши <Tab>
afx_msg void **OnMeasureItem**(int nIDCtl, LPMEASUREITEMSTRUCT lpMeasureItemStruct)	вызывается при создании изображения в списке, раскрывающемся списке или элементе меню; окно CWnd сообщает Windows о размерах элементов управления
afx_msg int **OnCharToItem**(UINT nChar, CListBox* pListBox, UINT nIndex)	вызывается списком с установленным стилем LBS_WANTKEYBOARDINPUT в ответ на сообщение WM_CHAR
afx_msg int **OnVKeyToItem**(UINT nKey, CListBox* pListBox, UINT nIndex)	вызывается полем списка в ответ на сообщение WM_KEYDOWN

Функция обработки сообщений MDI

afx_msg void **OnMDIActivate**(BOOL bActivate, CWnd* pActivateWnd, CWnd* pDeactivateWnd)	вызывается, когда активизируется или деактивизируется дочернее окно MDI

Функции обработки сообщений цикла меню

afx_msg void **OnEnterMenuLoop**(BOOL bIsTrackPopupMenu)	вызывается, когда начинается модальный цикл меню
afx_msg void **OnExitMenuLoop**(BOOL bIsTrackPopupMenu)	вызывается, когда закончен модальный цикл меню

Функции обработки основных сообщений

afx_msg void **OnStyleChanged**(int nStyleType, LPSTYLESTRUCT lpStyleStruct)	указывает, что изменились один или больше стилей окна
afx_msg void **OnStyleChanging**(int nStyleType, LPSTYLESTRUCT lpStyleStruct)	указывает, что один или больше стилей окна собираются измениться

afx_msg void **OnSizing**(UINT nSide, LPRECT lpRect)	указывает, что пользователь изменяет размеры прямоугольника
afx_msg void **OnMoving**(UINT nSide, LPRECT lpRect)	указывает, что пользователь перемещает объект окна CWnd
afx_msg void **OnCaptureChanged**(CWnd* pWnd)	посылает сообщение окну, которое теряет захват мыши
afx_msg BOOL **OnDeviceChange**(UINT nEventType, DWORD dwData)	сообщает об изменении аппаратной конфигурации

Переопределяемые и другие вспомогательные функции

virtual void **DoDataExchange**(CDataExchange* pDX)	предназначена для обмена данными с блоком диалога и проверки правильности; вызывается функцией UpdateData
virtual BOOL **PreTranslateMessage**(MSG* pMsg)	используется объектом CWinApp, чтобы фильтровать сообщения окна перед тем как послать их в функциям TranslateMessage и DispatchMessage
virtual BOOL **OnAmbientProperty**(COleControlSite* pSite, DISPID dispid, VARIANT* pvar)	вызывается для получения значений свойств окна, которое содержит элементы управления OLE
virtual LRESULT **WindowProc**(UINT message, WPARAM wParam, LPARAM lParam)	обеспечивает оконную процедуру для окна CWnd; по умолчанию посылает сообщения через карту сообщений
virtual BOOL **OnWndMsg**(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)	указывает, было ли обработано сообщение
LRESULT **Default**()	вызывает заданную по умолчанию процедуру, которая обеспечивает обработку любых сообщений окна, не обработанных приложением
virtual LRESULT **DefWindowProc**(UINT message, WPARAM wParam, LPARAM lParam)	вызывает заданную по умолчанию процедуру окна, которая обеспечивает обработку любых сообщений окна, не обработанных приложением

virtual void **PostNcDestroy**()	вызывается по умолчанию функцией OnNcDestroy после того, как окно было разрушено
virtual BOOL **OnChildNotify**(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)	вызывается родительским окном, чтобы дать сообщению возможность ответить на извещение от элементов управления
BOOL **ReflectChildNotify**(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)	функция-помощник, которая "отражает" сообщение к его источнику
static BOOL PASCAL **ReflectLastMsg**(HWND hWndChild, LRESULT* pResult = NULL)	отражает последнее сообщение дочернему окну
static void PASCAL **SendMessageToDescendants**(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam, BOOL bDeep, BOOL bOnlyPerm)	посылает сообщение всем дочерним окнам данного окна
DWORD **GetStyle**() const	возвращает текущий стиль окна
DWORD **GetExStyle**() const	возвращает расширенный стиль окна
static BOOL PASCAL **ModifyStyle**(HWND hWnd, DWORD dwRemove, DWORD dwAdd, UINT nFlags) и BOOL **ModifyStyle**(DWORD dwRemove, DWORD dwAdd, UINT nFlags = 0)	изменяют текущий стиль окна
static BOOL PASCAL **ModifyStyleEx**(HWND hWnd, DWORD dwRemove, DWORD dwAdd, UINT nFlags) и BOOL **ModifyStyleEx**(DWORD dwRemove, DWORD dwAdd, UINT nFlags = 0)	изменяют расширенный стиль окна

Приложение 3
Типы проектов и мастера

Типы проектов Visual C++

Тип проекта определяет состав и содержимое автоматически генерируемых файлов, действующие по умолчанию установки компилятора, подключаемые библиотеки и некоторые другие параметры.

Для выбора типа создаваемого проекта можно выбрать пункт меню File\New\Project Workspace (Файл\Создать\Проект) или Insert\Project (Вставка\Проект), если вы хотите вставить новый проект в уже существующий. При этом на экране появляется окно (рис. 1), в котором следует задать имя вашего приложения, его месторасположение и выбрать тип создаваемого проекта.

В Visual C++ версии 4.0 определены девять типов проектов:

- *MFC AppWizard (exe).* Состав проекта: файлы проекта (MDP- и MAK-файлы), исходные файлы будущего приложения (EXE-модуля) на базе классов библиотеки MFC, файлы ресурсов, каркас справочной системы (по выбору) и т. д. В дальнейшем возможно добавление файлов с исходными текстами, написанными как без, так и с использованием библиотеки классов MFC;
- *MFC AppWizard (dll).* Состав проекта: файлы проекта (MDP- и MAK-файлы), исходные файлы будущей библиотеки динамической компоновки (DLL-модуля) на базе классов библиотеки MFC, файлы ресурсов и т. д. В дальнейшем возможно добавление файлов с исходными текстами, написанными как без, так и с использованием библиотеки классов MFC;
- *OLE ControlWizard.* Состав проекта: файлы проекта (MDP- и MAK-файлы), исходные файлы будущего элемента управления OLE (OCX-файлы) на базе библиотеки классов MFC. В дальнейшем возможно добавление файлов с исходными текстами, написанными как без, так и с использованием библиотеки классов MFC;
- *Application.* Состав проекта: файлы проекта (MDP- и MAK-файлы) будущего приложения (EXE-модуля). В дальнейшем возможно дополнение исходными текстами, написанными как без, так и с использованием библиотеки классов MFC;
- *Dinamic-Link Library.* Состав проекта: файлы проекта (MDP- и MAK-файлы) будущей библиотеки динамической компоновки (DLL-модуля). В дальнейшем возможно дополнение исходными текстами, написанными как без, так и с использованием библиотеки классов MFC;
- *Console Application.* Состав проекта: файлы проекта (MDP- и MAK-файлы) будущего консольного приложения (EXE-модуля). В дальнейшем воз-

можно дополнение исходными текстами, использующими консольные функции Win32 API, которые обеспечивают эмуляцию текстового консольного терминала с поддержкой стандартных функций ввода/вывода;

- *Static Library*. Состав проекта: файлы проекта (MDP- и MAK-файлы) будущей статической библиотеки (LIB-файла). В дальнейшем возможно дополнение исходными текстами, написанными как без, так и с использованием библиотеки классов MFC, а также добавление объектных файлов;
- *Makefile*. Состав проекта: файлы проекта (MDP- и MAK-файлы), которые могут быть использованы для создания приложения при помощи версии Visual C++, отличной от текущей;
- *Custom AppWizard*. Состав проекта: файлы проекта (MDP- и MAK-файлы), исходные файлы будущего пользовательского генератора программ (AVX-файл) на базе классов библиотеки MFC, файлы ресурсов. В дальнейшем возможно добавление файлов с исходными текстами, написанными как без, так и с использованием библиотеки классов MFC. Необходимо добавить файлы-шаблоны, которые будут лежать в основе создаваемых приложений.

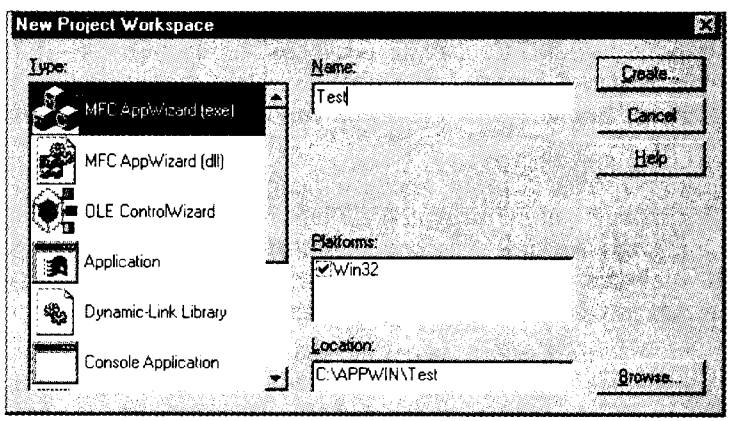


Рис. 1. Блок диалога для выбора типа создаваемого проекта

Если перечисленные типы проектов вас не удовлетворяют, можно создать свой тип на базе *Custom AppWizard* и добавить его к имеющемуся списку.

Мастера (Wizards)

Visual C++ содержит в своем составе два мастера — *AppWizard* и *ClassWizard*, а также панель инструмента *WizardBar*, которые, при использовании библиотеки классов MFC, позволяют максимально избавиться от рутинной работы, связанной с процессом программирования, и сосредоточиться на творчестве.

