МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Белгородский государственный технологический университет
им. В.Г. Шухова

А. В. Глухоедов

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Лабораторный практикум

Утверждено ученым советом университета в качестве учебного пособия для студентов очной и заочной формы обучения направлений 09.03.02 – Информационные системы и технологии, 09.03.03 – Прикладная информатика

Белгород
2017

УДК004.45 (07)
ББК 32.973я7
Г55

Рецензенты

старший преподаватель, Белгородского государственного технологического университета им. В.Г. Шухова *О.В. Веретенников*

старший преподаватель, Белгородского государственного национального исследовательского университета (НИУ «БелГУ») *А.Г. Смышляев*

Глухоедов, А. В.

Г55 Операционные системы: лабораторный практикум: учебное пособие / А. В. Глухоедов. – Белгород: Изд-во БГТУ, 2017. – 386 с.

ISBN 978-5-361-00393-8

Лабораторный практикум составлен в соответствии с рабочей программой дисциплины «Операционные системы», содержит теоретические материалы и задания к выполнению лабораторных работ и предназначен для приобретения студентами практических навыков программирования для Windows на языке C/C++ с применением Win32 API.

Лабораторный практикум предназначен для студентов очной и заочной формы обучения направлений 09.03.02 – Информационные системы и технологии, 09.03.03 – Прикладная информатика.

Данное издание публикуется в авторской редакции.

УДК 004.45(07)
ББК 32.973я7

ISBN 978-5-361-00393-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Microsoft Windows является слишком большой и сложной операционной системой. Ее изучение лучше всего начать с самого низкого уровня, чтобы получить представление о базовых механизмах операционной системы. Разобравшись в основах проще изучать механизмы высокого уровня, построенные именно на этом базисе. Поэтому особое внимание в данном учебном пособии уделяется основополагающим принципам разработки приложений для операционной системы Windows.

При разработке приложений для Windows приходится пользоваться огромным объемом документации, который изложен в справочной системе MSDN (Microsoft Developer Network), содержащей очень мало примеров и представленной только на английском языке. Часто приходится тратить много времени, чтобы понять, как работает та или иная функция. Положение ухудшается, если разработчик приложений обладает весьма поверхностными сведениями об устройстве операционной системы Windows. Данное учебное пособие изобилует примерами, которые могут оказаться весьма полезными в разработке приложений для Windows.

Без базовых знаний об устройстве Windows немыслимо рассмотрение такой важной технологии, как COM (Component Object Model – компонентная модель объектов). В данном учебном пособии прямо не затрагиваются вопросы, относящиеся к COM. Но в COM используются библиотеки DLL, процессы, потоки, межпроцессное взаимодействие и многое другое. Если знать, как устроены и работают эти фундаментальные механизмы операционной системы, то для освоения COM достаточно понять, как они применяются в этой технологии.

Также в данном учебном пособии не рассматривается технология CLR (Common Language Runtime – общеязыковая исполняющая среда) платформы Microsoft .NET Framework, которая является основой приложений с управляемым кодом. Однако CLR реализована в виде объекта COM внутри библиотеки DLL, которая загружается в процесс и используется потоки и межпроцессное взаимодействие. Так что полученные из данного учебного пособия знания о базовых механизмах операционной системы помогут и при написании управляемого кода.

В рамках одного учебного пособия невозможно осветить все аспекты разработки приложений для Windows. Однако хочется надеяться, что после получения базовых знаний студентам будет гораздо легче разобраться с другими разделами программирования для Windows.

ЛАБОРАТОРНАЯ РАБОТА № 1
ВВЕДЕНИЕ В WINDOWS API

Цель работы

Знакомство с понятием интерфейс прикладного программирования и получение практических навыков программирования для Windows на языке C/C++ с применением Win32 API.

Основные понятия

Для создания приложений (программ) работающих под управлением операционной системы Microsoft Windows используются разнообразные системные функции, которые предоставляет *интерфейс прикладного программирования* (Application Program Interface, API). Windows API включает в себя совокупность функций, принадлежащих ядру или службам Windows, а также соглашения об использовании этих функций.

Знание Windows API существенно облегчает изучение технологий программирования на более высоком уровне, например с использованием библиотек классов, таких как Borland Visual Component Library (VCL), Borland Object Windows Library (OWL), Microsoft Windows Template Library (WTL), Microsoft Foundation Classes (MFC) или Microsoft Windows Forms. Дело в том, что все эти библиотеки, с точки архитектуры, представляют собой надстройку над Windows API, так как их реализация основана на вызове функций Windows API.

Современные версии операционной системы Windows способны обеспечить поддержку API для 32- и 64-разрядных платформ, соответственно Win32 и Win64. Основными различиями между Win32 и Win64 являются размер указателей, соответственно 32 и 64 бита, и объем доступного виртуального адресного пространства – 4 Гбайта и 16 Эбайт. В данном курсе будет изучаться, только Win32 API.

Основные сведения о функциях Win32 API

Win32 API поддерживает свыше двух тысяч функций, которые можно использовать в своих приложениях. Все функции Win32 API реализованы в нескольких библиотеках DLL, а их прототипы объявлены в различных заголовочных файлах, на которые, как правило, ссылается Windows.h – главный заголовочный файл в Win32 API.

Вызов функций Win32 API в программе осуществляется аналогично вызову библиотечных функций C/C++. Основное отличие заключа-

ется в том, что код библиотечных функций C/C++ компоновщик связывает с кодом программы на этапе компоновки, в то время как для функций Win32 API это связывание откладывается. Связывание кода программы и кода функций Win32 API осуществляется только на этапе выполнения приложения, после того, как будет загружена необходимая библиотека DLL.

Основная часть функций Win32 API содержится в следующих библиотеках DLL:

- kernel32.dll – содержит низкоуровневые функции ввода/вывода, синхронизации, управления памятью, процессами и потоками;
- user32.dll – здесь находятся функции поддержки пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений;
- gdi32.dll – содержит функции, обеспечивающие графический вывод на различные устройства отображения;
- advapi32.dll – содержит функции, связанные с защитой объектов и работой с системным реестром;
- comdlg32.dll – содержит функции, связанные с использованием диалоговых окон общего назначения;
- shell32.dll – здесь в основном находятся функции для работы с проводником Windows.

Исчерпывающую документацию по функциям Win32 API можно получить из Platform Software Development Kit (Platform SDK), которая доступна по адресу: http://msdn.microsoft.com/.

Типы данных Win32 API

Разработчики Win32 API определяют собственные типы данных, чтобы в некоторой степени изолировать себя от языка C/C++. На самом деле все они определены посредством директив `typedef` или `#define` в заголовочных файлах Win32 API. В табл. 1.1 приведены некоторые наиболее часто встречающиеся типы данных.

Таблица 1.1. Некоторые типы данных в Win32 API

Тип данных	Соответствующий тип данных C/C++	Описание
`BOOL` `BOOLEAN`	`int` `unsigned char`	Булевский тип (принимает значения `TRUE` или `FALSE`)
`BYTE`	`unsigned char`	Целое 8-разрядное число без знака
`CHAR`	`char`	ANSI-символ

Продолжение табл. 1.1.

Тип данных	Соответствующий тип данных C/C++	Описание
DOUBLE	double	Число с плавающей точкой (десять значащих цифр)
DWORD ULONG	unsigned long	Целое 32-разрядное число без знака
DWORD64 ULONG64 UINT64	unsigned __int64	Целое 64-разрядное число без знака
FLOAT	float	Число с плавающей точкой (шесть значащих цифр)
INT	int	Целое 32-разрядное число со знаком
LONG	long	Целое 32-разрядное число со знаком
LONG64 INT64	__int64	Целое 64-разрядное число со знаком
LPARAM	long	Тип, используемый для описания параметра оконной процедуры
LPBOOL PBOOL	int*	Указатель на булевский тип
LPBYTE PBYTE	unsgined char*	Указатель на целое 8-разрядное число без знака
LPCSTR PCSTR	const char*	ANSI-строка (константа)
LPCVOID	const void*	Указатель на пустой тип (константа)
LPCWSTR PCWSTR	const wchar_t*	Unicode-строка (константа)
LPDWORD PDWORD	unsgined long*	Указатель на целое 32-разрядное число без знака
LPINT PINT	int*	Указатель на целое 32-разрядное число со знаком
LPLONG PLONG	long*	Указатель на целое 32-разрядное число со знаком
LPSHORT PSHORT	short*	Указатель на целое 16-разрядное число со знаком
LPSTR PSTR	char*	ANSI-строка
LPUINT PUINT	unsgined int*	Указатель на целое 32-разрядное число без знака

Окончание табл. 1.1.

Тип данных	Соответствующий тип данных C/C++	Описание
LPVOID PVOID	void*	Указатель на пустой тип
LPWORD PWORD	unsgined short*	Указатель на целое 16-разрядное число без знака
LPWSTR PWSTR	wchar_t*	Unicode-строка
LRESULT	long	Значение, возвращаемое оконной процедурой
PFLOAT	float*	Указатель на число с плавающей точкой
SHORT	short	Целое 16-разрядное число со знаком
UINT	unsigned int	Целое 32-разрядное число без знака
VOID	void	Пустой тип
WCHAR	wchar_t	Unicode-символ
WORD USHORT	unsigned short	Целое 16-разрядное число без знака
WPARAM	unsigned int	Тип, используемый для описания параметра оконной процедуры

Обратите внимание на то, что некоторые имена типов указателей начинаются с префикса LP (long pointer – длинный указатель). На самом деле в Win32 API нет длинных указателей, это обычные указатели, но из-за соображений обратной совместимости разных версий Windows, такие имена типов указателей перекочевали из Win16 API.

Дело в том, что раньше в модели памяти 16-разрядных операционных системах (таких, как DOS и Windows 3.1) существовало два типа указателей: *ближний* (near), который являлся быстрым, но был ограничен областью указывания в 64 Кбайта памяти; *дальний* (far) или длинный указатель мог ссылаться на 1 Мбайт памяти, но был более медленным. В 32-разрядных версиях Windows, необходимость в дальних указателях отпала, и используются только ближние указатели, так как 32-разрядные указатели способны ссылаться на 4 Гбайта памяти.

Некоторые функции Win32 API позволяют создавать объекты, которые представляют различные системные ресурсы. Каждый созданный объект идентифицируется своим *дескриптором*, который возвращает функция, создающая этот объект. Дескриптор еще могут называть *описателем* (handle). В Windows не допускается прямой доступ к данным объектов или системным ресурсам, которые представляют объекты. Поэтому для работы с ними используются функции Win32 API, которым в качестве параметра передается дескриптор объекта.

Кроме того, для большей безопасности в Windows сделано так, чтобы значения дескрипторов для некоторых объектов зависели от конкретного приложения. Поэтому если передать значение такого дескриптора другому приложению, любой вызов функции из этого приложения с полученным значением, закончится ошибкой.

В табл. 1.2 приведены типы некоторых дескрипторов, с которыми работают различные функции Win32 API. Полный перечень дескрипторов см. в документации Platform SDK.

Таблица 1.2. Дескрипторы в Win32 API

Дескриптор	**Описание**
HACCEL	Дескриптор таблицы быстрых клавиш
HANDLE	Дескриптор какого-либо объекта
HBRUSH	Дескриптор кисти
HCURSOR	Дескриптор курсора
HFONT	Дескриптор шрифта
HGLOBAL	Дескриптор глобального блока памяти
HICON	Дескриптор пиктограммы
HINSTANCE	Дескриптор экземпляра приложения
HKEY	Дескриптор ключа системного реестра
HLOCAL	Дескриптор локального блока памяти
HMENU	Дескриптор меню
HMODULE	Дескриптор модуля
HMONITOR	Дескриптор монитора
HRSRC	Дескриптор ресурса
HWND	Дескриптор окна

Также Win32 API предоставляет множество структур данных, которые используются при вызове различных функций, а также при отправке и обработке оконных сообщений. Информацию обо всех структурах Win32 API можно найти в документации Platform SDK. Некоторые из них будут рассмотрены в процессе изучения курса.

Обработка ошибок

Если по какой-либо причине функция Win32 API не может выполнить свою работу, она возвращает значение, свидетельствующее о том, что произошла ошибка. В табл. 1.3 показаны типы данных возвращаемых значений для большинства функций Win32 API.

Таблица 1.3. Типы значений, возвращаемые функциями Win32 API

Тип данных	Значения, свидетельствующие об ошибке
VOID	Функция почти всегда выполняется успешно. Таких функций в Win32 API очень мало
BOOL или BOOLEAN	Если вызов функции завершается неудачно, возвращается значение FALSE; в остальных случаях возвращаемое значение отлично от FALSE (не следует проверять его на соответствие TRUE, лучше проверять его на соответствие FALSE)
HANDLE	Если вызов функции завершается успешно, то возвращается дескриптор объекта, которым можно манипулировать; в случае ошибки – NULL или INVALID_HANDLE_VALUE (все зависит от конкретной функции). В документации Platform SDK для каждой функции четко указывается, что именно она возвращает при ошибке – NULL или INVALID_HANDLE_VALUE
PVOID	Если вызов функции завершается успешно, то возвращается адрес блока данных в памяти; в случае ошибки – NULL
LONG или DWORD	Если вызов функции завершается неудачно, то обычно возвращается 0 или -1 (все зависит от конкретной функции). В документации Platform SDK для каждой из таких функций указывается, каким именно значением она уведомляет об ошибке

Когда функция вернет значение, свидетельствующее о возникновении ошибки. Узнать какая именно это ошибка можно, вызвав функцию GetLastError, которая вернет код последней ошибки в текущем потоке:

```
DWORD GetLastError();
```

За каждой ошибкой закреплен свой уникальный код. Список кодов ошибок содержится в заголовочном файле WinError.h.

В случае если последний вызов функции Win32 API был завершен без ошибки, то GetLastError вернет код ERROR_SUCCESS, сообщающий о том, что функция была успешно завершена. Однако иногда при успешном выполнении некоторых функций вызов GetLastError может вернуть код отличный от ERROR_SUCCESS. Дело в том, что некоторые функции Win32 API могут завершаться успешно, но по разным причинам. Например, попытка создать объект с определенным именем может быть успешна либо потому, что объект действительно был создан, либо потому, что такой объект уже есть. Для возврата этой информации используется механизм установки кода последней ошибки. Сведения о таких функциях см. в документации Platform SDK.

Функцию GetLastError нужно вызывать сразу же после завершения функции Win32 API, иначе код может быть перезаписан кодом ошибки или кодом ERROR_SUCCESS в случае успешного завершения другой функции.

В процессе работы приложения может возникнуть необходимость получить описание обнаруженной ошибки. В Win32 API для этого есть специальная функция FormatMessage, которая превращает код ошибки в описательное сообщение, представляющее собой фразу на английском языке или любом другом из множества языков.

```
DWORD FormatMessage(DWORD dwFlags, LPCVOID lpSource,
    DWORD dwMessageId, DWORD dwLanguageId, LPTSTR lpBuffer,
    DWORD nSize, va_list *Arguments);
```

В следующем примере показано, как следует использовать функцию FormatMessage:

Листинг 1.1. Пример использования функции FormatMessage

```
1   void ShowMessageLastError()
2   {
3       // получаем код последней ошибки
4       DWORD dwError = GetLastError();
5
6       // будем использовать региональные настройки по умолчанию
7       DWORD dwLangId = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL);
8       // буфер, содержащий сообщение об ошибке
9       LPTSTR lpszSysMsg = NULL;
10
11      // получаем описание по коду
12      BOOL bRet = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS | FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL, dwError, dwLangId, (LPTSTR)&lpszSysMsg, 0, NULL);
13
14      if (FALSE != bRet)
15      {
16          // выводим сообщение об ошибке
17          MessageBox(NULL, lpszSysMsg, NULL, MB_OK|MB_ICONERROR);
18          // освободим буфер, содержащий сообщение об ошибке
19          LocalFree(lpszSysMsg);
20      } // if
21  } // ShowMessageLastError
```

Подробное описание функции FormatMessage можно найти в документации Platform SDK.

Также Microsoft позволяет использовать механизм обработки ошибок в ваших собственных функциях. Вызов какой-либо из этих функций по какой-то причине может завершиться с ошибкой, и нужно сообщить об этом. С этой целью внутри функции нужно установить код последней ошибки в потоке, а затем вернуть значение, которое будет свидетельствовать о возникновении ошибки. Чтобы установить код последней ошибки в потоке нужно вызвать `SetLastError` и передать ей нужный код.

```
VOID SetLastError(DWORD dwErrCode);
```

Можно использовать коды, уже определенные в WinError.h, при условии, что они подходят под те ошибки, о которых могут сообщать функции. Если ни один из кодов в WinError.h не годиться, можно определить свой код. Он представляет собой 32-битное значение, которое разбито на поля, показанные в табл. 1.4.

Таблица 1.4. Поля кода ошибки

Биты	**Содержимое**	**Значение**
31-30	Код степени «тяжести» (severity)	`0` – успех; `1` – информация; `2` – предупреждение; `3` – ошибка
29	Кем определен код	`0` – Microsoft; `1` – пользователь
28	Зарезервирован	`0`
27-16	Код подсистемы (facility code)	Первые 256 значений зарезервированы и определяются Microsoft
15-0	Код исключения (exception code)	Определяются Microsoft или пользователем

Встроенные средства отладки Visual C++

Отладчик в Visual C++ позволяет настраивать окно **Контрольные значения** (**Watch**) так, чтобы в нем показывался код и описание последней ошибки в текущем потоке. Для этого нужно в строке этого окна ввести «`$err,hr`» или «`@err,hr`». На рис. 1.1 показан пример, в котором функция `CreateFile` вернула значение `INVALID_HANDLE_VALUE` (`0xffffffff`) типа `HANDLE`, свидетельствующее о том, что эта функция была завершена с ошибкой. При этом в окне **Контрольные значения** (**Watch**) показан код ошибки `ERROR_FILE_NOT_FOUND`, равный значению `0x00000002`, и описание «*Не удается найти указанный файл.*» («*The system cannot find the file specified.*»).

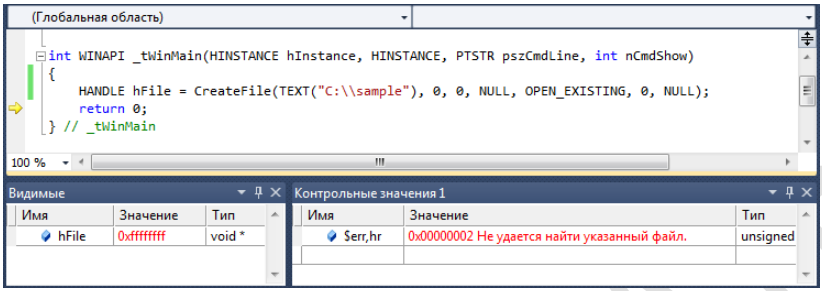

Рис. 1.1. Код последней ошибки в окне отладчика среды Visual C++

В состав Visual Studio также входит небольшая утилита «Поиск ошибки» (Error Lookup), которая позволяет получить описание ошибки по ее коду, как показано на рис. 1.2.

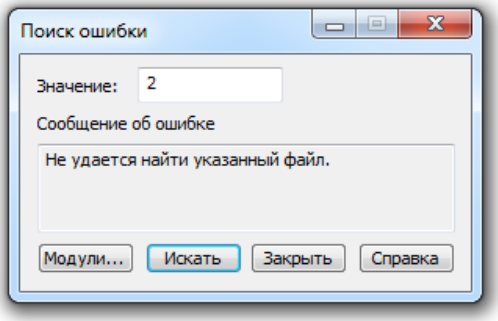

Рис. 1.2. Утилита «Поиск ошибки» (Error Lookup)

Работа с символами и строками в Windows

В Windows (начиная с Windows NT) поддерживается Unicode. Для представления Unicode-символов в Windows используется кодировка UTF-16, которая позволяет представить символы, составляющие алфавиты большинства языков мира.

Символьные и строковые типы данных для ANSI и Unicode

Компилятор Visual C++ поддерживает, как стандартные 8-битные символы ANSI, так и 16-битные символы Unicode (UTF-16). Для представления ANSI-символов используется тип `char`, а для Unicode-символов – `wchar_t`:

```
1   // 8-битный ANSI-символ
2   char c = 'A';
```

```
// Массив из 99 8-битных ANSI-символов с нулем на конце (также 8-битным)
char szBufferOfChar[100] = "Строка";

// 16-битный Unicode-символ
wchar_t w = L'A';
// Массив из 99 16-битных Unicode-символов с нулем на конце (также 16-битным)
wchar_t szBufferOfWChar[100] = L"Строка";
```

Символ «L» перед задаваемыми значениями сообщает компилятору, что следует использовать Unicode-символы.

Как уже было сказано разработчики Win32 API определяют собственные типы данных. Символы и строки не исключение. В заголовочном файле WinNT.h, определены следующие типы данных для работы с символами, указателями на символы и строками:

```
// ANSI типы
typedef char  CHAR;  // 8-битный ANSI-символ
typedef CHAR *PCHAR; // указатель на ANSI-символ

typedef CHAR       *LPSTR,  *PSTR;  // ANSI-строка
typedef CONST CHAR *LPCSTR, *PCSTR; // ANSI-строка (константа)

// Unicode типы
typedef wchar_t  WCHAR;  // 16-битный Unicode-символ
typedef WCHAR   *PWCHAR; // указатель на Unicode-символ

typedef WCHAR       *LPWSTR,  *PWSTR;  // Unicode-строка
typedef CONST WCHAR *LPCWSTR, *PCWSTR; // Unicode-строка (константа)
```

Кроме этого в заголовочном файле WinNT.h определены типы и макросы, позволяющие создавать программный код, который может компилироваться как с ANSI-, так и Unicode-символами:

```
#ifdef  UNICODE

typedef WCHAR TCHAR, *PTCHAR; // Unicode-символ и указатель на него
typedef LPWSTR  PTSTR,  LPTSTR;  // Unicode-строка
typedef LPCWSTR PCTSTR, LPCTSTR; // Unicode-строка (константа)

#define __TEXT(quote) L##quote

#else   /* UNICODE */
```

```
10
11  typedef CHAR TCHAR, *PTCHAR; // ANSI-символ и указатель на него
12  typedef LPSTR  PTSTR,  LPTSTR;  // ANSI-строка
13  typedef LPCSTR PCTSTR, LPCTSTR; // ANSI-строка (константа)
14
15  #define __TEXT(quote) quote
16
17  #endif /* !UNICODE */
18
19  #define TEXT(quote) __TEXT(quote)
```

Эти типы и макросы можно использовать следующим образом:

```
1   // 16-битный Unicode-символ, если определен UNICODE
2   // или 8-битный ANSI-символ в противном случае
3   TCHAR t = TEXT('A');
4   // Массив 16-битных Unicode-символов, если определен UNICODE
5   // либо 8-битных ANSI-символов в противном случае
6   TCHAR szBufferOfTChar[100] = TEXT("Строка");
```

Какой именно тип `wchar_t` или `char` будет использоваться, зависит от того, определена или нет константа `UNICODE` в период компиляции.

ANSI- и Unicode-функции

Начиная с Windows NT, все ключевые функции Win32 API для создания окон, вывода текста, работы с файлами и т.д. требуют Unicode-строки. Если любой из этих функций передавать при вызове ANSI-строку, эта функция сначала преобразует ANSI-строку в Unicode, и только после этого передаст ее операционной системе. Если некоторая функция должна возвращать ANSI-строку, операционная система преобразует Unicode-строку в ANSI и возвращает результат приложению. Все эти преобразования выполняются незаметно для программиста и, естественно, вызывают дополнительный расход памяти и времени.

В качестве примера, рассмотрим две версии функции `GetUserName`: одна принимает строки в ANSI, а другая – в Unicode:

```
BOOL GetUserNameA(LPSTR lpBuffer, LPDWORD lpnSize);

BOOL GetUserNameW(LPWSTR lpBuffer, LPDWORD lpnSize);
```

Функция `GetUserNameW` – это Unicode-версия. Буква «`W`» в конце имени функции – первая буква слова wide (широкий). Символы Unicode иногда называют *широкими символами* (wide characters), так как каждый такой символ занимает в памяти 2 байта (в два раза больше чем ANSI-символ). Буква «`A`» в конце имени `GetUserNameA` указывает, что данная версия функции принимает ANSI-строки.

Но обычно `GetUserNameW` или `GetUserNameA` напрямую не вызывают, а обращаются к макросу `GetUserName` определенному в WinBase.h:

```
#ifdef UNICODE
#define GetUserName GetUserNameW
#else
#define GetUserName GetUserNameA
#endif // !UNICODE
```

Какая именно версия функции `GetUserName` будет вызываться, зависит от того, определена или нет константа `UNICODE`.

Подобно Win32 API, библиотека C времени выполнения (C Run-Time Library, CRT) поддерживает два набора функция: один для манипуляции с ANSI-строками и символами, а другой – для работы с Unicode-строками и символами. Но, в отличие от функций Win32 API, здесь ANSI-функции не преобразуют внутренне полученные строки в Unicode и не вызывают затем Unicode-версии тех же функций. Аналогично Unicode-функций сами делают то, что им положено, а не вызывают ANSI-версии.

Примером может служить функция `strlen`, возвращающая длину ANSI-строк, а эквивалентом для Unicode-строк – `wcslen`. Прототипы обеих функций объявлены в заголовочном файле string.h. В код программы, которая может компилироваться как с ANSI-, так и с Unicode-строками, необходимо включить заголовочный файл tchar.h, в котором определен следующий макрос:

```
#ifdef _UNICODE
#define _tcslen  wcslen
#else
#define _tcslen  strlen
#endif // !_UNICODE
```

Если определена константа `_UNICODE`, то будет вызываться `wcslen`, в противном случае – `strlen`.

По умолчанию в новых проектах Visual C++ определены `UNICODE` и `_UNICODE`, т.е. используются Unicode-строки и символы, а также функции для работы с ними. Если необходимо явно указать, какой из наборов использовать Unicode или ANSI, выполните следующие действия:

1. Откройте решение, содержащее нужный проект (если это не было сделано заранее).
2. В меню **Проект** (**Project**) выберите **Свойства** (**Settings**).

   Откроется диалоговое окно **Страницы свойств** (**Property Pages**).

3. В открывшемся окне выберите **Свойства конфигурации** (**Configuration Properties**) → **Общие** (**General**).
4. Отредактируйте значение параметра **Набор символов** (**Character Set**), как показано на рис. 1.3, и нажмите кнопку **OK**.

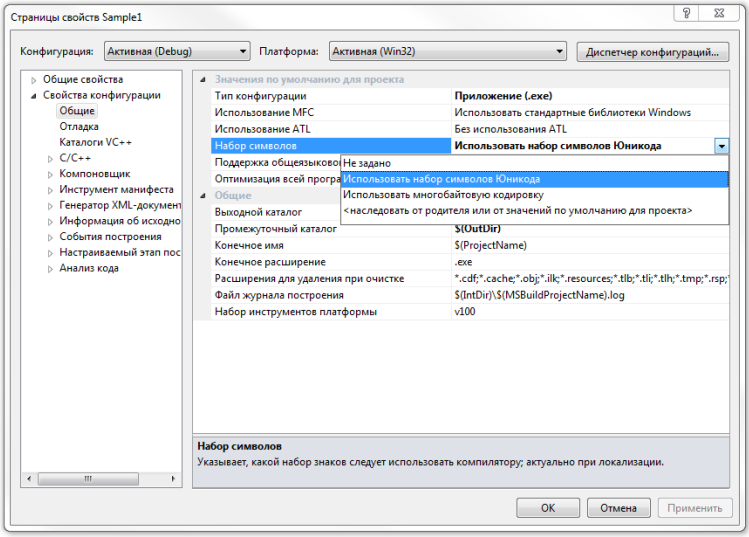

Рис. 1.3. Настройка параметров проекта Visual C++

Безопасные строковые функции

Любая функция, изменяющая строку, несет потенциальную угрозу безопасности: если результирующая строка больше, чем предназначенный для нее буфер, содержимое памяти будет испорчено. Рассмотрим пример:

```
1  // этот код записывает 7 символов в буфер
2  // длиной 6 символов, повреждая содержимое памяти
3  TCHAR szBuffer[6];
4  _tcscpy(szBuffer, TEXT("Строка")); // ноль на конце - тоже символ
```

Проблема с функцией `_tcscpy` (как и с большинством других функций, манипулирующих со строками) состоит в том, что у нее нет параметра задающего максимальный размер буфера. Следовательно, такие функции не могут знать испортят ли они память.

Подобная уязвимость в безопасности активно используется вредоносными программами. Поэтому для каждой из существующих стро-

ковых функций, таких как `_tcscpy`, в библиотеку C была добавлена новая безопасная версия, которая включает имя старой функции и суффикс `_s` (от англ. secure – безопасный). (Впрочем, это не относиться к таким функциям, как `_tcslen`, так как они не изменяют переданную им строку.)

```
PTSTR _tcscpy(PTSTR _Dst, PCTSTR _Src);
errno_t _tcscpy_s(PTSTR _Dst, size_t _SizeInWords,
    PCTSTR _Src);
```

Теперь если функции передается в качестве параметра буфер, доступный для записи, то одновременно необходимо передать и его размер. Размер буфера задают в символах, чтобы его узнать, достаточно вызвать макрос `_countof`, определенный в stdlib.h.

Все безопасные (имеющие в названии суффикс `_s`) функции первым делом проверяют переданные им аргументы. Если проверка оканчивается неудачей (например, итоговая строка не умещается в буфер), безопасные функции вызывают диалоговое окно (рис. 1.4), после чего приложение завершается (это происходит в отладочной сборке, а в случае окончательной сборки приложение завершается сразу).

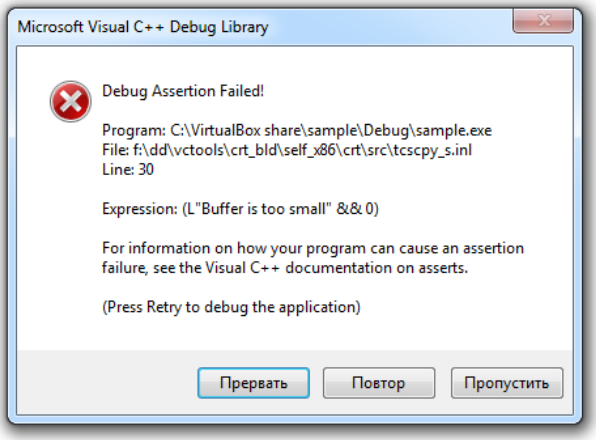

Рис. 1.4. Окно, отображаемое в случае ошибки при вызове безопасной сроковой функции

Помимо безопасных строковых функций (функций с суффиксом `_s` в именах) библиотека C поддерживает ряд новых безопасных функций для манипулирования со строками, определенных в заголовочном

файле StrSafe.h. Естественно, поддерживаются как ANSI- (A), так и Unicode-версии (W) этих функций. Вот прототипы некоторых из них:

```
STRSAFEAPI StringCchCopy(PTSTR pszDest, size_t cchDest,
    PCTSTR pszSrc);

STRSAFEAPI StringCchCat(PTSTR pszDest, size_t cchDest,
    PCTSTR pszSrc);

STRSAFEAPI StringCchPrintf(PTSTR pszDest, size_t cchDest,
    PCTSTR pszFormat, ...);

STRSAFEAPI StringCbCopy(PTSTR pszDest, size_t cbDest,
    PCTSTR pszSrc);

STRSAFEAPI StringCbCat(PTSTR pszDest, size_t cbDest,
    PCTSTR pszSrc);

STRSAFEAPI StringCbPrintf(PTSTR pszDest, size_t cbDest,
    PCTSTR pszFormat, ...);
```

Функции, содержащие в именах «Cch» (сокр. от «Count of characters»), принимают в виде параметра размер, выраженный в символах, обычно его значение получают с помощью макроса `_countof`. А функции, содержащие в именах «Cb» (сокр. от «Count of bytes»), принимают в виде параметра размер, выраженный в байтах, как правило, его значение получают с помощью оператора `sizeof`. Все эти функции возвращают одно из следующих значений:

- `S_OK` – успешный вызов;
- `STRSAFE_E_INVALID_PARAMETER` – неудачный вызов (в параметрах передан `NULL`);
- `STRSAFE_E_INSUFFICIENT_BUFFER` – неудачный вызов (исходная строка не уместилась в целевом буфере).

В отличие от безопасных функций (функций с суффиксом `_s` в именах), эти функции усекают строку, если она не умещается в буфере. В зависимости от поставленной задачи, усечение строки может быть допустимо, а, может, и нет. Именно поэтому данная ситуация по умолчанию считается неудачным вызовом.

Дополнительные возможности при работе со строками

Win32 API предлагает внушительный набор функций, работающих со строками. Рассмотрим некоторые из них, которые будут наиболее полезны при изучении данного курса. Подробнее эти функции и их применение описаны в документации Platform SDK.

Функция `MultiByteToWideChar` преобразует мультибайтовые символы строки в UTF-16:

```
int MultiByteToWideChar(UINT uCodePage, DWORD dwFlags,
    LPCSTR lpMultiByteStr, int cbMultiByte,
    LPWSTR lpWideCharStr, int cchWideChar);
```

Параметр *uCodePage* указывает кодовую страницу, связанную с мультибайтовой кодировкой (табл. 1.5). Параметр *dwFlags* обычно не используется и равен нулю. Параметр *lpMultiByteStr* указывает на преобразуемую строку, а *cbMultiByte* определяет ее длину в байтах. Если значение параметра *cbMultiByte* равно –1, функция самостоятельно определяет длину преобразуемой строки. Строка, полученная в результате преобразования, записывается в буфер по адресу, указанному в параметре *lpWideCharStr*. Максимальный размер этого буфера (в символах) задается в параметре *cchWideChar*. Если значение параметра *cchWideChar* равно нулю, функция ничего не преобразует, а просто возвращает размер буфера, необходимого для сохранения результата преобразования (с учетом нуль-символа в конце).

Таблица 1.5. Значения, указывающие на кодовые страницы

Значение	**Описание**
`CP_ACP`	Кодировка ANSI
`CP_OEMCP`	Кодировка OEM (Original Equipment Manufacturer), основанная на CP437 и содержит VGA-совместимую псевдографику
`CP_MACCP`	Кодировка Macintosh, совместимая с Apple MacOS
`CP_UTF7`	Кодировка UTF-7
`CP_UTF8`	Кодировка UTF-8

Функция `WideCharToMultiByte` выполняет обратное преобразование:

```
int WideCharToMultiByte(UINT uCodePage, DWORD dwFlags,
    LPCWSTR lpWideCharStr, int cchWideChar,
    LPSTR lpMultiByteStr, int cbMultiByte,
    LPCSTR lpDefaultChar, LPBOOL lpUsedDefaultChar);
```

Она очень похожа на `MultiByteToWideChar`. Параметр *uCodePage* определяет кодовую страницу (см. табл. 1.5) для строки – результата преобразования. Параметр *dwFlags* обычно не используется и равен нулю. Параметр *lpWideCharStr* указывает на преобразуемую строку, а *cchWideChar* определяет ее длину в символах. Функция самостоятельно определяет длину, если значение параметра *cchWideChar* равно –1.

Строка, полученная в результате преобразования, записывается в буфер по адресу, указанному в параметре *lpMultiByteStr*. Максимальный размер этого буфера (в байтах) задается в параметре *cbMultiByte*. Если значение параметра *cbMultiByte* равно нулю, функция возвращает размер буфера, необходимого для сохранения результата.

Параметры *lpDefaultChar* и *lpUsedDefaultChar* используются, только если в строке *lpWideCharStr* встречается символ, не представленный в кодовой странице, на которую указывает *uCodePage*. Если его преобразование не возможно, функция берет символ, на который указывает *lpDefaultChar*. Если этот параметр равен NULL, функция использует системный символ по умолчанию (обычно это символ «?»). Параметр *lpUsedDefaultChar* указывает на переменную типа BOOL, которую функция устанавливает в TRUE, если хотя бы один символ из строки *lpWideCharStr* не преобразован в свой мультибайтовый эквивалент. Если же все символы преобразованы успешно, функция устанавливает переменную как FALSE. Обычно этот параметр равен NULL.

Для сравнения строк с учетом языковых особенностей в Win32 API есть функция CompareString. Поддерживается как ANSI- (A), так и Unicode-версии (W) этой функции. Начиная с Windows Vista, в приложениях нужно использовать расширенную версию этой функции – CompareStringEx. Следует учесть, что она работает только с Unicode-строками.

```
int CompareString(LCID Locale, DWORD dwCmpFlags,
    LPCTSTR lpString1, int cchCount1,
    LPCTSTR lpString2, int cchCount2);

int CompareStringEx(LPCWSTR lpLocaleName, DWORD dwCmpFlags,
    LPCWSTR lpString1, int cchCount1,
    LPCWSTR lpString2, int cchCount2,
    LPNLSVERSIONINFO lpVersionInformation,
    LPVOID lpReserved, LPARAM lParam);
```

Параметр *Locale* задает так называемый идентификатор локализации (locale id) – это значение, определяющее конкретный язык. С помощью этого идентификатора функция CompareString сравнивает строки с учетом особенностей в данном языке. Она работает куда осмысленнее, но и существенно медленнее, чем функции библиотеки C. Для задания LCID можно использовать макрос MAKELANGID или одно из значений, перечисленных в табл. 1.6.

Параметр *lpLocaleName* содержит имя локализации (locale) или одно из значений, перечисленных в табл. 1.6. Начиная с Windows Vista, Microsoft рекомендует использовать имена локализаций вместо идентификаторов локализаций.

Таблица 1.6. Константы локализации

Идентификатор / Имя	**Описание**
`LOCALE_INVARIANT` `LOCALE_NAME_INVARIANT`	Инвариантная локализация (invariant locale), которая не зависит от языковых настроек
`LOCALE_SYSTEM_DEFAULT` `LOCALE_NAME_SYSTEM_DEFAULT`	Текущая локализация операционной системы Windows
`LOCALE_USER_DEFAULT` `LOCALE_NAME_USER_DEFAULT`	Текущая локализация пользователя, которая настраивается с помощью элемента **Язык и региональные стандарты** (**Regional and Language Options**) в панели управления Windows

Параметр *`dwCmpFlags`* указывает флаги, модифицирующие метод сравнения строк. Допустимые флаги перечислены в табл. 1.7.

Таблица 1.7. Флаги функции `CompareString(Ex)`

Флаг	**Действие**
`NORM_IGNORECASE`	Различия в регистре букв игнорируются
`NORM_IGNORENONSPACE`	Знаки, отличные от пробелов, игнорируются
`NORM_IGNORESYMBOLS`	Символы, отличные от алфавитно-цифровых, игнорируются
`NORM_IGNOREWIDTH`	Разница между одно- и двухбайтовым представлением одного и того же символа игнорируется
`SORT_STRINGSORT`	Знаки препинания обрабатываются так же, как и символы, отличные от алфавитно-цифровых

Четыре параметра *`lpString1`*, *`cchCount1`*, *`lpString2`* и *`cchCount2`* задают две строки и их длину в символах, соответственно. Если в качестве длины передать значение `-1`, функция сама рассчитывает длину строки, предполагается, что эта строка оканчивается нулем.

Последние три параметра функции `CompareStringEx` не используются. Поэтому параметры *`lpVersionInformation`* и *`lpReserved`* должны быть установлены в `NULL`, а значение параметра *`lParam`* должно равняться нулю.

Функция `CompareString(Ex)` возвращает значения, которые отличаются от значений, возвращаемые функциями `strcmp` и `wcscmp` из библиотеки C. В случае ошибки функция возвращает ноль, а в противном случае одно из следующих значений:

- `CSTR_LESS_THAN` – если *`lpString1`* < *`lpString2`*;
- `CSTR_EQUAL` – если *`lpString1`* = *`lpString2`*;
- `CSTR_GREATER_THAN` – если *`lpString1`* > *`lpString2`*.

Работа с системной информацией Windows

Некоторые функции Win32 API используются для получения или изменения различной информации, касающейся операционной системы Windows.

Имя компьютера

Для получения имени локального компьютера используются функции `GetComputerName` и `GetComputerNameEx`:

```
BOOL GetComputerName(LPTSTR lpBuffer, LPDWORD lpnSize);
BOOL GetComputerNameEx(COMPUTER_NAME_FORMAT NameType,
    LPTSTR lpBuffer, LPDWORD lpnSize);
```

Параметр *`NameType`* определяет тип имени. В табл. 1.8. перечислены некоторые значения, которые может принимать этот параметр. Полный список значений *`NameType`* можно найти в документации Platform SDK.

Таблица 1.8. Некоторые значения *`NameType`*

Значение	Описание
`ComputerNamePhysicalDnsDomain`	Имя домена, в который входит локальный компьютер
`ComputerNamePhysicalDnsFullyQualified`	Полное доменное имя, которое однозначно идентифицирует локальный компьютер
`ComputerNamePhysicalDnsHostname`	Доменное имя, назначенное локальному компьютеру
`ComputerNamePhysicalNetBIOS`	Имя, назначенное локальному компьютеру

Параметр *`lpBuffer`* представляет собой указатель на буфер, в который сохраняется полученное имя компьютера. Параметр *`lpnSize`* указывает на размер буфера в символах. Функция запишет в переменную, на которую указывает *`lpnSize`*, число символов записанных в буфер (нуль-символ в конце не учитывается).

Выполнение функции завершится ошибкой, если размер буфера будет меньше чем величина (`MAX_COMPUTERNAME_LENGTH + 1`). Следует отметить, что длина доменного имени компьютера может быть больше чем значение константы `MAX_COMPUTERNAME_LENGTH`, так как в DNS разрешены длинные имена.

Если необходимо определить размер буфера, параметр *`lpBuffer`* должен быть установлен в `NULL`, а значение переменной, на которую указывает параметр *`lpnSize`*, должно быть равным нулю. В этом случае

функция вернет через параметр *lpnSize*, необходимый размер буфера в символах (с учетом нуль-символа в конце).

Для того чтобы изменить имя локального компьютера используются функции SetComputerName и SetComputerNameEx:

```
BOOL SetComputerName(LPCTSTR lpBuffer);

BOOL SetComputerNameEx(COMPUTER_NAME_FORMAT NameType,
    LPCTSTR lpBuffer);
```

Параметр *NameType* определяет тип изменяемого имени. Этот параметр может принимать значение ComputerNamePhysicalDnsDomain, ComputerNamePhysicalDnsHostname и ComputerNamePhysicalNetBIOS (см. табл. 1.8). Параметр *lpBuffer* указывает на строку, содержащую новое имя компьютера. Новое имя не должно содержать управляющие символы, начальные и конечные пробелы, или любой из следующих символов: " / \ [ ] : | < > + = ; , ?.

Функции GetComputerName(Ex) и SetComputerName(Ex) в случае успеха возвращают значение отличное от FALSE.