AppWizard

Мастер *AppWizard* — это инструмент, который вы можете использовать для создания приложений Windows на основе библиотеки классов MFC. Его можно использовать для быстрого создания каркаса приложений — для генерации исполняемых файлов (*.EXE) или библиотек динамической компоновки (файлов *.DLL).

Интерфейс *AppWizard* легок и прост в использовании, но вместе с тем он достаточно мощный и гибкий для того, чтобы быстро сгенерировать каркас приложений:

- основанных на однодокументном (SDI) или многодокументном (MDI) интерфейсе Windows, а также на основе блока диалога;
- поддерживающих интерфейс OLE и работу с базами данных (ODBC и DAO);
- имеющих панели инструментов (toolbars), строку состояния (statusbar), поддержку контекстно-зависимой справки и объемное изображение элементов управления;
- со встроенной поддержкой многих стандартных команд меню.

Доступ к мастеру вы получаете при создании проектов типа *MFC AppWizard (exe)* и *MFC AppWizard (dll)*. Настройка осуществляется за шесть шагов. На первом шаге (рис. 2) вы можете выбрать тип главного окна приложения, обеспечивающего поддержку одно- или многодокументного интерфейса, или базирующегося на блоке диалога. Кроме того, можно выбрать язык, который будет использован в исходных текстах программы (только из списка доступных).

На следующем шаге (рис. 3) вы можете определить возможность работы с базами данных.

Здесь предоставляются следующие варианты, задаваемые переключателями:

None (Нет) — не обеспечивается поддержка баз данных (по умолчанию).

Header files only (Только заголовочные файлы) — *AppWizard* создает и включает файл заголовка AFXDB.H, обеспечивая основную поддержку базы данных. Вы можете создавать наборы записей (recordset) и затем проверять и модифицировать их.

Database view without file support (Просмотр базы данных без файловой поддержки) — предписывает *AppWizard* создать основанное на формах приложение, в котором просмотр и модификация записей производится через набор записей. Приложение не поддерживает сериализацию документов, так как большинство приложений баз данных работает на основе записей быстрее, чем на основе файлов.

Database view with file support (Просмотр базы данных с файловой поддержкой) — предписывает создать основанное на формах приложение, в котором

просмотр и модификация записей также производится через набор записей. В данном случае приложение поддерживает сериализацию, что может использоваться, например, для обновления файла настроек пользователя.

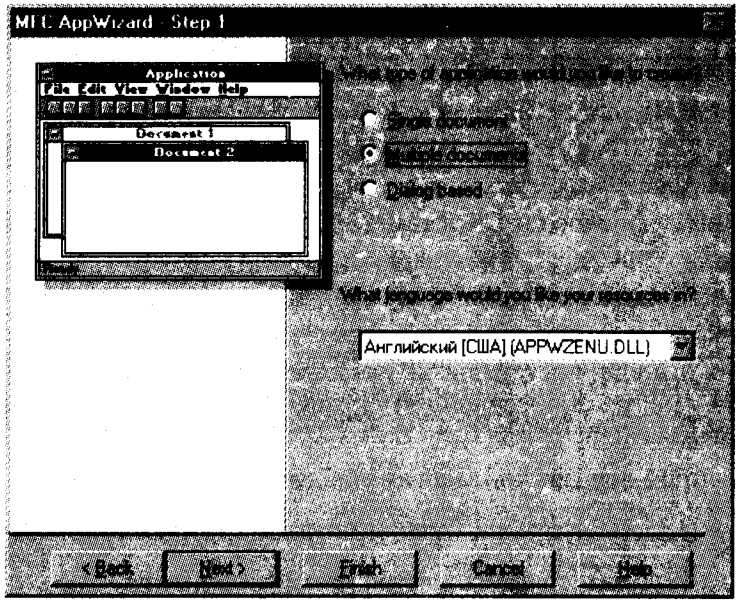


Рис. 2. Мастер AppWizard, шаг 1

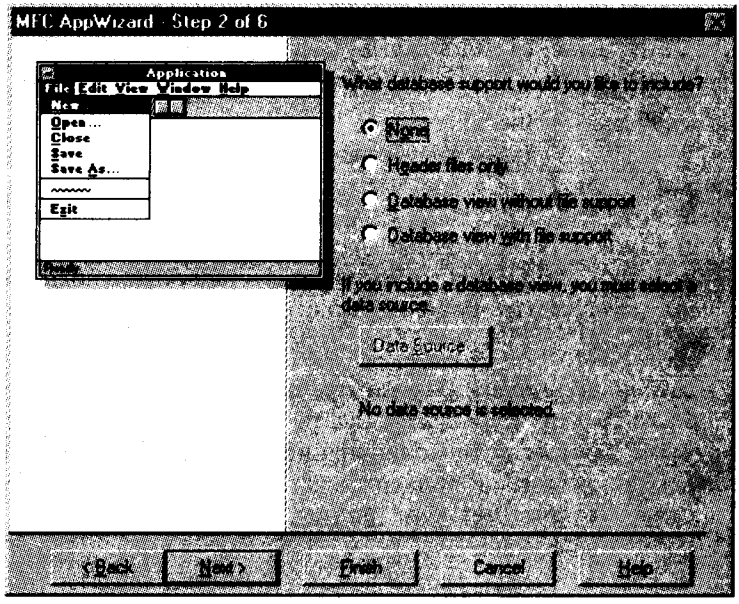


Рис. 3. Мастер AppWizard, шаг 2

Кнопка *Data Source* (Исходные данные) — открывает диалоговое окно Database Options (Опции баз данных) для задания типа базы данных, которая должна поддерживаться в вашем приложении. Эта кнопка доступна только, если включен просмотр базы данных.

На следующем шаге (рис. 4) вы можете определить возможность поддержки OLE.

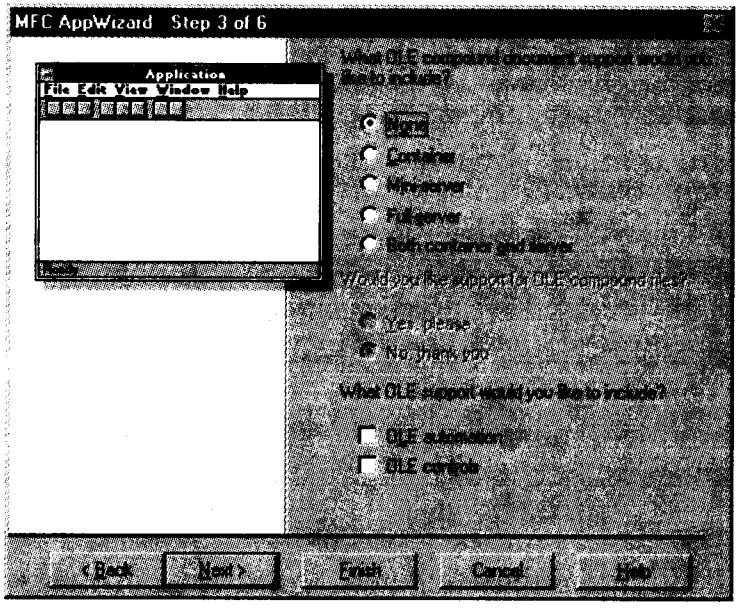


Рис. 4. Мастер AppWizard, шаг 3

Здесь имеются следующие опции:

Переключатель *None* (Нет) — выбирается, если не требуется поддержка OLE (по умолчанию).

Переключатель *Container* (Контейнер) — приложение может содержать связанные и внедренные объекты.

Переключатель *Mini-server* (Мини-Сервер) — при данном выборе приложение может создавать и управлять составными документами. Мини-сервер не может выполняться автономно и поддерживает только внедренные записи.

Переключатель *Full-server* (Полный Сервер) — позволяет создавать и управлять составными документами. Полный сервер способен выполняться автономно и поддерживает связанные и внедренные объекты.

Переключатель *Both container and server* (И контейнер и сервер) — выбирается, если нужно, чтобы ваше приложение было и контейнером и сервером одновременно. *Контейнер* — это приложение, которое может включать внедренные или связанные объекты в собственные документы. *Сервер* — это

приложение, которое может создавать объекты OLE для использования их контейнерами.

Переключатель *Yes, please* (Да, пожалуйста) — выбирается, чтобы обеспечить вашему документу "контейнер OLE" поддержку сериализации с использованием составного формата файла OLE. Составной формат файла предназначен для сохранения документа, который содержит один или больше объектов OLE в одном файле и позволяет иметь доступ к отдельным файлам объектов OLE.

Переключатель *No, thank you* (Нет, спасибо) — при данном выборе не поддерживается сериализация документов-контейнеров OLE.

Флажок *OLE automation* (Автоматизация OLE) — устанавливается, если вы хотите сделать ваше приложение доступным клиентам автоматизации OLE, например, Microsoft Excel.

Флажок *OLE controls* (Элементы управления OLE) — устанавливается, если вы хотите, чтобы приложение использовало элементы управления OLE.

На следующем шаге (рис. 5) вы можете определить стандартные возможности, которые *AppWizard* должен включить в ваше приложение:

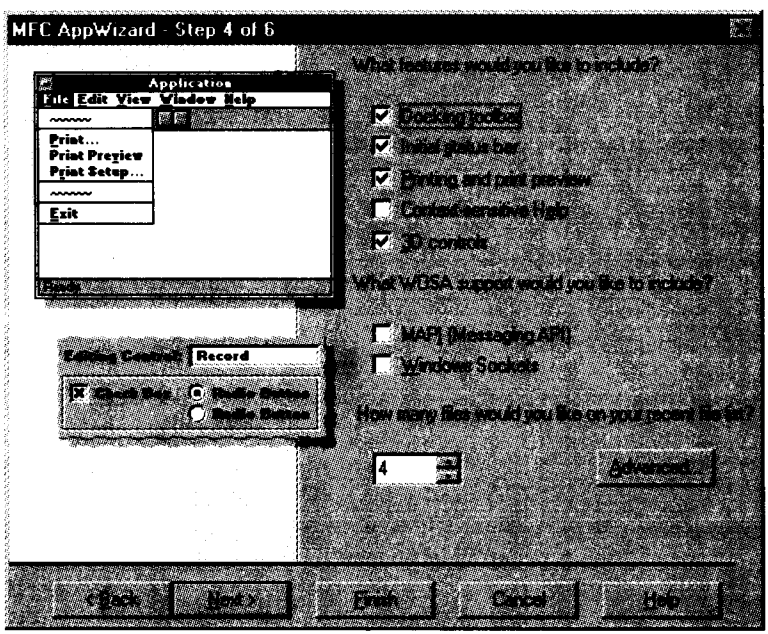


Рис. 5. Мастер AppWizard, шаг 4

Флажок *Docking toolbar* (Панель инструментов) — определяет, нужно ли создавать панель инструментов. По умолчанию панель инструментов содержит кнопки для создания нового документа, открытия и сохранения фай-

лов, вырезания, копирования, вставки, печати и вызова окна справки. Этот флажок установлен по умолчанию.

Флажок *Initial status bar* (Строка состояния) — определяет, будет ли главное окно приложения иметь строку состояния. Строка состояния содержит автоматические индикаторы для клавиш <Caps Lock>, <Num Lock> и <Scroll Lock> клавиатуры и строку для отображения информации о командах меню и кнопках панели инструментов. Флажок установлен по умолчанию.

Флажок *Printing and print preview* (Печать и просмотр печати) — *AppWizard* генерирует код для печати, настроек печати и предварительного просмотра печати. Флажок установлен по умолчанию.

Флажок *Context-sensitive Help* (Контекстно-зависимая справка) — *AppWizard* генерирует набор файлов справки, которые используются для создания контекстно-зависимой справки. При этом необходимо будет использовать *компилятор файлов справки* (help compiler).

Флажок *3D controls* (Трехмерные элементы управления) — включает использование затенения, что создает видимость объемности при изображении элементов управления. Флажок установлен по умолчанию.

Флажок *MAPI* (Messaging API) — позволяет генерировать приложение, которое создает, использует, передает и сохраняет почтовые сообщения.

Флажок *Windows Sockets* (Сокеты Windows) — позволяет приложению использовать сетевой протокол TCP/IP.

Поле *How many files would you like on your recent file list?* (Сколько файлов вы хотите запоминать в списке последних файлов?) — задает число файлов, запоминаемых в списке последних использованных файлов.

Кнопка *Advanced* (Дополнительно) — открывает дополнительное диалоговое окно, которое можно использовать для более тонкой настройки работы с файлами и стилей окон вашего приложения.

Следующий шаг (рис. 6) позволяет определить нужны ли вам встроенные комментарии, а также выбрать тип используемой библиотеки.

Переключатель *Yes, please* (Да, пожалуйста) — *AppWizard* генерирует и вставляет комментарии в файлы с исходными текстами, которые показывают, где вы должны добавить собственный текст программы. Этот переключатель выбран по умолчанию.

Переключатель *No, thank you* (Нет, спасибо) — *AppWizard* не вставляет комментарии в исходные тексты программы.

Переключатель *As a shared DLL* (Как библиотеку DLL) — приложение во время выполнения будет вызывать необходимые функции библиотеки MFC из динамически подключаемых библиотек.

Переключатель *As a statically linked library* (Как статически подключаемую библиотеку) — все необходимые функции библиотеки MFC будут подключены во время создания программы.

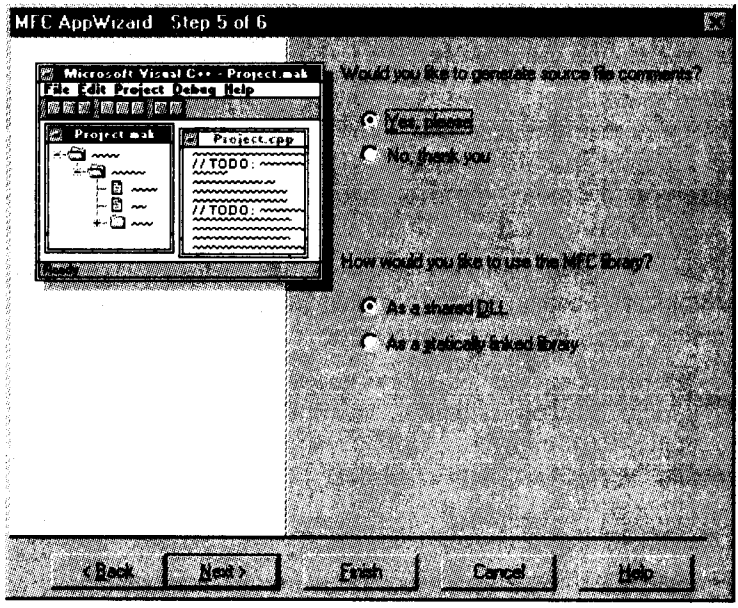


Рис. 6. Мастер AppWizard, шаг 5

Последний шаг (рис. 7) позволяет изменить имена классов, генерируемые *AppWizard*, а также изменить имена файлов с исходными текстами и заголовочных файлов.

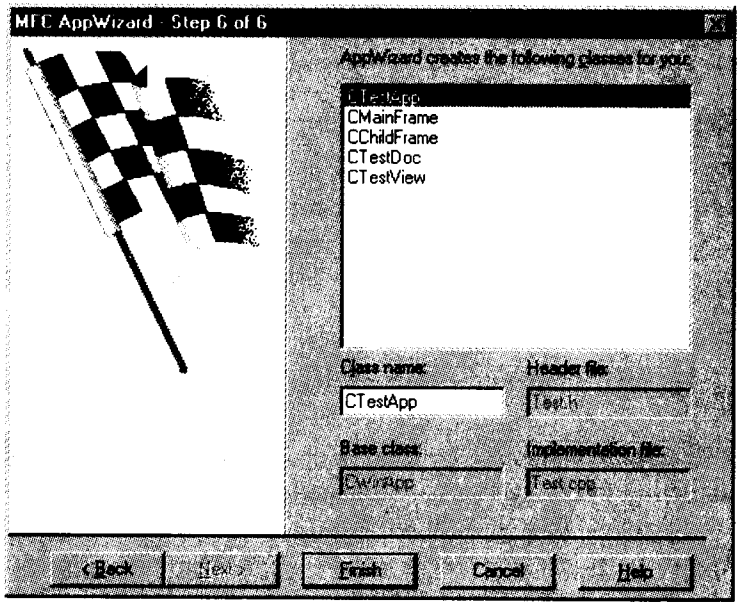


Рис. 7. Мастер AppWizard, шаг 6

После окончания процесса настройки *AppWizard* автоматически генерирует файлы, необходимые для построения приложения Windows: файлы с исходными текстами, файлы заголовков, файл ресурсов, файл проекта и так далее. Все коды, которые сгенерировал *AppWizard*, основаны на использовании библиотеки классов MFC и обеспечивают полную совместимость с *ClassWizard*. При этом вы получаете каркас работающего приложения с большим набором встроенных функциональных возможностей.

ClassWizard

Мастер *ClassWizard* помогает в создании классов, переменных, а также обработчиков сообщений и команд и доступен только для приложений, основанных на библиотеке классов MFC. Он также упрощает работу с базами данных и OLE. *ClassWizard* позволяет вам:

- создавать новые классы, производные от многих из основных, которые обрабатывают сообщения Windows;
- связывать сообщения с окнами, блоками диалога, элементами управления, пунктами меню и акселераторами;
- создавать новые и удалять существующие обработчики сообщений;
- посмотреть, какие сообщения уже имеют обработчики, и перейти непосредственно к нужному месту в исходном тексте;
- определить переменные, которые автоматически инициализируют и обновляют данные, вводимые пользователем в блоки диалога или формы;
- добавлять функции OLE при создании нового класса.

Чтобы запустить *ClassWizard*, достаточно в меню View выбрать пункт ClassWizard или нажать соответствующую кнопку на основной панели инструментов. В появившемся окне (рис. 8) представлены следующие вкладки:

Message Maps (Карты сообщений) — для просмотра извещений или других сообщений элементов управления, связанных с каждым объектом, и создания соответствующего обработчика.

Member Variables (Переменные) — позволяет добавлять новые или редактировать существующие переменные, которые связаны с блоком диалога, формой, элементом управления, просмотром записей или с полями набора записей.

OLE Automation (Автоматизация OLE) — используется для создания автоматизации OLE и увеличения функциональных возможностей вашей программы.

OLE Events (События OLE) — используется для поддержки событий OLE.

Class Info (Информация о классе) — предназначена для просмотра и редактирования характеристик выбранного класса.

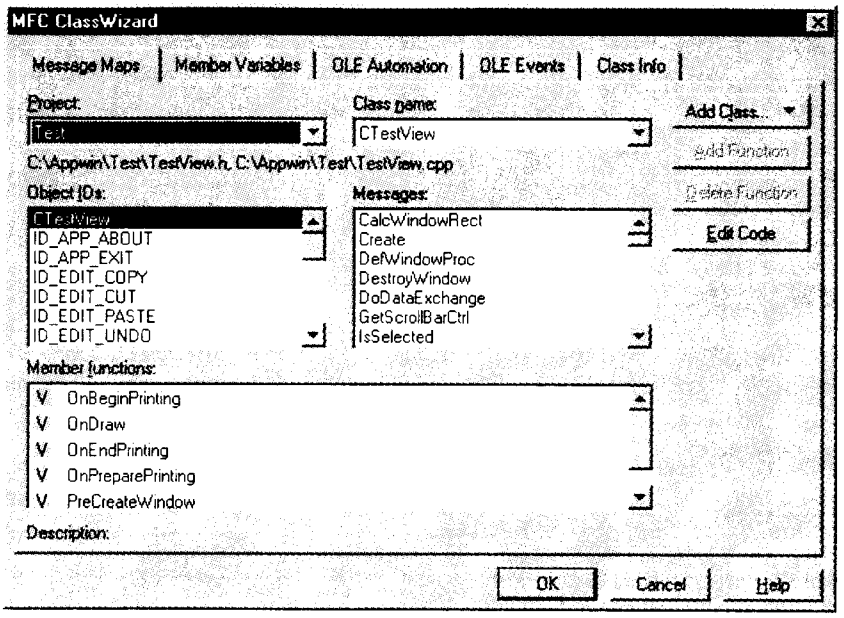


Рис. 8. Основное окно мастера ClassWizard

Вкладка *Message Maps* (Карты сообщений) открывает доступ к следующим опциям.