Имя пользователя

Для получения имени пользователя в текущем сеансе используются функции GetUserName и GetUserNameEx:

```
BOOL GetUserName(LPTSTR lpBuffer, LPDWORD lpnSize);

BOOLEAN GetUserNameEx(EXTENDED_NAME_FORMAT NameFormat,
    LPTSTR lpBuffer, LPDWORD lpnSize);
```

Параметр *NameFormat* определяет формат имени пользователя. Если учетная запись пользователя не находится в домене, то поддерживается только значение NameSamCompatible. В этом случае возвращается полное имя пользователя (например, Андрей-ПК\Андрей). Описание всех значений *NameFormat* см. в документации Platform SDK.

Параметр *lpBuffer* представляет собой указатель на буфер, в который сохраняется полученное имя пользователя. Параметр *lpnSize* указывает на размер буфера в символах. Функция запишет в переменную, на которую указывает *lpnSize*, число символов записанных в буфер (нуль-символ в конце не учитывается).

Выполнение функции завершится ошибкой, если размер буфера будет меньше чем величина (UNLEN + 1). Константа UNLEN определена в заголовочном файле LMCons.h. Если установить параметр *lpBuffer* в NULL, а значение переменной, на которую указывает параметр *lpnSize*, равным нулю, то функция вернет через параметр *lpnSize*, необходимый размер буфера в символах (с учетом нуль-символа в конце).

В случае успеха функции GetUserName и GetUserNameEx возвращают значение отличное от FALSE.

Чтобы использовать функцию GetUserNameEx необходимо подключить библиотеку Secur32.dll и заголовочный файл Security.h. Кроме того, перед заголовком Security.h нужно определить символическую константу SECURITY_WIN32.

Системные каталоги

Начиная с Windows XP, для определения путей к каталогам Windows следует вызывать функцию SHGetFolderPath, объявленную в заголовочном файле ShlObj.h.

```
HRESULT SHGetFolderPath(HWND hwndOwner, int nFolder,
    HANDLE hToken, DWORD dwFlags, LPTSTR pszPath);
```

Первый параметр, *hwndOwner*, всегда должен равняться NULL. Второй параметр, *nFolder*, – это значение CSIDL (Constant Special Item ID List), которое идентифицирует каталог, путь к которому необходимо узнать. Если такого каталога не существует, то можно его создать, скомбинировав значение CSIDL и значение CSIDL_FLAG_CREATE. В табл. 1.9 приводится описание некоторых значений CSIDL. Полный список значений CSIDL можно найти в документации Platform SDK.

Таблица 1.9. Значения CSIDL

Значение	**Описание**
CSIDL_APPDATA	Каталог, в котором размещены данные приложений пользователя **В Windows XP и выше:** %APPDATA%
CSIDL_COMMON_APPDATA	Каталог, в котором размещены данные приложений всех пользователей **В Windows XP:** %ALLUSERSPROFILE%\Start Menu\Programs\ Administrative Tools **В Windows Vista и выше:** %ALLUSERSPROFILE%
CSIDL_COMMON_DOCUMENTS	Каталог «Общие документы» **В Windows XP:** %ALLUSERSPROFILE%\Documents **В Windows Vista и выше:** %PUBLIC%\Documents

Продолжение табл. 1.9.

Значение	Описание
CSIDL_HISTORY	Каталог, в котором содержатся файлы истории Internet Explorer. **В Windows XP:** %USERPROFILE%\Local Settings\History **В Windows Vista и выше:** %LOCALAPPDATA%\Microsoft\Windows\History
CSIDL_INTERNET_CACHE	Каталог, в котором размещены временные файлы Интернета **В Windows XP:** %USERPROFILE%\Local Settings\ Temporary Internet Files **В Windows Vista и выше:** %LOCALAPPDATA%\Microsoft\Windows\ Temporary Internet Files
CSIDL_LOCAL_APPDATA	Каталог, в котором размещены данные приложений пользователя. Этот каталог привязан к локальному компьютеру и не копируется, когда пользовательский профиль перемещается на другой компьютер **В Windows XP:** %USERPROFILE%\Local Settings\Application Data **В Windows Vista и выше:** %LOCALAPPDATA%
CSIDL_PERSONAL	Каталог «Мои документы» **В Windows XP:** %USERPROFILE%\My Documents **В Windows Vista и выше:** %USERPROFILE%\Documents
CSIDL_PROGRAM_FILES	Каталог «Program Files» **В Windows XP и выше:** %PROGRAMFILES%
CSIDL_PROGRAM_FILES_COMMON	Каталог «Common Files» **В Windows XP и выше:** %COMMONPROGRAMFILES%

Окончание табл. 1.9.

Значение	Описание
CSIDL_SYSTEM	Системный каталог Windows **В Windows XP и выше:** %WINDIR%\system32
CSIDL_WINDOWS	Каталог Windows **В Windows XP и выше:** %WINDIR%

Третий параметр, *hToken*, представляет собой дескриптор маркера доступа, который идентифицирует пользователя и содержит информацию о привилегиях. Обычно этот параметр установлен в NULL. Параметр *hToken* используется только, если необходимо узнать путь к каталогу для определенного пользователя.

Четвертый параметр, *dwFlags*, определяет каталог, путь к которому возвращает функция в случае, если каталог, указанный в CSIDL не существует:

- SHGFP_TYPE_CURRENT – возвращает текущий каталог;
- SHGFP_TYPE_DEFAULT – возвращает каталог по умолчанию.

Пятый параметр, *pszPath*, представляет собой указатель на буфер, в который будет сохранен путь после вызова функции. Размер буфера должен равняться величине (MAX_PATH + 1).

Функция SHGetFolderPath возвращает S_OK в случае успеха, в противном случае – значение ошибки или S_FALSE если каталог не существует.

В Windows Vista и более новых версий функция SHGetFolderPath является лишь «оберткой» для функции SHGetKnownFolderPath, которую также можно использовать.

Следует отметить, что в Win32 API существуют функции вроде GetTempPath, GetWindowsDirectory, GetSystemDirectory и т.п., которые не следует использовать, так как они были добавлены только для обратной совместимости со старыми версиями Windows.

Версия операционной системы

Для получения информации о версии Windows используется функция GetVersionEx:

```
BOOL GetVersionEx(LPOSVERSIONINFO lpVersionInfo);
```

Параметр *lpVersionInfo* – указатель на структуру OSVERSIONINFO, в которую будет сохранена информация о версии.

Структура OSVERSIONINFO определена следующим образом:

```
typedef struct _OSVERSIONINFO {
  DWORD dwOSVersionInfoSize; // размер структуры в байтах
  DWORD dwMajorVersion; // идентификатор операционной системы
  DWORD dwMinorVersion; // идентификатор версии
  DWORD dwBuildNumber; // идентификатор сборки
  DWORD dwPlatformId; // платформа
  TCHAR szCSDVersion[128]; // дополнительная информация
                           // об операционной системе
} OSVERSIONINFO;
```

Первое поле, *dwOSVersionInfoSize*, должно содержать значение равное размеру структуры в байтах, как правило, его получают с помощью оператора sizeof.

Поля *dwMajorVersion* и *dwMinorVersion* указывают соответственно основной и дополнительный номер версии операционной системы. Наиболее актуальные версии Windows перечислены в табл. 1.10.

Таблица 1.10. Версии Windows

Операционная система	Версия	dwMajorVersion	dwMinorVersion
Windows 10 Windows Server 2016	10.0	10	0
Windows 8.1 Windows Server 2012 R2	6.3	6	3
Windows 8 Windows Server 2012	6.2	6	2
Windows 7 Windows Server 2008 R2	6.1	6	1
Windows Vista Windows Server 2008	6.0	6	0
Windows Server 2003 R2 Windows Server 2003	5.2	5	2
Windows XP	5.1	5	1

Четвертое поле, *dwBuildNumber*, указывает номер сборки операционной системы. Пятое поле, *dwPlatformId*, идентифицирует платформу операционной системы. Начинает с Windows XP это поле принимает значение VER_PLATFORM_WIN32_NT.

Последнее поле, *szCSDVersion*, – строка, которая указывает последнюю версию установленного пакета обновлений (например, «Service Pack 3»). Если пакет обновлений не установлен, строка пуста.

В случае успеха функции GetVersionEx возвращает значение отличное от FALSE. Для получения дополнительной информации о версии операционной системы следует передать в функцию GetVersionEx указатель на структуру OSVERSIONINFOEX (подробнее см. в документации Platform SDK).

Системные метрики и параметры

В Win32 API имеется функция GetSystemMetrics, позволяющая получить значения различных системных метрик и системных параметров настроек конфигурации операционной системы. *Системная метрика* (system metric) – это выраженный в пикселях размер элементов отображения Windows.

```
int GetSystemMetrics(int nIndex);
```

Параметр *nIndex* указывает метрику или настройку конфигурации значение, которой необходимо получить. Для задания метрики и настройки конфигурации в заголовочном файле WinUser.h определены константы с префиксом SM_. Все метрики с префиксом SM_CX определяют ширину элемента, с префиксом SM_CY – высоту. Список констант SM_* см. в документации Platform SDK.

Для получения или изменения системных параметров используется функция SystemParametersInfo:

```
BOOL SystemParametersInfo(UINT uiAction, UINT uiParam,
    PVOID pvParam, UINT fWinIni);
```

Первый параметр, *uiAction*, указывает запрашиваемый или устанавливаемый системный параметр. Для указания параметра применяют символические константы с префиксами SPI_GET и SPI_SET, определенные в заголовочном файле WinUser.h. Для запроса системных параметров используются константы с префиксом SPI_GET, для установки – SPI_SET. Список констант SPI_GET* и SPI_SET* можно найти в документации Platform SDK.

Параметры *uiParam* и *pvParam* зависят от параметра *uiAction*. При запросе системного параметра значение параметра *uiParam* должно быть равно нулю (в большинстве случаев), а параметр *pvParam* должен указывать на переменную, в которую будет записано текущее значение системного параметра. При установке системного параметра в зависимости от того, что это за параметр, его новое значение передается либо через параметр *uiParam*, либо через параметр *pvParam*.

Последний параметр, *fWinIni*, указывает нужно ли обновить настройки в профиле пользователя (файл win.ini). Этот параметр должен принимать значение равное нулю, если не требуется обновлять

настройки или может быть одним (или комбинацией) из следующих значений:

- SPIF_UPDATEINIFILE – записывает новое значение параметра настроек в профиле пользователя;
- SPIF_SENDCHANGE – после обновления профиля отсылает всем главным окнам сообщение WM_SETTINGCHANGE;
- SPIF_SENDWININICHANGE то же, что SPIF_UPDATEINIFILE.

В случае успеха функция SystemParametersInfo возвращает значение отличное от FALSE.

В листинге 1.2 приводится пример получения двух системных параметров с помощью функции SystemParametersInfo:

Листинг 1.2. Пример получения системных параметров

```
1  // определим включено ли сглаживание шрифтов
2  BOOL bIsFontSmooth;
3  SystemParametersInfo(SPI_GETFONTSMOOTHING, 0, &bIsFontSmooth, 0);
4
5  // определим размер рабочего стола без панели задач
6  RECT rcWorkArea;
7  SystemParametersInfo(SPI_GETWORKAREA, 0, &rcWorkArea, 0);
```

Системная дата и время

Операционная система Windows хранит системную дату и время в формате *универсального координированного времени* (Coordinated Universal Time, UTC), который является современной версией *среднего времени по Гринвичу* (Greenwich Mean Time, GMT).

Чтобы получить системную дату и время следует использовать функцию GetSystemTime, а чтобы извлечь системную дату и время по местному времени – GetLocalTime:

```
void GetSystemTime(LPSYSTEMTIME lpSystemTime);
void GetLocalTime(LPSYSTEMTIME lpSystemTime);
```

Параметр *lpSystemTime* – указатель на структуру SYSTEMTIME, которая имеет следующее определение:

```
typedef struct _SYSTEMTIME {
  WORD wYear; // год
  WORD wMonth; // месяц (1 – январь, 2 – февраль и т.д.)
  WORD wDayOfWeek; // день недели (0 – воскресенье,
                   // 1 – понедельник, 2 – вторник и т.д.)
  WORD wDay; // день месяца
  WORD wHour; // час
```

```
  WORD wMinute; // минута
  WORD wSecond; // секунда
  WORD wMilliseconds; // миллисекунда
} SYSTEMTIME, *PSYSTEMTIME;
```

Поля структуры SYSTEMTIME не нуждаются в дополнительных комментариях.

Для задания системной даты и времени используются функции SetSystemTime (в формате UTC) и SetLocalTime (в формате местного времени).

```
BOOL SetSystemTime(const SYSTEMTIME *lpSystemTime);
```

```
BOOL SetLocalTime(const SYSTEMTIME *lpSystemTime);
```

Параметр *lpSystemTime* – указатель на структуру SYSTEMTIME, которая определяет дату и время. При этом значение поля *wDayOfWeek* игнорируется.

В случае успеха функции SetSystemTime и SetLocalTime возвращают значение отличное от FALSE.

Для преобразования даты и времени из формата UTC в формат, соответствующий указанному часовому поясу, и наоборот следует использовать следующие функции:

```
BOOL SystemTimeToTzSpecificLocalTime(
    LPTIME_ZONE_INFORMATION lpTimeZone,
    LPSYSTEMTIME lpUniversalTime, LPSYSTEMTIME lpLocalTime);
```

```
BOOL TzSpecificLocalTimeToSystemTime(
    LPTIME_ZONE_INFORMATION lpTimeZone,
    LPSYSTEMTIME lpLocalTime, LPSYSTEMTIME lpUniversalTime);
```

Первый параметр, *lpTimeZone*, – указатель на структуру TIME_ZONE_INFORMATION, которая определяет настройки часового пояса. Если этот параметр установлен в NULL, то функция использует текущий часовой пояс. Описание структуры TIME_ZONE_INFORMATION см. в документации Platform SDK.

Параметры *lpUniversalTime* и *lpLocalTime* представляют собой указатели на структуру SYSTEMTIME, в которой определяется дата и время, соответственно, в формате UTC и в формате, соответствующему указанному часовому поясу.

В случае успеха функции SystemTimeToTzSpecificLocalTime и TzSpecificLocalTimeToSystemTime возвращают значение отличное от FALSE.

Кроме того, в Win32 API имеются функции GetDateFormat и GetTimeFormat, которые возвращают соответственно дату и время в виде строк, форматируя их в определенном формате с учетом языковых

особенностей. Начиная с Windows Vista, в приложениях нужно использовать функции GetDateFormatEx и GetTimeFormatEx. Следует отметить, что эти функции работают только с Unicode-строками.

```
int GetDateFormat(LCID Locale, DWORD dwFlags,
    const SYSTEMTIME *lpDate, LPCTSTR lpFormat,
    LPTSTR lpDateStr, int cchDate);

int GetTimeFormat(LCID Locale, DWORD dwFlags,
    const SYSTEMTIME *lpTime, LPCTSTR lpFormat,
    LPTSTR lpTimeStr, int cchTime);

int GetDateFormatEx(LPCWSTR lpLocaleName, DWORD dwFlags,
    const SYSTEMTIME *lpDate, LPCWSTR lpFormat,
    LPWSTR lpDateStr, int cchDate, LPCWSTR lpCalendar);

int GetTimeFormatEx(LPCWSTR lpLocaleName, DWORD dwFlags,
    const SYSTEMTIME *lpTime, LPCWSTR lpFormat,
    LPWSTR lpTimeStr, int cchTime);
```

Параметр *Locale* содержит идентификатор локализации, который можно задать с помощью макроса MAKELANGID. Параметр *lpLocaleName* представляет собой имя локализации. Для задания параметров *Locale* и *lpLocaleName* могут также использоваться константы локализации (см. табл. 1.6). Начиная с Windows Vista, Microsoft рекомендует использовать имена локализаций вместо идентификаторов локализаций.

Параметр *dwFlags* определяет режимы работы функций, если параметр *lpFormat* установлен в NULL. Для параметра *dwFlags* можно использовать значения, перечисленные в табл. 1.11.

Таблица 1.11. Константы, которые можно использовать для *dwFlags*

Константа	Описание
LOCALE_NOUSEROVERRIDE	Результирующая строка будет получена в формате, который используется операционной системой по умолчанию для указанного параметра локализации
DATE_SHORTDATE	Сокращенный формат даты (например, 02.03.1986)
DATE_LONGDATE	Полный формат даты (например, 2 марта 1986 г.)
DATE_YEARMONTH	Не использовать дни месяца (только для Window Vista и выше)
TIME_NOMINUTESORSECONDS	Не использовать минуты или секунды
TIME_NOSECONDS	Не использовать секунды
TIME_NOTIMEMARKER	Не использовать маркер (A, P, AM и PM)
TIME_FORCE24HOURFORMAT	Всегда использовать 24-часовой формат времени

Параметр *lpDate* (*lpTime*) – указатель на структуру SYSTEMTIME, содержащую дату и время. Если этот параметр установлен в NULL будут использоваться текущая дата и время.

Параметр *lpFormat* задает строку формата, в соответствии с которым будет отформатирована выходная строка. Строка формата может содержать специальные символы (табл. 1.12), пробелы и произвольные символы, заключенные в кавычки. Вместо специальных символов будут вставлены отдельные компоненты времени. Например, чтобы получить строку «2 марта 1986 г.», нужно использовать следующую строку формата «d MMMM yyyy г.».

Таблица 1.12. Специальные символы строки формата даты и времени

Символ	**Компонент даты и времени**
d	День месяца без ведущего нуля
dd	День месяца с ведущим нулем
ddd	Трехбуквенное сокращение дня недели
dddd	Полное название дня недели
M	Номер месяца без ведущего нуля
MM	Номер месяца с ведущим нулем
MMM	Трехбуквенное сокращение названия месяца
MMMM	Полное название месяца
y	Двухзначное обозначение года без ведущего нуля (последние две цифры года)
yy	Двухзначное обозначение года с ведущим нулем
yyyy	Полный номер года
gg	Название периода или эры
h	Часы без ведущего нуля в 12-часовом формате
hh	Часы с ведущим нулем в 12-часовом формате
H	Часы без ведущего нуля в 24-часовом формате
HH	Часы с ведущим нулем в 24-часовом формате
m	Минуты без ведущего нуля
mm	Минуты с ведущим нулем
s	Секунды без ведущего нуля
ss	Секунды с ведущим нулем
t	Маркер (такой как A или P)
tt	Маркер (такой как AM или PM)

Если параметр *lpFormat* установлен в NULL, будет использован формат по умолчанию для указанного параметра локализации.

Параметр *lpDateStr* (*lpTimeStr*) – указатель на буфер, в который сохраняется результат выполнения функции.

Параметр, *cchDate* (*cchTime*), содержит размер буфера. Если необходимо определить размер буфера, установите это параметр равным нулю.

Последний параметр функции GetDateFormatEx не используется и поэтому параметр *lpCalendar* должен быть установлен в NULL.

Функции GetDateFormat(Ex) и GetTimeFormat(Ex) в случае успеха возвращают число символов, скопированных в буфер (нуль-символ в конце не учитывается). Если параметр *cchDate* (*cchTime*) содержит значение равное нулю, функции возвращают необходимый размер (в символах) буфера, включая нуль-символ в конце. В случае ошибки возвращается ноль.

Консольные приложения Win32

Чтобы в процессе выполнения данной лабораторной работы полностью сосредоточиться на знакомстве с Win32 API и избежать необходимости углубляться в сложности, связанные с созданием и управлением окнами в приложении Windows, будем использовать консольное приложение Win32.

Консольное приложение (console application) представляет собой программу, использующую *интерфейс командной строки* (Console User Interface, CUI). Консольные приложения создаются, как правило, в тех случаях, когда требуется создать простейшую программу, обладающую самым простым интерфейсом. Информация консольному приложению передается в аргументах командной строки и через стандартный поток ввода. Для вывода информации консольное приложение использует стандартный поток вывода.

Создание консольного приложения Win32

Среда разработки Visual C++ позволяет создавать консольные приложения Win32. Для этого необходимо выполнить следующие действия:

1. В меню **Файл** (**File**) выберите **Создать** (**New**) → **Проект** (**Project**).

   Откроется диалоговое окно **Создать проект** (**New Project**).

2. В области шаблонов проектов **Visual C++** выберите группу **Win32** и затем выберите элемент **Консольное приложение Win32** (**Win32 Console Application**), как показано на рис. 1.5.

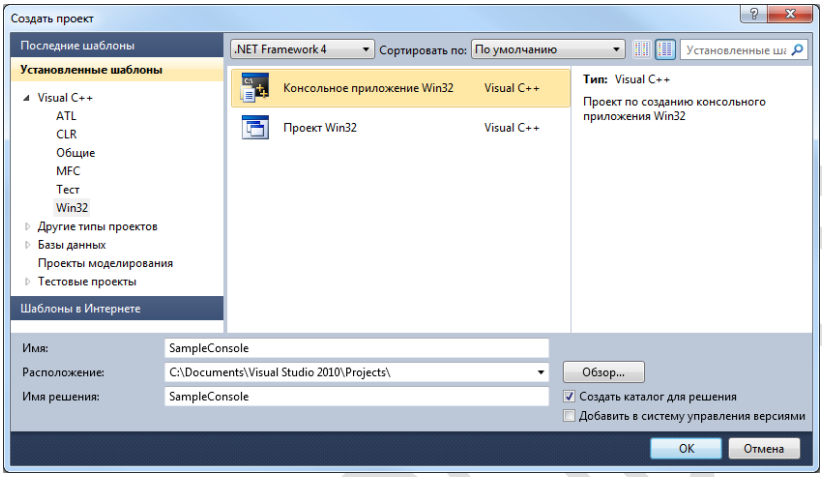

Рис. 1.5. Создание проекта консольного приложения Win32

3. Введите имя проекта.

   По умолчанию имя решения, содержащего проект, совпадает с именем проекта, однако можно ввести другое имя. Также можно указать другое расположение для проекта.

4. Нажмите кнопку **ОК**, чтобы создать проект.

5. В открывшемся диалоговом окне **Мастер приложений Win32** (**Win32 Application Wizard**) нажмите кнопку **Далее** (**Next**), выберите вариант **Пустой проект** (**Empty Project**) и нажмите кнопку **Готово** (**Finish**).

6. Если окно **Обозреватель решений** (**Solution Explorer**) не открыто, выберите в меню **Вид** (**View**) пункт **Обозреватель решений** (**Solution Explorer**).

7. Добавьте новый файл исходного кода в проект, выполнив следующие действия:

   а. В окне **Обозреватель решений** (**Solution Explorer**) щелкните правой кнопкой мыши папку **Файлы исходного кода** (**Source Files**) и последовательно выберите пункты **Добавить** (**Add**) и **Создать элемент** (**New Item**);

   б. В узле **Код** (**Code**) выберите элемент **Файл C++ (.cpp)** (**C++ File (.cpp)**), введите имя файла и нажмите кнопку **Добавить** (**Add**), как показано на рис. 1.6.

8. В файле, открывшемся в редакторе Visual Studio, введите программный код на языке C/C++ и сохраните его.

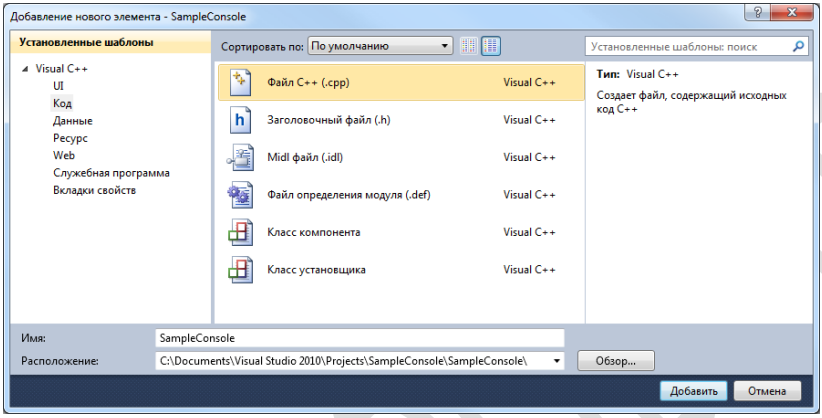

Рис. 1.6. Создание файла исходного кода C++

Функция main

Каждое написанное на языке C/C++ консольное приложение должно иметь специальную функцию `main`, которая является главной функцией всех программ C/C++. При написании приложения, которое использует набор символов Unicode, можно использовать `wmain`.

Функции `main` и `wmain` должны быть определены одним из следующих образов:

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

int wmain();
int wmain(int argc, wchar_t *argv[]);
int wmain(int argc, wchar_t *argv[], wchar_t *envp[]);
```

Можно также использовать макрос `_tmain`, который объявлен в заголовочном файле tchar.h. В зависимости от определения константы `_UNICODE` используется либо `main`, либо `wmain`.

```
int _tmain();
int _tmain(int argc, LPTSTR argv[]);
int _tmain(int argc, LPTSTR argv[], LPTSTR envp[]);
```

Параметр *argc* – это число аргументов командной строки, передаваемых в `main`/`wmain`. Параметр *argv* представляет собой указатель на массив аргументов командной строки. Все аргументы командной стро-

ки являются строками – введенные числа должны конвертироваться в соответствующее внутреннее представление. Параметр *envp* содержит указатель на массив переменных окружения (environment variable), которые используются в Windows для хранения текстовых строк пользователя и информации о системных настройках. Массив переменных окружения заканчивается значением NULL.

В листинге 1.3 приводится пример консольного приложения Win32, использующего параметры *argc*, *argv* и *envp*.

Листинг 1.3. Пример консольного приложения Win32

```
1   #include <Windows.h>
2   #include <tchar.h>
3   #include <locale.h>
4
5   int _tmain(int argc, LPTSTR argv[], LPTSTR envp[])
6   {
7       _tsetlocale(LC_ALL, TEXT("Russian"));
8
9       _tprintf(TEXT("> Аргументы командной строки:\n"));
10
11      for (int i = 0; i < argc; ++i)
12      {
13          _tprintf(TEXT("> %s\n"), argv[i]);
14      } // for
15
16      _tprintf(TEXT("\n> Переменные окружения:\n"));
17
18      for (LPTSTR *p = envp; NULL != *p; ++p)
19      {
20          _tprintf(TEXT("> %s\n"), *p);
21      } // for
22  } // _tmain
```

Как видно из примера, в теле функции main/wmain может не быть оператора return. Компилятор C/C++ должен интерпретировать это как return 0.

Кроме того, значения параметров *argc*, *argv* и *envp* также всегда доступны через соответствующие глобальные переменные *__argc*, *__argv*, *__wargv*, *_environ* и *_wenviron*.

Точка входа в консольное приложение

Если в качестве главной функции используется main, то точкой входа при запуске консольного приложения является специальная функция mainCRTStartup. Функция mainCRTStartup инициализирует

различные библиотеки C/C++, загружает необходимые DLL, создает и инициализирует все глобальные переменные. Когда все это будет сделано, она вызывает функцию `main`.

Если же в качестве главной функции используется `wmain`, то точкой входа является функция `wmainCRTStartup`, которая вызывает `wmain`, но в остальном она делает все тоже, что и `mainCRTStartup`.

После того как функция `main` (или `wmain`) вернет управление, функция `mainCRTStartup` (или `wmainCRTStartup`) удалит все глобальные переменные, выгрузит DLL и освободит ресурсы, выделенные для различных библиотек C/C++.

Статические и динамические библиотеки

Вместо того чтобы каждый раз реализовывать одни и те же функции в каждом создаваемом приложении, можно создать библиотеку, реализующую эти функции, и затем использовать ее в приложениях. В Visual C++ можно создавать статические и динамически подключаемые библиотеки.

Статической библиотекой (static library) называется файл с расширением .lib. Компоновщик копирует данные, функции, классы и методы классов, содержащиеся в статической библиотеке, в исполнимый файл (.exe) приложения при его создании. После этого содержимое статической библиотеки становиться частью приложения. Такой подход называется *статическим подключением* (static linking). Основным недостатком такого подхода является то, что при внесении изменений в библиотеку придется пересоздать все зависимые от нее файлы.

Динамически подключаемой библиотекой (dynamic link library, DLL) является библиотека, содержащая данные, функции, классы и методы классов, которые не копируются в исполнимый файл при создании приложения, а загружаются во время его запуска или в процессе выполнения. Подход, при котором библиотека DLL загружается в уже исполняемое приложение, называется *динамическим подключением* (dynamic linking). При внесении изменений в библиотеку DLL нет необходимости пересоздавать все зависимые от нее файлы.

В общем случае, файл, являющийся динамически подключаемой библиотекой, имеет расширение .dll. Однако некоторые виды библиотек DLL могут иметь другое расширение:

- .cpl – библиотека DLL, представляющая собой элемент панели управления Windows;
- .ocx – библиотека DLL, содержащая объекты ActiveX;
- .drv – библиотека DLL, представляющая собой драйвер какого-либо устройства.

Пример создания статической библиотеки

Для того чтобы создать статическую библиотеку в Visual C++ необходимо выполнить следующие действия:

1. В меню **Файл** (**File**) выберите **Создать** (**New**) → **Проект** (**Project**).

   Откроется диалоговое окно **Создать проект** (**New Project**).

2. В области шаблонов проектов **Visual C++** выберите группу **Win32** и затем выберите элемент **Проект Win32** (**Win32 Project**).

   В поле **Имя** (**Name**) введите имя создаваемого проекта, например, SampleLibrary. Нажмите кнопку **ОК**.

3. В открывшемся диалоговом окне **Мастер приложений Win32** (**Win32 Application Wizard**) нажмите кнопку **Далее** (**Next**).

   Выберите **Статическая библиотека** (**Static library**) и снимите флажок с параметра **Предварительно скомпилированный заголовок** (**Precompiled header**), как показано на рис. 1.7.

   Нажмите кнопку **Готово** (**Finish**).

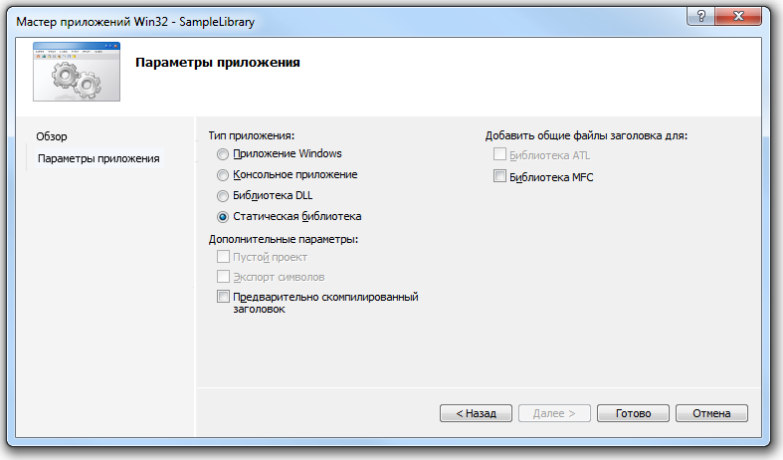

Рис. 1.7. Создание статической библиотеки в Visual C++

4. Добавьте новый заголовочный файл в проект, выполнив следующие действия:

   a. В окне **Обозреватель решений** (**Solution Explorer**) щелкните правой кнопкой мыши папку **Заголовочные файлы** (**Header Files**) и последовательно выберите пункты **Добавить** (**Add**) и **Создать элемент** (**New Item**);

б. В узле **Код** (**Code**) выберите элемент **Заголовочный файл (.h)** (**Header File (.h)**). В поле **Имя** (**Name**) введите имя файла, например, SampleLibrary.h и нажмите кнопку **Добавить** (**Add**).

5. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 1.4 и сохраните этот файл.

Листинг 1.4. Заголовочный файл SampleLibrary.h

```
#pragma once

#if !defined (__SAMPLE_LIBRARY_H__)
#define __SAMPLE_LIBRARY_H__

// функция PrintComputerName
// выводит в стандартный поток NetBIOS имя компьютера
BOOL PrintComputerName();

// функция PrintDNSHostname
// выводит в стандартный поток DNS имя компьютера
BOOL PrintDNSHostname();

#endif /* __SAMPLE_LIBRARY_H__ */
```

6. Добавьте новый исходный файл в проект, выполнив следующие действия:

   а. В окне **Обозреватель решений** (**Solution Explorer**) щелкните правой кнопкой мыши папку **Файлы исходного кода** (**Source Files**) и последовательно выберите пункты **Добавить** (**Add**) и **Создать элемент** (**New Item**);

   б. В узле **Код** (**Code**) выберите элемент **Файл C++ (.cpp)** (**C++ File (.cpp)**). В поле **Имя** (**Name**) введите имя файла, например, SampleLibrary.cpp. Нажмите кнопку **Добавить** (**Add**).

7. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 1.5 и сохраните этот файл.

8. В меню **Построение** (**Build**) выберите команду **Построить решение** (**Build Solution**).

Листинг 1.5. Файл исходного кода SampleLibrary.cpp

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

```

```
5   // ------------------------------------------------------------
6   BOOL PrintComputerName()
7   {
8       TCHAR szBuffer[MAX_COMPUTERNAME_LENGTH + 1];
9       DWORD cch = _countof(szBuffer);
10
11      // получим NetBIOS имя компьютера
12      BOOL bRet = GetComputerName(szBuffer, &cch);
13
14      if (FALSE != bRet)
15      {
16          // выведем полученное имя в стандартный поток
17          _tprintf(TEXT("Name: %s\n"), szBuffer);
18      } // if
19
20      return bRet;
21  } // PrintComputerName
22
23  // ------------------------------------------------------------
24  BOOL PrintDNSHostname()
25  {
26      LPTSTR lpszBuffer = NULL;
27      DWORD cch = 0;
28
29      // определим размер буфера
30      GetComputerNameEx(ComputerNamePhysicalDnsHostname, NULL,
    &cch);
31
32      // выделим память под буфер
33      lpszBuffer = new TCHAR[cch];
34
35      // получим DNS имя компьютера
36      BOOL bRet = GetComputerNameEx(
    ComputerNamePhysicalDnsHostname, lpszBuffer, &cch);
37
38      if (FALSE != bRet)
39      {
40          // выведем полученное имя в стандартный поток
41          _tprintf(TEXT("Hostname: %s\n"), lpszBuffer);
42      } // if
43
44      // освободим память, выделенную под буфер
45      delete[]lpszBuffer;
46      return bRet;
47  } // PrintDNSHostname
```

Подключение статической библиотеки

Статические библиотеки C/C++ можно подключать к написанным на языке C/C++ приложениям, а также к другим библиотекам.

Рассмотрим пример того, как подключить созданную в предыдущем разделе библиотеку SampleLibrary.lib к консольному приложению Win32. Для других проектов Win32 подключение статических библиотек происходит аналогично.

1. Создайте пустой проект консольного приложения Win32 (например, SampleRefsLib).
2. В меню **Проект** (**Project**) выберите **Свойства** (**Properties**).

   Откроется диалоговое окно **Страницы свойств** (**Property Pages**).
3. В открывшемся окне выберите **Свойства конфигурации** (**Configuration Properties**) → **Каталоги VC++** (**VC++ Directories**).
4. Отредактируйте значения параметров **Каталоги включения** (**Include Directories**) и **Каталоги библиотек** (**Library Directories**), добавив пути к каталогам, в которых находятся заголовочный файл SampleLibrary.h и библиотека SampleLibrary.lib.
5. Нажмите кнопку **OK**, чтобы сохранить изменения.
6. Добавьте в проект файл исходного кода SampleRefsLib.cpp.
7. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 1.6 и сохраните этот файл.

Листинг 1.6. Файл исходного кода SampleRefsLib.cpp

```
#include <Windows.h>
#include <tchar.h>
#include <locale.h>
#include <SampleLibrary.h>

int _tmain()
{
    _tsetlocale(LC_ALL, TEXT(""));

    // выведем NetBIOS имя компьютера
    PrintComputerName();
    // выведем DNS имя компьютера
    PrintDNSHostname();
} // _tmain
```

8. Подключите статическую библиотеку SampleLibrary.lib, выполнив одно из следующих действий:

– Добавьте в файл исходного кода SampleRefsLib.cpp следующую директиву:

```
#pragma comment(lib, "SampleLibrary.lib")
```

– В диалоговом окне **Страницы свойств** (**Property Pages**) выберите **Свойства конфигурации** (**Configuration Properties**) → **Компоновщик** (**Linker**) → **Ввод** (**Input**).

Отредактируйте значение параметра **Дополнительные зависимости** (**Additional Dependencies**), указав в нем SampleLibrary.lib, как показано на рис. 1.8. Нажмите кнопку **OK**.

9. В меню **Отладка** (**Debug**) выберите **Запуск без отладки** (**Start Without Debugging**).

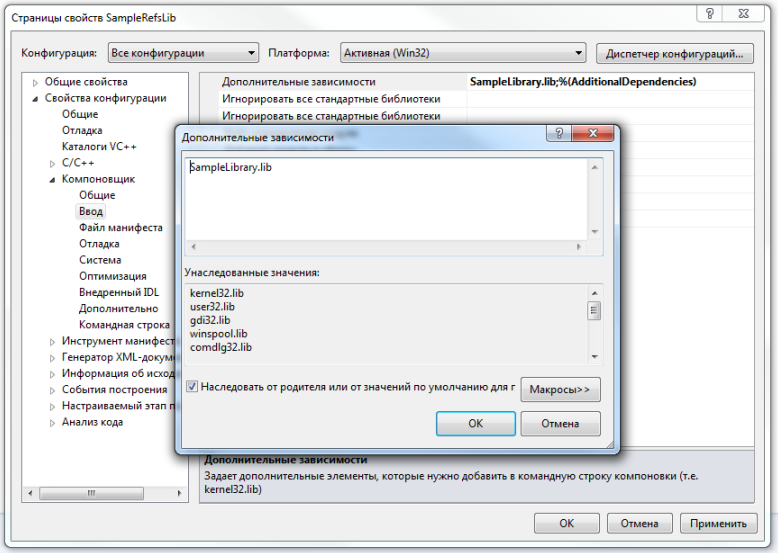

Рис. 1.8. Добавление библиотеки в Visual C++

Пример создания библиотеки DLL

Для того чтобы создать библиотеку DLL в Visual C++ необходимо выполнить следующие действия:

1. В меню **Файл** (**File**) выберите **Создать** (**New**) → **Проект** (**Project**).

   Откроется диалоговое окно **Создать проект** (**New Project**).

2. В области шаблонов проектов **Visual C++** выберите группу **Win32** и затем выберите элемент **Проект Win32** (**Win32 Project**).

В поле **Имя** (**Name**) введите имя создаваемого проекта (например, SampleDLL). Нажмите кнопку **ОК**.

3. В открывшемся диалоговом окне **Мастер приложений Win32** (**Win32 Application Wizard**) нажмите кнопку **Далее** (**Next**).

   Выберите **Библиотека DLL** (**DLL**) и выберите параметр **Пустой проект** (**Empty Project**), как показано на рис. 1.9.

   Нажмите кнопку **Готово** (**Finish**).

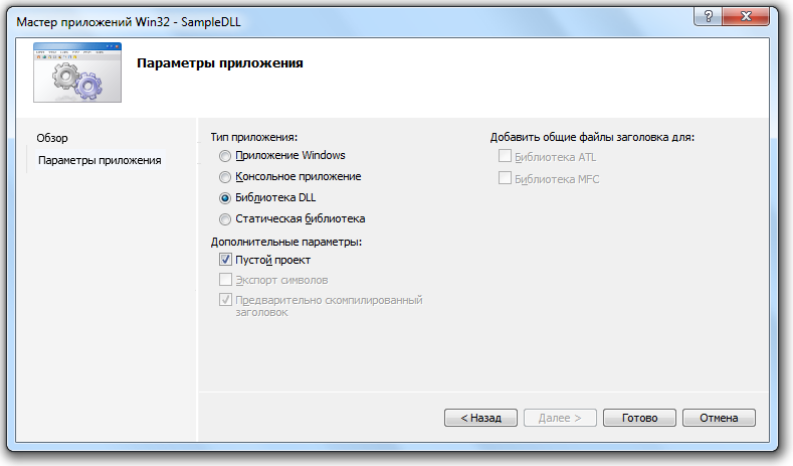

Рис. 1.9. Создание библиотеки DLL в Visual C++

4. Добавьте в проект заголовочный файл SampleDLL.h.
5. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 1.7 и сохраните этот файл.

Листинг 1.7. Заголовочный файл SampleDLL.h

```
1   #pragma once
2
3   #if !defined(__SAMPLE_DLL_H__)
4   #define __SAMPLE_DLL_H__
5
6   #ifdef SAMPLEDLL_EXPORTS
7   #define SAMPLEDLL_API __declspec(dllexport)
8   #else
9   #define SAMPLEDLL_API __declspec(dllimport)
10  #endif
11
```

```
12  #ifdef __cplusplus
13  extern "C" {
14  #endif
15
16  // функция PrintSysDirectories выводит в стандартный поток
17  // список указанных системных каталогов
18  SAMPLEDLL_API void PrintSysDirectories(const long csidl[],
    unsigned long nCount);
19
20  #ifdef __cplusplus
21  }
22  #endif
23
24  #endif /* __SAMPLE_DLL_H__ */
```

Как видно из этого примера (строки 6-10) в случае определения `SAMPLEDLL_EXPORTS` в проекте константа `SAMPLEDLL_API` определяется как **`__declspec(dllexport)`**; в случае же отсутствия такого определения константа `SAMPLEDLL_API` определяется как **`__declspec(dllimport)`**.

В Visual C++ с помощью ключевых слов `__declspec(dllexport)` и `__declspec(dllimport)` осуществляется экспорт и импорт данных, функций и классов в библиотеке DLL.

Константа `SAMPLEDLL_EXPORTS` определена по умолчанию в проекте SampleDLL (см. в меню **Проект** (**Project**) → **Свойства** (**Properties**), а затем **Свойства конфигурации** (**Configuration Properties**) → **C/C++** → **Препроцессор** (**Preprocessor**) → **Определения препроцессора** (**Preprocessor Definitions**), как показано на рис. 1.10).

Таким образом, один и тот же заголовочный файл SampleDLL.h может быть использован и в проекте SampleDLL, и в проекте, который будет использовать созданную библиотеку SampleDLL.dll.

Важно отметить то, что библиотеки DLL, созданные в Visual C++, отличаются от DLL в Windows. Первые можно использовать только в приложениях, написанных на языке C++, тогда как вторые – в приложениях, написанных не только на языке C++.

Дело в том, что компилятор Visual C++ осуществляет *декорирование имен* (name mangling) функций и методов классов. При этом имя каждой функции (или метода класса) кодируется с учетом формальных параметров, указанных в ее прототипе. Например, имя функции `PrintSysDirectories` будет преобразовано компилятором Visual C++ в `?PrintSysDirectories@@YAXQAJJ@Z`. Это позволяет компоновщику C++ различать перегруженные функции и методы класса.

Чтобы этого избежать, необходимо использовать модификатор **`extern "C"`** для всех экспортируемых функций – тогда компилятор не будет декорировать их имена.