Раскрывающиеся списки *Project* (Проект) и *Class Name* (Имя класса) — позволяют выбрать имя проекта и имя класса в проекте.

Список *Object IDs* (Идентификаторы объектов) — содержит идентификаторы объектов, таких как пункты меню, блоки диалога и т. п. Первое имя в списке — имя текущего класса.

Список *Messages* (Сообщения) — содержит сообщения, которые объект, выбранный в списке Object IDs (Идентификаторы объектов), может обрабатывать. Сообщения, выделенные жирным шрифтом, уже имеют обработчики.

Список *Member Functions* (Функции-члены) — содержит функции выбранного класса: отмеченные символом "V", являются виртуальными функциями, отмеченные "W" — обработчиками сообщений Windows.

Кнопка *Add Class* (Добавить класс) — позволяет добавить в проект новый класс. Можно создать совсем новый класс или импортировать существующий из файла или библиотеки OLE.

Кнопка *Add Function* (Добавить функцию) — позволяет добавить в список виртуальную функцию или обработчик сообщения в зависимости от выбора, сделанного в списке Messages (Сообщения).

Кнопка *Delete Function* (Удалить функцию) — позволяет удалить объявление функции из заголовочного файла.

Кнопка *Edit Code* (Редактировать код) — открывает окно редактирования на выбранной функции. Этот код определяет обработчик сообщения или виртуальную функцию для объекта.

Если нажать кнопку AddClass (Добавить класс), а затем выбрать команду New (Новый), то на экране появится блок диалога (рис. 9) для выбора параметров создаваемого класса.

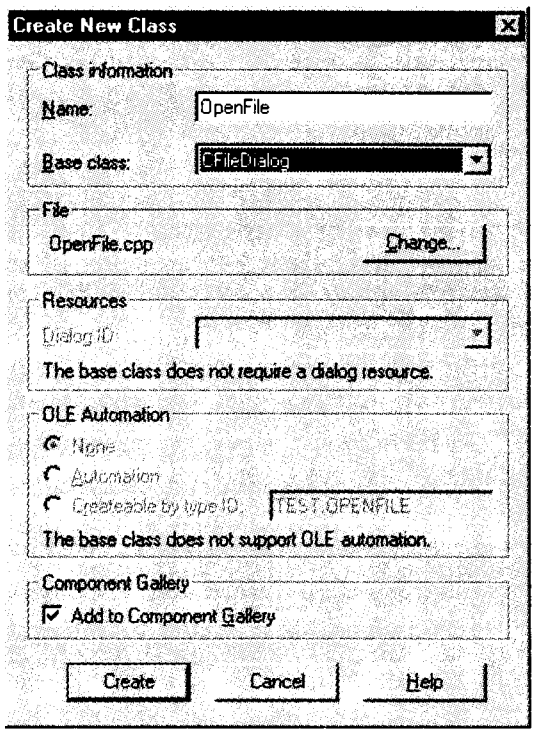


Рис. 9. Блок диалога для создания нового класса

В нем доступны следующие опции:

Поле *Name* (Имя) — имя класса, который вы хотите добавить к приложению. Имя файла, в который будет помещена реализация этого класса, отобразится в поле File (Файл).

Раскрывающийся список *Base Class* (Базовый класс) — базовый класс, на основе которого будет создаваться новый.

Кнопка *Change* (Изменить) — открывает диалоговое окно, позволяющее изменить имена файлов исходных текстов и заголовочных файлов.

Раскрывающийся список *Dialog ID* (Идентификатор диалога) — позволяет изменить идентификатор ресурса. Эта опция доступна только для классов, которые связаны с идентификаторами ресурсов, таких как блоки диалога.

Переключатель *None* (Нет) — автоматизация OLE не используется.

Переключатель *Automation* (Автоматизация) — предоставляет класс клиентам автоматизации OLE. Доступен только для некоторых классов.

Переключатель *Createable by Type ID* (Создаваемы по типу идентификатора) — подобен переключателю Automation (Автоматизация), но дополнительно предоставляет клиентам автоматизации OLE возможность непосредственно использовать запрос к объекту вашего класса. Доступен только для некоторых классов.

Флажок *Add to Component Gallery* (Добавить к галерее компонентов) — добавляет новый класс и связанные с ним ресурсы к Component Gallery — набору шаблонов объектов, которые можно вставлять в ваше приложение.

WizardBar

WizardBar — это панель инструментов, тесно связанная с *ClassWizard*, которая упрощает стандартные задачи определения обработчиков сообщений и переопределения виртуальных функций MFC (рис. 10). *WizardBar* позволяет:

- просматривать сообщения Windows, связанные с текущим окном, элементами управления, пунктами меню и акселераторами;
- создавать новые и удалять существующие обработчики сообщений;
- переходить к исходному тексту обработчика сообщений.



Рис. 10. Панель инструментов WizardBar

Ее основные компоненты:

Раскрывающийся список *Class Object IDs* (Идентификаторы объектов класса) — список идентификаторов, изображен в левой части панели *WizardBar* и отображает имя класса и идентификаторы класса, который открыт в данный момент. Первая запись в списке — имя самого класса, который позволяет переопределять виртуальные функции и сообщения Windows.

Раскрывающийся список *Messages List* (Список сообщений) — содержит виртуальные функции, сообщения Windows и процедуры, связанные с компонентом класса *CCmdTarget*, выбранным в списке Class Object IDs (Идентификаторы объектов класса).

Кнопка *Delete Button* (Кнопка удаления) — если вы выбрали переопределенную виртуальную функцию или обработчик сообщения Windows в списке Messages List (Список сообщений), то эта кнопка становится доступной и позволяет удалить прототип связанной функции в определении класса.

Кнопка *Open Header File Button* (Кнопка открытия заголовочного файла) — эта кнопка открывает заголовочный файл (.H) текущего файла .CPP.

Приложение 4. Настройки проекта

Для того чтобы определить настройки проекта, из пункта меню Build (Построитель) следует выбрать пункт Settings (Настройки). При этом на экране появится диалоговое окно (рис. 11) Project Settings (Настройки проекта).

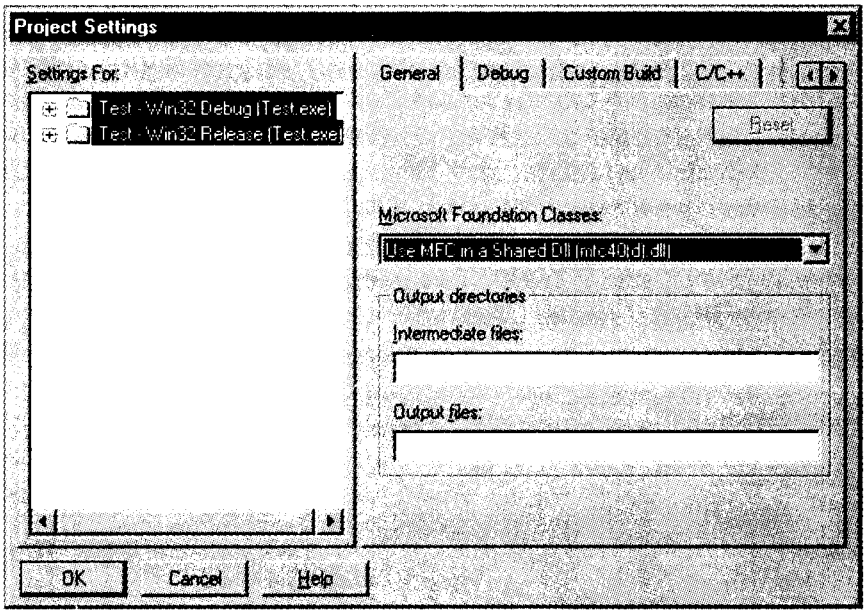


Рис. 11. Окно настроек проекта

Вы можете выбрать вкладку с названием типа опций, которые вы хотите установить. Доступными являются следующие вкладки: General (Общие), Debug (Отладка), Custom Build (Общие настройки), C/C++ (Настройки компилятора), Link (Настройки компоновщика), Resources (Ресурсы), OLE Types (Типы OLE) и Browser Info (Информация программы просмотра). Выбрав вкладки C/C++ или Link можно установить необходимые опции для компилятора и, соответственно, компоновщика.

Настройка опций компилятора

При выборе в раскрывающемся списке Category (Категория) необходимой категории опций в блоке диалога (рис. 12) будут отображаться опции, относящиеся к выбранной категории.

Определены следующие категории опций:

- General (Общие) — содержит наиболее часто используемые опции.
- Language (Язык) — определяет представление указателей C++ на члены классов, управление обработкой исключений и управление созданием скрытых виртуальных конструкторов/деструкторов в классах, базирующихся на виртуальных.
- Code Generation (Генерация кода) — определяет тип процессора, библиотеку времени выполнения, соглашения о вызовах и выравнивание элементов структур.
- Customize (Настройка) — содержит опции, которые отключают расширения языка Microsoft, допускают компоновку на уровне функций, устраняют дублирование строк, допускают минимальное перепостроение проекта и пошаговую компиляцию, отключают сообщения при запуске.
- Listing Files (Файлы листинга) — содержит опции создания файлов с информацией для просмотра (.BSC) и файлов с листингом кода.
- Optimizations (Оптимизация) — содержит опции, определяющие, как компилятор настраивает скоростные показатели работы приложения.
- Precompiled Headers (Прекомпиляция) — содержит опции, позволяющие ускорить время компиляции за счет прекомпиляции кода C или C++.
- Preprocessor (Препроцессор) — содержит опции, управляющие символами, макросами и включающие пути, используемые препроцессором C/C++.

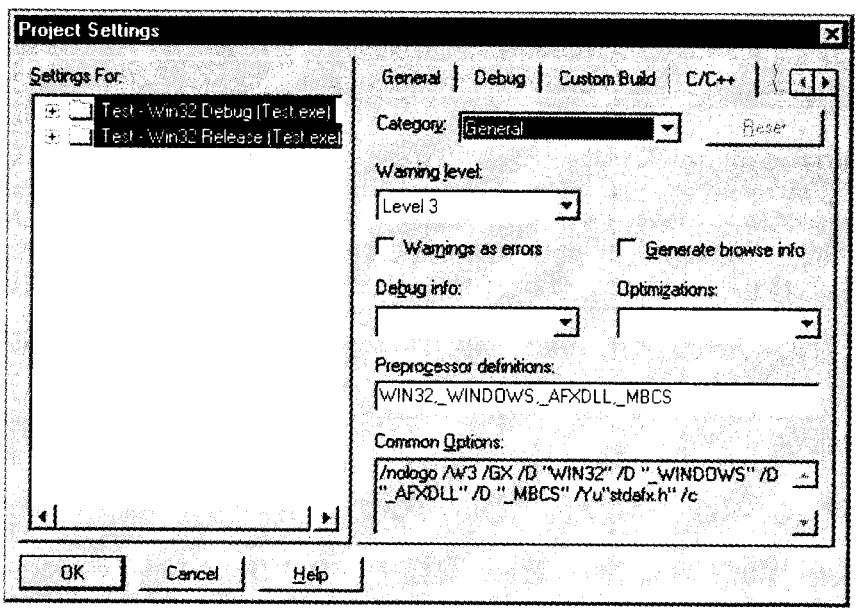


Рис. 12. Окно настроек проекта, вкладка C/C++

Ниже приведено описание категории наиболее часто используемых опций — General (Общие):

- Раскрывающийся список *Warning Level* (Уровень предупреждений) — устанавливает уровень выдачи предупреждений об ошибках (только для файлов с исходными текстами, не для объектных (.OBJ) файлов):

  None (Нет) — отключает все предупреждения. Параметр командной строки /W0 или /W;

  Level 1 (Уровень 1) — выводит только строгие предупреждения; параметр командной строки /W1;

  Level 2 (Уровень 2)— выводит менее строгие предупреждения, такие как использование функций без объявления типа возвращаемого значения, ошибка передачи возвращаемого значения для функции, тип возвращаемого значения которой не *void*, ошибки преобразования данных, которые могут привести к потере данных или точности; параметр командной строки /W2;

  Level 3 (Уровень 3) — выводит еще менее строгие предупреждения, такие как вызов функций до их объявления; параметр командной строки /W3;

  Level 4 (Уровень 4) — выводит наименее строгие предупреждения; параметр командной строки /W4.

- Флажок *Warnings as errors* (Считать предупреждения ошибками) — задает останов компиляции в случае появления предупреждений. Параметр командной строки /WX.

- Флажок *Generate browse info* (Генерировать информацию программы просмотра) — задает создание файлов .SBR с полной информацией программы просмотра. В процессе построения проекта утилита BSCMAKE использует файлы .SBR для создания файла .BSC, который можно открыть в окне просмотра. Параметр командной строки /FR.

- Раскрывающийся список *Debug info* (Отладочная информация) — определяет, какой тип информации создается для отладки, и где она хранится (в объектных файлах или в базе данных программы PDB):

  None (Нет) — без информации для отладки компиляция производится быстрее; нет параметров для командной строки;

  Line numbers only (Только номера строк) — создает объектные (.OBJ) файлы или исполняемые (.EXE) файлы, содержащие только глобальные и внешние переменные и информацию о номерах строк; используйте эту опцию только если хотите уменьшить размер файла .EXE или не собираетесь использовать вычисление значений выражений при отладке; параметр командной строки /Zd;

  C7 compatible (C7-совместимая) — производит объектные (.OBJ) и исполняемые (.EXE) файлы, содержащие номера строк и полную символьную информацию для отладчика; параметр командной строки /Z7; (имеет то же значение, что и /Zi в версии 7 Microsoft C/C++);

Program database (База данных программы) — производит базу данных программы (PDB), которая содержит символьную информацию для отладчика; параметр командной строки /Zi; если вы используете эту опцию, то размеры объектных файлов уменьшаются, так как информация для отладки содержится не в них, а в файле базы данных программы.

- Раскрывающийся список *Optimizations* (Оптимизация) — позволяет настроить показатели эффективности программы. Четыре типа опций оптимизации из пяти в списке (Default (По умолчанию), Disable (Debug) (Отладчик), Maximize Speed (Максимизировать скорость), Minimize Size (Минимизировать размер) не требуют дополнительной настройки. Если вы выберете вариант Customize (Настроить), то сможете более точно настроить параметры оптимизации программы.
- Поле *Preprocessor definitions* (Определения препроцессора) — позволяет задать один или несколько макросов для работы препроцессора.
- Поле *Common Options* (Общие установки) — отображает все выбранные установки.

Настройка опций компоновщика

При выборе в окне Project Settings (Настройки проекта) вкладки с надписью Link (Настройки компоновщика) появляется блок диалога (рис. 13), позволяющий задать параметры для 32-разрядного компоновщика (LINK.EXE).

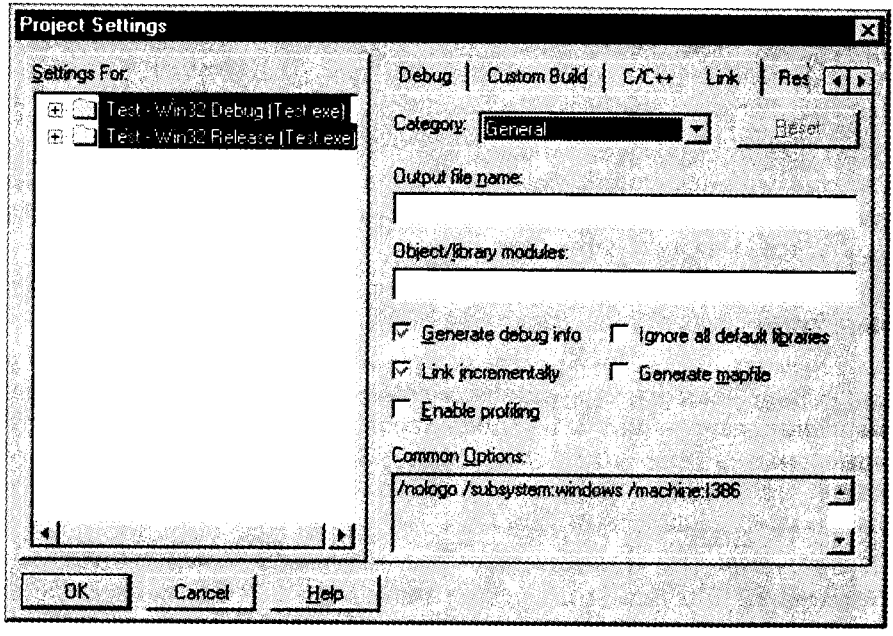


Рис. 13. Окно настроек проекта, вкладка Link

Возможна установка следующих категорий опций:

- General (Общие) — содержит наиболее часто используемые опции;
- Customize (Настроить) — содержит опции, управляющие процессом компоновки;
- Debug (Отладка) — содержит опции, управляющие процессом создания отладочной информации;
- Input (Вход) — содержит опции, управляющие параметрами компоновщика, необходимыми при использовании библиотек;
- Output (Выход) — содержит опции, управляющие компоновщиком при создании приложения Win32.

Ниже приводится описание наиболее часто используемых опций категории General (Общие):

- Поле *Output file name* (Имя выходного файла) — переопределяет заданное по умолчанию имя приложения и его местоположение. Параметр командной строки: /OUT:*имя_файла*.
- Поле *Object/library modules* (Объектные/библиотечные модули) — передает компоновщику объектные файлы или стандартную библиотеку (статическую или импортируемую). Параметр командной строки: *имя_файла*.
- Флажок *Generate debug info* (Генерировать отладочную информацию) — задает создание отладочной информации для EXE- или DLL-файлов. Параметр командной строки: /DEBUG.
- Флажок *Link incrementally* (Компоновать по шагам) — определяет пошаговую компоновку. Параметр командной строки: /INCREMENTAL:{YES|NO}.
- Флажок *Enable profiling* (Разрешить измерение временных характеристик) — создает на выходе файл, который может использоваться профайлером (утилитой для измерения временных характеристик программы). Параметр командной строки /PROFILE.
- Флажок *Ignore all default libraries* (Игнорировать все библиотеки, заданные по умолчанию) — предписывает компоновщику удалить все библиотеки, заданные по умолчанию, при разрешении им внешних ссылок. Параметр командной строки /NODEFAULTLIB.
- Флажок *Generate mapbile* (Создать MAP-файл) — задает создание специального файла, содержащего дополнительную информацию.