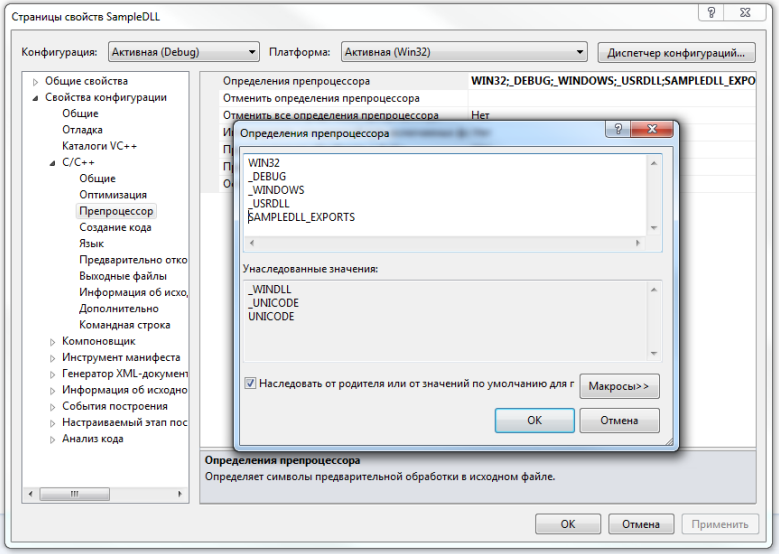

Рис. 1.10. Определения препроцессора Visual C++

6. Добавьте в проект исходный файл SampleDLL.cpp.
7. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 1.8 и сохраните этот файл.
8. В меню **Построение** (**Build**) выберите команду **Построить решение** (**Build Solution**).

Листинг 1.8. Файл исходного кода SampleDLL.cpp

```
1   #include <Windows.h>
2   #include <stdio.h>
3   #include <tchar.h>
4   #include <ShlObj.h>
5
6   #include "SampleDLL.h"
7
8   // ----------------------------------------------------------------
9   BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD dwReason, LPVOID
    lpvReserved)
10  {
```

```
11      return TRUE;
12  } // DllMain
13
14  // ------------------------------------------------------------
15  SAMPLEDLL_API void PrintSysDirectories(const long csidl[],
    unsigned long nCount)
16  {
17      TCHAR szBuffer[MAX_PATH + 1];
18
19      for (unsigned long i = 0; i < nCount; ++i)
20      {
21          HRESULT hr = SHGetFolderPath(NULL, csidl[i], NULL,
    SHGFP_TYPE_CURRENT, szBuffer);
22
23          if (S_OK == hr)
24              _tprintf(TEXT("%d: %s\n"), i+1, szBuffer);
25      } // for
26  } // PrintSysDirectories
```

Подключение DLL

Библиотеки DLL можно подключать к любым приложениям и другим библиотекам. Существует два способа подключения: явный и неявный. *Неявное подключение* DLL заключается в том, что при компоновке приложения используется библиотека импорта подключаемой DLL. *Библиотека импорта* (import library) автоматически создается компоновщиком при построении DLL и содержит названия и адреса экспортируемых функций и переменных, когда же сами функции и переменные содержаться в DLL. Файл библиотеки импорта называется так же, как и файл соответствующей библиотеки DLL, но имеет расширение .lib. Например, для библиотеки SampleDLL.dll библиотека импорта будет называться SampleDLL.lib.

Неявное подключение DLL применяется чаще всего, поскольку при этом не требуется дополнительного программного кода. Разработчику приложения достаточно просто подключить библиотеку импорта и необходимые заголовочные файлы (см. пример подключения статической библиотеки). Однако неявное подключение DLL имеет несколько ограничений:

- файл библиотеки DLL обязан иметь расширение .dll;
- библиотека DLL загружается, даже в том случае, если не используется никаких функций из этой библиотеки.

Хотя неявное подключение представляет собой самый простой способ использовать данные, функции, классы и методы классов, экс-

портируемые библиотекой DLL, в некоторых случаях предпочтительнее применять явное подключение. *Явное подключение* DLL заключается в использовании специальных функций Win32 API, позволяющих загружать и выгружать DLL, а также определять адрес функции или переменной, к которой необходимо обратиться. Подобный подход имеет следующие преимущества перед неявным подключением:

- файл библиотеки DLL может иметь любое расширение;
- библиотека DLL загружается только в том случае, когда она фактически используется.

Для загрузки и выгрузки DLL в Win32 API имеются, соответственно, функции `LoadLibrary` и `FreeLibrary`:

```
HMODULE LoadLibrary(LPCTSTR lpFileName);
BOOL FreeLibrary(HMODULE hModule);
```

Параметр *lpFileName* указывает на строку, которая именует загружаемый модуль (например, файл SampleDLL.dll), а параметр *hModule* представляет собой дескриптор загруженного модуля.

При успешном выполнении функция `LoadLibrary` возвращает дескриптор загруженного модуля; в случае ошибки – `NULL`.

В случае успеха функция `FreeLibrary` возвращает значение отличное `FALSE`. Следует отметить, что на самом деле функция `FreeLibrary` только уменьшает число ссылок на загруженный модуль. Когда же число ссылок достигнет нуля, модуль выгружается из адресного пространства приложения.

Для извлечения адреса экспортируемой функции или переменной из загруженной DLL используется функция `GetProcAddress`:

```
FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName);
```

Параметр *lpProcName* указывает на строку, которая определяет функцию или имя переменной. Если функция `GetProcAddress` завершается успешно, она возвращает адрес экспортируемой функции или переменной, в противном случае возвращается `NULL`.

В листинге 1.9 рассмотрен пример явного подключения библиотеки SampleDLL.dll и вызова функции `PrintSysDirectories`.

Листинг 1.9. Пример явного подключения DLL

```
1  #include <Windows.h>
2  #include <tchar.h>
3  #include <locale.h>
4  #include <ShlObj.h>
5
```

```
6   // определяем тип: указатель на функцию PrintSysDirectories
7   typedef void (*PRINT_SYSDIRECTORIES_PROC)(const long csidl[],
    unsigned long nCount);
8
9   int _tmain()
10  {
11      // массив идентификаторов системных каталогов
12      const long csidl[] = {
13          CSIDL_APPDATA,
14          CSIDL_COMMON_APPDATA,
15          CSIDL_LOCAL_APPDATA,
16          CSIDL_PERSONAL,
17          CSIDL_PROGRAM_FILES,
18          CSIDL_PROGRAM_FILES_COMMON,
19          CSIDL_SYSTEM,
20          CSIDL_WINDOWS
21      };
22
23      _tsetlocale(LC_ALL, TEXT(""));
24
25      // загружаем библиотеку SampleDLL.dll
26      HMODULE hDLL = LoadLibrary(TEXT("SampleDLL.dll"));
27
28      if (NULL != hDLL)
29      {
30          // извлекаем адрес функции PrintSysDirectories
31          PRINT_SYSDIRECTORIES_PROC func =
    (PRINT_SYSDIRECTORIES_PROC)GetProcAddress(hDLL,
    "PrintSysDirectories");
32          // если функция найдена, вызываем ее
33          if (NULL != func) func(csidl, _countof(csidl));
34
35          // выгружаем библиотеку SampleDLL.dll
36          FreeLibrary(hDLL);
37      } // if
38  } // _tmain
```

Точка входа в DLL

Когда DLL загружается в адресное пространство запущенного приложения, вызывается функция `_DllMainCRTStartup`, которая инициализирует различные библиотеки C/C++ и создает все глобальные переменные. Когда все это будет сделано, она вызывает функцию `DllMain` (если она существует) из загруженной библиотеки DLL.

Перед тем как DLL будет выгружена из адресного пространства, вновь вызывается функция `_DllMainCRTStartup`, которая снова вызовет

функцию DllMain (если она существует), и после этого удалит все глобальные переменные и освободит ресурсы, выделенные для различных библиотек C/C++.

Если в библиотеке DLL реализуется функция DllMain, она должна иметь следующий прототип.

```
BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD dwReason,
    LPVOID lpvReserved);
```

Первый параметр, *hInstDLL*, – дескриптор модуля DLL, который может использоваться при вызове функции, которые требуют дескриптор модуля.

Второй параметр, *dwReason*, указывает, почему вызывается функция DllMain. Этим параметром может быть одно из значений, перечисленных в табл. 1.13.

Таблица 1.13. Значения *dwReason*

Значение	**Описание**
DLL_PROCESS_ATTACH	DLL загружается в результате запуска приложения или в результате вызова функции LoadLibrary. Можно использовать этот момент, чтобы инициализировать какие-либо данные
DLL_THREAD_ATTACH	Текущий процесс создает новый поток. Когда это происходит, система вызывает функцию точки входа всех неявно подключенных DLL
DLL_THREAD_DETACH	Поток успешно завершил свою работу
DLL_PROCESS_DETACH	DLL выгружается в результате неудачной загрузки, завершения работы приложения или вызова функции FreeLibrary. Можно использовать этот момент, чтобы удалить какие-либо данные

Если значение параметра *dwReason* равно DLL_PROCESS_ATTACH, то функция DllMain должна возвращать TRUE в случае успеха и FALSE в случае неудачи. При этом если функция DllMain возвращает FALSE, то в случае неявного подключения DLL приложение тут же завершается, а в случае явного подключения библиотека DLL выгружается и функция LoadLibrary возвращает NULL.

Если значение параметра *dwReason* не равно DLL_PROCESS_ATTACH, то возвращаемое функцией DllMain значение игнорируется.

Третий параметр, *lpvReserved*, принимает значение равное NULL, если используется явное подключение DLL и не равное NULL, если используется неявное подключение.

Отложенная загрузка DLL

Visual C++ поддерживает *отложенную загрузку* DLL, которая значительно упрощает работу с библиотеками DLL. Отложенная загрузка DLL (delay-load DLL) – это неявное подключение библиотеки DLL, которая не загружается до тех пор, пока не используются функции из этой библиотеки.

Отложенная загрузка DLL корректно работает во всех версиях Windows и может быть полезна, например, если приложение использует какую-то новую функцию. Попытка запуска этой программы в более старой версии операционной системы может привести к ошибке, поскольку в этой системе может не быть нужной функции. Эта проблема легко решается с помощью отложенной загрузки DLL. Достаточно в процессе выполнения выяснить, что приложение работает в старой версии системы, и не вызывать новую функцию. Таким образом, если новая функция не будет вызвана, библиотека DLL загружаться не будет.

Следует отметить, что отложенная загрузка DLL кроме преимуществ имеет еще и несколько ограничений:

- не возможна отложенная загрузка kernel32.dll, так как эта библиотека необходима для вызова `LoadLibrary` и `GetProcAddress`;
- нельзя вызывать функции отложенной загрузки в `DllMain`;
- для выгрузки, отложено загруженной DLL, ни при каких условиях не следует вызывать `FreeLibrary`.

Рассмотрим, как использовать отложенную загрузку DLL на примере подключения библиотеки SampleDLL.dll. Первым делом, создайте (или откройте) проект приложения, а потом выполните следующие действия:

1. В меню **Проект** (**Project**) выберите **Свойства** (**Properties**).

   Откроется диалоговое окно **Страницы свойств** (**Property Pages**).

2. В открывшемся окне выберите **Свойства конфигурации** (**Configuration Properties**) → **Компоновщик** (**Linker**) → **Ввод** (**Input**).

3. Отредактируйте значение параметра **Дополнительные зависимости** (**Additional Dependencies**), указав в нем две библиотеки DelayImp.lib и SampleDLL.lib.

4. Отредактируйте значение параметра **Отложено загружаемые DLL** (**Delay Loaded DLLs**), указав в нем SampleDLL.dll, как показано на рис. 1.11.

5. Нажмите кнопку **OK**, чтобы сохранить настройки.

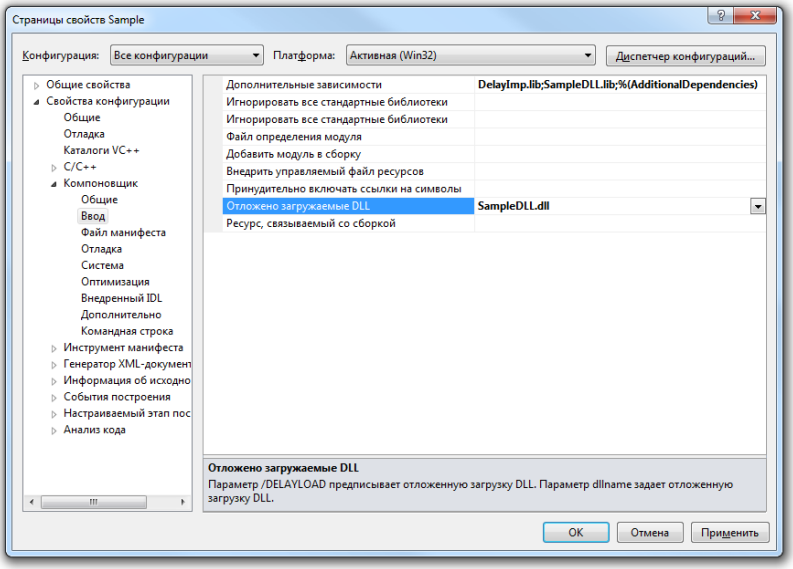

Рис. 1.11. Настройка отложено загружаемых DLL в Visual C++

Отложено загруженные DLL можно выгружать, если в них больше нет необходимости и нужно освободить системные ресурсы. Чтобы можно было выгружать отложено загружаемые DLL, нужно выполнить следующие действия:

1. В диалоговом окне **Страницы свойств** (**Property Pages**) выберите **Свойства конфигурации** (**Configuration Properties**) → **Компоновщик** (**Linker**) → **Дополнительно** (**Advanced**).
2. Отредактируйте значение параметра **Выгрузить отложено загружаемые DLL** (**Delay Loaded DLL**), выбрав в нем **`/DELAY:UNLOAD`**, как показано на рис. 1.12.
3. Нажмите кнопку **OK**, чтобы сохранить настройки.

Теперь чтобы выгрузить отложено загруженную DLL, нужно вызвать функцию `__FUnloadDelayLoadedDLL2`, прототип которой определен в заголовочном файле DelayImp.h:

```
BOOL __FUnloadDelayLoadedDLL2(LPCSTR szDll);
```

Параметр *`szDll`* указывает на строку, которая определяет имя отложено загруженной библиотеки DLL. В случае успеха функция возвращает значение отличное от `FALSE`.

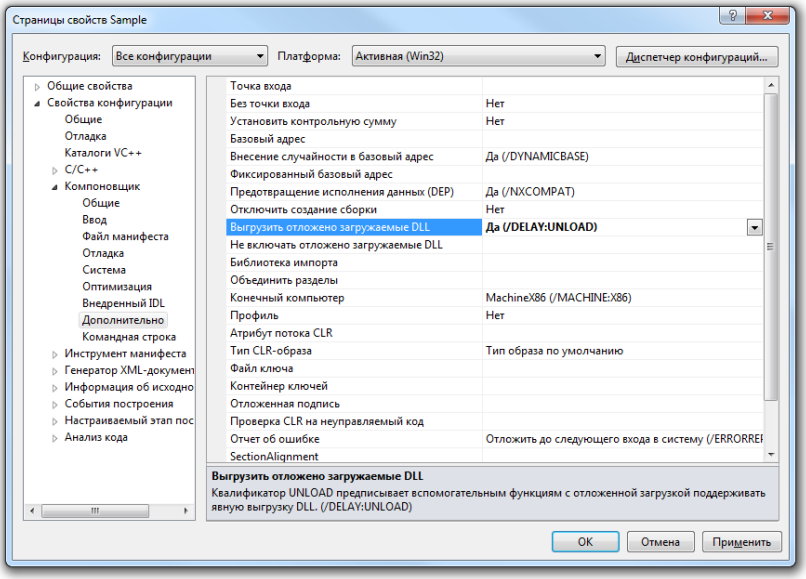

Рис. 1.12. Настройка отложено загружаемых DLL в Visual C++

В листинге 1.10 рассмотрен пример программного кода, в котором используется функция `PrintSysDirectories` (см. листинг 1.7) из библиотеки SampleDLL.dll, которая подключается с использованием отложенной загрузки DLL.

Листинг 1.10. Пример использования отложено загруженной DLL

```
1   const long csidl = CSIDL_WINDOWS;
2
3   // лучше помещать все вызовы функций отложено загруженных DLL
4   // в блоки __try __except
5   __try
6   {
7       // вызываем функцию PrintSysDirectories
8       // из библиотеки SampleDLL.dll (будет выполнена загрузка DLL)
9       PrintSysDirectories(&csidl, 1);
10
11      // вызываем функцию PrintSysDirectories
12      // из библиотеки SampleDLL.dll (DLL уже загружена)
13      PrintSysDirectories(&csidl, 1);
14  } // __try
15  __except (EXCEPTION_EXECUTE_HANDLER)
16  {
```

```
17      switch (GetLastError())
18      {
19      case ERROR_MOD_NOT_FOUND:
20          /* библиотека SampleDLL.dll не найдена */
21          break;
22      case ERROR_PROC_NOT_FOUND:
23          /* функция в библиотеке SampleDLL.dll не найдена */
24          break;
25      } // switch
26  } // __except
27
28  // выгружаем DLL, так как она более не нужна
29  // если DLL не была загружена, ничего не происходит
30  __FUnloadDelayLoadedDLL2("SampleDLL.dll");
```

Задание к работе

1. Разработать в Visual C++ статическую библиотеку, в которой будут реализованы функции вывода в стандартный поток следующей информации:
   - имя локального компьютера;
   - доменное имя, назначенное локальному компьютеру;
   - имя пользователя в текущем сеансе;
   - полное имя пользователя в текущем сеансе.
2. Разработать в Visual C++ библиотеку DLL, в которой будут реализованы функции вывода в стандартный поток следующей информации:
   - пути к системным каталогам, перечисленным в табл. 1.9;
   - версия операционной системы;
   - текущая дата и время (*выводятся согласно варианту задания*).
3. Изучить системные метрики и параметры, указанные в варианте задания. Включить в отчет назначение и описание изученных системных метрик и параметров.
4. Разработать в Visual C++ библиотеку DLL, в которой будут реализованы функции вывода в стандартный поток значений системных метрик, изученных в п. 3.
5. Разработать в Visual C++ библиотеку DLL, в которой будут реализованы функции вывода в стандартный поток значений системных параметров, изученных в п. 3.

6. Разработать в Visual C++ консольное приложение Win32, в котором будут использоваться функции, реализованные в следующих библиотеках:
   - статическая библиотека, созданная в п.1;
   - DLL, созданная в п.2 (*использовать неявное подключение*);
   - DLL, созданная в п.4 (*использовать явное подключение*);
   - DLL, созданная в п.5 (*использовать отложенную загрузку*).
7. Протестировать работу приложения, созданного в п. 6, на компьютере под управлением Windows XP (или выше). Результаты тестирования отразить в отчете.
8. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Формат даты	Формат времени	Системные метрики	Системные параметры
1,16	yyyy.MM.dd	hh:mm tt	SM_ARRANGE SM_CXHTHUMB SM_CYVTHUMB SM_CXSMICON SM_CYSMICON	SPI_GETACCESSTIMEOUT SPI_GETBEEP
2,17	yyyy-MM-dd	hh:mm:ss tt	SM_CLEANBOOT SM_CXICON SM_CYICON SM_CXSMSIZE SM_CYSMSIZE	SPI_GETFOCUSBORDERHEIGHT SPI_GETFOCUSBORDERWIDTH
3,18	yyyy/MM/dd	h:m t	SM_CMONITORS SM_CXICONSPACING SM_CYICONSPACING SM_CXVIRTUALSCREEN SM_CYVIRTUALSCREEN	SPI_GETMOUSEKEYS SPI_GETCONTACTVISUALIZATION
4,19	yyyy-M-d	h:m:s t	SM_CMOUSEBUTTONS SM_CXMAXIMIZED SM_CYMAXIMIZED SM_CXVSCROLL SM_CYVSCROLL	SPI_GETSERIALKEYS SPI_GETDEFAULTINPUTLANG
5,20	yyyy/M/d	HH:mm	SM_CXBORDER SM_CYBORDER SM_CXMAXTRACK SM_CYMAXTRACK SM_CYCAPTION	SPI_GETSHOWSOUNDS SPI_GETGESTUREVISUALIZATION

№	Формат даты	Формат времени	Системные метрики	Системные параметры
6,21	d.M.yyyy	HH:mm:ss	SM_CXCURSOR SM_CYCURSOR SM_CYMENU SM_CXMENUCHECK SM_CYMENUCHECK	SPI_GETSOUNDSENTRY SPI_GETKEYBOARDCUES
7,22	d-M-yyyy	H:m	SM_CXDLGFRAME SM_CYDLGFRAME SM_CXMENUSIZE SM_CYMENUSIZE SM_NETWORK	SPI_GETDROPSHADOW SPI_GETKEYBOARDPREF
8,23	d/M/yyyy	H:m:s	SM_CXDOUBLECLK SM_CYDOUBLECLK SM_CXMIN SM_CYMIN SM_REMOTECONTROL	SPI_GETFLATMENU SPI_GETKEYBOARDSPEED
9,24	dd.MM.yyyy	hh:mm tt	SM_CXDRAG SM_CYDRAG SM_CXMINIMIZED SM_CYMINIMIZED SM_REMOTESESSION	SPI_GETFONTSMOOTHING SPI_GETMOUSE
10,25	dd-MM-yyyy	hh:mm:ss tt	SM_CXEDGE SM_CYEDGE SM_CXMINSPACING SM_CYMINSPACING SM_SHOWSOUNDS	SPI_GETWORKAREA SPI_GETMOUSEWHEELROUTING
11,26	dd/MM/yyyy	h:m t	SM_CXFIXEDFRAME SM_CYFIXEDFRAME SM_CXMINTRACK SM_CYMINTRACK SM_STARTER	SPI_GETICONMETRICS SPI_GETSNAPTODEFBUTTON
12,27	MM/dd/yyyy	h:m:s t	SM_CXFOCUSBORDER SM_CYFOCUSBORDER SM_CXPADDEDBORDER SM_MOUSEPRESENT SM_MOUSEWHEELPRESENT	SPI_GETICONTITLELOGFONT SPI_GETWHEELSCROLLCHARS
13,28	M/d/yyyy	HH:mm	SM_CXFRAME SM_CYFRAME SM_CXSCREEN SM_CYSCREEN SM_CYSMCAPTION	SPI_GETICONTITLEWRAP SPI_GETWHEELSCROLLLINES

№	Формат даты	Формат времени	Системные метрики	Системные параметры
14,29	d MMMM yyyy	HH:mm:ss	SM_CXFULLSCREEN SM_CYFULLSCREEN SM_CXSIZE SM_CYSIZE SM_MIDEASTENABLED	SPI_ICONHORIZONTALSPACING SPI_GETMENUSHOWDELAY
15,30	d MMM yyyy	H:m	SM_CXHSCROLL SM_CYHSCROLL SM_CXSIZEFRAME SM_CYSIZEFRAME SM_MEDIACENTER	SPI_ICONVERTICALSPACING SPI_GETMINIMIZEDMETRICS

Контрольные вопросы

1. Что такое Windows API?
2. Чем Win32 API отличается от Win64 API?
3. В каких библиотеках находятся основные функции Win32 API?
4. Какие типы данных определены в Win32 API? Что такое дескриптор объекта?
5. Каким образом осуществляется обработка ошибок в Win32 API?
6. Каким образом осуществляется работа с символами и строками в Win32 API? Что такое безопасные строковые функции?
7. Какие в Win32 API имеются дополнительные возможности при работе со строками?
8. Какие функции Win32 API следует использовать для того, чтобы получить имя компьютера?
9. Какие функции Win32 API следует использовать для того, чтобы получить имя пользователя текущего сеанса?
10. Какую функцию Win32 API следует использовать для того, чтобы получить пути к различным каталогам Windows?
11. Какую функцию Win32 API следует использовать для того, чтобы определить версию Windows?
12. Какие функции Win32 API следует использовать для того, чтобы получить значения системных метрик и параметров Windows?
13. Какие функции Win32 API следует использовать при работе с системной датой и временем?
14. Что такое консольное приложение Win32? Как в Visual C++ создать проект консольного приложения Win32?

15. Для чего в консольном приложении Win32 предназначена функция `main`? Какие существуют варианты функции `main`?
16. Что такое точка входа в консольное приложение Win32?
17. Что называют статической библиотекой? Как в Visual C++ создать проект статической библиотеки?
18. Как в Visual C++ подключить статическую библиотеку к проекту Win32?
19. Что называют динамически подключаемой библиотекой (DLL)? Как в Visual C++ создать проект библиотеки DLL?
20. Как в Visual C++ подключить библиотеку DLL к проекту Win32? Что такое явное и неявное подключение DLL?
21. Что такое точка входа в DLL?
22. Что такое отложенная загрузка DLL? Как в Visual C++ применить отложенную загрузку DLL?

ЛАБОРАТОРНАЯ РАБОТА № 2
ОКОННЫЕ ПРИЛОЖЕНИЯ WINDOWS

Цель работы

Получение практических навыков создания оконных приложений Windows на языке C/C++ с применением Win32 API.

Основные понятия

Основой приложений Windows являются *окна*, взаимодействие между которыми осуществляется через *сообщения*. Сообщение является уведомлением о том, что произошло некоторое событие, которое может требовать, а может и не требовать выполнения определенных действий. Это событие может быть следствием действий пользователя, например, перемещение курсора или изменение размеров окна. Кроме того, событие может генерироваться приложением или же самой операционной системой.

Окна и элементы управления

Окно в Windows – это прямоугольная область экрана, где приложение отображает выводимую и принимает вводимую информацию от пользователя. В один и тот же момент времени только одно окно может принимать данные от пользователя.

При запуске Windows автоматически создается *окно рабочего стола* (desktop window), которое окрашивает фон экрана и является владельцем всех окон, создаваемых приложениями.

Каждое приложение создает, по крайней мере, одно окно, называемое *главным окном* (main window), которое служит основным окном программы. Главное окно относится к так называемым *перекрывающим окнам* (overlapped windows), которые способны перекрывать окна других приложений. Перекрывающие окна обычно включают область заголовка (title bar) и рамку (frame). Дополнительно такие окна могут иметь (а могут и не иметь) область меню (menu bar), системное меню (system menu) и кнопки управления окном: кнопка свертывания окна (minimize button), кнопка развертывания окна (maximize button), кнопка восстановления окна (restore button) и кнопка закрытия окна (close button).

В дополнение к главному окну приложения также используют диалоговые окна и элементы управления. *Диалоговые окна*, или *окна диалога* (dialog box), – это *всплывающие окна* (pop-up windows), кото-

рые используются для получения от пользователя дополнительной информации, а также вывода результатов работы приложения. Всплывающие окна могут иметь область заголовка и рамку окна.

Элементами управления (controls) являются *дочерние окна* (child windows), предназначенные для отображения и редактирования различной информации, а также для выполнения пользователем определенных действий.

Каждое окно содержит *клиентскую область* (client area), в которой можно выводить текст и графику. Окно также может иметь (а может и не иметь) *не клиентскую область* (nonclient area), которая включает в себя область заголовка, область меню и рамку окна. Управлением клиентской областью окна занимается приложение, не клиентской областью – операционная система.

Системы координат

Прежде чем перейти к дальнейшему рассмотрению, следует отметить, что в функциях Win32 API, работающих с окнами, может использоваться одна из следующих систем координат:

- экранные координаты (screen coordinates);
- координаты клиентской области (client coordinates).

На рис. 2.1 показано главное окно программы на экране, а также взаимоотношение между системами координат.

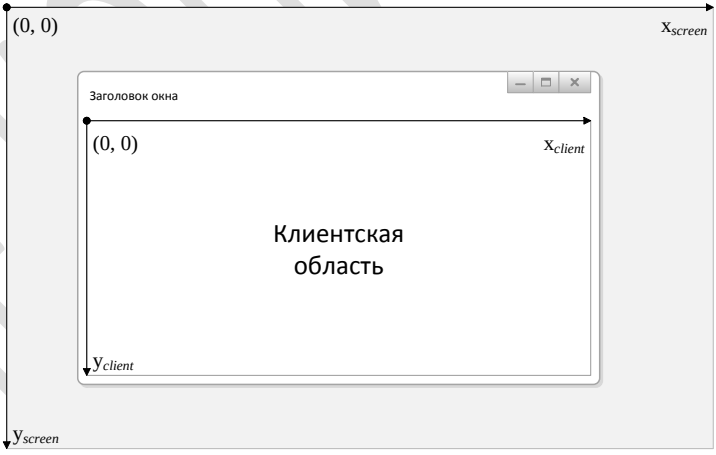

Рис. 2.1. Системы координат Windows

Для преобразования координат клиентской области окна в экранные координаты в Win32 API есть функция ClientToScreen:

```
BOOL ClientToScreen(HWND hWnd, LPPOINT lpPoint);
```

Здесь параметр *hWnd* содержит дескриптор окна, а параметр *lpPoint* указывает на переменную типа POINT, в которой хранятся координаты клиентской области. В случае успеха функция возвращает значение отличное от FALSE и записывает результат в переменную, на которую указывает *lpPoint*.

Обратное преобразование выполняет функция ScreenToClient:

```
BOOL ScreenToClient(HWND hWnd, LPPOINT lpPoint);
```

Она очень похожа на ClientToScreen. Здесь параметр *hWnd* также содержит дескриптор окна, а параметр *lpPoint* указывает на переменную типа POINT, в которой хранятся экранные координаты. В случае успеха функция возвращает значение отличное от FALSE и записывает результат в переменную, на которую указывает *lpPoint*.

Оконный класс

Любое окно в Windows создается на основе оконного класса. *Оконный класс* (window class) – это структура, определяющая основные характеристики окна. К ним относятся стиль класса и связанные с ним ресурсы, такие как меню (menu), пиктограмма (icon), курсор (cursor) и кисть (brush) для закрашивания фона окна. Кроме того, эта структура содержит адрес функции, предназначенной для обработки сообщений любого окна данного класса. (Не следует путать понятие *оконного класса Windows* с понятием *класса C++*.)

Для создания нового оконного класса необходимо заполнить структуру WNDCLASSEX, а затем передать адрес этой структуры в виде аргумента функции RegisterClassEx.

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

Функция RegisterClassEx это расширенная версия функции RegisterClass из предыдущих версий Windows. Можно пользоваться и функцией RegisterClass, передавая ей адрес структуры WNDCLASS. В случае успеха функции RegisterClassEx и RegisterClass возвращают не нулевое значение.

Структура WNDCLASSEX определена следующим образом:

```
typedef struct tagWNDCLASSEX {
    UINT cbSize; // размер структуры в байтах
    UINT style; // стиль оконного класса
    WNDPROC lpfnWndProc; // указатель на оконную процедуру
```

```
    int cbClsExtra; // число дополнительных байтов,
                    // которые должны быть распределены
                    // в конце структуры
    int cbWndExtra; // число дополнительных байтов,
                    // которые должны быть распределены
                    // вслед за экземпляром окна
    HINSTANCE hInstance; // дескриптор экземпляра приложения,
                         // в котором находится оконная
                         // процедура для этого класса
    HICON hIcon; // дескриптор пиктограммы
    HCURSOR hCursor; // дескриптор курсора
    HBRUSH hbrBackground; // дескриптор кисти, используемой
                          // для закраски фона окна
    LPCTSTR lpszMenuName; // указатель на строку, содержащую
                          // имя меню, применяемого
                          // для этого класса
    LPCTSTR lpszClassName; // указатель на строку, содержащую
                           // имя этого класса
    HICON hIconSm; // дескриптор малой пиктограммы
} WNDCLASSEX;
```

Первое поле, *cbSize*, должно содержать значение равное размеру структуры в байтах, как правило, его получают с помощью оператора `sizeof`.

Второе поле, *style*, может содержать один или несколько стилей, объединенных с помощью операции поразрядного «ИЛИ» языка C/C++. Наиболее употребительные стили оконного класса перечисленных в табл. 2.1. Полный список таких стилей см. в документации Platform SDK.

Таблица 2.1. Некоторые стили оконного класса

Стиль	**Описание**
CS_DBLCLKS	Оконная процедура будет получать сообщения при двойном щелчке клавишей мыши (double click)
CS_GLOBALCLASS	Создается оконный класс, доступный всем приложениям. Другие приложения могут создавать окна этого класса
CS_HREDRAW	Клиентская область окна этого класса будет перерисовываться при изменении ширины окна
CS_NOCLOSE	Запретить команду «Закрыть» в системном меню
CS_OWNDC	Выделить уникальный контекст устройства для каждого окна, созданного при помощи этого класса
CS_VREDRAW	Клиентская область окна этого класса будет перерисовываться при изменении высоты окна

Третье поле, *lpfnWndProc*, содержит адрес оконной процедуры – функции, предназначенной для обработки сообщений любого окна данного класса (подробнее см. в разделе «Оконные сообщения»).

Поля *cbClsExtra* и *cbWndExtra* используются крайне редко. Подробнее см. в документации Platform SDK.

Шестое поле, *hInstance*, содержит дескриптор экземпляра приложения, в котором находится оконная процедура для оконного класса.

Седьмое поле *hIcon* содержит дескриптор пиктограммы, которая появляется на панели задач Windows. Значение *hIcon* обычно получают с помощью функции LoadIcon:

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

Эта функция загружает указанный ресурс пиктограммы, заданный параметром *lpIconName*, из экземпляра приложения, на который указывает параметр *hInstance*. Чтобы загрузить одну из системных (предопределенных) пиктограмм, аргумент *hInstance* должен быть NULL, а параметр *lpIconName* должен содержать константу, идентификатор которой начинается с префикса IDI_ (например, IDI_APPLICATION).

Восьмое Поле *hCursor* содержит дескриптор курсора мыши, используемого приложением в клиентской области окна. Значение *hCursor* обычно получают вызовом функции LoadCursor:

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

Эта функция загружает ресурс курсора, на который указывает параметр *lpCursorName*, из экземпляра приложения, заданного параметром *hInstance*. Чтобы загрузить один из системных (предопределенных) курсоров, аргумент *hInstance* должен быть NULL, а параметр *lpCursorName* должен содержать константу, идентификатор которой начинается с префикса IDC_ (например, IDC_ARROW).

Девятое поле, *hbrBackground*, содержит дескриптор кисти, которая должна использоваться для закрашивания фона клиентской области окна. На самом деле, значение этого поля может быть, как дескриптором кисти, так и кодом цвета. Код цвета задается константой одного из системных цветов, к которой добавляться 1. Код цвета обязательно должен приводиться к типу HBRUSH. Константы системных цветов начинаются с префикса COLOR_ (например, COLOR_WINDOW, которая соответствует цвету фона окна).

Если значение поля *hbrBackground* равно NULL, приложение должно обрабатывать сообщение WM_ERASEBKGND самостоятельно закрашивая фон клиентской области окна.

Десятое поле, *lpszMenuName*, указывает на строку, содержащую имя ресурса меню, применяемого для всех окон данного класса. Если для

идентификации ресурса меню используется целое число, нужно использовать макрос MAKEINTRESOURCE. Если значение этого поля равно NULL, окна этого класса по умолчанию не имеют никакого меню.

Поле *lpszClassName* указывает на строку, содержащую имя оконного класса. Имена оконных классов должны быть уникальны только в пределах одного запущенного приложения.

Последнее поле, *hIconSm*, содержит дескриптор маленькой пиктограммы, которая отображается в области заголовка окна. Значение поля *hIconSm* получают также как и значение поля *hIcon* – с помощью функции LoadIcon. Кроме того, если поле *hIconSm* установлено в NULL, система ищет ресурс пиктограммы, указанный в поле *hIcon*.

В листинге 2.1 приводится пример регистрации нового оконного класса «MyWindowClass».

Листинг 2.1. Пример регистрации оконного класса

```
1   WNDCLASSEX wcex = { sizeof(WNDCLASSEX) };
2
3   wcex.style = CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS;
4   wcex.lpfnWndProc = WindowProc;
5   wcex.hInstance = hInstance;
6   wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
7   wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
8   wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
9   wcex.lpszMenuName = NULL;
10  wcex.lpszClassName = TEXT("MyWindowClass");
11  wcex.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
12
13  if (0 != RegisterClassEx(&wcex))
14  {
15      /* оконный класс успешно зарегистрирован! */
16  } // if
```

Оконный класс удаляется, когда приложение, в котором он был зарегистрирован, завершается. Приложение может также использовать функцию UnregisterClass, чтобы удалить оконный класс и освободить ресурсы, связанные с ним.

```
BOOL UnregisterClass(LPCTSTR lpClassName, HINSTANCE hInstance);
```

Первый параметр, *lpClassName*, указывает на строку, содержащую имя оконного класса, а второй параметр, *hInstance*, содержит дескриптор экземпляра приложения, в котором находится оконная процедура для оконного класса.

В случае неудачи функция UnregisterClass возвращает FALSE.

Предопределенные оконные классы

В Win32 API имеются предопределенные оконные классы *элементов управления общего пользования* (common controls). Большинство из них реализовано в библиотеке ComCtl32.dll. В табл. 2.2 перечислены элементы управления общего пользования и соответствующие им оконные классы.

Таблица 2.2. Элементы управления общего пользования

Элемент управления	**Оконный класс**	**Префикс флага стиля**
Анимационное изображение (Animate)	SysAnimate32	ACS_
Выпадающий список (Combo box)	ComboBox ComboBoxEx32	CBS_ CBES_
Гиперссылка (SysLink)	SysLink	LWS_
Групповая рамка (Group box)	Button	BS_
Дерево просмотра (Tree view)	SysTreeView32	TVS_
Заголовок списка просмотра (Header)	SysHeader32	HDS_
Закладки (Tab control)	SysTabControl32	TCS_
Индикатор процесса (Progress bar)	msctls_progress32	PBS_
Календарь (Month calendar)	SysMonthCal32	MCS_
Кнопка (Button)	Button	BS_
Кнопка со стрелкой раскрывающегося списка (Split button)	Button	BS_
Надпись (Static text)	Static	SS_
Окно подсказки (Tooltip)	tooltips_class32	TTS_
Панель инструментов (Toolbar)	ToolbarWindow32	TBSTYLE_
Переключатель (Radio button)	Button	BS_
Поле ввода IP-адреса (IP address)	SysIPAddress32	
Поле ввода горячей клавиши (Hot key)	msctls_hotkey32	
Поле ввода даты или времени (Data time picker)	SysDateTimePick32	DTS_
Полоса прокрутки (Scroll bar)	ScrollBar	SBS_
Регулятор (Slider)	msctls_trackbar32	TBS_
Редактируемое поле (Edit box)	Edit	ES_
Список (List box)	ListBox	LBS_
Список просмотра (List view)	SysListView32	LVS_
Строка состояния (Status bar)	msctls_statusbar32	SBARS_

Окончание табл. 2.2.

Элемент управления	Оконный класс	Префикс флага стиля
Счетчик или стрелки (Spin)	`msctls_updown32`	`UDS_`

Текстовый редактор (Rich edit)	`RichEdit`
	`RichEdit20A`
	`RichEdit20W`
	`RichEdit50W`
		`ES_`

Флажок (Check box)	`Button`	`BS_`

Для полноты картины в табл. 2.3 перечислены еще несколько предопределенных оконных классов, которые доступны только операционной системе и окна этих классов нельзя создать в приложении.

Таблица 2.3. Системные оконные классы

Оконный класс	Описание
`ComboLBox`	Окно списка, содержащегося в элементе управления «выпадающий список» (combo box)
`#32768`	Окно меню
`#32769`	Окно рабочего стола (desktop window)
`#32770`	Диалоговое окно
`#32771`	Окно переключения задач (task switch window)

В документации Platform SDK сказано, чтобы использовать оконные классы элементов управления общего пользования (кроме классов `RichEdit*`) следует вызвать функцию `InitCommonControlsEx`, которая регистрирует оконные классы этих элементов управления. Эта функция определена в заголовочном файле CommCtrl.h и реализована в библиотеке ComCtl32.dll.

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

Параметр *`lpInitCtrls`* передает адрес структуры `INITCOMMONCONTROLSEX`, содержащей информацию о том, какие классы элементов управления должны быть зарегистрированы. Однако, несмотря на этот параметр, функция `InitCommonControlsEx` регистрирует все доступные оконные классы.

Если загрузить библиотеку ComCtl32.dll при помощи вызова функции `LoadLibrary`:

```
LoadLibrary(TEXT("ComCtl32.dll"));
```

то функция `DllMain` этой библиотеки регистрирует оконные классы всех реализованных в ней элементов управления. Следовательно, нет необходимости вызывать еще и функцию `InitCommonControlsEx`.

В Windows Vista и более новых версиях, библиотека ComCtl32.dll автоматически загружается при первом создании элемента управления из этой библиотеки.

Чтобы использовать элемент управления «текстовый редактор» (rich edit) также нужно загрузить DLL при помощи вызова функции `LoadLibrary`. В Windows имеется несколько версий текстового редактора, каждый из которых реализован в разных библиотеках DLL. В табл. 2.4 показано, какие библиотеки DLL соответствуют какой версии текстового редактора.

Таблица 2.4. Библиотеки DLL для разных версий Rich edit

Версия	DLL	Оконный класс
1.0	Riched32.dll	`RichEdit`
2.0	Riched20.dll	`RichEdit20A`, `RichEdit20W`
3.0	Riched20.dll	`RichEdit20A`, `RichEdit20W`
4.1	Msftedit.dll	`RichEdit50W`

Следует обратить внимание на то, что от версии 2.0 к версии 3.0 оконный класс и имя файла DLL не изменились. Это позволяет приложениям, изначально созданным для использования текстового редактора версии 2.0, использовать редактор версии 3.0, не изменяя программный код.

Таким образом, для работы с текстовым редактором, например, версии 4.1 нужно подключить заголовочный файл Richedit.h и загрузить библиотеку Msftedit.dll:

```
LoadLibrary(TEXT("Msftedit.dll"));
```

Создание окна

Для создания окна определенного оконного класса вызывается функция `CreateWindowEx`. Функция `CreateWindowEx` это расширенная версия функции `CreateWindow`, которую тоже можно использовать.

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

Первый параметр, *dwExStyle*, задает расширенный стиль окна, применяемый совместно со стилем, определенным в параметре

dwStyle. Например, в качестве расширенного стиля можно задать один или несколько флагов, определенных в табл. 2.5. Полный список расширенных стилей можно найти в документации Platform SDK.

Таблица 2.5. Расширенные стили окна

Стиль	**Описание**
WS_EX_ACCEPTFILES	Создается окно, способное принимать перетаскиваемые файлы методом drag-and-drop
WS_EX_CLIENTEDGE	Клиентская область создаваемого окна будет несколько углублена
WS_EX_CONTEXTHELP	Создаваемое окно включает кнопку вопросительного знака в область заголовка окна. (Этот стиль не может использоваться с WS_MINIMIZEBOX и WS_MAXIMIZEBOX.) При нажатии на эту кнопку курсор принимает форму вопросительного знака, и если затем пользователь выполняет нажатие над областью дочернего окна, то окно получает сообщение WM_HELP
WS_EX_STATICEDGE	Создается окно с объемным бордюром (используется для элементов управления, не принимающих ввод пользователя)
WS_EX_TOPMOST	Создается окно, которое будет помещено поверх других окон, даже если окно неактивно
WS_EX_WINDOWEDGE	Создается окно с выпуклой границей за счет некоторого углубления клиентской области

Параметр *lpClassName* – указатель на строку, содержащую допустимое имя оконного класса, на основе которого создается окно. Таким именем может быть имя оконного класса, зарегистрированного при помощи функции RegisterClassEx или RegisterClass, либо имя предопределенного оконного класса (см. табл. 2.2).