Приложение 5. Значения параметра функции GetSystemMetrics

SM_CMOUSEBUTTONS	число кнопок мыши, если мышь не установлена, то возвращается 0
SM_CXDORDER, SM_CYDORDER	ширина и высота рамки окна
SM_CXCURSOR, SM_CYCURSOR	ширина и высота курсора в пикселях, поддерживаемые драйвером дисплея
SM_CXDLGFRAME, SM_CYDLGFRAME	толщина в пикселях рамки окна, имеющего заголовок и фиксированный размер; SM_CXDLGFRAME – ширина вертикальной границы, а SM_CYDLGFRAME – высота горизонтальной; в Windows 95 и Windows NT 4.0 необходимо использовать SM_CXFIXEDFRAME и SM_CYFIXEDFRAME соответственно
SM_CXDOUBLECLK, SM_CYDOUBLECLK	ширина и высота в пикселях прямоугольной области, в которой курсор мыши должен находиться между двумя нажатиями на одну из кнопок для определения двойного нажатия; последнее также зависит от временного интервала
SM_CXFRAME, SM_CYFRAME	толщина рамки в пикселях окна с нефиксированным размером; SM_CXFRAME – ширина вертикальной границы, а SM_CYFRAME – высота горизонтальной; в Windows 95 и Windows NT 4.0 необходимо использовать SM_CXSIZEFRAME и SM_CYSIZEFRAME соответственно
SM_CXFULLSCREEN, SM_CYFULLSCREEN	ширина и высота рабочей области развернутого окна
SM_CXHSCROLL, SM_CYHSCROLL	высота горизонтальной полосы прокрутки и ширина ее кнопок со стрелками
SM_CXHTHUMB	ширина в пикселях бегунка горизонтальной полосы прокрутки
SM_CXICON, SM_CYICON	ширина и высота большой пиктограммы, заданная по умолчанию; обычно они равны 32x32, но могут изменяться в зависимости от установок пользователя[1]

[1] Функция *LoadIcon* в Windows 95 может загружать пиктограммы только этих размеров, если требуются другие, то следует использовать битовые массивы (bitmaps) и функцию *LoadImage*.

SM_CXICONSPACING, SM_CYICONSPACING	в Windows NT версии 3.51 и ниже ширина и высота прямоугольных областей, на которые разбивается клиентская область окна Program Manager, и в которые помещаются пиктограммы после выравнивания; в Windows 95 и Windows NT 4.0 — размеры ячеек для больших пиктограмм, используемые при выравнивании; значения всегда больше или равны тем, которые функция возвращает при задании в качестве параметров SM_CXICON и SM_CYICON соответственно
SM_CXMIN, SM_CYMIN	минимально допустимые ширина и высота окна
SM_CXMINTRACK, SM_CYMINTRACK	минимальные величины, до которых пользователь может уменьшить ширину и высоту окна; для их изменения необходимо обработать сообщение Windows WM_GETMINMAXINFO
SM_CXSCREEN, SM_CYSCREEN	ширина и высота экрана
SM_CXSIZE, SM_CYSIZE	ширина и высота кнопок заголовка окна
SM_CXVSCROLL, SM_CYVSCROLL	ширина вертикальной полосы прокрутки и высота ее кнопок со стрелками
SM_CYCAPTION	высота области заголовка окон
SM_CYKANJIWINDOW	для версий с двухбайтовой кодировкой символов — высота (в пикселях) окна Kanji в нижней части экрана
SM_CYMENU	высота полосы меню
SM_CYVTHUMB	высота в пикселях бегунка вертикальной полосы прокрутки
SM_DBCSENABLED	возвращает ненулевое значение, если установлена версия модуля USER.EXE, поддерживающая двухбайтовую кодировку символов (DBCS), иначе 0
SM_DEBUG	возвращает ненулевое значение, если установлена отладочная версия модуля USER.EXE, иначе 0
SM_MENUDROPALIGNMENT	возвращает ненулевое значение, если раскрывающиеся меню выровнены по правому краю относительно соответствующего пункта, и 0, если по левому краю
SM_MOUSEPRESENT	возвращает ненулевое значение, если мышь установлена, и 0 в противном случае
SM_PENWINDOWS	возвращает ненулевое значение, если установлена версия Windows для рукописного ввода, и 0 в противном случае
SM_SHOWSOUNDS	возвращает ненулевое значение, если пользователь требует наряду со звуковым сопровождением некоторых событий визуального отображения информации, и 0 в противном случае

SM_SWAPBUTTON	ненулевое значение, если левая и правая кнопки мыши обменялись своими функциями
SM_ARRANGE	в Windows 95 и Windows NT 4.0 возвращает флаги выравнивания свернутых окон
SM_CLEANBOOT	в Windows 95 и Windows NT 4.0 возвращает значение, определяющее режим загрузки системы: 0 — нормальная загрузка 1 — режим защиты от сбоев 2 — режим защиты от сбоев с сетевой поддержкой
SM_CMETRICS	в Windows 95 и Windows NT 4.0 возвращает число системных параметров (метрик) и флагов
SM_CXDRAG, SM_CYDRAG	в Windows 95 и Windows NT 4.0 — ширина и высота (в пикселях) прямоугольной области, до выхода курсора мыши из которой не начинается операция переноса объекта, с тем чтобы дать возможность пользователю нажать и отпустить кнопку мыши, не начиная при этом перенос объекта
SM_CXEDGE, SM_CYEDGE	в Windows 95 и Windows NT 4.0 — размеры трехмерной рамки
SM_CXFIXEDFRAME, SM_CYFIXEDFRAME	в Windows 95 и Windows NT 4.0 — толщина рамки окна, которое имеет заголовок, но не может изменить свои размеры
SM_CXMAXIMIZED, SM_CYMAXIMIZED	в Windows 95 и Windows NT 4.0 — предельно допустимые размеры развернутого окна
SM_CXMAXTRACK, SM_CYMAXTRACK	в Windows 95 и Windows NT 4.0 — максимальные величины, до которых пользователь может увеличить ширину и высоту окна; для их изменения необходимо обработать сообщение Windows WM_GETMINMAXINFO
SM_CXMENUCHECK, SM_CYMENUCHECK	в Windows 95 и Windows NT 4.0 — размеры (в пикселях) битового образа маркера выбора (check-mark) пункта меню
SM_CXMENUSIZE, SM_CYMENUSIZE	в Windows 95 и Windows NT 4.0 — размеры кнопок в полосе меню (например, в MDI-приложениях)
SM_CXMINIMIZED, SM_CYMINIMIZED	в Windows 95 и Windows NT 4.0 — предельно допустимые размеры свернутого окна
SM_CXMINSPACING SM_CYMINSPACING	в Windows 95 и Windows NT 4.0 — размеры сетки для размещения свернутых окон; каждое такое окно при выравнивании помещается в прямоугольную область с этими размерами; эти значения всегда больше или равны тем, которые функция возвращает при задании в качестве параметров SM_CXMINIMIZED и SM_CYMINIMIZED соответственно
SM_CXSIZEFRAME, SM_CYSIZEFRAME	в Windows 95 и Windows NT 4.0 — толщина рамки окна, которое может изменять свои размеры

SM_CXSMICON, SM_CYSMICON	в Windows 95 и Windows NT 4.0 — рекомендуемые размеры маленькой пиктограммы, которая обычно используется в заголовке окна
SM_CXSMSIZE, SM_CYSMSIZE	в Windows 95 и Windows NT 4.0 — размеры (в пикселях) кнопок заголовка окна половинной высоты, которое используется, например, для плавающих панелей инструментов
SM_CYSMCAPTION	в Windows 95 и Windows NT 4.0 — высота области уменьшенного заголовка
SM_MIDEASTENABLED	в Windows 95 и Windows NT 4.0 возвращает ненулевое значение, если система допускает использование арабских (Hebrew/Arabic) языков
SM_NETWORK	в Windows 95 и Windows NT 4.0 возвращает установленный самый младший бит, если сеть присутствует, остальные биты зарезервированы для будущего использования
SM_SECURE	в Windows 95 и Windows NT 4.0 возвращает ненулевое значение, если установлена защита, и 0 в противном случае
SM_SLOWMACHINE	в Windows 95 и Windows NT 4.0 возвращает ненулевое значение, если компьютер оснащен медленным процессором

Предметный указатель

A

AfxGetApp 165
AfxGetInstanceHandle 167
AfxMesageBox 172
AfxRegisterClass 175
AfxRegisterWndClass 175
AttachThreadInput 65

B

BeginPaint 51

C

caret См. каретка
CCdmUI
  SetText 295
CClientDC 270
CCmdTarget
  BeginWaitCursor 172, 191
  EndWaitCursor 172, 191
  OnCmdMsg 192
  RestoreWaitCursor 172, 191
CCmdUI 195, 277
  ContinueRouting 196
  Enable 196, 277, 290, 295
  m_nID 195
  m_nIndex 195
  m_pMenu 195
  m_pOther 195
  m_pSubMenu 195
  SetCheck 196, 267, 277
  SetRadio 196, 277
  SetText 196, 290
CColorDialog 350
  CColorDialog 365
  DoModal 365
  GetColor 365
  GetSavedCustomColors 365
  m_cc 363
  OnColorOK 365
  SetCurrentColor 365
CCommandLineInfo 170
CCommonDialog
  CCommonDialog 350
CControlBar 169, 272
  CalcFixedLayout 273
  EnableDocking 275
  GetBarStyle 273
  GetCount 273
  GetDockingFrame 275
  IsFloating 275
  m_bAutoDelete 273
  SetBarStyle 273
CCreateContex 225
CDataExchange
  Fail 345
  m_bSaveAndValidate 345
  m_pDlgWnd 345
  PrepareCtrl 345
  PrepareEditCtrl 345
CDC
  DrawText 269
  TextOut 270
CDialog 169, 206
  CDialog 316
  Create 320
  CreateIndirect 320
  DoModal 316
  EndDialog 321
  GetDefID 323
  GotoDlgCtrl 323
  InitModalIndirect 320
  MapDialogRect 322
  NextDlgCtrl 323
  OnCancel 322
  OnInitDialog 321
  OnOK 322
  OnSetFont 322
  PrevDlgCtrl 323
  SetDefID 323
CDialogBar 169, 272, 296
  CDialogBar 296
  Create 296
CFileDialog 350
  CFileDialog 357
  DoModal 357
  GetFileExt 357
  GetFileName 357
  GetFileTitle 357
  GetNextPathName 358
  GetPathName 357
  GetReadOnlyPref 358
  GetStartPosition 358
  m_ofn 351
  OnFileNameChange 358
  OnFileNameOK 358
  OnFolderChange 358
  OnInitDone 358
  OnLBSelChangedNotify 358
  OnShareViolation 358
  OnTypeChange 358
CFindReplace
  FindNext 378
CFindReplaceDialog 350
  CFindReplaceDialog 378
  Create 378
  GetFindString 379
  GetNotifier 379
  GetReplaceString 379
  IsTerminating 379
  m_fr 376
  MatchCase 379
  MatchWholeWord 379
  ReplaceAll 379
  ReplaceCurrent 379
  SearchDown 379
CFontDialog 350
  CFontDialog 373
  DoModal 373
  GetColor 373
  GetCurrentFont 373
  GetFaceName 373
  GetSize 373
  GetStyleName 373
  GetWeight 373
  IsItalic 373

IsStrikeOut 373
IsUnderLine 373
m_cf 369
CFormView 169
CFrameWnd 206
Create 210, 226
DockControlBar 275
EnableDocking 275
FloatControlBar 276
GetDockState 211
LoadAccelTable 210
LoadBarState 210, 275
LoadFrame 210, 225
m_bAutoMenuEnable 210
RecalcLayout 276
SaveBarState 210, 275
SetDockState 211
ShowControlBar 211, 275
CHOOSECOLOR 363
CHOOSEFONT 369
CloseHandle 110
CMDIChildWnd 206
Create 245
GetMDIFrame 245
m_hMenuShared 245
MDIActivate 245
MDIDestroy 245
MDIMaximize 245
MDIRestore 245
CMDIFrameWnd 206
CreateClient 244
GetWindowMenuPopup 244
MDIActivate 243
MDICascade 244
MDIGetActive 243
MDIIconArrange 243
MDIMaximize 243
MDINext 243
MDIRestore 243
MDISetMenu 243
MDITile 244
CObject 116
COleResizerBar 272
CommandLineToArgvW 65
CPageSetupDialog 350
CPageSetupDialog 393
CreatePrinterDC 393
DoModal 393
GetDeviceName 394
GetDevMode 394
GetDriverName 394
GetMargins 394
GetPaperSize 394
GetPortName 394
OnDrawPage 394
PreDrawPage 395
CPaintDC 268
CPoint 270
CPrintDialog 350, 388
CPrintDialog 388
CreatePrinterDC 388
DoModal 388
GetCopies 389
GetDefaults 388
GetDeviceName 388
GetDevMode 388
GetDriverName 388
GetFromPage 389
GetPortName 388
GetPrinterDC 388
GetToPage 389
m_pd 380, 389
PrintAll 389
PrintCollate 389
PrintRange 389
PrintSelection 389
CPropertyPage 169
CPropertySheet 169
CreateEvent 78
CreateFile 110
CreateMutex 76
CreateProcess 63
CreateRemoteThread 65
CreateSemaphore 79
CREATESTRUCT 227
CreateThread 64
CRect 269
CRITICAL_SECTION 72
CRuntimeClass 116
CSize 270
CStatusBar 272
Create 290
CStatusBar 290
GetPaneInfo 291
GetPaneStyle 291
GetPaneText 292
GetStatusBarCtrl 292
SetIndicators 291
SetPaneInfo 291
SetPaneStyle 291
SetPaneText 292
CStatusBarCtrl 292
DrawItem 294
GetBorders 294
GetParts 293
GetRect 294
GetText 292
GetTextLenght 292
SetMinHeight 294
SetParts 293
SetSimple 294
SetText 293
CTollBarCtrl
DeleteButton 285
CTollBatCtrl
Customize 284
CToolBar 169, 272, 277
CommandToIndex 280
Create 278
CToolBar 277
GetButtonInfo 280
GetButtonStyle 280
GetButtonText 281
GetItemID 280
GetItemRect 280
GetToolBarCtrl 281
LoadBitmap 279
LoadToolBar 278
SetBitmap 279
SetButtonInfo 281
SetButtons 279
SetButtonStyle 280
SetButtonText 281
SetHeight 279
SetSizes 279
CToolBarCtrl 281
AddBitmap 284
AddButtons 285
AddString 284
AddStrings 284
AutoSize 285
GetBitmapFlags 284
GetButton 283
GetButtonCount 283
GetRows 284
InsertButton 285
RestoreState 285
SaveState 285
SetOwner 283
SetRows 284
CView 206
CWaitCursor 172
CWinApp 166, 168
DoMessageBox 172
DoWaitCursor 172
Enable3dControls 168
Enable3dControlsStatic 168
EnableShellOpen 169
ExitInstance 172
GetProfileBinary 171
GetProfileInt 170
GetProfileString 170
HideApplication 172
InitInstance 171
LoadCursor 169
LoadIcon 170
LoadOEMCursor 169
LoadOEMIcon 170
LoadStandardCursor 169
LoadStandardIcon 170
LoadStdProfileSettings 168

m_hInstance 167
m_hPrevInstance 167
m_lpCmdLine 167
m_nCmdShow 167
m_pszAppName 167
m_pszExeName 167
m_pszHelpFilePath 167
m_pszProfileName 167
m_pszRegistryKey 167
OnIdle 172
ParseCommandLine 170
ProcessShellCommand 170
RegisterShellFileTypes 169
Run 172
SelectPrinter 393
SetDialogBkColor 168
SetRegistryKey 169
WriteProfileBinary 171
WriteProfileInt 170
WriteProfileString 170
CWinThread 166
m_pMainWnd 167, 210
CWinTread
PumpMessage 183
CWnd 184, 206
m_hWnd 207
ArrangeIconicWindows 208
Attach 207
BeginPaint 208, 268
BringWindowToTop 208, 261
CenterWindow 208
Create 207
CreateEx 207, 226
Default 196
Detach 207
DoDataExchange 344
EnableWindow 207, 240
EndPaint 208, 268
FindWindow 208, 260
FromHandle 207
FromHandlePermanent 207
GetActiveWnd 207
GetCapture 207
GetClientRect 208, 269
GetCurrentMessage 196
GetDC 208
GetDlgItem 323
GetExStyle 207, 236
GetFocus 207
GetFont 208
GetForegroundWindow 207, 261
GetIcon 208
GetParent 245
GetSafeWnd 207, 235
GetStyle 207, 236
GetWindowDC 208
GetWindowPlacement 208
GetWindowRect 208
GetWindowText 208
GetWindowTextLenght 208
Invalidate 208, 267
IsIconic 208, 260
IsWindowEnable 207, 240
IsZoomed 208
ModifyStyle 208, 236
ModifyStyleEx 236
MoveWindow 208, 233
PostMessage 197
PreCreateWindow 207
PreSubclassWindow 207
PreTranslateMessage 196
Print 208
PrintClient 208
ReleaseDC 208
SendMessage 197
SendNotifyMessage 197
SetActiveWnd 207
SetCapture 207
SetFocus 207
SetFont 208
SetForegroundWindow 207, 261
SetIcon 208
SetWindowPlacement 208
SetWindowPos 208, 228
SetWindowText 208
ShowWindow 208
SubclassWindow 207
UpdateData 343
UpdateWindow 208

D

DeleteCriticalSection 72
desktop window См. окно системы главное
DEVMODE 385
DEVNAMES 387
DispatchMessage 90, 172, 196
DLGITEMTEMPLATE 318
DLGTEMPLATE 317

E

EndPaint 51
EnterCriticalSection 72
EnumFontFamilies 56
EnumObject 56
ExcludeUpdateRgn 54
ExitProcess 64
ExitThread 64

F

FIFO 62
FINDREPLACE 376
FindWindow 40
FreeEnvironmentStrings 65

G

GDI См. интерфейс устройства графический
GetClassLong 234
GetCommandLine 65
GetCurrentObject 56
GetCurrentProcess 65
GetCurrentProcessId 65
GetCurrentThread 65
GetCurrentThreadId 65
GetEnvironmentVariable 65
GetExitCodeProcess 65
GetExitCodeThread 65
GetFileSize 110
GetMessage 62, 90, 183, 197
GetObject 56
GetOpenFileName 110
GetProcessVersion 65
GetSaveFileName 110
GetStartupInfo 65
GetSystemMetrics 238
GetUpdateRect 52
GetUpdateRgn 52
GetWindowLong 234

I

InitializeCriticalSection 72
InterlockedDecrement 73
InterlockedExchange 73
InterlockedIncrement 73
InvalidateRect 52
InvalidateRgn 52
IsIconic 41

L

LeaveCriticalSection 72

M

MDI См. интерфейс многодокументный
MINMAXINFO 238
MSG 90
MsgWaitForMultipleObjects 74, 75

O

OpenEvent 78
OPENFILENAME 352
OpenGL 43
OpenMutex 77
OpenProcess 65
OpenSemaphore 79

P

PAGESETUPDLG 390
PAINTSTRUCT 51
PeekMessage 60, 183, 197
PostMessage 18
PreTranslateMessage 172
PRINTDLG 380
process См. процесс
PulseEvent 78

R

ReadFile 110
RECT 269
RedrawWindow 55
RegisterWindowMessage 191, 197
ReleaseMutex 77
ReleaseSemaphore 80
ResetEvent 78
ResumeThread 65
RGB 168

S

SDI См. интерфейс однодокументный
SelectObject 57
SendMessage 18
SetClassLong 233
SetEndOfFile 110
SetEvent 78
SetForegroundWindow 41
SetPriorityClass 66
SetThreadPriority 67
SetWindowLong 234
ShowWindow 41
SuspendThread 65