Третий параметр, *lpWindowName*, – указатель на строку, содержащую имя окна. Место отображения этого имени зависит от вида окна. Например, для главного окна приложения имя выводится как заголовок окна, а для окна класса Edit размещается в клиентской области.

Параметр *dwStyle* задает стиль окна, который может состоять из значений, указанных в табл. 2.6. Для элементов управления общего пользования дополнительно могут быть добавлены стили, идентификаторы которых начинаются с префикса, указанного в табл. 2.2. Полный список стилей можно найти в документации Platform SDK.

Параметры *X* и *Y* определяют позицию верхнего левого угла окна. Для перекрывающего окна эти параметры определяются в экранных координатах, а для дочерних окон – в клиентских координатах роди-

тельского окна. Если позиция окна не важна, то можно установить значение параметра *X* равное CW_USEDEFAULT. В этом случае операционная система сама определяет положение окна. Если параметр *X* имеет значение CW_USEDEFAULT, то значение *Y* игнорируется.

Таблица 2.6. Стили окна

Стиль	**Описание**
WS_BORDER	Создается окно с тонкой рамкой
WS_CAPTION	Создается окно с областью заголовка
WS_CHILD	Создается дочернее окно (не допускается совместное использование с WS_POPUP)
WS_CLIPCHILDREN	В создаваемом окне клиентская область, занятая дочерним окном, исключается при перерисовке
WS_CLIPSIBLINGS	При перерисовке создаваемого дочернего окна все остальные дочерние окна, накрадывающиеся на клиентскую область окна, будут отсечены при перерисовке
WS_HSCROLL	Создается окно с горизонтальной полосой прокрутки
WS_MAXIMIZEBOX	Создается окно с кнопкой развертывания
WS_MINIMIZEBOX	Создается окно с кнопкой свертывания
WS_OVERLAPPED	Создается перекрывающее окно, которое будет иметь область заголовка и рамку
WS_OVERLAPPEDWINDOW	Сочетание стилей WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX
WS_POPUP	Создается всплывающее окно (не допускается совместное использование с WS_CHILD)
WS_POPUPWINDOW	Сочетание стилей WS_BORDER, WS_POPUP и WS_SYSMENU. Чтобы сделать системное меню видимым, необходимо добавить стиль WS_CAPTION
WS_SYSMENU	Создается окно с системным меню в области заголовка
WS_THICKFRAME	Создается окно с рамкой, которая позволяет изменять его размеры
WS_VISIBLE	Создается окно, которое сразу же является видимым
WS_VSCROLL	Создается окно с вертикальной полосой прокрутки

Параметры *nWidth* и *nHeight* определяют соответственно ширину и высоту окна в пикселях. Если параметр *nWidth* имеет значение CW_USEDEFAULT, то значение *nHeight* игнорируется и операционная система сама определяет размер окна.

Девятый параметр, *hWndParent*, содержит дескриптор родительского окна (окна-владельца). Следует заметить, что если между окнами

существует связь *родительское – дочернее*, дочернее окно всегда отображается только на поверхности родительского окна. Если окно не имеет родительского окна (обычно это главное окно приложения), параметр *hWndParent* должен быть установлен в NULL.

Десятый параметр, *hMenu*, содержит дескриптор меню или идентификатор элемента управления, который содержится в сообщениях родительскому окну, поступающих от элемента управления. Интерпретация значения этого параметра зависит от вида окна.

Параметр *hInstance* – дескриптор экземпляра приложения.

Последний параметр, *lpParam*, может быть использован для передачи окну дополнительных данных в момент его создания. Если параметр *lpParam* не используется, он должен принимать значение NULL.

В случае успеха функция CreateWindowEx возвращает дескриптор созданного окна. Если по какой-то причине создать окно не удалось, то функция возвращает NULL.

В листинге 2.2 представлен пример создания главного окна и двух элементов управления, как показано на рис. 2.2.

Листинг 2.2. Пример создания окна с элементами управления

```
1   // создание окна
2   HWND hWnd = CreateWindowEx(0,
3       TEXT("MyWindowClass"), TEXT("Главное окно"),
4       WS_OVERLAPPEDWINDOW,
5       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
6       NULL, NULL, hInstance, NULL);
7
8   if (NULL != hWnd) // если окно успешно создано
9   {
10      // создание элемента управления для созданного окна
11      CreateWindowEx(WS_EX_CLIENTEDGE,
12          TEXT("Edit"), TEXT("Редактируемое поле"),
13          ES_AUTOHSCROLL|ES_LEFT|WS_CHILD|WS_VISIBLE,
14          10, 10, 200, 25,
15          hWnd, (HMENU)IDC_EDIT1, hInstance, NULL);
16
17      // создание элемента управления для созданного окна
18      CreateWindowEx(0,
19          TEXT("Button"), TEXT("Кнопка"),
20          BS_CENTER|BS_SPLITBUTTON|WS_CHILD|WS_VISIBLE,
21          220, 10, 100, 25,
22          hWnd, (HMENU)IDC_BUTTON1, hInstance, NULL);
23  } // if
```

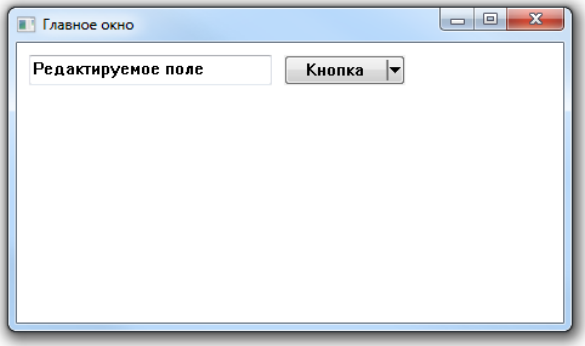

Рис. 2.2. Создание окна с элементами управления

Обратите внимание на то, что в этом примере для идентификации элементов управления используются две константы `IDC_EDIT1` и `IDC_BUTTON1`, которые могут быть определены следующим образом:

```
1   #define IDC_EDIT1       2001
2   #define IDC_BUTTON1     2002
```

Для некоторых элементов управления общего пользования существует альтернатива вызову функции `CreateWindowEx` – это вызов специальной функции создания элемента управления, которая в тоже время может выполнять некоторую инициализацию. В табл. 2.7 приведены такие функции для некоторых элементов управления.

Таблица 2.7. Элементы управления и функции их создания

Элемент управления	**Функция создания**
Панель инструментов (Toolbar)	`CreateToolbarEx`
Строка состояния (Status bar)	`CreateStatusWindow`
Счетчик или стрелки (Spin)	`CreateUpDownControl`

Диалоговые окна

Различают *модальные* (modal) и *немодальные* (modeless) диалоговые окна. Как правило, модальные диалоговые окна используются чаще, чем немодальные.

Модальным называется окно, которое не позволяет пользователю работать с другими окнами приложения до тех пор, пока работа с модальным окном не будет завершена. Однако пользователь может работать с окнами других приложений. Немодальные диалоговые окна не требуют своего завершения для работы с другими окнами. Это значит,

что пользователь может свободно переключаться между диалоговым окном и другими окнами приложения.

Создание диалогового окна

Диалоговое окно создается с использованием шаблона, который можно определить в файле ресурсов приложения, используя редактор диалоговых окон Visual Studio. Для определения шаблона диалогового окна нужно выполнить следующие действия:

1. В окне **Обозреватель решений** (**Solution Explorer**) выберите нужный проект.
2. В меню **Проект** (**Project**) выберите **Добавить ресурс** (**Add Resource**).

   Откроется диалоговое окно **Добавление ресурса** (**Add Resource**).
3. В области **Тип ресурса** (**Resource type**) выберите **Dialog**.

   Нажмите кнопку **Создать** (**New**).
4. В открывшемся редакторе Visual Studio (рис. 2.3), отредактируйте и сохраните созданный шаблон диалогового окна.

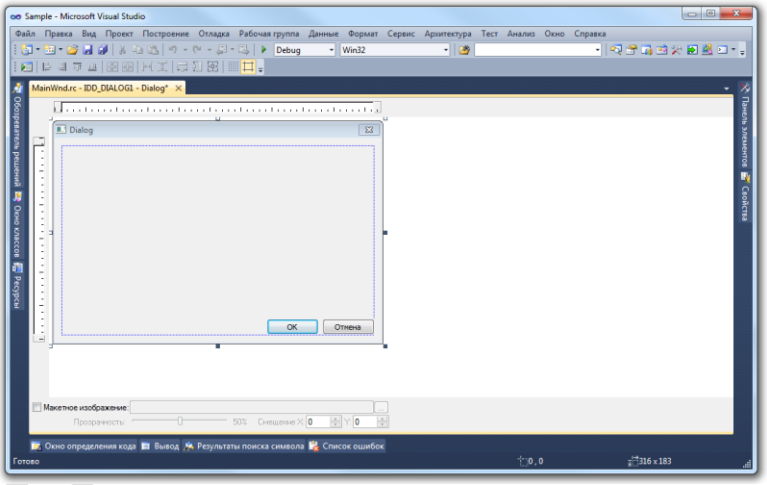

Рис. 2.3. Шаблон диалогового окна в редакторе Visual Studio

В Win32 API имеются следующие функции для создания диалогового окна:

```
INT_PTR DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate,
    HWND hWndParent, DLGPROC lpDialogFunc);
```

```
INT_PTR DialogBoxParam(HINSTANCE hInstance,
    LPCTSTR lpTemplate, HWND hWndParent,
    DLGPROC lpDialogFunc, LPARAM dwInitParam);

HWND CreateDialog(HINSTANCE hInstance, LPCTSTR lpTemplate,
    HWND hWndParent, DLGPROC lpDialogFunc);

HWND CreateDialogParam(HINSTANCE hInstance,
    LPCTSTR lpTemplate, HWND hWndParent,
    DLGPROC lpDialogFunc, LPARAM dwInitParam);
```

Первый параметр, *hInstance*, задает дескриптор модуля, который содержит шаблон диалогового окна.

Второй параметр, *lpTemplate*, определяет шаблон диалогового окна. Чтобы задать этот параметр, можно использовать макрос MAKEINTRESOURCE, в котором следует указать идентификатор ресурса шаблона диалогового окна.

Третий параметр, *hWndParent*, – дескриптор окна, которое будет владеть диалоговым окном.

Четвертый параметр, *lpDialogFunc*, указывает на процедуру диалогового окна, которая обрабатывает отправляемые окну сообщения (подробнее см. в разделе «Оконные сообщения»).

Параметр *dwInitParam* устанавливает значение, передаваемое в процедуру диалогового окна при инициализации диалогового окна.

Первые две функции, DialogBox и DialogBoxParam, создают модальное диалоговое окно и возвращают управление только после закрытия этого окна. При успешном завершении эти функции возвращают значение, указанное при завершении работы диалогового окна, в противном случае 0 или -1. Если параметр *hWndParent* определяет недопустимое значение, возвращаемое значение равно нулю.

Функции CreateDialog и CreateDialogParam создают не модальное диалоговое окно и возвращают управление сразу же. В случае успеха возвращается дескриптор созданного диалогового окна, а в случае ошибки – NULL.

Диалоговые окна созданные, с помощью функции DialogBox или DialogBoxParam, должны уничтожаться функцией EndDialog, вызываемой из процедуры диалогового окна.

```
BOOL EndDialog(HWND hDlg, INT_PTR nResult);
```

Первый параметр, *hDlg*, – дескриптор диалогового окна. Второй параметр, *nResult*, определяет значение, возвращаемое функцией, которая создала диалоговое окно.

Если функция EndDialog завершается успешно, возвращается значение отличное от FALSE.

В следующем примере показано создание модального диалогового окна, с помощью функции `DialogBox`.

Листинг 2.3. Пример создания модального диалогового окна

```
INT_PTR nDlgRes = DialogBox(hInstance,
    MAKEINTRESOURCE(IDD_DIALOG1), // идентификатор шаблона
    hWnd, // дескриптор окна владельца
    DialogProc); // процедура диалогового окна

if (IDOK == nDlgRes)
{
    /* функция DialogBox вернула значение IDOK */
} // if
```

Окно сообщений

Простейшим типом диалогового окна является окно сообщений. Окно сообщений содержит заголовок, текст сообщения, а также сочетание предопределенных пиктограмм и кнопок. Создавать окно сообщений позволяет функция `MessageBox`:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption,
    UINT uType);
```

Первый параметр, *`hWnd`*, задает дескриптор окна-владельца. Если этот параметр установлен в `NULL`, диалоговое окно не имеет владельца.

Второй параметр, *`lpText`*, – указатель на строку, которая содержит текст выводимого сообщения. Можно использовать управляющий символ «\n», если необходимо сделать сообщение многострочным.

Параметр *`lpCaption`* – указатель на строку, которая содержит заголовок диалогового окна. Если этот параметр установлен в `NULL`, используется заданный по умолчанию заголовок «Ошибка» («Error»).

Последний параметр, *`uType`*, определяет внешний вид и режим работы диалогового окна. Этот параметр может принимать одно или несколько значений, перечисленных в табл. 2.8 – 2.11.

Для указания состава отображаемых кнопок используются значения, перечисленные в табл. 2.8.

Таблица 2.8. Флаги, определяющие состав кнопок

Флаг	**Описание**
`MB_ABORTRETRYIGNORE`	Диалоговое окно содержит три кнопки: **Прекратить** (**Abort**), **Повторить** (**Retry**) и **Пропустить** (**Ignore**)
`MB_CANCELTRYCONTINUE`	Диалоговое окно содержит три кнопки: **Отменить** (**Cancel**), **Повторить** (**Retry**) и **Продолжить** (**Continue**)

Окончание табл. 2.8.

Флаг	Описание
MB_HELP	К кнопкам диалогового окна добавляется кнопка **Справка** (**Help**). Когда пользователь щелкает по этой кнопке, окно-владелец получает сообщение WM_HELP, а диалоговое окно продолжает работу
MB_OK	Диалоговое окно содержит кнопку **OK**. Это значение используется по умолчанию
MB_OKCANCEL	Диалоговое окно содержит две кнопки: **OK** и **Отменить** (**Cancel**)
MB_RETRYCANCEL	Диалоговое окно содержит две кнопки: **Повторить** (**Retry**) и **Отменить** (**Cancel**)
MB_YESNO	Диалоговое окно содержит две кнопки: **Да** (**Yes**) и **Нет** (**No**)
MB_YESNOCANCEL	Диалоговое окно содержит три кнопки: **Да** (**Yes**), **Нет** (**No**) и **Отменить** (**Cancel**)

Чтобы указать отображаемую пиктограмму, используются значения, перечисленные в табл. 2.9.

Таблица 2.9. Флаги, определяющие отображаемую пиктограмму

Флаг	Пиктограмма
MB_ICONEXCLAMATION MB_ICONWARNING	
MB_ICONINFORMATION MB_ICONASTERISK	
MB_ICONASTERISK MB_ICONQUESTION	
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	

Для указания кнопки по умолчанию, используются значения из табл. 2.10. *Кнопка по умолчанию* (default button) нажимается автоматически при нажатии клавиши «Enter» или «Пробел».

Таблица 2.10. Флаги, определяющие кнопку по умолчанию

Флаг	Описание
MB_DEFBUTTON1	Первая кнопка
MB_DEFBUTTON2	Вторая кнопка

Окончание табл. 2.10.

Флаг	Описание
MB_DEFBUTTON3	Третья кнопка
MB_DEFBUTTON4	Четвертая кнопка

Чтобы указать правила поведения диалогового окна, используются значения, перечисленные в табл. 2.11. Есть и другие, реже применяемые флаги, которые перечислены в документации Platform SDK.

Таблица 2.11. Флаги, определяющие поведение

Флаг	Описание
MB_APPLMODAL	Окно-владелец блокируется до тех пор, пока работа диалогового окна не завершена. Это значение используется по умолчанию
MB_SYSTEMMODAL	Сочетание MB_APPLMODAL и MB_TOPMOST
MB_TASKMODAL	То же самое, что и MB_APPLMODAL за исключением того, что блокируются все окна приложения, если диалоговое окно не имеет владельца
MB_TOPMOST	Диалоговое окно всегда располагается поверх остальных окон, даже если оно неактивно

Если функция MessageBox завершается ошибкой, возвращаемое значение равно нулю. В случае успеха функция возвращает одно из следующих значений:

- IDABORT – была нажата кнопка **Прекратить** (**Abort**);
- IDCANCEL – была нажата кнопка **Отменить** (**Cancel**);
- IDCONTINUE – была нажата кнопка **Продолжить** (**Continue**);
- IDIGNORE – была нажата кнопка **Пропустить** (**Ignore**);
- IDNO – была нажата кнопка **Нет** (**No**);
- IDOK – была нажата кнопка **OK**;
- IDRETRY – была нажата кнопка **Повторить** (**Retry**);
- IDYES – была нажата кнопка **Да** (**Yes**).

В листинге 2.4 представлен пример создания изображенного на рис. 2.4 окна сообщений.

Листинг 2.4. Пример создания окна сообщений

```
1   int mbResult = MessageBox(hWnd, TEXT("Текст сообщения"),
    TEXT("Заголовок"), MB_YESNOCANCEL | MB_ICONINFORMATION |
    MB_DEFBUTTON3);
```

```
2
3   switch (mbResult)
4   {
5   case IDYES: // нажата кнопка Да
6       break;
7   case IDNO: // нажата кнопка Нет
8       break;
9   case IDCANCEL: // нажата кнопка Отмена
10      break;
11  } // switch
```

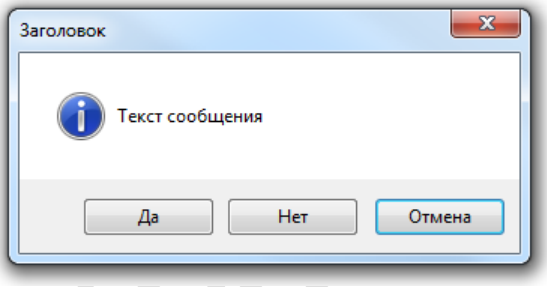

Рис. 2.4. Пример окна сообщений

Диалоговые окна общего пользования

В Win32 API имеется набор функций, которые создают *диалоговые окна общего пользования* (common dialog box). В табл. 2.12 перечислены диалоговые окна общего пользования и функции их создания.

Таблица 2.12. Функции создания диалоговых окон общего пользования

Диалоговое окно	**Функция создания**
Модальное диалоговое окно **Цвет** (**Color**)	`ChooseColor`
Модальное диалоговое окно **Шрифт** (**Font**)	`ChooseFont`
Немодальное диалоговое окно **Найти** (**Find**)	`FindText`
Модальное диалоговое окно **Открыть** (**Open**)	`GetOpenFileName`
Модальное диалоговое окно **Сохранить как** (**Save As**)	`GetSaveFileName`
Модальное диалоговое окно **Печать** (**Print**)	`PrintDlg`
Немодальное диалоговое окно **Заменить** (**Replace**)	`ReplaceText`

Подробное описание диалоговых окон общего пользования можно найти в документации Platform SDK.

В следующем примере проиллюстрировано создание диалогового окна **Печать** (**Print**), изображенного на рис. 2.5:

Листинг 2.5. Пример создания диалогового окна для настройки печати

```
PRINTDLG pd = { sizeof(PRINTDLG) };

pd.Flags = PD_ALLPAGES|PD_USEDEVMODECOPIESANDCOLLATE;
pd.hwndOwner = hWnd; // дескриптор окна-владельца
pd.nCopies = 1; // число копий
pd.nFromPage = 15; // начальная страница
pd.nToPage = 48; // конечная страница
pd.nMinPage = 1;  // минимальное значение диапазона страниц
pd.nMaxPage = 65535; // максимальное значение диапазона страниц

// создание диалогового окна Печать
if (PrintDlg(&pd) != FALSE)
{
    /* Диалоговое окно закрыто. Нажата кнопка Печать */
} // if
```

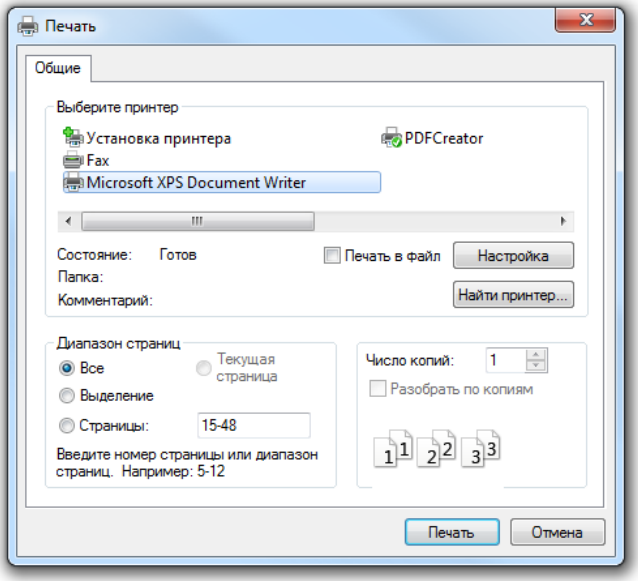

Рис. 2.5. Диалоговое окно для настройки печати

Функции работы с окнами

Win32 API предоставляет широкий набор функций, которые работают с созданными окнами. В табл. 2.13 перечислены наиболее употребительные функции. Подробное описание этих и других функций, работающих с окнами, см. в документации Platform SDK.

Таблица 2.13. Функции для работы с окнами

Функция	**Описание**
`BringWindowToTop`	Активизирует окно и переносит его поверх остальных окон, если оно находится позади
`CheckDlgButton`	Изменяет состояние элемента управления «переключатель» или «флажок»
`CloseWindow`	Сворачивает окно
`DestroyWindow`	Уничтожает окно и все его дочерние окна. Нельзя уничтожить окна других приложений
`EnableWindow`	Разрешает или запрещает окну ввод данных с клавиатуры или с помощью мыши
`EnumChildWindows`	Перечисляет дочерние окна
`EnumWindows`	Перечисляет все окна
`FindWindow(Ex)`	Возвращает дескриптор окна, найденного по имени окна и имени оконного класса
`GetActiveWindow`	Возвращает дескриптор активного окна
`GetClassInfo`	Возвращает информацию об оконном классе
`GetClassLong`	Возвращает значение указанного атрибута оконного класса
`GetClassName`	Возвращает имя класса окна
`GetClientRect`	Возвращает координаты клиентской области окна (в системе координат клиентской области)
`GetDesktopWindow`	Возвращает дескриптор окна рабочего стола
`GetDlgCtrlID` `GetWindowID`	Возвращает идентификатор окна
`GetDlgItem`	Возвращает дескриптор элемента управления в окне
`GetDlgItemInt`	Возвращает текст, связанный с элементом управления в окне, преобразуя его в целочисленное значение
`GetDlgItemText`	Возвращает текст, связанный с элементом управления в окне
`GetFirstChild`	Возвращает дескриптор первого дочернего окна
`GetFirstSibling`	Возвращает дескриптор первого окна среди окон, принадлежащих к тому же окну

Продолжение табл. 2.13.

Функция	**Описание**
GetFocus	Возвращает дескриптор окна, имеющего фокус ввода
GetForegroundWindow	Возвращает дескриптор приоритетного окна (окна, с которым пользователь в настоящее время работает)
GetLastSibling	Возвращает дескриптор последнего окна среди окон, принадлежащих тому же окну
GetNextSibling	Возвращает дескриптор следующего соседнего окна, которое принадлежит тому же окну
GetParent	Возвращает дескриптор родительского окна или окна-владельца
GetPrevSibling	Возвращает дескриптор предыдущего соседнего окна, которое принадлежит тому же окну
GetWindowExStyle	Возвращает расширенный стиль окна
GetWindowInfo	Возвращает информацию об окне
GetWindowInstance	Возвращает дескриптор экземпляра приложения, в котором создано окно
GetWindowLong	Возвращает значение указанного атрибута окна
GetWindowOwner	Возвращает дескриптор окна-владельца
GetWindowRect	Возвращает координаты окна (в экранной системе координат)
GetWindowStyle	Возвращает стиль окна
GetWindowText	Возвращает имя окна
GetWindowTextLength	Возвращает длину имени окна
IsChild	Проверяет, является ли окно дочерним окном по отношению к указанному окну
IsDlgButtonChecked	Определяет состояние элемента управления «переключатель» или «флажок»
IsIconic IsMinimized	Проверяет, свернуто ли окно
IsWindow	Определяет, идентифицирует ли дескриптор окна существующее окно
IsWindowEnabled	Проверяет, разрешен или запрещен ввод данных в окно
IsWindowVisible	Проверяет, состояние видимости окна
IsZoomed IsMaximized	Проверяет, развернуто ли окно
SetActiveWindow	Активизирует окно

Окончание табл. 2.13.

Функция	Описание
SetClassLong	Изменяет атрибуты оконного класса
SetDlgItemInt	Изменяет текст элемента управления в окне на строковое представление указанного целочисленного значения
SetDlgItemText	Изменяет текст элемента управления в окне
SetFocus	Устанавливает в окне фокус клавиатуры
SetParent	Заменяет родительское окно
SetWindowLong	Изменяет атрибуты окна
SetWindowPos MoveWindow	Изменяет размер и позицию окна
SetWindowText	Изменяет имя окна
ShowWindow	Устанавливает состояние отображения окна

В листинге 2.6 представлен небольшой пример, в котором используются функции для работы с окнами – `FindWindow` и `CloseWindow`.

Листинг 2.6. Пример использования функций для работы с окнами

```
1   // поиск окна с заголовком "Калькулятор"
2   HWND hWnd = FindWindow(NULL, TEXT("Калькулятор"));
3
4   if (NULL != hWnd) // если окно успешно найдено
5   {
6       CloseWindow(hWnd); // свернуть окно
7   } // if
```

Отображение окна

Для отображения на экране созданного окна используется функция `ShowWindow`:

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

Первый параметр, *hWnd*, задает дескриптор отображаемого окна, а второй параметр, *nCmdShow*, определяет, в каком виде будет показано окно. Для задания параметра *nCmdShow* можно использовать любое из значений, приведенных в табл. 2.14.

Таблица 2.14. Значения параметра *nCmdShow*

Значение	Описание
SW_HIDE	Скрыть указанное окно. Если окно активно, активизируется другое окно

Окончание табл. 2.14.

Значение	Описание
SW_MAXIMIZE	Развернуть указанное окно. Активное окно остается активным
SW_MINIMIZE	Свернуть указанное окно. Если окно активно, активизируется другое окно
SW_RESTORE	Активизировать и отобразить указанное окно. Если окно свернуто или развернуто, оно восстанавливается в первоначальных размерах и позиции
SW_SHOW	Активизировать и отобразить указанное окно. Окно отображается в последних размерах и позиции
SW_SHOWDEFAULT	Выполняет заданное по умолчанию отображение указанного окна
SW_SHOWMAXIMIZED	Активизировать и развернуть указанное окно
SW_SHOWMINIMIZED	Активизировать и свернуть указанное окно
SW_SHOWMINNOACTIVE	Свернуть указанное окно. Активное окно остается активным
SW_SHOWNOACTIVATE	Отобразить указанное окно в последнем размере и позиции. Активное окно остается активным

Функция ShowWindow в случае успеха возвращает значение отличное от FALSE.

Оконные сообщения

Операционная система помещает каждое *оконное сообщение* (windows message), адресованное окнам приложения, в *очередь сообщений* (message queue) этого приложения. Приложение извлекает сообщения из очереди в *цикле обработки сообщений* (message loop). После извлечения сообщения передаются *оконной процедуре* (window procedure), в которой выполняется обработка этих сообщений.

Цикл обработки сообщений

В каждом приложении Windows должен находиться цикл обработки сообщений, который обычно выглядит следующим образом:

Листинг 2.7. Цикл обработки сообщений

```
1   MSG  msg;
2   BOOL bRet;
3
4   while ((bRet = GetMessage(&msg, NULL, 0, 0)) != FALSE)
5   {
6       if (bRet == -1)
7       {
```

```
8           /* обработка ошибки и возможно выход из цикла */
9       } // if
10      else
11      {
12          TranslateMessage(&msg);
13          DispatchMessage(&msg);
14      } // else
15  } // while
```

Структура MSG содержит информацию о сообщении, полученном из очереди сообщений. Эта структура определена в заголовочном файле WinUser.h следующим образом:

```
typedef struct tagMSG {
  HWND hwnd; // дескриптор окна
  UINT message; // код (идентификатор) сообщения
  WPARAM wParam; // дополнительная информация о сообщении
  LPARAM lParam; // дополнительная информация о сообщении
  DWORD time; // время создания сообщения
  POINT pt; // позиция курсора (в экранных координатах)
            // в момент создания сообщения
} MSG, *PMSG, *LPMSG;
```

Первое поле, *hwnd*, содержит дескриптор окна, которому адресовано полученное сообщение.

Второе поле, *message*, – код сообщения, который идентифицирует полученное сообщение. Для всех оконных сообщений в заголовочном файле WinUser.h определены символические константы, что облегчает чтение и понимание программ.

Поля *wParam* и *lParam* содержат дополнительную информацию о полученном сообщении, которая зависит от кода сообщения.

Последние два поля, *time* и *pt*, в дополнительных комментариях не нуждаются.

Извлечение сообщения из очереди осуществляется с помощью функции GetMessage:

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

Первый параметр, *lpMsg*, указывает на структуру MSG, в которую помещается информация об извлекаемом сообщении.

Второй параметр, *hWnd*, содержит дескриптор окна, для которого извлекается сообщение. Обычно этот параметр устанавливают в NULL, что позволяет извлекать сообщения для любого окна приложения.

Параметры *wMsgFilterMin* и *wMsgFilterMax* задают диапазон извлекаемых сообщений. Если оба этих параметра принимают значение

равное нулю, то функция извлекает из очереди любое очередное сообщение.

Функция GetMessage возвращает значение TRUE при получении любого сообщения, кроме WM_QUIT. При получении сообщения WM_QUIT, возвращается значение FALSE, что приводит к выходу из цикла обработки сообщений (см. листинг 2.7). В случае возникновения ошибки GetMessage возвращает значение –1.

Нужно отметить, что если в очереди нет сообщений, функция GetMessage, не возвращает управление программе, а ждет, когда сообщение появится в очереди. Однако в некоторых приложениях может возникнуть необходимость выполнять фоновую обработку каких-либо данных, если в очереди нет сообщений. Для решения этой проблемы, Win32 API предоставляет функцию PeekMessage, которая проверяет наличие сообщения в очереди и при необходимости извлекает его.

Прототип функции PeekMessage практически идентичен прототипу функции GetMessage:

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

Данная функция возвращает значение TRUE, если в очереди имеется сообщение, иначе – FALSE. Первые четыре параметра функции PeekMessage такие же, как и у функции GetMessage. Последний параметр, *wRemoveMsg*, определяет, как именно сообщение извлекается из очереди. Для задания параметра *wRemoveMsg* можно использовать одно из следующих значений:

- PM_NOREMOVE – сообщение остается в очереди;
- PM_REMOVE – сообщение удаляется из очереди.

В листинге 2.8 представлен пример цикла обработки сообщений, в котором используется функция PeekMessage.

Листинг 2.8. Цикл обработки сообщений (без ожидания сообщения)

```
1   MSG  msg;
2   BOOL bRet;
3
4   for (;;)
5   {
6       // определяем наличие сообщений в очереди
7       while (!PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
8       {
9           /* пока нет сообщений, выполняется какая-нибудь работа */
10      } // while
11
```

```
12      // извлекаем сообщение из очереди
13      bRet = GetMessage(&msg, NULL, 0, 0);
14
15      if (bRet == -1)
16      {
17          /* обработка ошибки и возможно выход из цикла */
18      } // if
19      else if (FALSE == bRet)
20      {
21          break; // получено WM_QUIT, выход из цикла
22      } // if
23      else
24      {
25          TranslateMessage(&msg);
26          DispatchMessage(&msg);
27      } // else
28  } // for
```

В теле цикла обработки сообщений можно увидеть вызовы функций `TranslateMessage` и `DispatchMessage`, которые имеют следующие прототипы:

```
BOOL TranslateMessage(const MSG *lpMsg);

LRESULT DispatchMessage(const MSG *lpmsg);
```

Функция `TranslateMessage` используется только в тех случаях, когда необходимо обрабатывать ввод с клавиатуры. Дело в том, что в Windows реализована двух уровневая схема обработки сообщений от клавиатуры, в которой сначала из очереди извлекается сообщение, содержащее *виртуальный код* (virtual key) нажатой клавиши, а затем сообщение, содержащее код символа (ANSI или Unicode) этой клавиши. Сообщения с виртуальными кодами клавиш генерируются операционной системой, тогда как сообщения с кодами символов помещаются в очередь функцией `TranslateMessage`. Например, если будет получено сообщение `WM_KEYDOWN` (была нажата клавиша), в котором содержится виртуальный код нажатой клавиши, то функция `TranslateMessage` добавит в начало очереди сообщение `WM_CHAR`, которое содержит код символа нажатой клавиши.

В случае если сообщение (с кодом символа нажатой клавиши) было добавлено в очередь функция `TranslateMessage` возвращает `TRUE`, иначе – `FALSE`. Следует также отметить, что функция `TranslateMessage` не изменяет сообщение, указанное параметром *`lpMsg`*.

Функция `DispatchMessage` передает полученное сообщение оконной процедуре того окна, которому оно было адресовано. Это может

быть оконная процедура окна (созданного приложением) или оконная процедура одного из элементов управления общего пользования.

Функция `DispatchMessage` возвращает значение, которое возвращает оконная процедура и которое, как правило, игнорируется.

Цикл обработки сообщений диалоговых окон

Чтобы обрабатывать сообщения модального диалогового окна, операционная система запускает собственный цикл обработки сообщений, временно забирая у приложения управление очередью сообщений.

Обработка сообщений немодального диалогового окна, напротив, выполняется в цикле обработки сообщений приложения, с помощью вызова функции `IsDialogMessage`, например, следующим образом:

Листинг 2.9. Обработка сообщений немодального диалогового окна

```
MSG  msg;
BOOL bRet;

while ((bRet = GetMessage(&msg, NULL, 0, 0)) != FALSE)
{
    if (bRet == -1)
    {
        /* обработка ошибки и возможно выход из цикла */
    } // if
    else if (IsDialogMessage(hDlg, &msg) == FALSE)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    } // if
} // while
```

В этом примере используется глобальная переменная `hDlg`, которая содержит дескриптор немодального диалогового окна.

Функция `IsDialogMessage` определяет, предназначено ли сообщение указанному диалоговому окну и, если это так, обрабатывает его.

```
BOOL IsDialogMessage(HWND hDlg, LPMSG lpMsg);
```

Первый параметр, *`hDlg`*, дескриптор диалогового окна, сообщение которого нужно обработать. Второй параметр, *`lpMsg`*, указывает на сообщение, которое нужно обработать.

Если сообщение было обработано функцией `IsDialogMessage`, возвращается значение `TRUE`, иначе – `FALSE`.

Оконная процедура

Обработкой оконных сообщений в приложении занимается специальная функция, называемая *оконной процедурой*. Для каждого окна создается своя копия оконной процедуры, что позволяет использовать в ней статические переменные.

Имя оконной процедуры может быть любым. Оконная процедура имеет следующую сигнатуру:

```
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam);
```

Первый параметр, *hWnd*, содержит дескриптор окна, получающего сообщение. Второй параметр, *uMsg*, – код сообщения. Параметры *wParam* и *lParam* содержат дополнительную информацию, которая зависит от кода полученного сообщения.

В теле оконной процедуры должна выполняться обработка полученных сообщений. Все сообщения, которые не обрабатываются оконной процедурой, должны передаваться в функцию `DefWindowProc`, которая выполняет стандартную обработку сообщений. В этом случае оконная процедура должна вернуть то значение, которое вернет функция `DefWindowProc`.

Функция `DefWindowProc` фактически является оконной процедурой по умолчанию. Поэтому прототип данной функции имеет туже сигнатуру, что и оконная процедура:

```
LRESULT CALLBACK DefWindowProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam);
```

В листинге 2.10 представлен пример оконной процедуры, которая обрабатывает сообщения `WM_CREATE` и `WM_DESTROY`, а остальные передает в функцию `DefWindowProc`.

Листинг 2.10. Пример оконной процедуры

```
1   LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
2   {
3       switch (uMsg)
4       {
5       case WM_CREATE: // создание окна
6           /* здесь выполняется обработка сообщения */
7           return 0;
8       case WM_DESTROY: // уничтожение окна
9           /* здесь выполняется обработка сообщения */
10          return 0;
11      } // switch
```

```
12
13      // передача необработанного сообщения
14      // оконной процедуре по умолчанию
15      return DefWindowProc(hWnd, uMsg, wParam, lParam);
16  } // WindowProc
```

Процедура диалогового окна

Каждое диалоговое окно принадлежит предопределенному оконному классу (см. табл. 2.3), которому соответствует своя оконная процедура. Приложения не имеют прямого доступа к этой оконной процедуре, но могут использовать так называемую *процедуру диалогового окна* (dialog procedure). Эта процедура похожа на обычную оконную процедуру, но в отличие от оконной процедуры, процедура диалогового окна никогда не вызывает функцию `DefWindowProc`. Вместо этого, она возвращает значение `TRUE`, если она обрабатывает оконное сообщение, или `FALSE`, если она этого не делает.

Имя процедуры диалогового окна, как имя оконной процедуры, может быть любым. Процедура диалогового окна имеет следующую сигнатуру:

```
INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg,
    WPARAM wParam, LPARAM lParam);
```

Параметры процедуры диалогового окна такие же, как у оконной процедуры. Первый параметр, *hwndDlg*, содержит дескриптор диалогового окна, получающего сообщение. Второй параметр, *uMsg*, – код сообщения. Параметры *wParam* и *lParam* содержат дополнительную информацию, которая зависит от кода полученного сообщения.

Если процедура диалогового окна обрабатывает сообщение, которое требует специального возвращаемого значения, то она может передать это значение через макрос `SetDlgMsgResult`, определенный в заголовочном файле WindowsX.h:

```
SetDlgMsgResult(hwnd, msg, result)
```

Первый параметр, *hwnd*, определяет дескриптор диалогового окна. Второй параметр, *msg*, идентифицирует сообщение. Третий параметр, *result*, задает значение, которая должна вернуть процедура диалогового окна.

В этом случае процедура диалогового окна должна вернуть то значение, которое вернет макрос `SetDlgMsgResult`. Подробнее о том, каким образом процедура диалогового окна возвращает значения см. в документации Platform SDK.

В листинге 2.11 представлен пример процедуры диалогового окна, которая обрабатывает сообщения WM_INITDIALOG и WM_DESTROY.

Листинг 2.11. Пример процедуры диалогового окна

```
1   INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM
    wParam, LPARAM lParam)
2   {
3       switch (uMsg)
4       {
5       case WM_INITDIALOG: // инициализация диалогового окна
6           /* здесь выполняется обработка сообщения */
7           return SetDlgMsgResult(hwndDlg, WM_INITDIALOG, TRUE);
8       case WM_DESTROY: // уничтожение диалогового окна
9           /* здесь выполняется обработка сообщения */
10          return TRUE;
11      } // switch
12
13      return FALSE;
14  } // DialogProc
```

Часто используемые сообщения

В этом разделе рассмотрены оконные сообщения, обработка которых в оконной процедуре встречается наиболее часто. Полный перечень оконных сообщений, а также их описание, можно найти в документации Platform SDK.

WM_CLOSE

Сообщение WM_CLOSE уведомляет окно о том, что оно должно быть закрыто. Приложение, обрабатывая это сообщение, может запросить у пользователя подтверждение, перед разрушением окна, и при подтверждении выбора вызвать функцию DestroyWindow.

Параметры *wParam* и *lParam* не используются. Если это сообщение обрабатывается, оконная процедура должна вернуть значение 0.

Сообщение WM_CLOSE может обрабатываться, например, следующим образом:

Листинг 2.12. Пример обработки сообщения WM_CLOSE

```
1   case WM_CLOSE:
2       {
3           int mbResult = MessageBox(hWnd, TEXT("Хотите завершить
    работу?"), TEXT("Выход"), MB_YESNO | MB_ICONQUESTION |
    MB_DEFBUTTON2);
4
5           if (IDYES == mbResult) // пользователь выбрал «Да»
```

```
6           {
7               DestroyWindow(hWnd); // уничтожаем окно
8           } // if
9       }
10      return 0;
```

По умолчанию функция `DefWindowProc` вызывает `DestroyWindow`, чтобы разрушить окно.

WM_COMMAND

Сообщение `WM_COMMAND` отправляется окну, когда пользователь выбирает пункт в меню окна, когда элемент управления общего пользования отправляет уведомление своему родительскому окну, или когда транслируется нажатие быстрых клавиш.

Старшее слово параметра *wParam* определяет код уведомления, если сообщение передано элементом управления. Если сообщение, передано в результате трансляции быстрых клавиш, это значение равно `1`. Если же сообщение от меню, то значение равно `0`. Младшее слово параметра *wParam* определяет идентификатор пункта меню, элемента управления или быстрой клавиши.

Получить значения, запакованные в параметре *wParam*, можно с помощью двух макросов `LOWORD` и `HIWORD`, которые возвращают соответственно младшее слово и старшее слово.

Параметр *LParam* – дескриптор элемента управления, посылающего сообщение. Может принимать значение `NULL`.

Если это сообщение обрабатывается, оконная процедура должна вернуть значение `0`.

В следующем примере продемонстрирована обработка уведомления `BN_CLICKED` (была нажата кнопка) для двух элементов управления «кнопка» (button).

Листинг 2.13. Пример обработки сообщения `WM_COMMAND`

```
1   case WM_COMMAND:
2       {
3           // идентификатор элемента управления
4           WORD id = LOWORD(wParam);
5           // код уведомления
6           WORD codeNotify = HIWORD(wParam);
7
8           if (BN_CLICKED == codeNotify)
9           {
10              switch (id)
11              {
```

```
12              case IDC_BUTTON1: // обработка нажатия кнопки 1
13                  return 0;
14              case IDC_BUTTON2: // обработка нажатия кнопки 2
15                  return 0;
16              } // switch
17          } // if
18      }
19      break;
```

WM_CREATE

Сообщение `WM_CREATE` отправляется окну, создаваемому функцией `CreateWindowEx` или `CreateWindow`. (Отправление происходит, перед тем как функция, создавшая окно, возвращает значение.) Оконная процедура окна принимает это сообщение после создания окна, но перед отображением.

Параметр *wParam* не используется. Параметр *lParam* указывает на структуру `CREATESTRUCT`, которая содержит информацию о создаваемом окне. Описание структуры `CREATESTRUCT` см. в документации Platform SDK.

Если это сообщение обрабатывается, оконная процедура должна вернуть значение `0`. Если оконная процедура вернет `-1`, то окно будет уничтожено, а функция `CreateWindow(Ex)` вернет `NULL`.

WM_DESTROY

Сообщение `WM_DESTROY` отправляется уничтожаемому окну, после того, как оно исчезает с экрана. Параметры *wParam* и *lParam* не используются.

Если это сообщение обрабатывается, оконная процедура должна вернуть значение `0`.

WM_INITDIALOG

Сообщение `WM_INITDIALOG` отправляется диалоговому окну непосредственно перед тем, как оно будет отображено на экране. Обработка этого сообщения позволяет выполнить инициализацию данных, связанных с диалоговым окном.

Параметр *wParam* представляет собой дескриптор элемента управления, который по умолчанию принимает фокус клавиатуры, если процедура диалогового окна вернет значение `TRUE`.

Параметр *lParam* определяет дополнительные данные инициализации. Эти данные передаются, как параметр *lParam* при вызове функций `DialogBoxParam` или `CreateDialogParam`. Этот параметр принимает значение `NULL`, если для создания использовались другие функции.

Если процедура диалогового окна возвращает значение TRUE, элемент управления, дескриптор которого указан в параметре *wParam*, автоматически принимает фокус клавиатуры. Если в процедуре диалогового окна для какого-либо элемента управления устанавливается фокус клавиатуры, используя функцию SetFocus, то должно возвращаться значение FALSE.

WM_MOVE

Сообщение WM_MOVE отправляется окну после того, как оно будет перемещено. Параметр *wParam* не используется, а параметр *lParam* содержит координаты левого и верхнего угла окна. Младшее слово параметра *lParam* содержит *x*-координату, а старшее слово содержит *y*-координату. Координаты указываются в экранных координатах для перекрывающих и всплывающих окон, а для дочерних окон в координатах клиентской области родительского окна.

Получить *x*- и *y*-координату можно с помощью двух макросов, определенных в заголовочном файле WindowsX.h – соответственно GET_X_LPARAM и GET_Y_LPARAM. Следующий пример демонстрирует, как это можно сделать:

```
1   WORD x = GET_X_LPARAM(lParam);
2   WORD y = GET_Y_LPARAM(lParam);
```

Если сообщение WM_MOVE обрабатывается, оконная процедура должна вернуть значение 0.

WM_NOTIFY

Сообщение WM_NOTIFY отправляется окну элементом управления общего пользования, когда произошло некоторое событие или элемент управления требует некоторой информации.

Параметр *wParam* идентифицирует элемент управления. Однако нет гарантии, что этот идентификатор будет уникальным. Поэтому чтобы идентифицировать элемент управления следует использовать одно из полей структуры NMHDR (*hwndFrom* или *idFrom*), на которую указывает параметр *lParam*. Структура NMHDR описывается следующим образом:

```
typedef struct tagNMHDR {
  HWND hwndFrom; // дескриптор окна элемента управления
  UINT_PTR idFrom; // идентификатор элемента управления
  UINT code; // код уведомления
} NMHDR;
```

Первое поле, *hwndFrom*, содержит дескриптор окна элемента управления, от которого пришло уведомление. Второе поле, *idFrom*, – это

идентификатор этого элемента управления. Третье поле, *code*, содержит код уведомления. Для каждого элемента управления существуют свои специфические уведомления (см. в документации Platform SDK).

Для некоторых уведомлений параметр *lParam* указывает на другую структуру, имеющую NMHDR в качестве первого поля.

В следующем примере продемонстрирована обработка уведомления NM_CLICK (щелчок левой кнопкой мыши) для элемента управления «список просмотра» (list view):

Листинг 2.14. Пример обработки сообщения WM_NOTIFY

```
1   case WM_NOTIFY:
2       {
3           LPNMHDR lpnmhdr = (LPNMHDR)lParam;
4
5           switch (lpnmhdr->code)
6           {
7           case NM_CLICK:
8               if (lpnmhdr->idFrom == IDC_LISTVIEW1)
9               {
10                  LPNMITEMACTIVATE lpnmitem =
    (LPNMITEMACTIVATE)lpnmhdr;
11                  /* обработка уведомления */
12                  return 0;
13              } // if
14          } // switch
15      }
16      break;
```

WM_SIZE

Сообщение WM_SIZE отправляется окну после изменения его размера. Параметр *wParam* может принимать одно из следующих значений:

- SIZE_MAXHIDE – сообщение было послано всем всплывающим окнам после того, как другое окно было развернуто;
- SIZE_MAXIMIZED – окно было развернуто;
- SIZE_MAXSHOW – сообщение было послано всем всплывающим окнам после того, как другое окно было восстановлено до прежних размеров;
- SIZE_MINIMIZED – окно было свернуто;
- SIZE_RESTORED – размеры окна были изменены, но окно не было свернуто или развернуто.

Младшее слово параметра *lParam* содержит новую ширину клиентской области. Старшее слово параметра *lParam* содержит новую

высоту клиентской области. Получить ширину и высоту из параметра *lParam* можно с помощью макросов LOWORD и HIWORD.

Если это сообщение обрабатывается, оконная процедура должна вернуть значение 0.

Отправка сообщений

В Win32 API имеется набор функций, предназначенных для отправки оконных сообщений. Основными из них являются PostMessage и SendMessage:

```
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam,
    LPARAM lParam);

LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam,
    LPARAM lParam);
```

Первый параметр, *hWnd*, содержит дескриптор окна, оконная процедура которого примет сообщение. Если этот параметр принимает значение равное HWND_BROADCAST, то сообщение отправляется всем окнам верхнего уровня (такой тип сообщений не отправляется дочерним окнам).

Второй параметр, *uMsg*, – код сообщения, которое будет отправлено. Для отправки собственных сообщений в пределах оконного класса, можно использовать любое значение в диапазоне от WM_USER до 0x7FFF.

Последние параметры, *wParam* и *lParam*, содержат дополнительную информацию, которая зависит от оконного сообщения.

Основное различие между этими двумя функциями состоит в том, что функция PostMessage помещает сообщение в очередь сообщений и возвращает управление без ожидания обработки этого сообщения, тогда как функция SendMessage вызывает оконную процедуру заданного окна и не возвращает управление до тех пор, пока оконная процедура не обработает переданное сообщение. В первом случает речь идет о *синхронных* сообщениях, а во втором – об *асинхронных* сообщениях.

Если функция PostMessage завершает успешно, возвращается значение отличное от FALSE.

Функция SendMessage возвращает значение, которое вернула оконная процедура в результате обработки сообщения.

Отправка сообщений элементам управления

После создания элемента управления общего пользования приложение управляет его действиями, отправляя необходимые сообщения при помощи функции SendMessage. Кроме общих оконных сообщений таких, как, например, WM_SETFONT и WM_SETTEXT, каждый элемент управ-

ления поддерживает свои специфические сообщения (см. в документации Platform SDK).

Например, для задания максимальной длины вводимого текста в элементе управления «редактируемом поле» (edit) с дескриптором окна `hwndEdit` необходимо отправить этому элементу управления сообщение `EM_LIMITTEXT`:

```
SendMessage(hwndEdit, EM_LIMITTEXT, 255, 0);
```

Альтернативой вызовам функции `SendMessage` является использование макросов, определенных в заголовочных файлах CommCtrl.h и WindowsX.h. Например, чтобы отправить элементу управления сообщение `EM_LIMITTEXT` можно использовать макрос `Edit_LimitText`:

```
Edit_LimitText(hwndEdit, 255);
```

Не трудно заметить, что программный код с макросом значительно проще для чтения и понимания.

К сожалению, как видно из табл. 2.15, в Win32 API имеются макросы не для всех элементов управления общего пользования.

Таблица 2.15. Префиксы макросов для элементов управления

Элемент управления	**Оконный класс**	**Префикс макроса**
Анимационное изображение (Animate)	`SysAnimate32`	`Animate_`
Выпадающий список (Combo box)	`ComboBox` `ComboBoxEx32`	`ComboBox_`
Дерево просмотра (Tree view)	`SysTreeView32`	`TreeView_`
Заголовок списка просмотра (Header)	`SysHeader32`	`Header_`
Закладки (Tab control)	`SysTabControl32`	`TabCtrl_`
Календарь (Month calendar)	`SysMonthCal32`	`MonthCal_`
Кнопка (Button), Переключатель (Radio button), Флажок (Check box)	`Button`	`Button_`
Надпись (Static text)	`Static`	`Static_`
Поле ввода даты или времени (Data time picker)	`SysDateTimePick32`	`DateTime_`
Полоса прокрутки (Scroll bar)	`ScrollBar`	`ScrollBar_`
Редактируемое поле (Edit box)	`Edit`	`Edit_`
Список (List box)	`ListBox`	`ListBox_`
Список просмотра (List view)	`SysListView32`	`ListView_`

Кроме того, для отправки сообщений элементам управления можно использовать функцию SendDlgItemMessage:

```
LRESULT SendDlgItemMessage(HWND hDlg, int nIDDlgItem,
    UINT Msg, WPARAM wParam, LPARAM lParam);
```

Первый параметр, *hDlg*, содержит дескриптор окна, которому принадлежит элемент управления. Второй параметр, *nIDDlgItem*, задает идентификатор элемента управления. Последние три параметра (*Msg*, *wParam* и *lParam*) аналогичны параметрам функции SendMessage.

Функция SendDlgItemMessage работает точно так же, как и функция SendMessage, то есть не возвращает управление до тех пор, пока оконная процедура не обработает сообщение.

Отправка WM_QUIT

Для отправки сообщения WM_QUIT в Win32 API используется функция PostQuitMessage:

```
void PostQuitMessage(int nExitCode);
```

Параметр *nExitCode* определяет код завершения. Значение этого параметра будет записано в поле *wParam* структуры MSG.

Не следует посылать сообщение WM_QUIT с помощью функций PostMessage или SendMessage.

Нужно отметить, что сообщение WM_QUIT не может быть получено оконной процедурой, т.к. оно не связано с каким-либо окном. Только функции GetMessage и PeekMessage могут его получить.

Распаковщики сообщений

В заголовочном файле WindowsX.h определены макросы *распаковщики сообщений* (message crackers), которые упрощают обработку сообщений в оконной процедуре. При их использовании придется постоянно заглядывать в исходный текст файла WindowsX.h, так как в документации Platform SDK распаковщики не описаны.

Распаковщики сообщений позволяют обойти ряд трудностей, которые могут возникнуть при разработке оконной процедуры. Во-первых, как правило, программный код оконной процедуры (кроме простейших примеров) разрастается до огромных размеров, так что становится крайне трудно его читать и отлаживать. Во-вторых, в оконной процедуре может сосредотачиваться большое количество переменных, часть из которых используется для обработки конкретного сообщения, тогда как остальные в этот момент не используются, бесполезно занимая место в памяти.

В WindowsX.h для большинства сообщения вида `WM_XXX` определен свой макрос с именем `HANDLE_WM_XXX`. Например, для сообщения `WM_CREATE` определен следующий макрос:

```
HANDLE_WM_CREATE(hwnd, wParam, lParam, fn)
```

Параметры всех таких макросов одинаковые. Параметр *hwnd* – это дескриптор окна, получающего сообщение. Параметры *wParam* и *lParam* содержат дополнительную информацию. Последний параметр, *fn*, – имя функции, которая будет обрабатывать сообщение.

Макрос распаковщика сообщения должен использоваться внутри конструкции `switch` оконной процедуры. Например, следующий фрагмент программного кода:

```
1   case WM_CREATE:
2       return HANDLE_WM_CREATE(hWnd, wParam, lParam, OnCreate);
```

в результате раскрытия представляет собой следующий программный код (подробнее см. исходный текст файла WindowsX.h):

```
1   case WM_CREATE:
2       return ((OnCreate)((hWnd),(LPCREATESTRUCT)(lParam)) ? 0L :
    (LRESULT)-1L);
```

Таким образом, можно видеть, что макрос распаковщика сообщения осуществляет распаковку параметров, вызов функции-обработчика сообщения и анализ возвращаемого результата.

Чтобы распаковщик правильно распознал функцию-обработчика, она должна иметь определенную сигнатуру, которая зависит от кода сообщения. В WindowsX.h перед определением соответствующего макроса приводится сигнатура функции-обработчика:

```
1   /* BOOL Cls_OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct) */
2   #define HANDLE_WM_CREATE(hwnd, wParam, lParam, fn) \
3       ((fn)((hwnd), (LPCREATESTRUCT)(lParam)) ? 0L : (LRESULT)-1L)
```

Из этого примера видно, что функция-обработчик сообщения `WM_CREATE` имеет следующую сигнатуру:

```
BOOL Cls_OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct);
```

Имя функции-обработчика не обязательно должно быть таким, как указано в файле WindowsX.h, оно может быть любым.

Кроме того, в файле WindowsX.h определен небольшой макрос `HANDLE_MSG`, который упрощает использование распаковщиков сообщений в конструкции `switch` оконной процедуры:

```
HANDLE_MSG(hwnd, message, fn)
```

Здесь параметр *hwnd* – дескриптор окна, получающего сообщение. Параметр *message* – код сообщения. Последний параметр, *fn*, – имя функции-обработчика.

Этот макрос используется в конструкции `switch` оконной процедуры следующим образом:

```
1   switch (uMsg)
2   {
3       HANDLE_MSG(hWnd, WM_CREATE, OnCreate);
4       HANDLE_MSG(hWnd, WM_DESTROY, OnDestroy);
5   } // switch
```

Упаковка сообщений

В заголовочном файле WindowsX.h также определены макросы `FORWARD_WM_XXX`, которые решают обратную задачу – принимают данные сообщения, упаковывают их в параметры *wParam* и *lParam*, а затем вызывают функцию, указанную в качестве аргумента макроса, передавая ей упакованные параметры. Например, для сообщения `WM_CREATE` определен следующий макрос:

```
FORWARD_WM_CREATE(hwnd, lpCreateStruct, fn)
```

В качестве вызываемой функции может использоваться функция `PostMessage`, `SendMessage` или любая другая функция, имеющая аналогичный набор параметров.

Приложения Win32

Приложение Win32 (Win32 application) основано на *графическом интерфейсе пользователя* (Graphical User Interface, GUI). Такие приложения создают окна, имеют меню, взаимодействуют с пользователем через диалоговые окна и т.п. Почти все стандартные приложения Windows (такие как «Блокнот», «Калькулятор» и др.) являются приложениями Win32.

Вместе с тем различие между приложением Win32 и консольным приложением весьма условно. Можно, например, создать консольное приложение, в котором будут отображаться различные окна. В то же время можно создать приложение Win32, способное выводить данные в стандартный поток вывода, как это делает консольное приложение.

Функция WinMain

По аналогии с консольными приложениями C/C++, каждое приложение Win32 должно иметь главную функцию – `WinMain`. При написании приложения, которое использует набор символов Unicode, можно использовать `wWinMain`.

Можно также использовать макрос _tWinMain, который объявлен в заголовочном файле tchar.h. В зависимости от определения константы _UNICODE используется либо WinMain, либо wWinMain.

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

int WINAPI wWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow);

int WINAPI _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow);
```

Параметр *hInstance* – дескриптор экземпляра приложения, который система присваивает запущенному приложению. Параметр *hPrevInstance* остался от Windows 3.1 для совместимости версий, в Win32 этот параметр не используется и поэтому всегда установлен в NULL. Параметр *lpCmdLine* указывает на строку, которая содержит аргументы приложения, при запуске в режиме командной строки. Параметр *nCmdShow* содержит значение (см. табл. 2.14), которое может быть передано функции ShowWindow.

Точка входа приложения Win32

Если в качестве главной функции используется WinMain, точкой входа при запуске приложения Windows является специальная функция WinMainCRTStartup. Функция WinMainCRTStartup инициализирует различные библиотеки C/C++, загружает необходимые DLL и создает и все глобальные переменные. Когда все это будет сделано, она вызывает функцию WinMain.

Если же в качестве главной функции используется wWinMain, то точкой входа является функция wWinMainCRTStartup, которая вызывает wWinMain, но в остальном она делает все тоже, что и WinMainCRTStartup.

После того как функция WinMain (wWinMain) вернет управление, функция WinMainCRTStartup (wWinMainCRTStartup) удалит все глобальные переменные, выгрузит DLL и освободит ресурсы, выделенные для различных библиотек C/C++.

Пользовательский интерфейс

Каждая новая версия Windows предлагает улучшенный вариант пользовательского интерфейса. Помимо того, что в библиотеку ComCtl32.dll добавляются новые элементы управления общего пользования, изменяется внешний вид уже существующих элементов управления. Например, на рис. 2.6 проиллюстрировано как изменялся внешний вид элемента управления «кнопка».

а *б* *в*

Рис. 2.6. Кнопка: *а* – традиционное оформление; *б* – оформление в Windows XP; *в* – оформление в Windows Vista

Операционная система Windows XP и более новые версии используют библиотеку ComCtl32.dll версии 6. Все более ранние версии Windows используют ComCtl32.dll версии 5 с традиционным стилем оформления элементов управления.

По умолчанию приложения Win32, создаваемые в Visual C++, используют традиционный стиль оформления элементов управления. Чтобы использовать улучшенный стиль оформления элементов управления, в проект приложения нужно включить специальный ресурс – манифест. *Манифест приложения* (application manifest) представляет собой документ в формате XML, содержащий всю информацию, необходимую для взаимодействия приложения и библиотеки ComCtl32.dll версии 6. Пример файла манифеста представлен в листинге 2.15.

Листинг 2.15. Пример файла манифеста

```
1   <?xml version='1.0' encoding='UTF-8' standalone='yes'?>
2   <assembly xmlns='urn:schemas-microsoft-com:asm.v1'
    manifestVersion='1.0'>
3       <assemblyIdentity name='CompanyName.ProductName.YourApp'
    processorArchitecture='x86' version='1.0.0.0' type='win32' />
4     <description>Your application description here.</description>
5     <dependency>
6         <dependentAssembly>
7             <assemblyIdentity type='win32'
    name='Microsoft.Windows.Common-Controls' version='6.0.0.0'
    processorArchitecture='x86' publicKeyToken='6595b64144ccf1df'
    language='*' />
8         </dependentAssembly>
9     </dependency>
10  </assembly>
```

Чтобы включить файл манифеста в проект Visual C++ выполните следующие действия:

1. Откройте решение, содержащее нужный проект (если это не было сделано заранее).
2. В меню **Проект** (**Project**) выберите **Свойства** (**Settings**).

   Откроется диалоговое окно **Страницы свойств** (**Property Pages**).

3. В открывшемся окне выберите **Свойства конфигурации** (**Configuration Properties**) → **Компоновщик** (**Linker**) → **Файл манифеста** (**Manifest File**).
4. Отредактируйте значение параметра **Создавать манифест** (**Generate Manifest**), выбрав в нем **`/MANIFEST`**.
5. Отредактируйте значение параметра **Дополнительные зависимости манифеста** (**Additional Manifest Dependencies**), как показано на рис. 2.7, указав следующее значение (в одну строку):

```
type='win32' name='Microsoft.Windows.Common-Controls'
version='6.0.0.0' processorArchitecture='x86'
publicKeyToken='6595b64144ccf1df' language='*'
```

6. Нажмите кнопку **OK**, чтобы сохранить изменения.

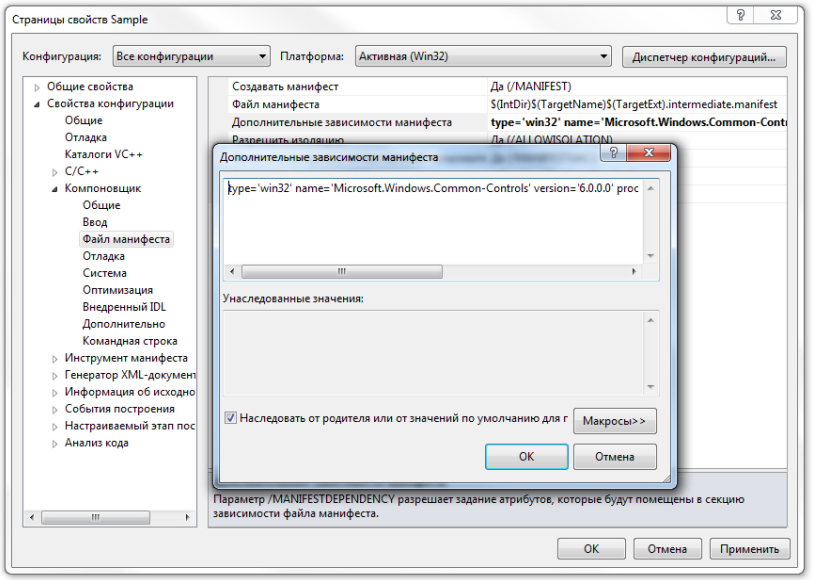

Рис. 2.7. Добавление манифеста в проект Visual C++

Ресурсы приложения Win32

Ресурсы являются составной часть практически каждого приложения Win32. В них определяются такие объекты, как пиктограммы, курсоры, растровые изображения, таблицы строк, меню, диалоговые окна и многие другие. Ресурсы могут находиться, как в исполняемом (EXE) файле, так в библиотеках DLL.

Для поддержки ресурсов в проектах Visual C++ создается файл описания ресурсов, который обычно называется так же, как и проект, но имеет расширение .rc. Например, для проекта Sample файл описания ресурсов будет называться Sample.rc. Также вместе с файлом описания ресурсов создается заголовочный файл resource.h, содержащий определения идентификаторов используемых ресурсов.

При создании нового ресурса или добавлении существующего ресурса Visual Studio открывает соответствующий редактор ресурсов. Редакторы ресурсов содержат инструменты для быстрого и удобного изменения ресурсов приложения. В составе Visual Studio обязательно имеются редакторы для таких ресурсов, как пиктограммы, курсоры, растровые изображения, диалоговые окна, меню, таблица быстрых клавиш, таблица строк, панели инструментов и информация о версии приложения.

Меню

Меню является важнейшим элементом большинства оконных приложений Windows. Меню расположенное ниже заголовка главного окна приложения называется *главным меню*. Окно, имеющее заголовок, может предоставлять доступ к *системному меню*, которое вызывается щелчком левой кнопки мыши на пиктограмме, расположенной в левой части главного окна.

Иногда в приложениях используются *контекстные меню*, появляющиеся под курсором при щелчке правой кнопкой мыши. Такие меню обычно ассоциируются с некоторым объектом, на который указывает курсор мыши в момент щелчка.

Любое меню содержит *пункты меню*. Пункт меню обозначается своим именем: словом или короткой фразой. Различают два типа пунктов меню:

- *пункт-команда* – пункт меню, имеющий уникальный идентификатор и который посылает приложению сообщение `WM_COMMAND`, заставляя его выполнить некоторое действие;
- *пункт-подменю* – пункт меню, представляет собой заголовок вызываемого меню следующего, более низкого уровня.

Пункты меню могут быть *разрешенными* (enabled), *запрещенными* (disabled) или *недоступными* (grayed). По умолчанию пункт меню является разрешенным. Запрещенный и недоступный пункты с точки зрения поведения одинаковы. Различаются запрещенный и недоступный пункты только своим внешним видом: запрещенный выглядит также как и разрешенный, а недоступный отображается серым цветом.

Иногда пункт меню используется в роли *флажка* (check box), который может быть установлен или сброшен. Пункты меню могут так же использоваться в роли *переключателей* (radio button). На рис. 2.8 показано окно, в котором вызвано меню, содержащее флажки и переключатели.

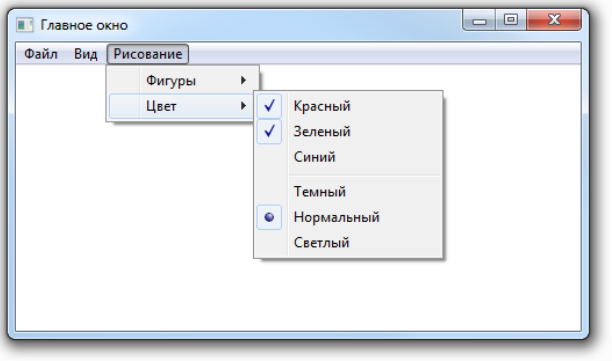

Рис. 2.8. Пункты меню флажки и переключатели

Меню можно создать одним из следующих способов:

- на основе шаблона меню, определенного в файле описания ресурсов;
- при помощи функций `CreateMenu`, `AppendMenu` и `InsertMenuItem`;
- на основе шаблона меню, определяемого во время выполнения программы при помощи функции `LoadMenuIndirect`.

Чаще всего используют первый способ. Для того чтобы создать шаблон меню в файле описания ресурсов, нужно выполнить следующие действия:

1. В окне **Обозреватель решений** (**Solution Explorer**) выберите нужный проект.
2. В меню **Проект** (**Project**) выберите **Добавить ресурс** (**Add Resource**).

   Откроется диалоговое окно **Добавление ресурса** (**Add Resource**).
3. В области **Тип ресурса** (**Resource type**) выберите **Menu**.

   Нажмите кнопку **Создать** (**New**).
4. В открывшемся редакторе Visual Studio (рис. 2.9), отредактируйте и сохраните созданный шаблон меню.

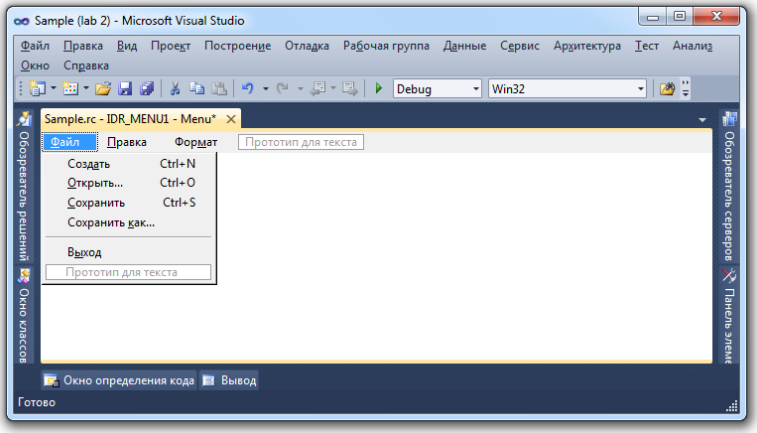

Рис. 2.9. Шаблон меню в редакторе Visual Studio

После создания шаблона меню оно еще не появится в составе окна приложения. Чтобы это случилось, меню нужно присоединить к окну одним из нескольких способов.

Наиболее традиционным способом является присваивание полю *lpszMenuName* структуры `WNDCLASSEX` значения указателя на имя меню, при регистрации оконного класса. Если имя меню определено, как целочисленная константа, нужно использовать макрос `MAKEINTRESOURCE`.

Можно связать меню с окном при создании окна, передав дескриптор меню параметру *hMenu* функции `CreateWindowEx`. В этом способе меню сначала нужно загрузить, используя функцию `LoadMenu`:

```
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpMenuName);
```

Первый параметр, *hInstance*, задает дескриптор модуля, который содержит шаблон меню.

Второй параметр, *lpMenuName*, определяет имя меню. Чтобы задать этот параметр, можно использовать макрос `MAKEINTRESOURCE`, в котором следует указать идентификатор шаблона меню.

При успешном завершении функция `LoadMenu` вернет дескриптор меню, в противном случае `NULL`.

Есть еще один способ назначения меню – использовать функцию `SetMenu`, которая имеет следующий прототип:

```
BOOL SetMenu(HWND hWnd, HMENU hMenu);
```

Первый параметр, *hWnd*, – дескриптор окна, к которому нужно присоединить меню. Второй параметр, *hMenu*, – дескриптор меню.

В случае успеха функция SetMenu возвращает TRUE, в противном случае FALSE. Новое меню заменяет старое меню, если оно уже было.

В Win32 API имеется несколько функций (табл. 2.16), которые можно использовать для работы с меню. Подробное описание этих функций см. в документации Platform SDK.

Таблица 2.16. Функции для работы с меню

Функция	**Описание**
AppendMenu	Добавляет новый пункт меню в конец меню
CreateMenu	Создает меню
DeleteMenu	Удаляет пункт меню
EnableMenuItem	Делает пункт меню разрешенным, запрещенным или недоступным
GetMenu	Возвращает дескриптор меню, связанного с указанным окном
GetSubMenu	Возвращает дескриптор подменю
InsertMenuItem	Вставляет в меню новый пункт
RemoveMenu	Удаляет пункт меню
SetMenuDefaultItem	Делает пункт меню «применяемым по умолчанию»
SetMenuItemInfo	Изменяет информацию о пункте меню (в том числе об отметке пунктов-флажков и пунктов-переключателей)
TrackPopupMenuEx	Отображает контекстное меню в заданном месте

Таблица быстрых клавиш

Быстрая клавиша (keyboard accelerator) – это клавиша или комбинация клавиш, которые при нажатии генерируют сообщение WM_COMMAND или WM_SYSCOMAND.

Обычно быстрые клавиши дублируют пункты меню, предоставляя пользователям альтернативный способ вызова команд (например, стандартное приложение «Блокнот» использует комбинацию Ctrl+N для создания нового текстового файла). Однако быстрые клавиши могут также генерировать команды, которых нет в меню.

Чтобы добавить в приложение обработку быстрых клавиш, первым делом нужно создать таблицу быстрых клавиш. Для этого выполните следующие действия:

1. В окне **Обозреватель решений** (**Solution Explorer**) выберите нужный проект.
2. В меню **Проект** (**Project**) выберите **Добавить ресурс** (**Add Resource**).

Откроется диалоговое окно **Добавление ресурса** (**Add Resource**).

3. В области **Тип ресурса** (**Resource type**) выберите **Accelerator**. Нажмите кнопку **Создать** (**New**).
4. В открывшемся редакторе Visual Studio (рис. 2.10), отредактируйте и сохраните созданную таблицу быстрых клавиш.

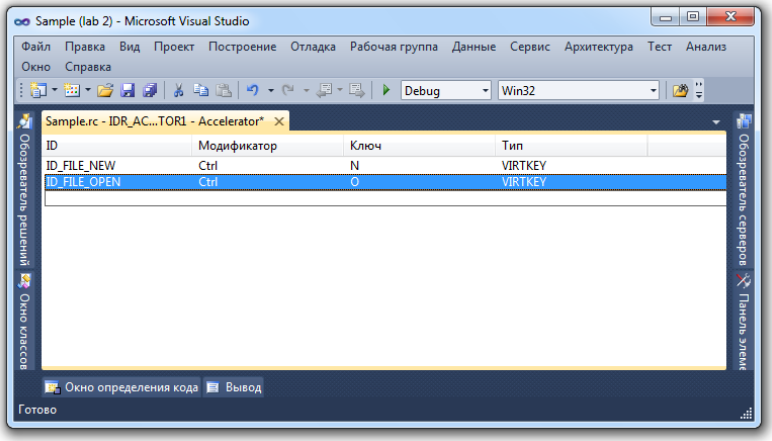

Рис. 2.10. Таблица быстрых клавиш в редакторе Visual Studio

Когда таблица быстрых клавиш будет создана, необходимо обеспечить загрузку этой таблицы во время запуска приложения. Для этого используется функция `LoadAccelerators`:

```
HACCEL LoadAccelerators(HINSTANCE hInstance,
    LPCTSTR lpTableName);
```

Первый параметр, *`hInstance`*, задает дескриптор модуля, который содержит таблицу быстрых клавиш.

Второй параметр, *`lpTableName`*, определяет имя таблицы быстрых клавиш. Чтобы задать этот параметр, можно использовать макрос `MAKEINTRESOURCE`, в котором следует указать идентификатор ресурса таблицы быстрых клавиш.

При успешном завершении функция `LoadAccelerators` возвращает дескриптор, загруженной таблицы быстрых клавиш, в противном случае `NULL`.

Для обработки быстрых клавиш приложение должно перехватывать сообщения клавиатуры, анализировать их коды и в случае совпадения с кодом, определенным в таблице быстрых клавиш, направлять

соответствующее сообщение в оконную процедуру главного окна. Все это может выполнить функция TranslateAccelerator:

```
int TranslateAccelerator(HWND hWnd, HACCEL hAccTable,
    LPMSG lpMsg);
```

Первый параметр, *hWnd*, дескриптор окна, которому будут отправляться сообщения WM_COMMAND или WM_SYSCOMAND, если таблица быстрых клавиш содержит код нажатой виртуальной клавиши.

Второй параметр, *hAccTable*, дескриптор загруженной таблицы быстрых клавиш.

Последний параметр, *lpMsg*, указывает на сообщение, которое нужно обработать.

Если функция TranslateAccelerator, возвращается ненулевое значение, это значит, что преобразование комбинации клавиш и обработка сообщения завершились успешно. В этом случае приложение не должно повторно обрабатывать эту комбинацию при помощи функций TranslateMessage и DispatchMessage.

Данное требование можно выполнить, если организовать цикл обработки сообщений, например, следующим образом:

Листинг 2.16. Обработка быстрых клавиш

```
1   MSG  msg;
2   BOOL bRet;
3
4   while ((bRet = GetMessage(&msg, NULL, 0, 0)) != FALSE)
5   {
6       if (!TranslateAccelerator(hWnd, hAccel, &msg))
7       {
8           TranslateMessage(&msg);
9           DispatchMessage(&msg);
10      } // if
11  } // while
```

В этом примере hWnd – дескриптор главного окна приложения, а hAccel – дескриптор загруженной таблицы быстрых клавиш.

Создание приложения Win32

Для того чтобы создать приложение Win32 в Visual C++ необходимо выполнить следующие действия:

1. В меню **Файл** (**File**) выберите **Создать** (**New**) → **Проект** (**Project**).

   Откроется диалоговое окно **Создать проект** (**New Project**).

2. В области шаблонов проектов **Visual C++** выберите группу **Win32** и затем выберите элемент **Проект Win32** (**Win32 Project**).

   В поле **Имя** (**Name**) введите имя создаваемого проекта (например, SampleWin32). Нажмите кнопку **ОК**.

3. В открывшемся диалоговом окне **Мастер приложений Win32** (**Win32 Application Wizard**) нажмите кнопку **Далее** (**Next**).

   Выберите **Приложение Windows** (**Windows application**) и параметр **Пустой проект** (**Empty Project**), как показано на рис. 2.11.

   Нажмите кнопку **Готово** (**Finish**).

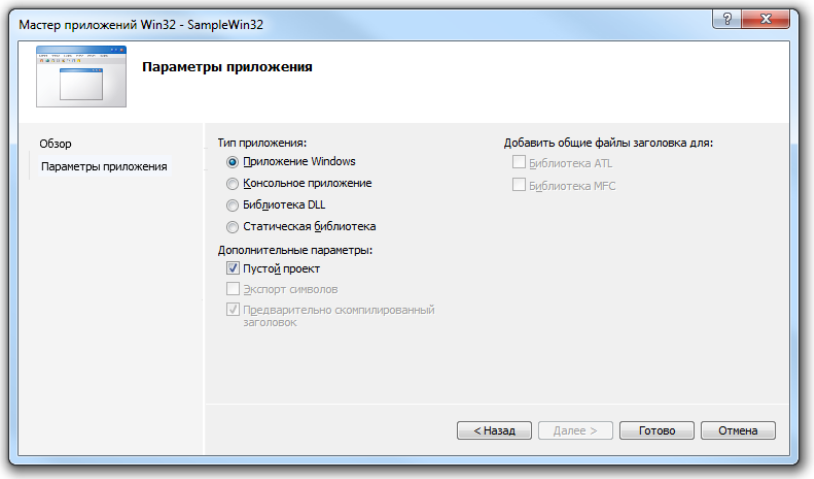

Рис. 2.11. Создание приложения Win32 в Visual C++

4. В меню **Проект** (**Project**) выберите **Добавить ресурс** (**Add Resource**).

   Откроется диалоговое окно **Добавление ресурса** (**Add Resource**).

5. В области **Тип ресурса** (**Resource type**) выберите **Dialog**.

   Нажмите кнопку **Создать** (**New**).

6. В открывшемся редакторе Visual Studio отредактируйте созданный шаблон диалогового окна, как показано на рис. 2.12.

7. В меню **Проект** (**Project**) выберите **Добавить ресурс** (**Add Resource**).

   Откроется диалоговое окно **Добавление ресурса** (**Add Resource**).

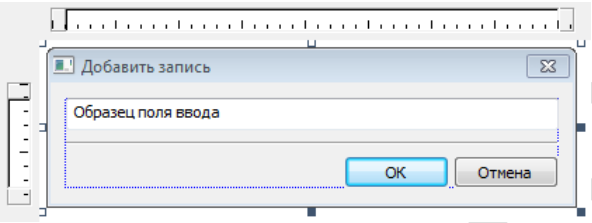

Рис. 2.12. Шаблон диалогового окна

8. В области **Тип ресурса** (**Resource type**) выберите **Menu**.

   Нажмите кнопку **Создать** (**New**).

9. В открывшемся редакторе Visual Studio отредактируйте созданный шаблон меню, добавив в него пункты меню (табл. 2.17).

Таблица 2.17. Пункты меню

Идентификатор	Надпись	Пункт меню
ID_NEW_RECORD	&Добавить запись...\tCtrl+N	&Правка
ID_NEW_RECORD2	Добавить &несколько записей...	&Правка
ID_DEL_RECORD	&Удалить запись\tDel	&Правка
ID_FIND_RECORD	Найти &запись\tCtrl+F	&Правка
ID_FORMAT_FONT	&Шрифт...	Форм&ат

10. В меню **Проект** (**Project**) выберите **Добавить ресурс** (**Add Resource**).

    Откроется диалоговое окно **Добавление ресурса** (**Add Resource**).

11. В области **Тип ресурса** (**Resource type**) выберите **Accelerator**.

    Нажмите кнопку **Создать** (**New**).

12. В открывшемся редакторе Visual Studio отредактируйте созданную таблицу быстрых клавиш, как показано на рис. 2.13.

ID	Модификатор	Ключ	Тип
ID_DEL_RECORD	Нет	VK_DELETE	VIRTKEY
ID_FIND_RECORD	Ctrl	F	VIRTKEY
ID_NEW_RECORD	Ctrl	N	VIRTKEY

Рис. 2.13. Таблица быстрых клавиш

13. Добавьте в проект исходный файл SampleWin32.cpp.

14. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 2.17 и сохраните этот файл.
15. Включите в проект файл манифеста.
16. В меню **Построение** (**Build**) выберите команду **Построить решение** (**Build Solution**).