T

TerminateProcess 64
TerminateThread 64
thread См. поток
TranslateMessage 90, 172, 196

U

UpdateWindow 55

V

ValidateRect 52
ValidateRgn 52

W

WaitCommEvent 75
WaitForMultipleObjects 74
WaitForMultipleObjectsEx 74
WaitForSingleObject 74
WaitForSingleObjectEx 74
WaitForSingleObjectsEx 74
WinExec 65
WinMain 39, 165
WM_CHAR 90
WM_ERASEBKGND 51
WM_NCPAINT 51
WM_PAINT 50
WM_QUIT 90
WM_TIMER 60
WNDCLASS 175
WNDCLASSEX 175
WriteFile 110

* * *

А

Активное ожидание 62

Б

Базовый уровень приоритета 69

В

Взаимоисключение 76

Д

Десинхронизация ввода 17

И

Интерфейс
  многодокументный 18
  однодокументный 18
  устройства графический 28

К

Каретка 51
Класс процессов
  High 67
  Idle 66
  Normal 66
  Realtime 67

М

Модуль 61

О

Окно
  системы главное 55

П

Перегрузка 63
Поток 61
Процесс 61

С

Семафор 79
Событие 77
Состояние
  блокировки 61
  выполнения 61
  готовности 61
Структура
  NMHDR 287
  TBNOTIFY 288

Т

Тупик 74

Содержание

Введение **2**

ЧАСТЬ I. ОБЗОР WINDOWS 95 И WINDOWS NT **7**

Глава 1. Основные черты **9**

Требования к операционной системе 90-х 9
Различные версии Windows 10
Windows 95 12
Windows NT 13
Интерфейс прикладного программирования 15
Система, управляемая сообщениями 16
SDI и MDI 18

Глава 2. Пользовательский интерфейс **19**

Окна, элементы управления, модальные и немодальные блоки диалога 19
Ресурсы приложения 25

Глава 3. Управление графическим выводом **27**

Идеология графического вывода 27
Графические объекты 27
Аппаратно-независимый графический вывод 28
Контексты устройств 28
Графические режимы 29
Типы контекстов устройств 29
Экран 30
Принтер 31
Объект в памяти 31
Информационный контекст 32
Работа со шрифтами. Шрифты True Type 32

ЧАСТЬ II. ПРОГРАММИРОВАНИЕ ПОД MS WINDOWS **35**

Глава 4. Интерфейс прикладного программирования Win32 API **37**

Отличие 16- и 32-разрядных платформ 37
Функция WinMain 39

Новые возможности для создания программ ... 41
APIShell — пример приложения ... 43
Графический вывод ... 50
Сообщение WM_PAINT ... 51
Доступ к графическим объектам, их замена и удаление ... 56

Глава 5. Вытесняющая многозадачность ... 59

Процессы и потоки ... 63
Классы процессов и приоритеты потоков ... 66
Класс Idle ... 66
Класс Normal ... 66
Класс High ... 67
Класс Realtime ... 67
Относительные приоритеты потоков ... 67
Взаимодействие программ ... 70
Синхронизация потоков ... 71
Критические секции ... 71
Защищенный доступ к переменным ... 73
Ожидающие функции ... 74
Объекты синхронизации ... 75
APISync — пример приложения ... 80

Глава 6. Структура приложения Windows ... 88

Стандартный цикл обработки сообщений ... 90
Оконная процедура ... 91
Основные типы оконных сообщений Windows ... 92
Аппаратные сообщения ... 93
Сообщения обслуживания окна ... 95
Сообщения об организации интерфейса пользователя ... 97
Сообщения о завершении работы приложения и системы ... 98
Частные сообщения ... 99
Информационные сообщения системных ресурсов ... 99
Сообщения об обмене данными ... 100
Внутрисистемные сообщения ... 101
APIWin — пример приложения ... 101

ЧАСТЬ III. БИБЛИОТЕКА КЛАССОВ MICROSOFT FOUNDATION CLASSES (MFC) ... 111

Глава 7. Обзор ... 113

Иерархия классов MFC ... 114
CObject — основа всех классов ... 116
Сериализация ... 120
Классы, определяющие архитектуру приложения ... 126
Классы приложения и поддержки потоков ... 127
Классы маршрутизации команд ... 128
Классы документов ... 128
Классы шаблонов документов ... 129
Окна, блоки диалога и элементы управления ... 129
Классы фреймов окон ... 130

Классы представлений 130
Классы блоков диалога 132
Классы элементов управления 133
Классы панелей элементов управления 135
Классы для вывода на экран и печать 136
Классы контекста устройств 136
Классы графических объектов 136
Классы простых типов данных 137
Массивы, списки и ассоциативные списки 138
Классы шаблонов массивов, списков и ассоциативных списков 138
Готовые к использованию классы массивов 139
Готовые к использованию классы списков 139
Готовые к использованию классы ассоциативных списков 139
Классы для работы с файлами и базами данных 139
Классы файлов ввода/вывода 140
Классы для работы с DAO 140
Классы для работы с ODBC 141
Классы для работы в Internet и сетях 142
Классы ISAPI 142
Классы сокетов Windows 143
Классы для работы с OLE 143
Контейнерные классы OLE 143
Серверные классы OLE 144
Классы OLE для передачи данных и drag-and-drop 144
Классы автоматизации OLE 145
Классы, связанные с OLE 145
Классы для отладки и обработки исключений 146
Классы поддержки отладки 146
Классы обработки исключений 146

Глава 8. Макросы, глобальные функции и переменные 148

Типы данных 148
Получение информации о приложении 149
Модель объекта времени выполнения 150
Диагностика объектов 151
Основные макросы 151
Основные глобальные переменные 152
Форматирование строк и окно сообщения 152

ЧАСТЬ IV. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ КЛАССОВ MFC 157

Глава 9. Основные составляющие приложения на базе библиотеки классов MFC 159

Соглашения об именах MFC 163
Включаемые файлы 163
Функция WinMain 164
Класс CWinApp 166
Минимальная программа для Windows 173

Регистрация класса окна 175
Создание окна 180

Глава 10. Обработка сообщений в библиотеке классов MFC 182

Цикл обработки сообщений MFC 182
Категории сообщений 184
Карта сообщений 185
Компоненты карты сообщений 187
Класс CCmdTarget 191
Стандартный маршрут команды 192
Команды обновления и класс CCmdUI 194
Функции для работы с сообщениями 196

ЧАСТЬ V. СОЗДАНИЕ SDI И MDI ПРИЛОЖЕНИЙ НА БАЗЕ БИБЛИОТЕКИ MFC 199

Глава 11. Классы окон библиотеки MFC 201

Окна, определенные в системе Windows 202
Класс CWnd 207
Класс CFrameWnd 209
Создание окна и библиотека MFC 211
Создание главного окна SDI-приложения 224
Создание дочерних окон 230
Ограничение размеров окна 237
Функция GetSystemMetrics 238
Ограничение доступа к окну 240

Глава 12. Создание многодокументных приложений 241

Класс CMDIFrameWnd 242
Класс CMDIChildWnd 244
MDI — пример приложения 246
Обработка команд меню 262
UNICODE 264
Обработка сообщения WM_PAINT 268

Глава 13. Панели элементов управления 272

Класс CControlBar 272
Класс CToolBar 277
Класс CToolBarCtrl 281
Класс CStatusBar 289
Класс CStatusBarCtrl 292
Класс CDialogBar 295
ControlBars — пример приложения 297

ЧАСТЬ VI. БЛОКИ ДИАЛОГА 311

Глава 14. Модальные и немодальные блоки диалога 313

Класс CDialog 316
Template — пример приложения 328

Блок диалога в качестве главного окна приложения 341
Обмен данными с блоком диалога 342
Создание блока диалога на основе шаблона в памяти 346

Глава 15. Стандартные блоки диалога 350

Класс CFileDialog 351
Класс CColorDialog 363
Класс CFontDialog 369
Класс CFindReplaceDialog 375
Класс CPrintDialog 379
Класс CPageSetupDialog 389

Заключение 396

ПРИЛОЖЕНИЯ 397

Приложение 1. Иерархия классов MFC 4.1 399

Приложение 2. Список функций класса CWnd 402

Атрибуты 402
Конструкторы и создание окон 402
Доступ к дереву окон 404
Общие функции работы с сообщениями 405
Обработка сообщений модального диалога 405
Функции работы с текстом окна 405
Функции меню (кроме дочерних окон) 406
Функции изменения размеров и расположения окна 406
Функции преобразования координат 407
Функции обновления и рисования 407
Функции использования таймера 409
Функции для работы со всплывающими подсказками 409
Функции состояния окна 409
Функции для работы с блоками диалога 410
Функции прокрутки 411
Функции доступа к окну 412
Функции обслуживания 413
Функции работы с буфером обмена (Clipboard) 413
Функции каретки 414
Функция взаимодействия 414
Функции работы с пиктограммами 414
Функции контекстно-зависимой справки 415
Функция поддержки диалога 415
Функции расположения и другие 415
Функции управления OLE 415
Функции обработки сообщений управления окнами 416
Функции обработки сообщений нерабочей области окна 418
Функции обработки системных сообщений 419
Функции обработки входных сообщений 421
Функции обработки инициализирующих сообщений меню 422

Функции обработки сообщений буфера обмена 422
Функции обработки управляющих сообщений 423
Функция обработки сообщений MDI 424
Функции обработки сообщений цикла меню 424
Функции обработки основных сообщений 424
Переопределяемые и другие вспомогательные функции 425

Приложение 3. Типы проектов и мастера 427

Типы проектов Visual C++ 427
Мастера (Wizards) 428
AppWizard 429
ClassWizard 435
WizardBar 438

Приложение 4. Настройки проекта 439

Настройка опций компилятора 439
Настройка опций компоновщика 442

Приложение 5. Значения параметра функции GetSystemMetrics 444

Предметный указатель 448