Листинг 2.17. Файл исходного кода SampleWin32.cpp

```
#include <Windows.h>
#include <WindowsX.h>
#include <CommCtrl.h>
#include <tchar.h>

#include "resource.h"
#include "afxres.h"

// идентификаторы элементов управления на главном окне
#define IDC_LIST            2001

// код собственного сообщения WM_ADDITEM,
// которое будет использоваться для добавления записей
#define WM_ADDITEM          WM_USER + 1

HWND hWnd = NULL; // дискриптор главного окна

HACCEL hAccel = NULL; // дискриптор таблицы быстрых клавиш

// дескрипторы немодальных диалоговых окон
HWND hDlg = NULL;
HWND hFindDlg = NULL;

TCHAR szBuffer[100] = TEXT("");

FINDREPLACE findDlg; // структура для диалогового окна "Найти"
UINT uFindMsgString = 0; // код сообщения FINDMSGSTRING

HFONT hFont = NULL; // дескриптор шрифта

// оконная процедура главного окна
LRESULT CALLBACK MyWindowProc(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);

// функция, которая вызывается в цикле обработки сообщений,
// пока в очереди нет сообщений
void OnIdle(HWND hwnd);
// функция, которая вызывается в цикле обработки сообщений
```

```
38  // перед тем, как сообщение будет передано в оконную процедуру
39  BOOL PreTranslateMessage(LPMSG lpMsg);
40
41  // далее идут обработчики сообщений главного окна:
42  // обработчик сообщения WM_CREATE
43  BOOL OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct);
44  // обработчик сообщения WM_DESTROY
45  void OnDestroy(HWND hwnd);
46  // обработчик сообщения WM_SIZE
47  void OnSize(HWND hwnd, UINT state, int cx, int cy);
48  // обработчик сообщения WM_COMMAND
49  void OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
50  // обработчик сообщения WM_ADDITEM
51  void OnAddItem(HWND hwnd);
52  // обработчик сообщения FindMsgString
53  void OnFindMsgString(HWND hwnd, LPFINDREPLACE lpFindReplace);
54
55  // процедура диалогового окна
56  INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM
    wParam, LPARAM lParam);
57
58  // далее идут обработчики сообщений диалогового окна:
59  // обработчик сообщения WM_INITDIALOG
60  BOOL Dialog_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM
    lParam);
61  // обработчик сообщения WM_CLOSE
62  void Dialog_OnClose(HWND hwnd);
63  // обработчик сообщения WM_COMMAND
64  void Dialog_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT
    codeNotify);
65
66  // ----------------------------------------------------------------
67  int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE, LPTSTR
    lpszCmdLine, int nCmdShow)
68  {
69      // регистрируем оконный класс главного окна...
70
71      WNDCLASSEX wcex = { sizeof(WNDCLASSEX) };
72      wcex.style = CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS;
73      wcex.lpfnWndProc = MyWindowProc; // оконная процедура
74      wcex.hInstance = hInstance;
75      wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
76      wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
77      wcex.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
78      wcex.lpszMenuName = MAKEINTRESOURCE(IDR_MENU1);
79      wcex.lpszClassName = TEXT("MyWindowClass"); // имя класса
```

```
80      wcex.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
81
82      if (0 == RegisterClassEx(&wcex)) // регистрируем класс
83      {
84          // не удалось зарегистрировать новый оконный класс
85          return -1; // завершаем работу приложения
86      } // if
87
88      // загружаем библиотеку
89      // элементов управления общего пользования
90      LoadLibrary(TEXT("ComCtl32.dll"));
91
92      // загружаем таблицу быстрых клавиш
93      hAccel = LoadAccelerators(hInstance, MAKEINTRESOURCE(
    IDR_ACCELERATOR1));
94
95      // создаем главное окно на основе нового оконного класса
96      hWnd = CreateWindowEx(0, TEXT("MyWindowClass"),
    TEXT("SampleWin32"), WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
    CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
97
98      if (NULL == hWnd) // если не удалось создать окно
99      {
100         return -1; // завершаем работу приложения
101     } // if
102
103     ShowWindow(hWnd, nCmdShow); // отображаем главное окно
104
105     // далее идет цикл обработки сообщения...
106
107     MSG  msg;
108     BOOL bRet;
109
110     for (;;)
111     {
112         // определяем наличие сообщений в очереди
113         while (!PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE))
114         {
115             OnIdle(hWnd);
116         } // while
117
118         // извлекаем сообщение из очереди
119         bRet = GetMessage(&msg, NULL, 0, 0);
120
121         if (bRet == -1)
122         {
```

```
123             /* обработка ошибки и возможно выход из цикла */
124         } // if
125         else if (FALSE == bRet)
126         {
127             break; // получено WM_QUIT, выход из цикла
128         } // if
129         else if (!PreTranslateMessage(&msg))
130         {
131             TranslateMessage(&msg);
132             DispatchMessage(&msg);
133         } // if
134     } // for
135 
136     return (int)msg.wParam; // завершаем работу приложения
137 } // _tWinMain
138 
139 // ----------------------------------------------------------------
140 LRESULT CALLBACK MyWindowProc(HWND hWnd, UINT uMsg, WPARAM
    wParam, LPARAM lParam)
141 {
142     switch (uMsg)
143     {
144         HANDLE_MSG(hWnd, WM_CREATE, OnCreate);
145         HANDLE_MSG(hWnd, WM_DESTROY, OnDestroy);
146         HANDLE_MSG(hWnd, WM_SIZE, OnSize);
147         HANDLE_MSG(hWnd, WM_COMMAND, OnCommand);
148     case WM_ADDITEM:
149         OnAddItem(hWnd);
150         return 0;
151     } // switch
152 
153     if (uFindMsgString == uMsg) // сообщение FINDMSGSTRING
154     {
155         OnFindMsgString(hWnd, (LPFINDREPLACE)lParam);
156         return 0;
157     } // if
158 
159     // передача необработанного сообщения
160     // оконной процедуре по умолчанию
161     return DefWindowProc(hWnd, uMsg, wParam, lParam);
162 } // MyWindowProc
163 
164 // ----------------------------------------------------------------
165 void OnIdle(HWND hwnd)
166 {
167     /* здесь можно выполнять обработку каких-либо данных */
```

```
168 } // OnIdle
169
170 // ---------------------------------------------------------------
171 BOOL PreTranslateMessage(LPMSG lpMsg)
172 {
173     BOOL bRet = TRUE;
174
175     if (!TranslateAccelerator(hWnd, hAccel, lpMsg))
176     {
177         bRet = IsDialogMessage(hDlg, lpMsg);
178
179         if (FALSE == bRet)
180             bRet = IsDialogMessage(hFindDlg, lpMsg);
181     } // if
182
183     return bRet;
184 } // PreTranslateMessage
185
186 // ---------------------------------------------------------------
187 BOOL OnCreate(HWND hwnd, LPCREATESTRUCT lpCreateStruct)
188 {
189     // создаем список
190     CreateWindowEx(0, TEXT("ListBox"), NULL,
191         WS_CHILD|WS_VISIBLE|WS_BORDER|LBS_STANDARD, 10, 10, 250,
    410, hwnd, (HMENU)IDC_LIST, lpCreateStruct->hInstance, NULL);
192
193     // создаем кнопку "Добавить запись"
194     CreateWindowEx(0, TEXT("Button"), TEXT("Добавить запись"),
195         WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, 270, 10, 200, 40,
    hwnd, (HMENU)ID_NEW_RECORD, lpCreateStruct->hInstance, NULL);
196
197     // создаем кнопку "Добавить неск. записей"
198     CreateWindowEx(0, TEXT("Button"), TEXT("Добавить неск.
    записей"),
199         WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, 270, 55, 200, 40,
    hwnd, (HMENU)ID_NEW_RECORD2, lpCreateStruct->hInstance, NULL);
200
201     // создаем кнопку "Удалить запись"
202     CreateWindowEx(0, TEXT("Button"), TEXT("Удалить запись"),
203         WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, 270, 100, 200, 40,
    hwnd, (HMENU)ID_DEL_RECORD, lpCreateStruct->hInstance, NULL);
204
205     // создаем кнопку "Найти запись"
206     CreateWindowEx(0, TEXT("Button"), TEXT("Найти запись"),
207         WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, 270, 145, 200, 40,
    hwnd, (HMENU)ID_FIND_RECORD, lpCreateStruct->hInstance, NULL);
```

```
208
209     // создаем кнопку "Изменить шрифт"
210     CreateWindowEx(0, TEXT("Button"), TEXT("Изменить шрифт"),
211         WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, 270, 210, 200, 40,
    hwnd, (HMENU)ID_FORMAT_FONT, lpCreateStruct->hInstance, NULL);
212
213     return TRUE;
214 } // OnCreate
215
216 // ----------------------------------------------------------------
217 void OnDestroy(HWND hwnd)
218 {
219     // удаляем созданный шрифт
220     if (NULL != hFont)
221         DeleteObject(hFont), hFont = NULL;
222
223     PostQuitMessage(0); // отправляем сообщение WM_QUIT
224 } // OnDestroy
225
226 // ----------------------------------------------------------------
227 void OnSize(HWND hwnd, UINT state, int cx, int cy)
228 {
229     if (state != SIZE_MINIMIZED)
230     {
231         // получим дескриптор списка
232         HWND hwndCtl = GetDlgItem(hwnd, IDC_LIST);
233         // изменяем высоту списка
234         MoveWindow(hwndCtl, 10, 10, 250, cy-20, TRUE);
235     } // if
236 } // OnSize
237
238 // ----------------------------------------------------------------
239 void OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
240 {
241     // получим дескриптор экземпляра приложения
242     HINSTANCE hInstance = GetWindowInstance(hwnd);
243
244     switch (id)
245     {
246     case ID_NEW_RECORD: // нажата кнопка "Добавить запись"
247         {
248             // создаем модальное диалоговое окно
249             int nDlgResult = DialogBox(hInstance,
    MAKEINTRESOURCE(IDD_DIALOG1), hwnd, DialogProc);
250
251             if (IDOK == nDlgResult)
```

```
252             {
253                 // отправляем окну сообщение о том, что нужно
    добавить запись
254                 SendMessage(hwnd, WM_ADDITEM, 0 ,0);
255             } // if
256         }
257         break;
258 
259     case ID_NEW_RECORD2: // нажата кнопка "Добавить неск.
    записей"
260         // если немодальное диалоговое окно еще не создано
261         if (IsWindow(hDlg) == FALSE)
262         {
263             // создаем немодальное диалоговое окно
264             hDlg = CreateDialog(hInstance, MAKEINTRESOURCE(
    IDD_DIALOG1), hwnd, DialogProc);
265             // отображаем немодальное диалоговое окно
266             ShowWindow(hDlg, SW_SHOW);
267         } // if
268         break;
269 
270     case ID_DEL_RECORD: // нажата кнопка "Удалить запись"
271         {
272             // получим дескриптор списка
273             HWND hwndCtl = GetDlgItem(hwnd, IDC_LIST);
274 
275             // определим текущий выделенный элемент в списке
276             int iItem = ListBox_GetCurSel(hwndCtl);
277 
278             if (iItem != -1)
279             {
280                 int mbResult = MessageBox(hwnd, TEXT("Удалить
    выбранный элемент?"), TEXT("SampleWin32"),
    MB_YESNO|MB_ICONQUESTION);
281 
282                 if (mbResult == IDYES)
283                 {
284                     // удаляем выделенный элемент из списка
285                     ListBox_DeleteString(hwndCtl, iItem);
286                 } // if
287             } // if
288         }
289         break;
290 
291     case ID_FIND_RECORD: // нажата кнопка "Найти запись"
292         if (0 == uFindMsgString)
```

```
293         {
294             // получим код сообщения FINDMSGSTRING
295             uFindMsgString = RegisterWindowMessage(
    FINDMSGSTRING);
296         } // if
297 
298         // если диалоговое окно "Найти" еще не создано
299         if (IsWindow(hFindDlg) == FALSE)
300         {
301             findDlg.lStructSize = sizeof(FINDREPLACE);
302 
303             // указываем дескриптор экземпляра приложения
304             findDlg.hInstance = hInstance;
305             // указываем дескриптор окна владельца
306             findDlg.hwndOwner = hwnd;
307 
308             // указываем строковый буфер
309             findDlg.lpstrFindWhat = szBuffer;
310             // указываем размер буфера
311             findDlg.wFindWhatLen = _countof(szBuffer);
312 
313             // создаем диалоговое окно "Найти"
314             hFindDlg = FindText(&findDlg);
315         } // if
316         break;
317 
318     case ID_FORMAT_FONT: // нажата кнопка "Изменить шрифт"
319         {
320             CHOOSEFONT cf = { sizeof(CHOOSEFONT) };
321 
322             // указываем дескриптор экземпляра приложения
323             cf.hInstance = hInstance;
324             // указываем дескриптор окна владельца
325             cf.hwndOwner = hwnd;
326 
327             LOGFONT lf;
328             ZeroMemory(&lf, sizeof(lf));
329 
330             cf.lpLogFont = &lf; // указываем структуру, которая
    будет использоваться для создания шрифта
331 
332             BOOL bRet = ChooseFont(&cf);
333 
334             if (FALSE != bRet)
335             {
336                 // создаем новый шрифт
```

```
337                 HFONT hNewFont = CreateFontIndirect(
    cf.lpLogFont);
338 
339                 if (NULL != hNewFont)
340                 {
341                     // удаляем созданный ранее шрифт
342                     if (NULL != hFont) DeleteObject(hFont);
343                     // устанавливаем новый шрифт для списка
344                     hFont = hNewFont;
345                     SendDlgItemMessage(hwnd, IDC_LIST,
    WM_SETFONT, (WPARAM)hFont, (LPARAM)TRUE);
346                 } // if
347             } // if
348         }
349         break;
350     } // switch
351 } // OnCommand
352 
353 // ------------------------------------------------------------
354 void OnAddItem(HWND hwnd)
355 {
356     // получим дескриптор списка
357     HWND hwndCtl = GetDlgItem(hwnd, IDC_LIST);
358 
359     // добавляем новый элемент в список
360     int iItem = ListBox_AddString(hwndCtl, szBuffer);
361 
362     // выделяем новый элемент
363     ListBox_SetCurSel(hwndCtl, iItem);
364 } // OnAddItem
365 
366 // ------------------------------------------------------------
367 void OnFindMsgString(HWND hwnd, LPFINDREPLACE lpFindReplace)
368 {
369     if (lpFindReplace->Flags & FR_FINDNEXT) // нажата кнопка
    "Найти далее"
370     {
371         // получим дескриптор списка
372         HWND hwndCtl = GetDlgItem(hwnd, IDC_LIST);
373 
374         // определим текущий выделенный элемент в списке
375         int iItem = ListBox_GetCurSel(hwndCtl);
376 
377         // выполним поиск указанного текста в списке
378         // сразу после текущего выделенного элемента
```

```
379         iItem = ListBox_FindString(hwndCtl, iItem, lpFindReplace-
    >lpstrFindWhat);
380
381         // выделяем найденный элемент
382         ListBox_SetCurSel(hwndCtl, iItem);
383
384         if (LB_ERR == iItem) // элемент не найден
385         {
386             MessageBox(hFindDlg, TEXT("Поиск завершен"),
    TEXT("SampleWin32"), MB_OK|MB_ICONINFORMATION);
387         } // if
388     } // if
389 } // OnFindMsgString
390
391 // ------------------------------------------------------------
392 INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM
    wParam, LPARAM lParam)
393 {
394     switch (uMsg)
395     {
396     case WM_INITDIALOG:
397         {
398             BOOL bRet = HANDLE_WM_INITDIALOG(hwndDlg, wParam,
    lParam, Dialog_OnInitDialog);
399             return SetDlgMsgResult(hwndDlg, uMsg, bRet);
400         }
401     case WM_CLOSE:
402         HANDLE_WM_CLOSE(hwndDlg, wParam, lParam, Dialog_OnClose);
403         return TRUE;
404     case WM_COMMAND:
405         HANDLE_WM_COMMAND(hwndDlg, wParam, lParam,
    Dialog_OnCommand);
406         return TRUE;
407     } // switch
408
409     return FALSE;
410 } // DialogProc
411
412 // ------------------------------------------------------------
413 BOOL Dialog_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM
    lParam)
414 {
415     // получим дескриптор окна редактируемого поля
416     HWND hwndEdit = GetDlgItem(hwnd, IDC_EDIT1);
417
418     // задаем максимальную длину текста в редактируемом поле
```

```
419     Edit_LimitText(hwndEdit, _countof(szBuffer)-1);
420
421     // задаем серый (фоновый) текст в редактируемом поле
422     Edit_SetCueBannerText(hwndEdit, L"Название новой записи");
423
424     return TRUE;
425 } // Dialog_OnInitDialog
426
427 // ----------------------------------------------------------------
428 void Dialog_OnClose(HWND hwnd)
429 {
430     if (hwnd == hDlg)
431     {
432         // уничтожаем немодальное диалоговое окно
433         DestroyWindow(hwnd);
434     } // if
435     else
436     {
437         // завершаем работу модального диалогового окна
438         EndDialog(hwnd, IDCLOSE);
439     } // else
440 } // Dialog_OnClose
441
442 // ----------------------------------------------------------------
443 void Dialog_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT
    codeNotify)
444 {
445     switch (id)
446     {
447     case IDOK: // нажата кнопка "OK"
448         {
449             // получим содержимое редактируемого поля
450             int cch = GetDlgItemText(hwnd, IDC_EDIT1, szBuffer,
    _countof(szBuffer));
451
452             if (0 == cch) // в редактируемого поле нет текста
453             {
454                 // получим дескриптор окна редактируемого поля
455                 HWND hwndEdit = GetDlgItem(hwnd, IDC_EDIT1);
456
457                 EDITBALLOONTIP ebt = { sizeof(EDITBALLOONTIP) };
458
459                 ebt.pszTitle = L"SampleWin32";
460                 ebt.pszText = L"Укажите название новой записи";
461                 ebt.ttiIcon = TTI_WARNING;
462
```

```
463                 Edit_ShowBalloonTip(hwndEdit, &ebt);
464             } // if
465             else if (hwnd == hDlg)
466             {
467                 // очистим редактируемое поле
468                 SetDlgItemText(hwnd, IDC_EDIT1, NULL);
469                 // отправляем окну-владельцу сообщение о том, что
    нужно добавить запись
470                 SendMessage(GetParent(hwnd), WM_ADDITEM, 0, 0);
471             } // if
472             else
473             {
474                 // завершаем работу модального диалогового окна
475                 EndDialog(hwnd, IDOK);
476             } // else
477         } // if
478         break;
479
480     case IDCANCEL: // нажата кнопка "Отмена"
481         if (hwnd == hDlg)
482         {
483             // уничтожаем немодальное диалоговое окно
484             DestroyWindow(hwnd);
485         } // if
486         else
487         {
488             // завершаем работу модального диалогового окна
489             EndDialog(hwnd, IDCANCEL);
490         } // else
491         break;
492     } // switch
493 } // Dialog_OnCommand
```

Задание к работе

1. Изучить элементы управления общего пользования, указанные в варианте задания. Особое внимание уделить тому, какие стили они поддерживают, какие сообщения принимают и какие уведомления отправляют родительскому окну. Включить в отчет назначение и описание изученных элементов управления.
2. Изучить диалоговые окна общего пользования, указанные в варианте задания. Включить в отчет назначение и описание изученных диалоговых окон.

3. Изучить оконное сообщение `WM_TIMER` и сообщения, указанные в варианте задания. Включить в отчет описание изученных оконных сообщений.

4. Разработать в Visual C++ оконное приложение Win32, которое:
   - должно создавать главное окно, содержащее меню и элементы управления, указанные в варианте задания;
   - должно обрабатывать комбинации быстрых клавиш;
   - должно создавать одно или несколько диалоговых окон с использованием шаблона;
   - должно создавать диалоговое окно сообщений;
   - должно создавать диалоговые окна, указанные в варианте задания;
   - должно обрабатывать оконное сообщение `WM_TIMER` и оконные сообщения, указанные в варианте задания.

   При необходимости можно также использовать и другие элементы управления.

5. Протестировать работу приложения, разработанного в п. 4, на компьютере под управлением Windows XP (или выше). Результаты тестирования отразить в отчете.

6. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Элементы управления	Диалоговое окно	Сообщения

1,16	Выпадающий список (Combo box)
	Гиперссылка (SysLink)
	Кнопка (Button)
	Поле ввода IP-адреса (IP address)
		Цвет (Color)
		Найти (Find)
			`WM_CHAR`
			`WM_KEYDOWN`
			`WM_LBUTTONDBLCLK`
			`WM_LBUTTONDOWN`
			`WM_LBUTTONUP`
			`WM_MOUSEMOVE`
			`WM_MOVE`

2,17	Групповая рамка (Group box)
	Дерево просмотра (Tree view)
	Надпись (Static text)
	Поле ввода горячей клавиши (Hot key)
		Шрифт (Font)
		Заменить (Replace)
			`WM_CHAR`
			`WM_KEYUP`
			`WM_MBUTTONDBLCLK`
			`WM_MBUTTONDOWN`
			`WM_MBUTTONUP`
			`WM_MOUSEWHEEL`
			`WM_MOVING`

№	Элементы управления	Диалоговое окно	Сообщения
3,18	Закладки (Tab control) Индикатор процесса (Progress bar) Переключатель (Radio button) Поле ввода даты или времени (Data time picker)	Открыть (Open) Найти (Find)	WM_MOUSEMOVE WM_RBUTTONDBLCLK WM_RBUTTONDOWN WM_RBUTTONUP WM_SIZE WM_SYSCHAR WM_SYSKEYDOWN
4,19	Календарь (Month calendar) Редактируемое поле (Edit box) Список (List box) Счетчик или стрелки (Spin)	Сохранить как (Save As) Заменить (Replace)	WM_LBUTTONDBLCLK WM_LBUTTONDOWN WM_LBUTTONUP WM_MOUSEWHEEL WM_SIZING WM_SYSCHAR WM_SYSKEYUP
5,20	Кнопка (Button) Поле ввода IP-адреса (IP address) Регулятор (Slider) Список просмотра (List view)	Печать (Print) Найти (Find)	WM_CHAR WM_KEYDOWN WM_MBUTTONDBLCLK WM_MBUTTONDOWN WM_MBUTTONUP WM_MOUSEMOVE WM_MOVE
6,21	Выпадающий список (Combo box) Надпись (Static text) Поле ввода горячей клавиши (Hot key) Флажок (Check box)	Цвет (Color) Заменить (Replace)	WM_CHAR WM_KEYUP WM_MOUSEWHEEL WM_MOVING WM_RBUTTONDBLCLK WM_RBUTTONDOWN WM_RBUTTONUP
7,22	Гиперссылка (SysLink) Дерево просмотра (Tree view) Переключатель (Radio button) Поле ввода даты или времени (Data time picker)	Шрифт (Font) Найти (Find)	WM_LBUTTONDBLCLK WM_LBUTTONDOWN WM_LBUTTONUP WM_MOUSEMOVE WM_SIZE WM_SYSCHAR WM_SYSKEYDOWN
8,23	Закладки (Tab control) Индикатор процесса (Progress bar) Редактируемое поле (Edit box) Счетчик или стрелки (Spin)	Открыть (Open) Заменить (Replace)	WM_MBUTTONDBLCLK WM_MBUTTONDOWN WM_MBUTTONUP WM_MOUSEWHEEL WM_SIZING WM_SYSCHAR WM_SYSKEYUP

№	Элементы управления	Диалоговое окно	Сообщения
9,24	Календарь (Month calendar) Поле ввода IP-адреса (IP address) Регулятор (Slider) Список (List box)	Сохранить как (Save As) Найти (Find)	WM_CHAR WM_KEYDOWN WM_MOUSEMOVE WM_MOVE WM_RBUTTONDBLCLK WM_RBUTTONDOWN WM_RBUTTONUP
10,25	Поле ввода горячей клавиши (Hot key) Список просмотра (List view) Счетчик или стрелки (Spin) Флажок (Check box)	Печать (Print) Заменить (Replace)	WM_CHAR WM_KEYUP WM_LBUTTONDBLCLK WM_LBUTTONDOWN WM_LBUTTONUP WM_MOUSEWHEEL WM_MOVING
11,26	Выпадающий список (Combo box) Кнопка (Button) Поле ввода даты или времени (Data time picker) Регулятор (Slider)	Цвет (Color) Найти (Find)	WM_MBUTTONDBLCLK WM_MBUTTONDOWN WM_MBUTTONUP WM_MOUSEMOVE WM_SIZE WM_SYSCHAR WM_SYSKEYDOWN
12,27	Гиперссылка (SysLink) Дерево просмотра (Tree view) Индикатор процесса (Progress bar) Редактируемое поле (Edit box)	Шрифт (Font) Заменить (Replace)	WM_MOUSEWHEEL WM_RBUTTONDBLCLK WM_RBUTTONDOWN WM_RBUTTONUP WM_SIZING WM_SYSCHAR WM_SYSKEYUP
13,28	Групповая рамка (Group box) Закладки (Tab control) Поле ввода IP-адреса (IP address) Регулятор (Slider)	Открыть (Open) Найти (Find)	WM_CHAR WM_KEYDOWN WM_LBUTTONDBLCLK WM_LBUTTONDOWN WM_LBUTTONUP WM_MOUSEMOVE WM_MOVE
14,29	Индикатор процесса (Progress bar) Кнопка (Button) Поле ввода горячей клавиши (Hot key) Список (List box)	Сохранить как (Save As) Заменить (Replace)	WM_CHAR WM_KEYUP WM_MBUTTONDBLCLK WM_MBUTTONDOWN WM_MBUTTONUP WM_MOUSEWHEEL WM_MOVING

№	Элементы управления	Диалоговое окно	Сообщения
15,30	Календарь (Month calendar) Поле ввода даты или времени (Data time picker) Список просмотра (List view) Флажок (Check box)	Печать (Print) Найти (Find)	WM_MOUSEMOVE WM_RBUTTONDBLCLK WM_RBUTTONDOWN WM_RBUTTONUP WM_SIZE WM_SYSCHAR WM_SYSKEYDOWN

Контрольные вопросы

1. Что в Windows называют окном?
2. Что такое главное окно приложения Windows? Сколько таких окон может быть у приложения?
3. Что в Windows называют диалоговое окном? Какие различают виды диалоговых окон?
4. Что в Windows называют элементами управления?
5. Что такое клиентская и не клиентская область окна? Что они в себя включают?
6. Какие системы координат используются в функциях Win32 API, работающих с окнами?
7. Какие функции Win32 API следует использовать для преобразования из одной системы координат в другую?
8. Что такое оконный класс? Как создать новый оконный класс?
9. Что такое предопределенные оконные классы элементов управления общего пользования? Какие предопределенные оконные классы существуют?
10. Какие функции Win32 API следует использовать для того, чтобы создать окно?
11. Какие функции Win32 API следует использовать для создания различных диалоговых окон?
12. Какие функции Win32 API возвращают дескриптор окна?
13. Какие функции Win32 API изменяют отображение окна?
14. Какие функции Win32 API возвращают различные характеристики окна или его оконного класса?
15. Что в Windows называют оконным сообщением?
16. Что такое очередь сообщений?

17. Что такое цикл обработки сообщений? Как обрабатываются сообщения диалоговых окон?
18. Что такое оконная процедура? Для чего применяется оконная процедура?
19. Что такое оконная процедура диалогового окна? Чем она отличается от обычной оконной процедуры?
20. Какие оконные сообщения используются наиболее часто?
21. Чем отличаются оконные сообщения `WM_CREATE` и `WM_INITDIALOG`?
22. Чем отличаются оконные сообщения `WM_DESTROY` и `WM_CLOSE`?
23. Чем отличаются оконные сообщения `WM_COMMAND` и `WM_NOTIFY`?
24. В чем разница между синхронными и асинхронными оконными сообщениями?
25. Какую функцию Win32 API следует использовать для того, чтобы отправить синхронное оконное сообщение?
26. Какую функцию Win32 API следует использовать для того, чтобы отправить асинхронное оконное сообщение?
27. Зачем отправлять оконные сообщения элементам управления? Какие функции Win32 API следует для этого использовать?
28. Какие существуют макросы для отправки оконных сообщений элементам управления?
29. Какую функцию Win32 API следует использовать для того, чтобы отправить сообщение `WM_QUIT`? Почему для этого нельзя использовать другие функции отправки оконных сообщений?
30. Как и для чего применяют распаковщики сообщений?
31. Что такое приложение Win32? Как в Visual C++ создать проект приложения Win32?
32. Для чего в приложении Win32 предназначена функция `WinMain`? Какие существуют варианты функции `WinMain`?
33. Что такое точка входа в приложение Win32?
34. Что такое и для чего применяется манифест приложения?
35. Что такое ресурсы приложения?
36. Как создать главное меню приложения?
37. Как обрабатываются комбинации быстрых клавиш?

ЛАБОРАТОРНАЯ РАБОТА № 3
ПРОЦЕССЫ И ПОТОКИ В WINDOWS

Цель работы

Знакомство с основами управления процессами и потоками, а также изучение механизмов синхронизации потоков. Получение практических навыков использования объектов ядра Windows.

Основные понятия

Процесс (process) представляет собой экземпляр исполняемого приложения, обладающий независимым виртуальным адресным пространством, в котором могут размещаться данные, недоступные другим процессам. В свою очередь, внутри каждого процесса могут независимо выполняться один или несколько *потоков* (threads). Именно потоки отвечают за исполнение кода, содержащегося в виртуальном адресном пространстве процесса. При создании процесса автоматически создается его первый поток, называемый *главным потоком* (main thread).

Объекты ядра

Объект ядра (kernel object) представляет собой структуру данных, созданную ядром операционной системы, и в которой содержится информация об объекте операционной системы (дескриптор безопасности, счетчик использования и др.).

Операционная система Windows позволяет создавать, открывать и выполнять различные операции с несколькими типами объектов ядра, к которым в том числе относятся процессы и потоки. Полный список типов объектов ядра Windows можно получить с помощью бесплатной утилиты WinObj (рис. 3.1), созданной Sysinternals (ее можно скачать по ссылке: http://download.sysinternals.com/files/WinObj.zip). Утилиту WinObj следует запускать от имени «Администратора».

Как и для всех объектов Windows, доступ к объектам ядра осуществляется через функции Win32 API, которым в качестве параметра передается дескриптор объекта ядра.

Создание объекта ядра

В Win32 API имеется множество функций, создающих объекты ядра. Какую именно из этих функций следует использовать, зависит от

типа объекта ядра. (Подробно функции создания объектов ядра будут рассматриваться по мере знакомства с этими объектами.)

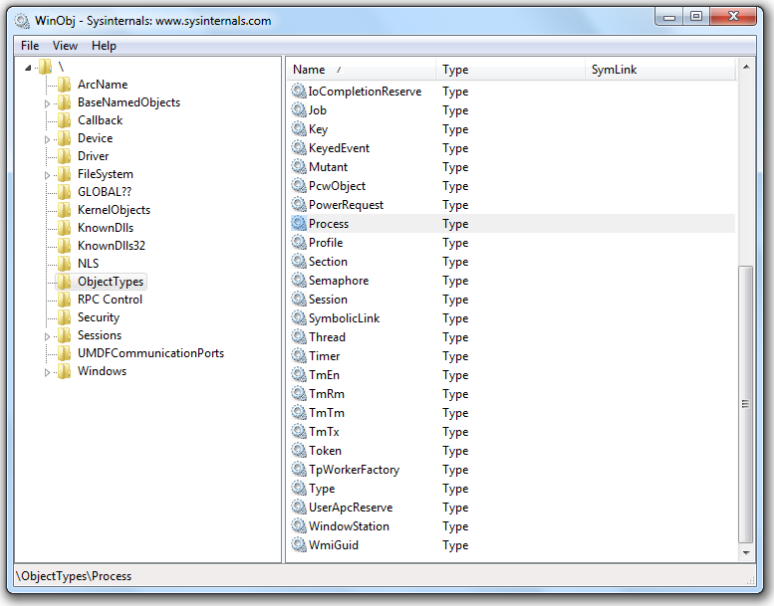

Рис. 3.1. Утилита WinObj (доступные типы объектов ядра)

Все функции, создающие объекты ядра, возвращают дескрипторы этих объектов. Значение дескриптора объекта ядра привязано к конкретному процессу и не действительно в других процессах. Это связано с тем, что на самом деле дескрипторы объектов ядра представляют собой индексы в специальной таблице дескрипторов, принадлежащей процессу, и в которой хранится информация обо всех открытых объектах ядра.

Таблица дескрипторов создается при инициализации процесса, и изначально она пуста. Но стоит одному из потоков процесса вызвать какую-нибудь функцию, создающую объект ядра, как тут же в таблицу дескрипторов данного процесса будет добавлена соответствующая запись.

Следует отметить, что конкретный объект ядра (но, не дескриптор этого объекта) может одновременно использоваться сразу несколькими процессами. Операционная система всегда знает, сколько процессов используют тот или иной объект ядра, поскольку в каждом объекте

есть счетчик его использования. Когда объект создается, счетчику присваивается значение 1. Если какой-либо процесс обращается к объекту, счетчик этого объекта увеличивается на 1.

Безопасность и наследование

Большинство функций, создающие объекты ядра, принимают в качестве аргумента указатель на структуру SECURITY_ATTRIBUTES, которая содержит *дескриптор безопасности* (security descriptor) объекта ядра и определят, будет ли дескриптор созданного объекта наследуемым.

Структура SECURITY_ATTRIBUTES описывается следующим образом:

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength; // размер структуры в байтах
  LPVOID lpSecurityDescriptor; // дескриптор безопасности
  BOOL bInheritHandle; // признак наследования
} SECURITY_ATTRIBUTES;
```

Первое поле, *nLength*, должно содержать значение равное размеру структуры в байтах, как правило, его получают с помощью оператора sizeof.

Второе поле, *lpSecurityDescriptor*, – дескриптор безопасности, который описывает, кто имеет право доступа к создаваемому объекту ядра. Если это поле установлено в NULL, то используются параметры безопасности по умолчанию: создатель объекта ядра и любой пользователь из группы «Администраторы» получают полный доступ к объекту, а все прочие к нему не допускаются.

Третье поле, *bInheritHandle*, определяет, является ли возвращаемый дескриптор наследуемым, когда процесс порождает дочерний процесс. Если это поле принимает значение равное TRUE, дескриптор наследуется. По умолчанию дескриптор объекта ядра не наследуется.

Если в функцию, создающую объект ядра, в качестве аргумента вместо указателя на структуру SECURITY_ATTRIBUTES передать NULL, то будет создан объект с параметрами безопасности и наследования по умолчанию.

Закрытие объекта ядра

Не зависимо от того, как именно был создан объект ядра, если он более не нужен, его следует закрыть с помощью функции CloseHandle:

```
BOOL CloseHandle(HANDLE hObject);
```

Параметр *hObject* – дескриптор объекта ядра, который необходимо закрыть. Если дескриптор не верен, функция CloseHandle возвращает значение FALSE.

Функция `CloseHandle` удаляет соответствующую запись из таблицы дескрипторов и уменьшает на 1 счетчик объекта ядра. Как только счетчик объекта ядра обнулится, этот объект уничтожается; но, если счетчик не обнулен, объект остается. Независимо от того уничтожен объект ядра или нет, получить доступ к этому объекту уже нельзя, так как запись о нем была удалена из таблицы дескрипторов.

Следует отметить, что если забыть закрыть объект ядра, утечки не произойдет. По завершении процесса гарантировано закрываются все открытые объекты ядра. Если при этом счетчик какого-либо объекта ядра обнуляется, этот объект будет уничтожен.

Совместное использование объектов ядра

Время от времени возникает необходимость в совместном использовании одних и тех же объектов ядра несколькими процессами. Рассмотрим по порядку механизмы, благодаря которым это возможно.

Наследование дескриптора объекта ядра

Наследование применяется в случае, когда процесс решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам ядра.

Во-первых, для этого нужно, чтобы дескрипторы созданных объектов ядра были наследуемыми. Поэтому при создании объекта ядра, процесс должен инициализировать структуру `SECURITY_ATTRIBUTES` (присвоив полу *`bInheritHandle`* значение `TRUE`), а затем передать адрес этой структуры в функцию, создающую объект ядра.

Во-вторых, процесс должен порождать дочерний процесс с помощью функции `CreateProcess`, передав в параметре *`bInheritHandles`* значение `TRUE`. В результате дочерний процесс наследует дескрипторы родительского процесса. (Подробно функция `CreateProcess` рассматривается в разделе «Процессы».)

При создании дочернего процесса, как обычно формируется новая таблица дескрипторов (пока пустая). После этого в нее копируются все наследуемые дескрипторы из таблицы дескрипторов родительского процесса. При этом важно то, что значения дескрипторов сохраняются, следовательно, они будут идентичны в обоих процессах (родительском и дочернем).

Помимо копирования записей из таблицы дескрипторов, также увеличиваются на 1 значения счетчиков соответствующих объектов ядра, поскольку эти объекты используются еще одним процессом. Поэтому, если сразу после создания дочернего процесса, родительский процесс закроет свой дескриптор объекта ядра, это никак не отразится на работе дочернего процесса с этим объектом ядра.

Следует отметить, что наследуются только дескрипторы объектов ядра, существующие на момент создания дочернего процесса. Если родительский процесс создаст после этого новые объекты ядра с наследуемыми дескрипторами, то эти дескрипторы будут уже недоступны созданному дочернему процессу.

При наследовании дескрипторов объектов ядра дочерний процесс не знает, какие дескрипторы он унаследовал. Поэтому для передачи значений наследуемых дескрипторов используются различные формы межпроцессной связи. Например, значение дескриптора объекта ядра можно передавать в дочерний процесс как аргумент в командной строке. Другие формы межпроцессной связи тоже годятся.

Время от времени, может возникать ситуация, когда процесс создает объект ядра с наследуемым дескриптором, а затем порождает два дочерних процесса. Но наследуемый дескриптор нужен только одному из них. Для этого в Win32 API имеется функция `SetHandleInformation`, которая способна изменять свойства дескриптора объекта ядра.

```
BOOL SetHandleInformation(HANDLE hObject, DWORD dwMask,
    DWORD dwFlags);
```

Первый параметр, *`hObject`*, – дескриптор объекта ядра, свойства которого необходимо изменить.

Второй параметр, *`dwMask`*, определяет какой флаг (или флаги) дескриптора объекта ядра необходимо изменить (используются те же значения, что и для параметра *`dwFlags`*). Третий параметр, *`dwFlags`*, устанавливает или сбрасывает флаги. Этот параметр может принимать значение ноль или одно (или несколько) из следующих значений:

- `HANDLE_FLAG_INHERIT` – если установить этот флаг, дескриптор объекта ядра будет являться наследуемым;
- `HANDLE_FLAG_PROTECT_FROM_CLOSE` – если установить этот флаг, дескриптор объекта ядра нельзя будет закрывать. Если какой-нибудь поток попытается вызвать функцию `CloseHandle` для такого дескриптора, это приведет к исключению.

Если функция `SetHandleInformation` завершается успешно, возвращаемое значение отлично от `FALSE`.

Например, чтобы сделать дескриптор объекта ядра наследуемым нужно установить флаг `HANDLE_FLAG_INHERIT`, а чтобы сделать его ненаследуемым – сбросить этот флаг:

```
1  // установка флага HANDLE_FLAG_INHERIT
2  SetHandleInformation(hObject, HANDLE_FLAG_INHERIT,
   HANDLE_FLAG_INHERIT);
```

```
3   // сброс флага HANDLE_FLAG_INHERIT
4   SetHandleInformation(hObject, HANDLE_FLAG_INHERIT, 0);
```

Чтобы определить, какие флаги установлены для дескриптора объекта ядра, следует использовать функцию `GetHandleInformation`:

```
BOOL GetHandleInformation(HANDLE hObject, LPDWORD lpdwFlags);
```

Первый параметр, *`hObject`*, – дескриптор объекта ядра. Второй параметр, *`lpdwFlags`*, указывает на переменную типа `DWORD`, в которую возвращаются текущие флаги для заданного дескриптора.

Если функция `GetHandleInformation` завершается успешно, возвращаемое значение отлично от `FALSE`.

Определить является ли дескриптор объекта ядра наследуемым или нет, можно следующим образом:

```
1   DWORD dwFlags = 0;
2   GetHandleInformation(hObject, &dwFlags);
3
4   if (dwFlags & HANDLE_FLAG_INHERIT)
5   {
6       /* дескриптор объекта ядра является наследуемым */
7   } // if
```

Дублирование дескриптора объекта ядра

Операция дублирования дескриптора объекта ядра заключается в том, что берется соответствующая запись в таблице дескрипторов одного процесса и создается ее копия в таблице другого. Для выполнения такой операции в Win32 API существует функция `DuplicateHandle`.

```
BOOL DuplicateHandle(HANDLE hSourceProcessHandle,
    HANDLE hSourceHandle, HANDLE hTargetProcessHandle,
    LPHANDLE lpTargetHandle, DWORD dwDesiredAccess,
    BOOL bInheritHandle, DWORD dwOptions);
```

Первый параметр, *`hSourceProcessHandle`*, представляет собой дескриптор процесса, который содержит дублируемый дескриптор объекта ядра. Второй параметр, *`hSourceHandle`*, – это исходный (дублируемый) дескриптор объекта ядра. Значение этого дескриптора хранится в таблице дескрипторов процесса, на который указывает *`hSourceProcessHandle`*.

Третий параметр, *`hTargetProcessHandle`*, представляет собой дескриптор процесса, который должен получить дублированный дескриптор объекта ядра. Четвертый параметр, *`lpTargetHandle`*, – указатель на переменную типа `HANDLE`, которая получает значение дублиро-

ванного дескриптора объекта ядра. Полученный дублированный дескриптор будет храниться в таблице дескрипторов процесса, на который указывает параметр *hTargetProcessHandle*.

Пятый параметр, *dwDesiredAccess*, задает флаги требуемого доступа к дублированному дескриптору. В некоторых случаях, дублированный дескриптор объекта ядра может иметь больше прав доступа, чем исходный (дублируемый) дескриптор объекта ядра. Однако в большинстве случаев, дублированный дескриптор не может иметь больше прав доступа, чем исходный дескриптор. Перечень флагов доступа, которые могут быть установлены для каждого типа объекта ядра, можно найти в документации Platform SDK.

Шестой параметр, *bInheritHandle*, определяет, будет ли дублированный дескриптор объекта ядра являться наследуемым или нет.

Последний параметр, *dwOptions*, может быть нулевой, или любой комбинацией следующих значений:

- DUPLICATE_CLOSE_SOURCE – исходный дескриптор объекта ядра закрывается, независимо от того, будет ли операция дублирования завершена успешно или с ошибкой;
- DUPLICATE_SAME_ACCESS – дублированный дескриптор объекта ядра имеет те же права доступа, что и исходный дескриптор объекта ядра (в этом случае параметр *dwDesiredAccess* игнорируется).

Если функция DuplicateHandle завершается успешно, возвращаемое значение отлично от FALSE.

Именованные объекты ядра

Многие объекты ядра (но не все) допускают именование. Имя объекта ядра указывается при вызове функции, создающей этот объект.

При создании именованного объекта ядра, сначала проверяется, существует ли объект с таким же именем. Если такого объекта не существует, создается именованный объект ядра и добавляется соответствующая запись в таблицу дескрипторов вызывающего процесса.

Если объект с таким именем уже существует, выполняется проверка типа объекта и прав доступа вызывающего процесса к этому объекту. Если типы двух объектов с одинаковыми именами не совпадают или вызывающий процесс не имеет прав доступа к существующему объекту, вызов функции, создающей этот объект, завершается ошибкой. Если же все в порядке, счетчик объекта ядра увеличивается на 1 и создается новая запись в таблице дескрипторов вызывающего процесса, а функция, создающая объект ядра, возвращает дескриптор существующего объекта. При этом функция GetLastError вернет зна-

чение ERROR_ALREADY_EXISTS, что позволит приложению определить: был ли создан новый объект ядра или просто открыт уже существующий.

В Win32 API имеется также набор функций, которые не создают объекты ядра, но способны их открывать. (Подробно эти функции будут рассматриваться по мере знакомства с объектами ядра.)

Все функции, открывающие объекты ядра, принимают в качестве параметров имя (или идентификатор) объекта ядра и флаги доступа к этому объекту. Каждая такая функция просматривает созданные объекты ядра, пытаясь найти совпадение. Если объекта с указанным именем нет, вызов такой функции завершается ошибкой. Если же объект ядра с заданным именем существует и если его тип идентичен тому, что был указан, выполняется проверка, разрешен ли к этому объекту для вызывающего процесса запрошенный уровень доступа. Если такой доступ разрешен, таблица дескрипторов вызывающего процесса обновляется, и счетчик объекта увеличивается на 1.

Чтобы исключить конфликты между несколькими пользовательскими сеансами, в которых запускается одно и то же приложение, создающее именованные объекты ядра, в каждом пользовательском сеансе формируется свое пространство имен. Таким образом, ни из какого сеанса нельзя получить доступ к объектам другого сеанса, даже если известны их имена. Объекты ядра, которые должны быть доступны приложениям всех пользовательских сеансов, используют единое глобальное пространство имен.

Каждый именованный объект ядра по умолчанию помещается в пространство имен пользовательского сеанса. Однако его можно поместить в глобальное пространство имен, поставив перед именем объекта префикс «Global\». Если необходимо явно указать, что объект ядра должен находиться в пространстве имен пользовательского сеанса, следует использовать префикс «Local\».

Процессы

Получить список процессов в Windows можно с помощью системной утилиты «Диспетчер задач» (Task Manager) (рис. 3.2).

Чтобы получить информацию об открытых процессом объектах ядра и загруженных модулях, можно воспользоваться бесплатной утилитой Process Explorer от Sysinternals (ее можно скачать по ссылке: http://download.sysinternals.com/files/ProcessExplorer.zip). Рабочая область Process Explorer состоит из двух окон (рис. 3.3). В верхнем окне отображается список запущенных процессов. Информация, которая отображается в нижнем окне (lower pane), зависит от выбранного

режима. В режиме дескрипторов отображаются все открытые дескрипторы выбранного в верхнем окне процесса, а в режиме библиотек DLL – все загруженные процессом динамические библиотеки и отображенные в память файлы.

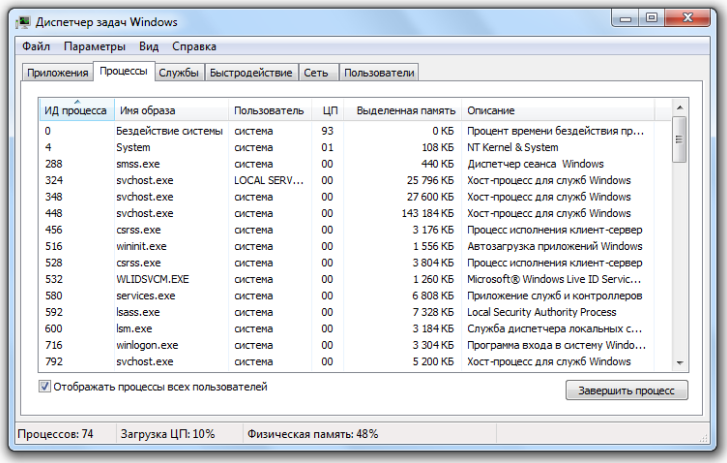

Рис. 3.2. Список запущенных процессов в Windows

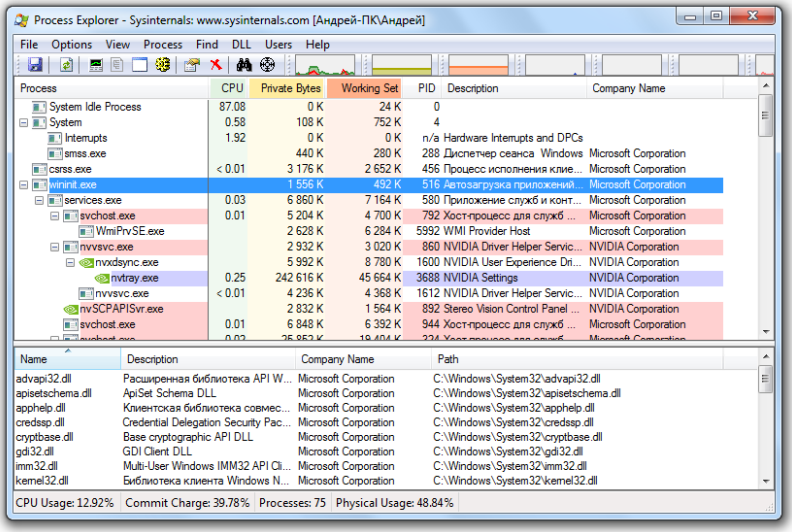

Рис. 3.3. Утилита Process Explorer

Создание процесса

При создании процесса, создается объект ядра «процесс» с начальным значением счетчика равным 1. Этот объект, содержит информацию о соответствующем процессе, которая позволяет операционной системе управлять этим процессом. Затем для нового процесса создается виртуальное адресное пространство, в которое загружается код исполняемого файла.

После этого для главного потока нового процесса создается объект ядра «поток» (счетчик равен 1), через который операционная система управляет потоком. Главный поток начинается с точки входа (`WinMainCRTStartup`, `wWinMainCRTStartup`, `mainCRTStartup` или `wmainCRTStartup`), в которой выполняется инициализация различных библиотек C/C++, загрузка необходимых DLL и создание глобальных переменные. Когда все это будет сделано, вызывается главная функция приложения (`WinMain`, `wWinMain`, `main` или `wmain`).

Создавая процессы и управляя ими, приложение может организовывать параллельное выполнение нескольких задач. Новый процесс создается с помощью функции `CreateProcess`:

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

Первый параметр, *lpApplicationName*, указывает на строку, которая содержит имя исполняемого файла, который нужно запустить. На самом деле, этот параметр используется только для поддержки подсистемы POSIX в Windows. Поэтому следует устанавливать этот параметр в `NULL`.

Второй параметр, *lpCommandLine*, позволяет указать полную командную строку, используемую при создании нового процесса. Предполагается, что первый аргумент командной строки представляет собой имя исполняемого файла. Если в имени этого файла не указано расширение, используется расширение по умолчанию (`.exe`). Если в имени исполняемого файла не задан полный путь, то этот файл ищется в следующих каталогах (в указанном порядке):

1. Каталог, содержащий исполняемый файл родительского процесса.

2. Текущий каталог родительского процесса.
3. Системный каталог Windows (%WINDIR%\system32).
4. Каталог Windows (%WINDIR%).
5. Каталоги, перечисленные в переменной окружения PATH.

Найдя нужный исполняемый файл, функция `CreateProcess` создает новый процесс.

Важно отметить тот факт, что параметр *`lpCommandLine`* указывает на строку, которая не является константой. Дело в том, что функция `CreateProcess` в процессе выполнения изменяет переданную командную строку, но перед возвратом управления восстанавливает ее. Из-за этого может возникать ошибка доступа. Например, следующий программный код приведет к ошибке, потому что строка «notepad» размещена в блоке памяти только для чтения:

```
CreateProcess(NULL, TEXT("notepad"), ...);
```

Лучший способ решения этой проблемы – перед вызовом функции `CreateProcess` копировать командную строку во временный буфер:

```
1  TCHAR szCmdLine[] = TEXT("notepad");
2  CreateProcess(NULL, szCmdLine, ...);
```

Параметры *`lpProcessAttributes`* и *`lpThreadAttributes`* позволяют определить параметры безопасности и наследования для создаваемых объектов ядра «процесс» и «поток» соответственно. Эти параметры можно установить в `NULL`, тогда эти объекты будут созданы с параметрами безопасности и наследования по умолчанию.

Пятый параметр, *`bInheritHandles`*, определяет, будет ли новый процесс наследовать, каждый наследуемый дескриптор родительского процесса. Если этот параметр принимает значение `TRUE`, дескрипторы наследуются.

Шестой параметр, *`dwCreationFlags`*, определяет флаги, которые влияют на то, как создается новый процесс. В табл. 3.1 перечислены флаги, которые могут быть использованы. Они могут быть установлены в любой комбинации, кроме тех, на которые обращено внимание. В этой таблице приведен неполный список возможных флагов. Полный список см. в документации Platform SDK.

Параметр *`dwCreationFlags`* разрешает также задавать класс приоритета процесса, который влияет на распределение процессорного времени между потоками процессов. Как правило, не рекомендуется задавать класс приоритета, так как операционная система присваивает новому процессу класс приоритета по умолчанию. Возможные классы перечислены в табл. 3.4 (см. раздел «Приоритет процесса»).

Таблица 3.1. Флаги создания процесса

Флаг	**Описание**
CREATE_NEW_CONSOLE	Приводит к созданию нового консольного окна для нового процесса. Этот флаг нельзя комбинировать с DETACHED_PROCESS
CREATE_NO_WINDOW	Не дает отображать никаких окон в новом процессе
CREATE_SUSPENDED	Позволяет создавать процесс и в тоже время приостанавливать его главный поток
CREATE_UNICODE_ENVIRONMENT	Сообщает, что блок переменных окружения нового процесса должен содержать Unicode-строки. По умолчанию этот блок формируется на основе ANSI-строк
DETACHED_PROCESS	Блокирует доступ процессу, порожденному процессом консольного приложения, к консольному окну этого приложения

Седьмой параметр, *lpEnvironment*, указывает на блок памяти, хранящий строки переменных окружения, которые будут передаваться новому процессу. Как правило, этот параметр устанавливают в NULL, в результате чего дочерний процесс наследует переменные окружения от родительского процесса.

Восьмой параметр, *lpCurrentDirectory*, позволяет задать рабочий каталог для нового процесса. Если, этот параметр установить в NULL, рабочий каталог нового процесса будет таким же, как и у родительского процесса.

Последние два параметра, *lpStartupInfo* и *lpProcessInformation*, указывают на структуры STARTUPINFO и PROCESS_INFORMATION соответственно. Эти структуры будут рассмотрены отдельно.

Функция CreateProcess возвращает значение TRUE, если новый процесс и его главный поток были успешно созданы. Следует отметить, что функция CreateProcess возвращает управление до окончания инициализации процесса. Это может привести к тому, что при возникновении ошибки во время инициализации (например, не будет найдена нужная библиотека DLL), дочерний процесс завершит свою работу, а родительский процесс ничего об этом не узнает.

Структура STARTUPINFO

Структура STARTUPINFO имеет восемнадцать полей, некоторые из которых имеют смысл, только если дочерний процесс создает консольное окно, а другие – если этот процесс создает перекрывающее окно, которым являет главное окно приложения.

```
typedef struct _STARTUPINFO {
  DWORD  cb; // размер структуры в байтах
  LPTSTR lpReserved; // зарезервировано
  LPTSTR lpDesktop; // имя рабочего стола
  LPTSTR lpTitle; // заголовок консольного окна
  DWORD dwX; // x-координата окна
  DWORD dwY; // y-координата окна
  DWORD dwXSize; // ширина окна в пикселях
  DWORD dwYSize; // высота окна в пикселях
  DWORD dwXCountChars; // ширина консольного окна в символах
  DWORD dwYCountChars; // высота консольного окна в символах
  DWORD dwFillAttribute; // атрибуты цвета в консольном окне
  DWORD dwFlags; // набор флагов
  WORD wShowWindow; // параметр отображения окна
  WORD cbReserved2; // зарезервировано
  LPBYTE lpReserved2; // зарезервировано
  HANDLE hStdInput; // дескриптор буфера ввода
  HANDLE hStdOutput; // дескриптор буфера вывода
  HANDLE hStdError; // дескриптор буфера ошибки
} STARTUPINFO, *LPSTARTUPINFO;
```

Первое поле, *cb*, должно содержать размер структуры в байтах, как правило, его получают с помощью оператора `sizeof`.

Второе поле, *lpReserved*, не используется и должно принимать значение равное `NULL`.

Третье поле, *lpDesktop*, идентифицирует имя рабочего стола, на котором должно запускаться приложение. Если это поле принимает значение равное `NULL`, используется текущий рабочий стол.

Четвертое поле, *lpTitle*, определяет заголовок консольного окна. Если это поле принимает значение равное `NULL`, в заголовок выводится имя исполняемого файла. Это поле имеет смысл только для консольного приложения.

Поля *dwX* и *dwY* указывают координаты (в пикселях) окна приложения. В оконных приложениях эти координаты используются при создании главного окна, если параметр *X* функции `CreateWindowEx` принимает значение `CW_USEDEFAULT`. В консольных приложениях эти координаты определяют верхний левый угол консольного окна.

Поля *dwXSize* и *dwYSize* определяют ширину и высоту (в пикселях) окна приложения. В оконных приложениях эти значения используются при создании главного окна, если параметр *nWidth* функции `CreateWindowEx` принимает значение `CW_USEDEFAULT`. В консольных приложениях эти значения определяют ширину и высоту консольного окна.

Поля *dwXCountChars* и *dwYCountChars* определяют ширину и высоту (в символах) консольного окна приложения. Это поле имеет смысл только для консольного приложения.

Одиннадцатое поле, *dwFillAttribute*, задает цвет текста и фона в консольном окне приложения. Это поле имеет смысл только для консольного приложения. Значением этого поля может быть комбинация предопределенных констант (см. в документации Platform SDK). Например, следующая комбинация производит красный текст на желтом фоне:

```
FOREGROUND_RED | BACKGROUND_RED | BACKGROUND_GREEN
```

Двенадцатое поле, *dwFlags*, содержит набор флагов, позволяющих управлять созданием дочернего окна. Эти флаги сообщают функции `CreateProcess`, содержат ли прочие поля структуры `STARTUPINFO` полезную информацию или их можно игнорировать. Список допустимых флагов приведен в табл. 3.2.

Таблица 3.2. Флаги поля *dwFlags*

Флаг	**Описание**
`STARTF_USECOUNTCHARS`	Если этот флаг не установлен, поля *dwXCountChars* и *dwYCountChars* игнорируются
`STARTF_USEFILLATTRIBUTE`	Если этот флаг не установлен, поле *dwFillAttribute* игнорируется
`STARTF_USEPOSITION`	Если этот флаг не установлен, поля *dwX* и *dwY* игнорируются
`STARTF_USESHOWWINDOW`	Если этот флаг не установлен, поле *wShowWindow* игнорируется
`STARTF_USESIZE`	Если этот флаг не установлен, поля *dwXSize* и *dwYSize* игнорируются
`STARTF_USESTDHANDLES`	Если этот флаг не установлен, поля *hStdInput*, *hStdOutput* и *hStdError* игнорируются

Тринадцатое поле, *wShowWindow*, определяет способ отображения окна. Значение этого поля передается через параметр *nCmdShow* в функцию `WinMain` (или `wWinMain`). Кроме того, значение этого поля используется, если при вызове функции `ShowWindow` передается значение `SW_SHOWDEFAULT`. Это поле имеет смысл только для оконного приложения.

Поля *cbReserved2* и *lpReserved2* зарезервированы и должны принимать значения `0` и `NULL` соответственно.

Последние три поля (*hStdInput*, *hStdOutput* и *hStdError*) определяют дескрипторы буферов для консольного ввода-вывода. Если эти

поля принимают значение равное NULL, используются стандартные буферы для консольного ввода-вывода.

Структура PROCESS_INFORMATION

Завершая свою работу, функция CreateProcess заполняет поля структуры PROCESS_INFORMATION, которая представляет собой следующее:

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess; // дескриптор процесса
  HANDLE hThread; // дескриптор потока
  DWORD dwProcessId; // идентификатор процесса
  DWORD dwThreadId; // идентификатор потока
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Первые два поля, *hProcess* и *hThread*, представляют собой соответственно дескриптор созданного процесса и дескриптор его главного потока. Эти дескрипторы хранятся в таблице дескрипторов вызывающего (родительского) процесса. Последние два поля, *dwProcessId* и *dwThreadId*, содержат идентификаторы созданного процесса и его главного потока соответственно.

Как уже было сказано, при создании процесса создаются объекты ядра «процесс» и «поток», для каждого из которого счетчик устанавливает в 1. Завершая свою работу, функция CreateProcess открывает эти объекты ядра и заносит их дескрипторы в поля *hProcess* и *hThread* структуры PROCESS_INFORMATION. При этом счетчики каждого из этих объектов ядра увеличиваются до 2.

Это означает, что родительский процесс должен вызвать функцию CloseHandle (и тем самым уменьшив счетчик объекта ядра) для дескриптора дочернего процесса и его главного потока, как только необходимость в них отпадет.

Кроме того созданному процессу присваивается уникальный идентификатор. Не может существовать двух процессов с одинаковыми идентификаторами. То же самое касается и потоков. При этом значения этих идентификаторов никогда не бывают нулевыми. Завершая свою работу, функция CreateProcess заносит значения этих идентификаторов в соответствующие поля *dwProcessId* и *dwThreadId* структуры PROCESS_INFORMATION.

Дочерние процессы

Создавая дочерние процессы, приложение может организовывать параллельное выполнение нескольких задач. При этом приложение

может подождать пока они завершаться или продолжить работу параллельно с ними.

В следующем примере показан программный код, в котором создается новый процесс и, в случае успеха, вызывается функция `WaitForSingleObject`, которая приостанавливает выполнение потока родительского процесса, до завершения порожденного им процесса.

Листинг 3.1. Пример создания процесса

```
1   TCHAR szCmdLine[] = TEXT("cmd.exe");
2
3   STARTUPINFO si = { sizeof(STARTUPINFO) };
4   PROCESS_INFORMATION pi;
5
6   // порождаем дочерний процесс
7   BOOL bRet = CreateProcess(NULL, szCmdLine, NULL, NULL, FALSE,
    CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
8
9   if (FALSE != bRet)
10  {
11      // закрывает дескриптор главного потока дочернего процесса
12      CloseHandle(pi.hThread);
13      // ожидаем завершения дочернего процесса
14      WaitForSingleObject(pi.hProcess, INFINITE);
15      // закрывает дескриптор дочернего процесса
16      CloseHandle(pi.hProcess);
17  } // if
```

Но все-таки чаще приложение создает *обособленные процессы* (detached processes) и продолжает работу параллельно с ними, не дожидаясь их завершения. Например, как в следующем примере:

Листинг 3.2. Пример создания обособленного процесса

```
1   TCHAR szCmdLine[] = TEXT("cmd.exe");
2
3   STARTUPINFO si = { sizeof(STARTUPINFO) };
4   PROCESS_INFORMATION pi;
5
6   // порождаем дочерний процесс
7   BOOL bRet = CreateProcess(NULL, szCmdLine, NULL, NULL, FALSE,
    CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
8
9   if (FALSE != bRet)
10  {
11      // закрывает дескриптор главного потока дочернего процесса
12      CloseHandle(pi.hThread);
```

```
13      // закрывает дескриптор дочернего процесса
14      CloseHandle(pi.hProcess);
15  } // if
```

Завершение процесса

По завершении процесса его код и выделенные ему ресурсы удаляются из памяти, а также уменьшаются на 1 счетчики связанных с этим процессом объектов ядра «процесс» и «поток». Однако эти объекты не удаляются, пока их счетчики не обнулятся. А это произойдет, когда все другие процессы, открывшие объекты ядра «процесс» и «поток», связанные с завершаемым процессом, закроют эти объекты.

Процесс завершает работу в следующих случаях:

- Главная функция (`WinMain`, `wWinMain`, `main` или `wmain`) приложения завершила свою работу и вернула управление. Это единственный случай, когда гарантировано корректное освобождение всех ресурсов, принадлежавших процессу.
- Один из потоков процесса вызвал функцию `ExitProcess`. Вызов этой функции является нежелательным, так как при этом вероятна утечка ресурсов.
- Поток другого процесса вызвал функцию `TerminateProcess`. Вызов этой функции также является нежелательным из-за возможной утечки ресурсов.
- Все потоки процесса завершили свою работу. Такая ситуация является редкой, но тем не менее возможной. Обнаружив, что в процессе не исполняется ни один поток, операционная система завершает этот процесс.

Следует отметить, что при завершении процесса прекращается выполнение всех его потоков. Но завершение процесса не приводит к завершению порожденных им дочерних процессов.

Функция `ExitProcess`

Функция `ExitProcess` завершает процесс, указав в качестве параметра *`uExitCode`* код завершения:

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

Возвращаемого значения у этой функции нет, так как в случае успеха произойдет немедленное завершение процесса.

Когда главная функция (`WinMain`, `wWinMain`, `main` или `wmain`) приложения возвращает управление, оно передается входной функции (`WinMainCRTStartup`, `wWinMainCRTStartup`, `mainCRTStartup` или `wmainCRT-`

Startup), которая выполнит удаление всех ресурсов, выделенных процессу, а затем обратится к функции ExitProcess, передав ей значение, которое вернула главная функция приложения. Вот почему возврат управления главной функции приложения приводит к завершению процесса.

Заметим, что вызов функции ExitProcess до того, как главная функция приложения вернет управление, приведет немедленному уничтожению процесса, но при этом весьма вероятна утечка ресурсов, выделенных при создании глобальных переменных и при инициализации различных библиотек. Таким образом, функцию ExitProcess не следует вызывать явно. Что касается объектов ядра, открытых в данном процессе, то здесь утечки не произойдет, так как в любом случае при завершении процесса они будут закрыты.

Функция TerminateProcess

Любой процесс может прекратить выполнение другого процесса при помощи функции TerminateProcess:

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

Первый параметр, *hProcess*, – дескриптор процесса, который нужно завершить.

Второй параметр, *uExitCode*, содержит код завершения процесса. Обычно этот параметр принимает нулевое значение.

Функция TerminateProcess завершится ошибкой, если дескриптор, который содержится в параметре *hProcess*, не обладает правом доступа PROCESS_TERMINATE. Если функция TerminateProcess завершается успешно, возвращаемое значение отлично от FALSE.

Использовать функцию TerminateProcess следует лишь в тех, когда иным способом завершить процесс не удается. При вызове функции TerminateProcess вероятна утечка ресурсов. Но все открытые процессом объекты ядра гарантированно будут закрыты.

Функция GetExitCodeProcess

Другой процесс может определить код завершения процесса, с помощью функции GetExitCodeProcess:

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);
```

Первый параметр, *hProcess*, – дескриптор процесса, для которого нужно узнать код завершения.

Второй параметр, *lpExitCode*, указывает на переменную типа DWORD, в которую будет записано значение кода завершения процесса.

Если процесс еще не завершился, параметр *lpExitCode* вернет значение STILL_ACTIVE.

Функция GetExitCodeProcess завершится ошибкой, если дескриптор, который содержится в параметре *hProcess*, не обладает правом доступа PROCESS_QUERY_INFORMATION. Если функция GetExitCodeProcess завершается успешно, возвращаемое значение отлично от FALSE.

Идентификаторы и псевдодескрипторы процессов

Процесс может получить дескриптор и идентификатор нового дочернего процесса из структуры PROCESS_INFORMATION. Для получения дескриптора и идентификатора текущего процесса в Win32 API имеются две функции – GetCurrentProcess и GetCurrentProcessId:

```
HANDLE GetCurrentProcess();

DWORD GetCurrentProcessId();
```

В действительности функция GetCurrentProcess возвращает *псевдодескриптор* (pseudo handle), представляющий собой константу типа HANDLE обычно –1, которая интерпретируется как дескриптор текущего процесса. Псевдодескриптор является не наследуемым и имеет максимально возможные права доступа.

Вызывающий процесс может использовать псевдодескриптор всякий раз, когда ему требуется его собственный дескриптор. Когда псевдодескриптор больше не нужен, закрывать его нет необходимости. Вызов функции CloseHandle с псевдодескриптором не уменьшает счетчик объекта ядра и, следовательно, не удаляет этот объект.

Реальный дескриптор текущего процесса можно получить, вызвав функцию DuplicateHandle, передав ей в качестве параметра псевдодескриптор, следующим образом:

Листинг 3.3. Получение дескриптора текущего процесса

```
1   HANDLE hProcess = NULL;
2
3   // получим псевдодескриптор текущего процесса
4   HANDLE hCurrentProcess = GetCurrentProcess();
5
6   // получим реальный дескриптор текущего процесса
7   BOOL bRet = DuplicateHandle(hCurrentProcess, hCurrentProcess,
    hCurrentProcess, &hProcess, 0, FALSE, DUPLICATE_SAME_ACCESS);
8
9   if (FALSE != bRet)
10  {
11      // закроем реальный дескриптор текущего процесса
```

```
12      CloseHandle(hProcess), hProcess = NULL;
13  } // if
```

Функция `GetCurrentProcessId` возвращает идентификатор текущего процесса. Для получения дескриптора процесса по его идентификатору применяется функция `OpenProcess`:

```
HANDLE OpenProcess(DWORD dwDesiredAccess,
    BOOL bInheritHandle, DWORD dwProcessId);
```

Первый параметр, *dwDesiredAccess*, определяет запрашиваемые права доступа к объекту ядра «процесс». Некоторые из возможных значений этого параметра перечислены в табл. 3.3. Полный перечень значений параметра *dwDesiredAccess* можно найти в документации Platform SDK.

Таблица 3.3. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
PROCESS_ALL_ACCESS	Все возможные права доступа
PROCESS_DUP_HANDLE	Разрешает дублирование дескрипторов процесса, используя функцию DuplicateHandle
PROCESS_QUERY_INFORMATION	Необходимо для получения информации о процессе
PROCESS_SET_INFORMATION	Необходимо для изменения информации о процессе
PROCESS_TERMINATE	Разрешает использование дескриптора процесса для завершения работы процесса, используя функцию TerminateProcess
SYNCHRONIZE	Разрешает использование дескриптора процесса в функциях ожидания для ожидания завершения процесса

Второй параметр, *bInheritHandle*, позволяет указать, должен ли полученный дескриптор быть наследуемым.

Третий параметр, *dwProcessId*, содержит идентификатор процесса, дескриптор которого нужно получить.

Если функция `OpenProcess` завершается успешно, она возвращает дескриптор запрашиваемого процесса, увеличивая на 1 счетчик соответствующего объекта ядра. В случае ошибки возвращается `NULL`.

Пренебрежение флагами, определяющими права доступа к объекту ядру (это касается всех объектов ядра, а не только процессов), может привести к тому, что приложение будет работать не так, как задумывалось. Начиная с Windows Vista, по соображениям безопасности

большинство приложений выполняются в ограниченном контексте безопасности, даже если текущий пользователь обладает правами администратора. Поэтому если необходимо получить права доступа, не подразумевающие внесения изменений в объект ядра, то не следует запрашивать полный доступ к этому объекту. Достаточно запросить только те права доступа, которые позволяют выполнить необходимые операции с объектом ядра.

В листинге 3.4 приводится пример функции, которая ожидает завершения процесса и при необходимости возвращает код завершения этого процесса. Заметим, что для этого не требуется полных прав доступа к процессу.

Листинг 3.4. Функция, ожидающая завершения процесса

```
1   BOOL WaitProcessById(DWORD dwProcessId, DWORD dwMilliseconds,
    LPDWORD lpExitCode)
2   {
3       // открываем процесс
4       HANDLE hProcess = OpenProcess(SYNCHRONIZE |
    PROCESS_QUERY_INFORMATION, FALSE, dwProcessId);
5
6       if (NULL == hProcess)
7       {
8           return FALSE; // не удалось открыть процесс
9       } // if
10
11      // ожидаем завершения процесса
12      WaitForSingleObject(hProcess, dwMilliseconds);
13
14      if (NULL != lpExitCode)
15      {
16          // получим код завершения процесса
17          GetExitCodeProcess(hProcess, lpExitCode);
18      } // if
19
20      // закрываем дескриптор процесса
21      CloseHandle(hProcess);
22
23      return TRUE;
24  } // WaitProcessById
```

В Win32 API также имеется функция `GetWindowThreadProcessId`, которая предоставляет возможность получить идентификатор процесса, используя дескриптор окна, созданного в этом процессе.

```
DWORD GetWindowThreadProcessId(HWND hWnd, LPDWORD lpProcessId);
```

Первый параметр, *hWnd*, представляет собой дескриптор окна. Второй параметр, *lpProcessId*, указывает на переменную типа DWORD, в которую будет записан полученный идентификатор процесса. Этот параметр может быть установлен в NULL.

Если функция GetWindowThreadProcessId завершает успешно, она возвращает идентификатор потока, в котором было создано указанное окно. В случае ошибки возвращается ноль.

Следующий пример демонстрирует программный код, в котором для ожидания процесса используется дескриптор окна, которое было в нем создано.

Листинг 3.5. Пример ожидания завершения процесса

```
1   // поиск окна с заголовком "Калькулятор"
2   HWND hWnd = FindWindow(NULL, TEXT("Калькулятор"));
3
4   if (NULL != hWnd) // если окно успешно найдено
5   {
6       DWORD dwProcessId = 0;
7
8       // определим идентификатор процесса,
9       // в котором было создано найденное окно
10      GetWindowThreadProcessId(hWnd, &dwProcessId);
11
12      // ожидаем завершения работы процесса
13      WaitProcessById(dwProcessId, INFINITE, NULL);
14  } // if
```

Приоритет процесса

Чем выше приоритет процесса, тем больше его потокам выделяется процессорного времени, чем ниже приоритет, тем меньше выделяется процессорного времени. Каждый процесс принадлежит одному из классов приоритета, перечисленных в табл. 3.4.

Таблица 3.4. Классы приоритета процесса

Класс приоритета	Описание
REALTIME_PRIORITY_CLASS (Реального времени)	Потоки процесса этого класса приоритета вытесняют потоки всех других процессов, включая процессы операционной системы
HIGH_PRIORITY_CLASS (Высокий)	Процессы этого класса приоритета, как правило, выполняют критические по времени задачи, которые должны быть выполнены немедленно. Потоки такого процесса вытесняют потоки процесса, запущенного с более низким приоритетом

Окончание табл. 3.4.

Класс приоритета	Описание
ABOVE_NORMAL_PRIORITY _CLASS (Выше среднего)	Потоки процесса этого класса приоритета вытесняют потоки процесса, запущенного с приоритетом NORMAL_PRIORITY_CLASS или ниже, но вытесняются потоками процесса, запущенного с приоритетом HIGH_PRIORITY_CLASS или выше
NORMAL_PRIORITY_CLASS (Средний)	Используется обычный приоритет. Потоки процесса с этим классом приоритета вытесняют потоки процесса, запущенного с более низким приоритетом, но вытесняются потоками процесса, запущенного с более высоким приоритетом
BELOW_NORMAL_PRIORITY _CLASS (Ниже среднего)	Потоки процесса этого класса приоритета вытесняют потоки процесса, запущенного с приоритетом IDLE_PRIORITY_CLASS, но вытесняются потоками процесса, запущенного с приоритетом NORMAL_PRIORITY_CLASS или выше
IDLE_PRIORITY_CLASS (Низкий)	Потоки процесса этого класса приоритета выполняются только тогда, когда операционная система неактивна и вытесняются потоками процесса, запущенного с более высоким приоритетом

По умолчанию, классом приоритета процесса является NORMAL_ PRIORITY_CLASS. Если процесс имеет приоритет IDLE_PRIORITY_CLASS или BELOW_NORMAL_PRIORITY_CLASS, порождаемый им процесс наследует этот класс приоритета. При создании дочернего процесса с помощью функции CreateProcess также можно задать класс приоритета для нового процесса.

Чтобы определить текущий класс приоритета процесса используется функция GetPriorityClass, чтобы изменить – SetPriorityClass:

```
DWORD GetPriorityClass(HANDLE hProcess);

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

Параметр *hProcess* – дескриптор процесса, для которого нужно определить или изменить текущий класс приоритета.

Параметр *dwPriorityClass* задает класс приоритета процесса (см. табл. 3.4).

Функция GetPriorityClass завершится с ошибкой, если дескриптор, который содержится в параметре *hProcess*, не обладает правом доступа PROCESS_QUERY_INFORMATION. Если функция GetPriorityClass завершается успешно, она возвращает класс приоритета процесса; в случае ошибки возвращается ноль.

Функция SetPriorityClass завершится с ошибкой, если дескриптор, который содержится в параметре *hProcess*, не обладает правом доступа PROCESS_SET_INFORMATION. Если функция SetPriorityClass завершается успешно, возвращаемое значение отлично от FALSE.

Начиная с Windows Vista, приложение может устанавливать класс приоритета процесса REALTIME_PRIORITY_CLASS, только если оно запущено от имени «Администратора».

Перечисление процессов

Изначально в Win32 API не было функций, которые позволяли бы перечислять выполняемые процессы. Однако с появлением Windows NT для решения этой задачи в Win32 API был добавлен набор функций под общим названием Process Status. Прототипы этих функций описаны в заголовочном файле Psapi.h, а сами они реализованы в библиотеке Psapi.dll. Более подробную информацию обо всех функциях Process Status можно найти в документации Platform SDK.

Чтобы получить список идентификаторов всех созданных процессов, следует использовать функцию EnumProcesses:

```
BOOL EnumProcesses(DWORD *pProcessIds, DWORD cb,
    DWORD *pBytesReturned);
```

Первый параметр, *pProcessIds*, указывает на буфер, в который будет сохранен список идентификаторов работающих процессов. Второй параметр, *cb*, содержит размер (в байтах) буфера, на который указывает параметр *pProcessIds*. Третий параметр, *pBytesReturned*, указывает на переменную типа DWORD, в которую будет записан размер (в байтах) полученного списка идентификаторов.

Если функция EnumProcesses завершается успешно, возвращаемое значение отлично от FALSE.

Имея дескриптор процесса можно получить список модулей, загруженных этим процессом. Для этого следует использовать функцию EnumProcessModules или EnumProcessModulesEx:

```
BOOL EnumProcessModules(HANDLE hProcess, HMODULE *lphModule,
    DWORD cb, LPDWORD lpcbNeeded);

BOOL EnumProcessModulesEx(HANDLE hProcess, HMODULE *lphModule,
    DWORD cb, LPDWORD lpcbNeeded, DWORD dwFilterFlag);
```

Первый параметр, *hProcess*, представляет собой дескриптор процесс. Этот дескриптор должен обладать правами доступа PROCESS_QUERY_INFORMATION и PROCESS_VM_READ

Второй параметр, *lphModule*, указывает на буфер, в который будет сохранен список дескрипторов загруженных модулей. Третий пара-

метр, *cb*, содержит размер (в байтах) буфера, на который указывает параметр *lphModule*. Четвертый параметр, *lpcbNeeded*, указывает на переменную типа DWORD, в которую будет записан размер (в байтах) полученного списка дескрипторов.

Последний параметр, *dwFilterFlag*, функции EnumProcessModulesEx определяет критерии фильтрации списка загруженных модулей. Этот параметр может принимать одно из следующих значений:

- LIST_MODULES_32BIT – полученный список модулей, будет содержать только 32-разрядные модули;
- LIST_MODULES_64BIT – полученный список модулей, будет содержать только 64-разрядные модули;
- LIST_MODULES_ALL – полученный список модулей, будет содержать все загруженные модули;
- LIST_MODULES_DEFAULT – список формируется по умолчанию.

Обычно интересует только дескриптор главного модуля процесса, поэтому при вызове функции EnumProcessModules(Ex) можно указать массив, состоящий только из одного элемента.

Если функция EnumProcessModules(Ex) завершается успешно, возвращаемое значение отлично от FALSE. Если эта функция вызывается из 32-разрядных приложений, работающих на 64-разрядной версии Windows, она может только перечислить модули 32-разрядного процесса. Если процесс представляет собой 64-разрядный процесс, функция EnumProcessModules(Ex) завершается ошибкой ERROR_PARTIAL_COPY.

Для того чтобы определить имя модуля, следует использовать функцию GetModuleBaseName или GetModuleFileNameEx:

```
DWORD GetModuleBaseName(HANDLE hProcess, HMODULE hModule,
    LPTSTR lpBaseName, DWORD nSize);

DWORD GetModuleFileNameEx(HANDLE hProcess, HMODULE hModule,
    LPTSTR lpFilename, DWORD nSize);
```

Первый параметр, *hProcess*, представляет собой дескриптор процесса. Этот дескриптор должен обладать правами доступа PROCESS_QUERY_INFORMATION и PROCESS_VM_READ.

Второй параметр, *hModule*, представляет собой дескриптор модуля. Если этот параметр установлен в NULL, эти функции получают дескриптор главного модуля указанного процесса и возвращают его имя.

Третий параметр, *lpBaseName* (*lpFilename*), указывает на строку, в которую будет записано имя модуля. Последний параметр, *nSize*, определяет максимальный размер (в символах) строки. Если имя модуля длиннее, чем максимальный размер строки, имя усекается.

В случае успеха и функция GetModuleBaseName, и функция GetModuleFileNameEx возвращают длину результирующей строки; в случае ошибки возвращается ноль.

Отличием функции GetModuleBaseName от GetModuleFileNameEx является то, что вторая возвращает полное имя модуля, которое включает полный путь к этому модулю. Следует также отметить, что функция GetModuleFileNameEx является расширенной версией функции GetModuleFileName, которая возвращает полное имя указанного модуля для текущего процесса.

В листингах 3.6,3.7 представлен программный код двух функций: первая функция выводит список всех работающих процессов, а вторая – список всех загруженных модулей для указанного процесса:

Листинг 3.6. Функция, перечисляющая процессы системы

```
1   void LoadProcessesToListBox(HWND hwndCtl)
2   {
3       // удалим все строки из списка
4       ListBox_ResetContent(hwndCtl);
5
6       // получим список идентификаторов процессов
7       DWORD aProcessIds[1024], cbNeeded = 0;
8       BOOL bRet = EnumProcesses(aProcessIds, sizeof(aProcessIds),
    &cbNeeded);
9
10      if (FALSE != bRet)
11      {
12          TCHAR szName[MAX_PATH], szBuffer[300];
13
14          for (DWORD i = 0,
15              n = cbNeeded / sizeof(DWORD); i < n; ++i)
16          {
17              DWORD dwProcessId = aProcessIds[i], cch = 0;
18              if (0 == dwProcessId) continue;
19
20              // открываем объект ядра "процесс"
21              HANDLE hProcess = OpenProcess(PROCESS_VM_READ |
    PROCESS_QUERY_INFORMATION, FALSE, dwProcessId);
22
23              if (NULL != hProcess)
24              {
25                  // определяем имя главного модуля процесса
26                  cch = GetModuleBaseName(hProcess, NULL, szName,
    MAX_PATH);
27                  CloseHandle(hProcess); // закрываем объект ядра
28              } // if
```

```
29
30              if (0 == cch)
31                  StringCchCopy(szName, MAX_PATH, TEXT("Неизвестный
    процесс"));
32
33              // формируем строку для списка
34              StringCchPrintf(szBuffer, _countof(szBuffer),
    TEXT("%s (PID: %u)"), szName, dwProcessId);
35
36              // добавляем в список новую строку
37              int iItem = ListBox_AddString(hwndCtl, szBuffer);
38
39              // сохраняем в новой строке идентификатор процесса
40              ListBox_SetItemData(hwndCtl, iItem, dwProcessId);
41          } // for
42      } // if
43  } // LoadProcessesToListBox
```

Листинг 3.7. Функция, перечисляющая модули процесса

```
1   void LoadModulesToListBox(HWND hwndCtl, DWORD dwProcessId)
2   {
3       // удалим все строки из списка
4       ListBox_ResetContent(hwndCtl);
5
6       // открываем объект ядра "процесс"
7       HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
    PROCESS_VM_READ, FALSE, dwProcessId);
8
9       if (NULL != hProcess)
10      {
11          // определяем размер (в байтах) списка модулей
12          DWORD cb = 0;
13          EnumProcessModulesEx(hProcess, NULL, 0, &cb,
    LIST_MODULES_ALL);
14
15          // вычисляем количество модулей
16          DWORD nCount = cb / sizeof(HMODULE);
17
18          // выделяем память для списка модулей
19          HMODULE *hModule = new HMODULE[nCount];
20
21          // получаем список модулей
22          cb = nCount * sizeof(HMODULE);
23          BOOL bRet = EnumProcessModulesEx(hProcess, hModule, cb,
    &cb, LIST_MODULES_ALL);
24
```

```
25          if (FALSE != bRet)
26          {
27              TCHAR szFileName[MAX_PATH];
28
29              for (DWORD i = 0; i < nCount; ++i)
30              {
31                  // получаем имя загруженного модуля
32                  bRet = GetModuleFileNameEx(hProcess, hModule[i],
    szFileName, MAX_PATH);
33                  if (FALSE != bRet) ListBox_AddString(hwndCtl,
    szFileName); // добавляем в список новую строку
34              } // for
35          } // if
36
37          delete[]hModule; // освобождаем память
38
39          CloseHandle(hProcess); // закрываем объект ядра
40      } // if
41  } // LoadModulesToListBox
```

Задания

Задание (job) – это объект ядра, который позволяет группировать процессы и накладывать на них различные ограничения. Один и тот же процесс может быть включен только в одно задание. Все процессы, порождаемые включенным в задание процессом, также будут включены в это задание. Кроме того, если процесс уже включен в задание, то его и порожденные им процессы уже нельзя исключить из этого задания. Такой защитный механизм не позволяет включенным в задание процессам обойти наложенные на них ограничения.

Создание задания

Новое задание создается с помощью функции `CreateJobObject`:

```
HANDLE CreateJobObject(LPSECURITY_ATTRIBUTES lpJobAttributes,
    LPCTSTR lpName);
```

Первый параметр, *`lpJobAttributes`*, указывает на структуру `SECURITY_ATTRIBUTES`, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «задание». Этот параметр можно установить в `NULL`, тогда объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Второй параметр, *`lpName`*, указывает на строку, которая определяет имя создаваемого объекта ядра. Если этот параметр установлен в `NULL`, будет создан безымянный объект ядра.

Если функция CreateJobObject выполняется успешно, возвращается дескриптор созданного объекта ядра «задание», иначе – NULL.

Разумеется, можно открыть уже созданное задание (если оно не безымянное) с помощью функции OpenJobObject:

```
HANDLE OpenJobObject(DWORD dwDesiredAccess,
    BOOL bInheritHandles, LPCTSTR lpName);
```

Первый параметр, *dwDesiredAccess*, определяет запрашиваемые права доступа к объекту ядра «задание». Некоторые из возможных значений этого параметра перечислены в табл. 3.5. Полный перечень значений параметра *dwDesiredAccess* можно найти в документации Platform SDK.

Таблица 3.5. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
JOB_OBJECT_ALL_ACCESS	Все возможные права доступа
JOB_OBJECT_ASSIGN_PROCESS	Разрешает использование дескриптора для включения процессов в задание
JOB_OBJECT_QUERY	Необходимо для получения информации о задании
JOB_OBJECT_SET_ATTRIBUTES	Необходимо для наложения ограничений на процессы в задании
JOB_OBJECT_SET_SECURITY_ATTRIBUTES	Необходимо для наложения ограничений, связанных с безопасностью, на процессы в задании
JOB_OBJECT_TERMINATE	Разрешает использование дескриптора для завершения всех процессов в задании

Второй параметр, *bInheritHandle*, позволяет указать, должен ли полученный дескриптор быть наследуемым. Третий параметр, *lpName*, определяет имя задания, дескриптор которого нужно получить.

Если функция OpenJobObject завершается успешно, она возвращает дескриптор запрашиваемого задания, увеличивая на 1 счетчик соответствующего объекта ядра. В случае ошибки возвращается NULL.

Ограничения, налагаемые на процессы в задании

На процессы в задании можно наложить ограничения следующих видов:

– базовые и расширенные ограничения, которые не позволяют процессам в задании монопольно захватывать системные ресурсы;

- базовые ограничения пользовательского интерфейса, которые не позволяют его изменять;
- ограничения, связанные с безопасностью и которые не позволяют процессам в задании обращаться к защищаемым объектам.

Ограничения в задании устанавливают с помощью вызова функции SetInformationJobObject:

```
BOOL SetInformationJobObject(HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInfoClass,
    LPVOID lpJobObjectInfo, DWORD cbJobObjectInfoLength);
```

Первый параметр, *hJob*, определяет дескриптор задания. Второй параметр, *JobObjectInfoClass*, – вид ограничений. Этот параметр может принимать значения, перечисленные в табл. 3.6. Третий параметр, *lpJobObjectInfo*, указывает на структуру данных, которая содержит информацию о задаваемых ограничениях. Четвертый параметр, *cbJobObjectInfoLength*, определяет размер структуры.

Таблица 3.6. Виды ограничений

Значение параметра *JobObjectInfoClass*	**Структура, указываемая в параметре** *lpJobObjectInfo*
JobObjectBasicLimitInformation (базовые ограничения)	JOBOBJECT_BASIC_LIMIT_INFORMATION
JobObjectExtendedLimitInformation (расширенные ограничения)	JOBOBJECT_EXTENDED_LIMIT_INFORMATION
JobObjectBasicUIRestrictions (базовые ограничения пользовательского интерфейса)	JOBOBJECT_BASIC_UI_RESTRICTIONS
JobObjectSecurityLimitInformation (ограничения, связанные с безопасностью)	JOBOBJECT_SECURITY_LIMIT_INFORMATION

Описание структур, приведенных в табл. 3.6, см. в документации Platform SDK.

Функция SetInformationJobObject завершится ошибкой, если дескриптор, который содержится в параметре *hJob*, не обладает правом доступа JOB_OBJECT_SET_ATTRIBUTES. Если необходимо задать ограничения, связанные с безопасностью, дескриптор задания должен также обладать правом доступа JOB_OBJECT_SET_SECURITY_ATTRIBUTES. Если функция SetInformationJobObject завершается успешно, возвращаемое значение отлично от FALSE.

Чтобы получить информацию о наложенных ограничениях, следует воспользоваться функцией QueryInformationJobObject:

```
BOOL QueryInformationJobObject(HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInfoClass,
    LPVOID lpJobObjectInfo, DWORD cbJobObjectInfoLength,
    LPDWORD lpReturnLength);
```

Первые четыре параметра этой функции аналогичны параметрам функции SetInformationJobObject.

Последний параметр, *lpReturnLength*, указывает на переменную типа DWORD, которая получает число байт, записанных в структуру данных, указанную параметром *lpJobObjectInfo*. Этот параметр может быть установлен в NULL.

Функция QueryInformationJobObject завершится ошибкой, если дескриптор, который содержится в параметре *hJob*, не обладает правом доступа JOB_OBJECT_QUERY. Если функция QueryInformationJobObject завершается успешно, возвращаемое значение отлично от FALSE.

Включение процесса в задание

Для включения процесса в задание в Win32 API имеется функция AssignProcessToJobObject:

```
BOOL AssignProcessToJobObject(HANDLE hJob, HANDLE hProcess);
```

Первый параметр, *hJob*, – дескриптор задания, который должен обладать правом доступа JOB_OBJECT_ASSIGN_PROCESS.

Второй параметр, *hProcess*, – дескриптор процесса, который должен обладать правами доступа PROCESS_SET_QUOTA и PROCESS_TERMINATE.

Если функция AssignProcessToJobObject завершается успешно, возвращаемое значение отлично от FALSE. При этом увеличивается на 1 значение счетчика соответствующего объекта ядра «задание».

Функцию AssignProcessToJobObject следует вызывать сразу после создания дочернего процесса и перед началом его работы. Поэтому, порождая процесс, необходимо вызывать функцию CreateProcess с флагом CREATE_SUSPENDED, который приостанавливает работу главного потока процесса сразу после его создания. Возобновить работу главного потока процесса можно с помощью функции ResumeThread (эта функция рассматривается при изучении потоков).

Важно отметить, что функция AssignProcessToJobObject позволяет включить в задание только процесс, который не включен ни в одно задание. Как только процесс был включен в какое-нибудь задание, его уже нельзя исключить из него или включить в другое задание.

Узнать принадлежит ли процесс к заданию, можно вызвав функцию IsProcessInJob:

```
BOOL IsProcessInJob(HANDLE ProcessHandle, HANDLE JobHandle,
    PBOOL Result);
```

Первый параметр, *ProcessHandle*, – дескриптор процесса, который должен обладать правом доступа PROCESS_QUERY_INFORMATION.

Второй параметр, *JobHandle*, – дескриптор задания, который должен обладать правом доступа JOB_OBJECT_QUERY. Если параметр *JobHandle* установлен в NULL, проверяется принадлежность процесса к любому заданию.

Последний параметр, *Result*, указывает на переменную типа BOOL, которая принимает значение TRUE, если процесс принадлежит заданию, или FALSE в противном случае.

Если функция IsProcessInJob завершается успешно, возвращаемое значение отлично от FALSE.

Кроме того, следует обратить внимание на то, что если процесс был включен в задание, все порождаемые им процессы автоматически включаются в тоже задание. Однако это правило можно изменить одним из следующих способов:

- Наложить ограничение на процессы в задании, добавив в поле *LimitFlags* структуры JOBOBJECT_BASIC_LIMIT_INFORMATION флаг JOB_OBJECT_LIMIT_BREAKAWAY_OK, тем самым указать, что порождаемый процесс может выполняться вне задания. Для этого, при порождении процесса, функцию CreateProcess следует вызывать с флагом CREATE_BREAKAWAY_FROM_JOB. Такой механизм пригодится на случай, если дочерний процесс тоже управляет заданиями.
- Наложить ограничение на процессы в задании, добавив в поле *LimitFlags* структуры JOBOBJECT_BASIC_LIMIT_INFORMATION флаг JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK, тем самым указать, что порождаемый процесс должен выполняться вне задания.

В листинге 3.8 приводится пример функции, в которой создаются задание и группа процессов, включенных в это задание.

Листинг 3.8. Функция создания группы процессов в одном задании

```
1   BOOL StartGroupProcessesAsJob(HANDLE hJob, LPCTSTR lpszCmdLine[],
    DWORD nCount, BOOL bInheritHandles, DWORD dwCreationFlags)
2   {
3       // определим, включен ли вызывающий процесс в задание
4       BOOL bInJob = FALSE;
5       IsProcessInJob(GetCurrentProcess(), NULL, &bInJob);
6
7       if (FALSE != bInJob) // если да (!)
8       {
9           // определим разрешено ли порождать процессы,
10          // которые не будут принадлежать этому заданию ...
```

```
11          JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
12
13          QueryInformationJobObject(NULL,
    JobObjectBasicLimitInformation, &jobli, sizeof(jobli), NULL);
14
15          DWORD dwLimitMask = JOB_OBJECT_LIMIT_BREAKAWAY_OK |
    JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;
16
17          if ((jobli.LimitFlags & dwLimitMask) == 0)
18          {
19              /* все порожденные процессы
20                 автоматически включаются в задание */
21              return FALSE;
22          } // if
23      } // if
24
25      // порождаем процессы...
26
27      TCHAR szCmdLine[MAX_PATH];
28
29      STARTUPINFO si = { sizeof(STARTUPINFO) };
30      PROCESS_INFORMATION pi;
31
32      for (DWORD i = 0; i < nCount; ++i)
33      {
34          StringCchCopy(szCmdLine, MAX_PATH, lpszCmdLine[i]);
35
36          // порождаем новый процесс,
37          // приостанавливая работу его главного потока
38          BOOL bRet = CreateProcess(NULL, szCmdLine, NULL, NULL,
    bInheritHandles, dwCreationFlags | CREATE_SUSPENDED |
    CREATE_BREAKAWAY_FROM_JOB, NULL, NULL, &si, &pi);
39
40          if (FALSE != bRet)
41          {
42              // добавляем новый процесс в задание
43              AssignProcessToJobObject(hJob, pi.hProcess);
44              // возобновляем работу потока нового процесса
45              ResumeThread(pi.hThread);
46
47              // закрывает дескриптор потока нового процесса
48              CloseHandle(pi.hThread);
49              // закрывает дескриптор нового процесса
50              CloseHandle(pi.hProcess);
51          } // if
52      } // for
```

```
53
54      return TRUE;
55  } // StartGroupProcessesAsJob
```

Эту функцию можно вызывать следующим образом:

```
1   HANDLE hJob = CreateJobObject(NULL, NULL); // создаем задание
2
3   if (NULL != hJob)
4   {
5       // установим ограничения для процессов в задании ...
6       JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
7
8       // процессы в задании должны выполняется с низким приоритетом
9       jobli.PriorityClass = IDLE_PRIORITY_CLASS;
10      // задаем установленные ограничения
11      jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS |
    JOB_OBJECT_LIMIT_BREAKAWAY_OK;
12
13      BOOL bRet = SetInformationJobObject(hJob,
    JobObjectBasicLimitInformation, &jobli, sizeof(jobli));
14
15      if (FALSE != bRet)
16      {
17          // порождаем процессы...
18
19          LPCTSTR szCmdLine[] = {
20              TEXT("cmd.exe"), TEXT("notepad"), TEXT("calc.exe")
21          };
22
23          bRet = StartGroupProcessesAsJob(hJob, szCmdLine,
    _countof(szCmdLine), FALSE, 0);
24      } // if
25
26      CloseHandle(hJob); // закрываем дескриптор задания
27  } // if
```

Завершение всех процессов в задании

Чтобы завершить работу всех процессов в задании, можно воспользоваться функцией `TerminateJobObject`:

```
BOOL TerminateJobObject(HANDLE hJob, UINT uExitCode);
```

Первый параметр, *`hJob`*, – дескриптор задания, процессы которого нужно завершить. Второй параметр, *`uExitCode`*, содержит код завершения процесса.

Функция TerminateJobObject завершится с ошибкой, если дескриптор, который содержится в параметре *hJob*, не обладает правом доступа JOB_OBJECT_TERMINATE. Если же функция завершается успешно, возвращаемое значение отлично от FALSE.

Вызов функции TerminateJobObject аналогичен вызовам функции TerminateProcess для каждого процесса. Поэтому каждый дескриптор процесса, добавленного в задание, должен обладать правом доступа PROCESS_TERMINATE.

Перечисление процессов в задании

Функцию QueryInformationJobObject можно использовать не только для получения информации о текущих ограничениях, установленных в задании. Этой функцией можно пользоваться и для получения другой информации. Например, чтобы получить список идентификаторов включенных в задание процессов.

Для этого в качестве второго параметра используется значение JobObjectBasicProcessIdList, а в качестве третьего – указатель на структуру JOBOBJECT_BASIC_PROCESS_ID_LIST, которая определена следующим образом:

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
  DWORD NumberOfAssignedProcesses; // максимальное число
                                   // идентификаторов
  DWORD NumberOfProcessIdsInList; // число идентификаторов
  ULONG_PTR ProcessIdList[1]; // список идентификаторов
} JOBOBJECT_BASIC_PROCESS_ID_LIST;
```

Первое поле, *NumberOfAssignedProcesses*, определяет максимальное число идентификаторов процесса, которые могут быть записаны в поле *ProcessIdList*.

Во второе поле, *NumberOfProcessIdsInList*, возвращается число идентификаторов процесса, которые были записаны в поле *ProcessIdList*. В третье поле, *ProcessIdList*, записывается список полученных идентификаторов процесса.

Чтобы получить список идентификаторов, нужно выделить блок памяти, в котором поместится список идентификаторов необходимого размера и структура JOBOBJECT_BASIC_PROCESS_ID_LIST. Затем вызвать функцию QueryInformationJobObject, передав ей в качестве аргумента адрес выделенного блока в памяти. В примере из листинга 3.9 показано, как это можно сделать. Функцию EnumProcessesInJob из этого примера, можно использовать практически так же, как и функцию EnumProcesses (см. листинг 3.6).

Листинг 3.9. Функция, которая возвращает список идентификаторов включенных в задание процессов

```
1   BOOL EnumProcessesInJob(HANDLE hJob, ULONG_PTR *lpidProcess,
    DWORD cb, LPDWORD lpcbNeeded)
2   {
3       // определяем максимальное количество идентификаторов,
4       // на которое расчитан буфер lpidProcess
5       DWORD nCount = cb / sizeof(ULONG_PTR);
6
7       if (NULL != lpidProcess && nCount > 0)
8       {
9           // определеям размер блока памяти (в байтах)
10          // для хранения идентификаторов и структуры
11          DWORD cbJobPIL = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST)
    + (nCount - 1) * sizeof(ULONG_PTR);
12
13          // выделяем блок памяти
14          JOBOBJECT_BASIC_PROCESS_ID_LIST *pJobPIL =
    (JOBOBJECT_BASIC_PROCESS_ID_LIST *)malloc(cbJobPIL);
15
16          if (NULL != pJobPIL)
17          {
18              // указываем максимальное количество идентификаторов,
19              // на которое расчитана выделенная память
20              pJobPIL->NumberOfAssignedProcesses = nCount;
21
22              // запрашиваем список идентификаторов процессов
23              BOOL bRet = QueryInformationJobObject(hJob,
    JobObjectBasicProcessIdList, pJobPIL, cbJobPIL, NULL);
24
25              if (FALSE != bRet)
26              {
27                  // определяем количество идетификаторов
28                  nCount = pJobPIL->NumberOfProcessIdsInList;
29                  // копируем в буфер список идентификаторов
30                  CopyMemory(lpidProcess, pJobPIL->ProcessIdList,
    nCount*sizeof(ULONG_PTR));
31
32                  // возвращаем размер блока памяти (в байтах),
33                  // в который скопирован список идентификаторов
34                  if (NULL != lpcbNeeded)
35                      *lpcbNeeded = nCount * sizeof(ULONG_PTR);
36              } // if
37
38              free(pJobPIL); // освобождаем память
39              return bRet;
```

```
40          } // if
41      } // if
42
43      return FALSE;
44  } // EnumProcessesInJob
```

Потоки

Как правило, приложения обходятся единственным, главным потоком. Однако можно создавать дополнительные потоки, что позволяет организовать параллельное выполнение нескольких задач в рамках одного процесса.

Создание потока

При создании потока, создается объект ядра «поток» с начальным значением счетчика равным 1. Операционная система использует этот объект ядра, чтобы управлять потоком. Для нового потока в виртуальном адресном пространстве процесса отводится место под его стек.

Главный поток процесса создается при создании процесса. Дополнительный поток создается при вызове функции `CreateThread`:

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

Первый параметр, *`lpThreadAttributes`*, указывает на структуру `SECURITY_ATTRIBUTES`, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «поток». Этот параметр можно установить в `NULL`, тогда объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Второй параметр, *`dwStackSize`*, определяет, какую часть адресного пространства процесса новый поток сможет использовать под свой стек. Значению `0` этого параметра соответствует размер стека по умолчанию, равный размеру стека главного потока (обычно это 1 Мбайт).

Третий параметр, *`lpStartAddress`*, определяет адрес функции потока, в которой создаваемый поток выполняет свою работу. В главном потоке такой функцией является `WinMainCRTStartup`, `wWinMainCRTStartup`, `mainCRTStartup` или `wmainCRTStartup`. Функция потока может иметь любое имя, но всегда соответствует следующей сигнатуре:

```
DWORD CALLBACK ThreadFunc(LPVOID lpParameter);
```

Параметр *`lpParameter`* функции `CreateThread` идентичен параметру *`lpParameter`* функции потока и позволяет предавать функции создава-

емого потока какое-либо значение. Оно может быть или просто числовым значением, или указателем на переменную, содержащую дополнительную информацию.

Пятый параметр, *dwCreationFlags*, определяет дополнительные флаги, управляющие созданием потока. Этот параметр может принимать одно из двух значение:

- `0` – исполнение потока начинает сразу после создания;
- `CREATE_SUSPENDED` – созданный поток приостанавливается до последующих указаний.

Последний параметр, *lpThreadId*, указывает на переменную типа `DWORD`, в которую функция `CreateThread` возвращает идентификатор нового потока. Этот параметр может быть установлен в `NULL`.

Если функция `CreateThread` выполняется успешно, возвращается дескриптор созданного объекта ядра «поток» с полными правами доступа, иначе – `NULL`.

Завершение потока

По завершении потока уничтожаются все выделенные ему ресурсы операционной системы, память под его стек освобождается и счетчик сопоставленного с ним объекта ядра «поток» уменьшается на 1. Однако этот объект не будет удален, пока его счетчик не обнулится.

Кроме того, при завершении потока операционная система уведомляет об этом все DLL, подключенные к процессу, которому принадлежит завершаемый поток.

Поток завершается в следующих случаях:

- Функция потока завершила работу и вернула управление. Это единственный случай, когда гарантировано корректное освобождение всех ресурсов, принадлежавших потоку.
- Поток вызвал функцию `ExitThread`. Вызов этой функции является нежелательным, так как при этом вероятна утечка ресурсов.
- Один из потоков процесса вызвал функцию `TerminateThread`. Вызов этой функции является нежелательным, так как при этом вероятна утечка ресурсов.
- Был завершен процесс, содержащий данный поток. Этот случай тоже является нежелательным.

Если завершаемый поток является последним активным потоком в процессе, завершается и сам процесс.

Функция ExitThread

Функция ExitThread завершает поток, указав в качестве параметра *uExitCode* код завершения:

```
VOID ExitThread(UINT uExitCode);
```

Возвращаемого значения у функции ExitThread нет, так как вызов этой функции приведет к немедленному завершению потока. Однако при этом вероятна утечка ресурсов, используемых потоком, поэтому функцию ExitThread не следует использовать без особых причин.

Функция TerminateThread

Прекратить выполнение потока можно также при помощи функции TerminateThread:

```
BOOL TerminateThread(HANDLE hThread, UINT uExitCode);
```

Первый параметр, *hThread*, – дескриптор потока, второй параметр, *uExitCode*, – код завершения потока.

Функция TerminateThread завершится с ошибкой, если дескриптор, который содержится в параметре *hThread*, не обладает правом доступа THREAD_TERMINATE. Если функция TerminateThread завершается успешно, возвращаемое значение отлично от FALSE.

Если поток завершается вызовом функции TerminateThread, его стек не освобождается, пока не завершится процесс, а загруженные библиотеки DLL не уведомляются о завершении потока, что может привести к некорректному завершению процесса. Поэтому следует избегать вызовов этой функции в своих приложениях.

Функция GetExitCodeThread

Код завершения потока можно определить с помощью вызова функции GetExitCodeThread:

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

Первый параметр, *hThread*, – дескриптор потока, для которого нужно узнать код завершения. Это дескриптор должен обладать правом доступа THREAD_QUERY_INFORMATION.

Второй параметр, *lpExitCode*, указывает на переменную типа DWORD, в которую будет записано значение кода завершения потока. Если поток не завершен, параметр *lpExitCode* вернет значение STILL_ACTIVE.

Если функция GetExitCodeThread завершается успешно, возвращаемое значение отлично от FALSE.

Функция `RtlUserThreadStart`

Новый поток всегда начинает свою работу с функции `RtlUserThreadStart` из библиотеки ntdll.dll. Эта функция вызывает функцию потока, предавая ей параметр *`lpParameter`*, который был ранее передан функции `CreateThread`.

Листинг 3.10. Функция `RtlUserThreadStart`

```
1   void RtlUserThreadStart(LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter)
2   {
3       __try
4       {
5           // вызов функции потока,
6           // когда она вернет управление поток завершится
7           ExitThread((lpStartAddress)(lpParameter));
8       } // __try
9       __except(UnhandledExceptionFilter(GetExceptionInformation()))
10      {
11          // завершение процесса
12          ExitProcess(GetExceptionCode());
13      } // __except
14  } // RtlUserThreadStart
```

Как видно из этого примера, если функция потока возвращает управление, вызывается функция `ExitThread`, в которую передается значение, возвращенное функцией потока. Если поток вызывает необрабатываемое им исключение, вызывается функция `ExitProcess` и завершается весь процесс. Таким образом, поток всегда завершается внутри функции `RtlUserThreadStart`.

Следует отметить, что главный поток так же начинает свою работу с функции `RtlUserThreadStart`. В этом случае функция `RtlUserThreadStart` обращается к `WinMainCRTStartup`, `wWinMainCRTStartup`, `mainCRTStartup` или `wmainCRTStartup`.

Функции Visual C++ для создания и завершения потока

Функции Win32 API `CreateThread` и `ExitThread`, которые используются для создания и завершения потока, разрабатывались еще для Windows 3.1 и не учитывают возникновение некоторых проблем, связанных с использованием стандартной библиотеки C/C++, которая не рассчитана на многопоточные приложения (подробнее см. в документации Visual C++).

Для решения этих проблем в Visual C++ были разработаны новые функции `_beginthreadex` и `_endthreadex`, которые следует использо-

вать для создания и завершения потока. Прототипы этих функций объявлены в заголовочном файле process.h:

```
uintptr_t _beginthreadex(void *_Security, unsigned _StackSize,
    unsigned (__stdcall *_StartAddress)(void *),
    void *_ArgList, unsigned _InitFlag, unsigned *_ThrdAddr);

void _endthreadex(unsigned _Retval);
```

Параметры функций _beginthreadex и _endthreadex имеют те же значения, что и параметры у функций CreateThread и ExitThread, но типы данных приведены в синтаксисе языка C/C++. Кроме того, функция потока теперь имеет немного другую сигнатуру:

```
unsigned __stdcall ThreadFunc(void *lpParameter);
```

Как и CreateThread, функция _beginthreadex возвращает дескриптор созданного потока. Однако из-за некоторого расхождения в типах данных, придется позаботиться о приведении значения к типу HANDLE:

```
1  HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0, ThreadFunc,
   NULL, 0, NULL);
2
3  if (NULL != hThread) // если поток успешно создан
4  {
5      CloseHandle(hThread); // закрываем дескриптор потока
6  } // if
```

Если поток был создан с помощью функции _beginthreadex, то когда функция потока возвращает управление, вызывается функция _endthreadex, в которую передается значение, возвращенное функцией потока.

На самом деле функции _beginthreadex и _endthreadex являются «обертками» для традиционных функций CreateThread и ExitThread. Функция _beginthreadex предварительно выделяет память для структуры _tiddata, которая обеспечивает в создаваемом потоке корректную работу функций стандартной библиотеки C/C++, а затем вызывает функцию CreateThread. Функция _endthreadex освобождает выделенную память и вызывает функцию ExitThread. Подробное описание действий этих функций, а также структуры _tiddata, можно найти в документации Visual C++.

Идентификаторы и псевдодескрипторы потоков

При создании потока можно получить его дескриптор и идентификатор. Для получения дескриптора и идентификатора текущего потока используются функции GetCurrentThread и GetCurrentThreadId:

```
HANDLE GetCurrentThread();
DWORD GetCurrentThreadId();
```

Функция `GetCurrentThread` возвращает псевдодескриптор текущего потока (константа типа `HANDLE` обычно `–1`). Псевдодескриптор потока является не наследуемым и имеет максимально возможные права доступа. Когда псевдодескриптор больше не нужен, закрывать его нет необходимости. Вызов функции `CloseHandle` с псевдодескриптором потока не уменьшает счетчик объекта ядра «поток» и, следовательно, не закрывает этот объект.

Реальный дескриптор текущего потока можно получить, вызвав функцию `DuplicateHandle`, передав ей значение псевдодескриптора.

Функция `GetCurrentThreadId` возвращает идентификатор текущего потока. Для получения дескриптора потока по его идентификатору используется функция `OpenThread`:

```
HANDLE OpenThread(DWORD dwDesiredAccess, BOOL bInheritHandle,
    DWORD dwThreadId);
```

Первый параметр, *`dwDesiredAccess`*, определяет запрашиваемые права доступа к объекту ядра «поток». Некоторые из возможных значений этого параметра перечислены в табл. 3.7. Полный перечень значений параметра *`dwDesiredAccess`* можно найти в документации Platform SDK.

Таблица 3.7. Права доступа (значения параметра *`dwDesiredAccess`*)

Значение	**Описание**
`SYNCHRONIZE`	Разрешает использование дескриптора потока в функциях ожидания для ожидания завершения потока
`THREAD_ALL_ACCESS`	Все возможные права доступа
`THREAD_QUERY_INFORMATION`	Необходимо для получения информации о потоке
`THREAD_SET_INFORMATION`	Необходимо для задания информации о потоке
`HREAD_SUSPEND_RESUME`	Разрешает использование дескриптора потока, для приостановления или возобновления работы потока
`THREAD_TERMINATE`	Разрешает использование дескриптора потока для завершения его работы, используя функцию `TerminateThread`

Второй параметр, *`bInheritHandle`*, позволяет указать, должен ли полученный дескриптор быть наследуемым.

Третий параметр, *dwThreadId*, содержит идентификатор потока, дескриптор которого нужно получить.

Если функция OpenThread завершается успешно, она возвращает дескриптор запрашиваемого потока, увеличивая на 1 счетчик соответствующего объекта ядра. В случае ошибки возвращается NULL.

Приоритет потока

Приоритет потока устанавливается относительно класса приоритета процесса. Относительные приоритеты потоков описаны в табл. 3.8. Исходя из класса приоритета процесса и относительного приоритета потока, операционная система формирует уровень приоритета потока от 0 (самый низкий) до 31 (самый высокий).

Таблица 3.8. Относительные приоритеты потоков

Относительный приоритет потока	**Описание**
THREAD_PRIORITY_TIME_CRITICAL (Критичный по времени)	Поток выполняется с приоритетом 31 для процессов реального времени и с приоритетом 15 для остальных процессов
THREAD_PRIORITY_HIGHEST (Максимальный)	Поток выполняется с приоритетом на два уровня выше обычного
THREAD_PRIORITY_ABOVE_NORMAL (Выше среднего)	Поток выполняется с приоритетом на один уровень выше обычного
THREAD_PRIORITY_NORMAL (Средний)	Поток выполняется с обычным приоритетом процесса данного класса
THREAD_PRIORITY_BELOW_NORMAL (Ниже среднего)	Поток выполняется с приоритетом на один уровень ниже обычного
THREAD_PRIORITY_LOWEST (Минимальный)	Поток выполняется с приоритетом на два уровня ниже обычного
THREAD_PRIORITY_IDLE (Простаивающий)	Поток выполняется с приоритетом 16 для процессов реального времени и с приоритетом 1 для остальных процессов

Потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом. В табл. 3.9 показано, как формируется уровень приоритета в Windows Vista. Однако в других версиях Windows формирование уровня приоритета может происходить иначе.

Созданный поток получает приоритет равный приоритету процесса. При изменении класса приоритета процесса, автоматически изменяются приоритеты всех его потоков.

Таблица 3.9. Формирование уровней приоритета в Windows Vista

Относительный приоритет потока	**Класс приоритета процесса**					
	Низкий	Ниже среднего	Средний	Выше среднего	Высокий	Реального времени
Критичный по времени	15	15	15	15	15	31
Максимальный	6	8	10	12	15	26
Выше среднего	5	7	9	11	14	25
Средний	4	6	8	10	13	24
Ниже среднего	3	5	7	9	12	23
Минимальный	2	4	6	8	11	22
Простаивающий	1	1	1	1	1	16

Чтобы определить относительный приоритет потока используется функция `GetThreadPriority`, а чтобы его задать – `SetThreadPriority`:

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

Параметр *hThread* – это дескриптор потока. Параметр *nPriority* задает относительный приоритет потока.

Функция `GetThreadPriority` завершится с ошибкой, если дескриптор, который содержится в параметре *hThread*, не обладает правом доступа `THREAD_QUERY_INFORMATION`. Если функция `GetThreadPriority` завершается успешно, она возвращает предыдущее значение относительного приоритета потока (см. табл. 3.8); в случае ошибки возвращается значение `THREAD_PRIORITY_ERROR_RETURN`.

Функция `SetThreadPriority` завершится с ошибкой, если дескриптор, который содержится в параметре *hThread*, не обладает правом доступа `THREAD_SET_INFORMATION`. Если функция `SetThreadPriority` завершается успешно, возвращаемое значение отлично от `FALSE`.

Приостановка, возобновление и переключение потоков

В каждом объекте ядра «поток» есть *счетчик приостановок* (suspend counter). Поток выполняется только в том случае, когда значение этого счетчика равно нулю. Если создавать поток в приостановленном состоянии, начальное значение его счетчика приостановок равно `1`, иначе – `0`. Уменьшить или увеличить значение счетчика приостановок потока можно с помощью функций `ResumeThread` и `SuspendThread` соответственно:

```
DWORD ResumeThread(HANDLE hThread);
```

```
DWORD SuspendThread(HANDLE hThread);
```

Параметр *hThread* – дескриптор потока, счетчик которого необходимо изменить. Этот дескриптор потока должен обладать правом доступа THREAD_SUSPEND_RESUME. В случае успешного выполнения обе функции возвращают предыдущее значение счетчика приостановок, иначе возвращаемое значение равно –1.

Также поток может приостановить выполнение на определенный период, вызвав функцию Sleep:

```
VOID Sleep(DWORD dwMilleseconds);
```

Параметр *dwMilleseconds* указывает время (в миллисекундах), на которое следует приостановить поток. Операционная система приостанавливает поток *приблизительно* на период, указанный параметром *dwMilleseconds*, хотя не исключено, что поток будет приостановлен на несколько секунд или даже минут дольше. Это зависит от того, какая ситуация сложится в операционной системе к тому времени.

Вызывая функцию Sleep, поток отказывается от остатка выделенного ему процессорного времени. Таким образом, если вызвать функцию Sleep и передать в параметре *dwMilleseconds* нулевое значение, можно заставить операционную систему предоставить процессорное время другому потоку. Однако операционная система может снова запустить вызывающий поток, если нет других готовых к исполнению потоков с тем же приоритетом.

Для того чтобы дать возможность выполняться потоку с более низким приоритетом (если нет потоков с тем же приоритетом) и которому не хватает процессорного времени следует использовать функцию SwitchToThread:

```
BOOL SwitchToThread();
```

Функция SwitchToThread возвращает FALSE, если на момент ее вызова нет ни одного потока готового к исполнению; в ином случае – ненулевое значение.

Синхронизация потоков

При написании многопоточных приложений может понадобиться *синхронизация потоков* (thread synchronization). Обычно синхронизацию потоков применяют в следующих случаях:

- когда несколько потоков должны совместно использовать некоторые общие ресурсы;
- когда нужно уведомлять другие потоки о завершении или о начале каких-либо операций.

Как известно, все потоки процесса, разделяют некоторые общие ресурсы – такие, как адресное пространство процесса или открытые файлы. В один момент времени с определенным разделяемым ресурсом должен работать только один поток, иначе может получиться так, что один поток пишет в блок памяти, из которого другой что-то считывает. Для решения этой проблемы используется механизм *взаимного исключения* (mutual exclusion) доступа к разделяемому ресурсу, который заключается в том, что в определенный момент времени только один поток может иметь монопольный доступ к разделяемому ресурсу. При этом доступ к разделяемому ресурсу блокируется для всех остальных потоков, которым нужен этот ресурс, и они вынуждены ждать, пока доступ к нему не будет разблокирован.

Кроме разграничения доступа к разделяемым ресурсам, потокам может потребоваться кооперация выполняемых ими действий. Например, может возникнуть ситуация, при которой потоку для продолжения своей работы требуется результат выполнения другого потока. В приведенном случае поток должен синхронизировать свои действия с другими потоками по готовности данных.

Существуют два подхода к синхронизации потоков: *синхронизация в пользовательском режиме* и *синхронизация с использованием объектов ядра*. Механизмы синхронизации в пользовательском режиме обеспечивают высокое быстродействие, но у них есть ряд ограничений. Механизмы синхронизации с использованием объектов ядра обладают меньшим быстродействием, но они предоставляют куда больше возможностей, например, их можно использовать для синхронизации потоков разных процессов.

Функции взаимоблокировки

В Win32 API имеется набор функций под общим названием *функции взаимоблокировки* (interlocked functions). Эти функции позволяют безопасно изменять содержимое переменных, используемых сразу несколькими потоками. При этом вызов функции взаимоблокировки не приводит к блокированию вызывающего потока.

Разграничение доступа к разделяемым ресурсам (в данном случае к переменным) с помощью функций взаимоблокировки возможно благодаря тому, что каждая такая функция гарантированно выполняется, как *атомарная операция* (atomic operation), которая не может быть прервана в ходе своего выполнения.

В табл. 3.10 перечислены наиболее часто используемые функции взаимоблокировки. Подробное описание этих и других функций взаимоблокировки приведено в документации Platform SDK.

Таблица 3.10. Некоторые функции взаимоблокировки

Функция	Описание
`InterlockedCompareExchange`	Выполняет сравнение и при необходимости изменяет значение указанной переменной, возвращает исходное значение
`InterlockedDecrement`	Уменьшает значение указанной переменной на 1 и возвращает результат
`InterlockedExchange`	Изменяет значение указанной переменной на заданное значение и возвращает исходное значение
`InterlockedExchangeAdd`	Изменяет значение указанной переменной на заданную величину и возвращает результат
`InterlockedIncrement`	Увеличивает значение указанной переменной на 1 и возвращает результат

Перечисленные функции взаимоблокировки имеют следующие прототипы:

```
LONG InterlockedCompareExchange(LONG volatile *Destination,
    LONG Exchange, LONG Comparand);

LONG InterlockedDecrement(LONG volatile *Addend);

LONG InterlockedExchange(LONG volatile *Target, LONG Value);

LONG InterlockedExchangeAdd(LONG volatile *Addend,
    LONG Value);

LONG InterlockedIncrement(LONG volatile *Addend);
```

Следующий пример демонстрирует использование функции взаимоблокировки `InterlockedIncrement`:

```
1   long cRef = 0;
2   InterlockedIncrement(&cRef); // cRef += 1, функция вернет 1
```

Следует отметить, что в Win32 API нет функции взаимоблокировки, которая просто считывает значение какой-нибудь переменной, не изменяя его. На самом деле такая функция и не нужна. Всегда можно обратиться непосредственно к самой переменной, даже если в тот же момент другой поток изменяет значение этой переменной с помощью какой-либо функций взаимоблокировки. В этом случае переменная вернет либо исходное, либо уже измененное значение.

Некоторые функции взаимоблокировки можно использовать для реализации механизма взаимного исключения:

```
// циклическая блокировка (spinlock)
// записываем значение «занято»
while (InterlockedExchange(&lock, 1) != 0) SwitchToThread();

/* здесь можно безопасно работать с разделяемым ресурсом */

InterlockedExchange(&lock, 0); // записываем значение «свободно»
```

В этом примере каждый поток, желающий получить доступ к разделяемому ресурсу, с помощью функции `InterlockedExchange` записывает в общую переменную `lock` условное значение «*занято*». Если предыдущее значение переменной `lock` было «*свободно*», считается, что данный поток получил доступ к ресурсу, в противном случае, поток крутится в цикле, пока в переменную `lock` не будет записано значение «*свободно*». После работы с разделяемым ресурсом поток должен записать в переменную условное значение «свободно». Переменная `lock`, которую используют синхронизируемые потоки, должна быть определена следующим образом:

```
long lock = 0; // начальное значение «свободно»
```

Критические секции

Область программного кода, в которой требуется монопольный доступ к разделяемым ресурсам, называется *критической секцией* (critical section). Механизм критических секций гарантирует, что в один момент времени монопольный доступ к какому-либо разделяемому ресурсу сможет получить только один поток и, пока этот поток не выйдет за границы критической секции, все остальные потоки, которым нужен этот же ресурс, будут вынуждены ждать.

Для использования критической секции нужно создать переменную типа `CRITICAL_SECTION` и проинициализировать ее с помощью функции `InitializeCriticalSection`:

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

После этого любой участок программного кода, работающий с разделяемым ресурсом, нужно заключить в вызовы функций `EnterCriticalSection` и `LeaveCriticalSection`:

```
EnterCriticalSection(&cs); // начало критической секции

/* здесь можно безопасно работать с разделяемым ресурсом */

LeaveCriticalSection(&cs); // конец критической секции
```

Функция EnterCriticalSection блокирует поток, если в данной критической секции присутствует другой поток. Ожидающий поток будет разблокирован только после того, как будет вызвана функция LeaveCriticalSection.

```
VOID EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection);

VOID LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection);
```

Вместо функции EnterCriticalSection можно использовать функцию TryEnterCriticalSection, которая никогда не приостанавливает выполнение вызывающего потока:

```
BOOL TryEnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection);
```

Функция TryEnterCriticalSection возвращает значение TRUE, если вызывающий поток получил доступ к ресурсу. Если ресурс уже занят другим потоком, функция возвращает значение FALSE.

Вызов функции TryEnterCriticalSection позволяет потоку проверить, доступен ли ресурс, и если нет заняться чем-то другим:

```
1  if (TryEnterCriticalSection(&cs) != FALSE)
2  {
3      /* здесь можно безопасно работать с разделяемым ресурсом */
4
5      LeaveCriticalSection(&cs);
6  } // if
```

Когда критическая секция более не нужна, ее следует удалить вызовом функции DeleteCriticalSection:

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

Тонкая блокировка чтения и записи

В отличие от критической секции *тонкая блокировка чтения и записи* (slim reader-writer lick) различает потоки, обращающиеся к разделяемому ресурсу для чтения и для записи. Тонкая блокировка позволяет нескольким «читающим» потокам одновременно обращаться к разделяемому ресурсу, поскольку чтение не грозит повреждением такого ресурса. Однако для «записывающих» потоков предоставляется монопольный доступ к разделяемому ресурсу.

Чтобы использовать механизм тонкой блокировки чтения и записи, нужно создать переменную типа SRWLOCK и проинициализировать ее с помощью функции InitializeSRWLock:

```
VOID InitializeSRWLock(PSRWLOCK SRWLock);
```

Теперь чтобы получить монопольный доступ к ресурсу, «записывающий» поток должен вызвать функцию AcquireSRWLockExclusive. После модификации ресурса снять блокировку можно вызовом функции ReleaseSRWLockExclusive.

```
VOID AcquireSRWLockExclusive(PSRWLOCK SRWLock);
VOID ReleaseSRWLockExclusive(PSRWLOCK SRWLock);
```

В случае «читающего» потока все аналогично, только используется другая пара функций – AcquireSRWLockShared и ReleaseSRWLockShared.

```
VOID AcquireSRWLockShared(PSRWLOCK SRWLock);
VOID ReleaseSRWLockShared(PSRWLOCK SRWLock);
```

Функции ожидания

Почти все объекты ядра находятся либо в *свободном состоянии* (signaled state), либо в *занятом состоянии* (nonsignaled state). Переход из одного состояния в другое для каждого объекта ядра осуществляется по-своему. Так объект ядра «процесс» находится в занятом состоянии, пока выполняется сопоставленный с ним процесс, и переходит в свободное состояние, когда процесс завершается. Точно такие же правила распространяются и на объект ядра «поток». Следует отметить, что объекты ядра «процесс» и «поток» никогда не возвращаются в занятое состояние.

Функции ожидания позволяют потоку приостановить свою работу и ждать освобождения какого-либо объекта ядра. Если в момент вызова функции ожидания заданный объект свободен, поток не будет приостановлен.

В Win32 API имеется сразу несколько функций ожидания, но чаще всего используется функция WaitForSingleObject:

```
DWORD WaitForSingleObject(
   HANDLE hHandle, DWORD dwMilliseconds);
```

Первый параметр, *hHandle*, – дескриптор объекта ядра, поддерживающий состояния «свободен» и «занят». Этот дескриптор должен обладать правом доступа SYNCHRONIZE.

Второй параметр, *dwMilliseconds*, определяет, сколько времени (в миллисекундах) следует ждать освобождения указанного объекта ядра. Если этот параметр принимает значение INFINITE, вызывающий поток будет ждать бесконечно (либо до завершения работы процесса).

Функция WaitForSingleObject возвращает одно из следующих значений:

- WAIT_OBJECT_0 – объект ядра перешел в свободное состояние;
- WAIT_TIMEOUT – время ожидания истекло, но объект ядра остался в занятом состоянии;
- WAIT_FAILED – возникла ошибка.

Если необходимо ожидать сразу несколько объектов ядра можно использовать функцию WaitForMultipleObjects:

```
DWORD WaitForMultipleObjects(DWORD nCount,
    const HANDLE *lpHandles, BOOL bWaitAll,
    DWORD dwMilliseconds);
```

Первый параметр, *nCount*, определяет количество ожидаемых объектов ядра. Его значение должно быть от 1 до MAXIMUM_WAIT_OBJECTS.

Второй параметр, *lpHandles*, указывает на массив дескрипторов объектов ядра. Каждый дескриптор в этом массиве должен обладать правом доступа SYNCHRONIZE.

Третий параметр, *bWaitAll*, определяет должен ли, вызывающий поток ждать, пока не освободятся все указанные объекты ядра. Если этот параметр принимает значение FALSE, вызывающий поток будет ждать освобождения любого из указанных объектов ядра.

Последний параметр, *dwMilliseconds*, задает, сколько времени (в миллисекундах) следует ждать освобождения указанных объектов ядра. Если этот параметр принимает значение INFINITE, вызывающий поток будет ждать вечно (либо до завершения работы процесса).

Функция WaitForMultipleObjects, как и функция WaitForSingleObject может вернуть значения WAIT_TIMEOUT и WAIT_FAILED, которые не требуют дополнительных комментариев. Если при вызове функции WaitForMultipleObjects в параметре *bWaitAll* было передано значение TRUE и все заданные объекты ядра перешли в свободное состояние, функция возвращает значение WAIT_OBJECT_0. Если же в параметре *bWaitAll* передано значение FALSE и любой из заданных объектов ядра перешел в свободное состояние, эта функция возвращает значение WAIT_OBJECT_0 + *i*, где *i* – индекс элемента в массиве дескрипторов объектов ядра, на который указывает параметр *lpHandles*.

Мьютексы

Мьютекс (mutex, mutual exclusion) – объект ядра, гарантирующий потокам взаимное исключение доступа к разделяемому ресурсу. Изначально предполагалось, что Windows будет поддерживать операционную систему OS/2, где мьютекс должен был иметь структуру, от которой требовалось, чтобы поток мог отказаться от объекта, оставив его недоступным. Поскольку подобное поведение для такого объекта считалось необычным, был создан объект, получивший название *мутант* (mutant). Со временем от поддержки OS/2 отказались, и объект стал использоваться в Windows под названием «мьютекс» (но при этом сохранил внутреннее имя «мутант»).

В каждом объекте ядра «мьютекс» есть переменная, в которой запоминается идентификатор захватившего его потока. Этот идентификатор может быть равен нулю, что означает – мьютекс не захвачен ни одним из потоков. Если мьютекс не захвачен, он находится в свободном состоянии, иначе – в занятом состоянии.

Для создания объектов ядра «мьютекс» предназначены две функции – `CreateMutex` и `CreateMutexEx`:

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

HANDLE CreateMutexEx(LPSECURITY_ATTRIBUTES lpMutexAttributes,
    LPCTSTR lpName, DWORD dwFlags, DWORD dwDesiredAccess);
```

Первый параметр, *`lpMutexAttributes`*, указывает на структуру `SECURITY_ATTRIBUTES`, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «мьютекс». Если этот параметр установлен в `NULL`, объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Параметр *`lpName`* указывает на строку, которая определяет имя создаваемого объекта ядра. Если этот параметр установлен в `NULL`, будет создан безымянный объект ядра.

Параметр *`bInitialOwner`* (*`dwFlags`*) определяет начальное состояние мьютекса. Если этот параметр принимает значение `FALSE` (`0`), созданный мьютекс будет находиться в свободном состоянии. Если же он принимает значение `TRUE` (`CREATE_MUTEX_INITIAL_OWNER`), созданный мьютекс захватывается вызывающим потоком, и потому будет находиться в занятом состоянии.

Параметр *`dwDesiredAccess`* позволяет указать уровень доступа для возвращаемого дескриптора объекта ядра «мьютекс». Возможные значения этого параметра перечислены в табл. 3.11.

Если функция CreateMutex(Ex) выполняется успешно, возвращается дескриптор созданного объекта ядра «мьютекс» с полными правами доступа или правами, указанными в параметре *dwDesiredAccess*, иначе возвращается NULL.

Таблица 3.11. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
MUTEX_ALL_ACCESS	Все возможные права доступа
SYNCHRONIZE	Разрешает использование дескриптора мьютекса в функциях ожидания

Кроме того, любой поток может открыть уже созданный мьютекс с помощью функции OpenMutex:

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

Первый параметр, *dwDesiredAccess*, определяет запрашиваемые права доступа к объекту ядра «мьютекс» (см. табл. 3.11). Второй параметр, *bInheritHandle*, позволяет указать, должен ли полученный дескриптор быть наследуемым. Третий параметр, *lpName*, определяет имя мьютекса, дескриптор которого нужно получить.

Если функция OpenMutex завершается успешно, она возвращает дескриптор запрашиваемого объекта ядра, увеличивая на 1 значение его счетчика. В случае ошибки возвращается NULL.

Не нужный объект ядра «мьютекс» следует, как всегда, закрыть вызовом функции CloseHandle.

Мьютексы используются практически так же, как и критические секции:

```
1  // захватываем мьютекс (ожидаем, если мьютекс уже захвачен)
2  DWORD dwResult = WaitForSingleObject(hMutex, INFINITE);
3
4  if (WAIT_FAILED != dwResult) // мьютекс успешно захвачен
5  {
6      /* здесь можно безопасно работать с разделяемым ресурсом */
7
8      ReleaseMutex(hMutex); // освобождаем мьютекс
9  } // if
```

С помощью функции WaitForSingleObject (или другой функции ожидания) вызывающий поток захватывает мьютекс. При этом объект ядра «мьютекс» переходит в занятое состояние. Если мьютекс уже захвачен другим потоком, вызывающий поток ждет, пока этот мьютекс

не освободится. Мьютекс можно перевести в свободное состояние с помощью функции ReleaseMutex:

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

Здесь параметр *hMutex* – дескриптор мьютекса, который нужно освободить. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции ReleaseMutex, она вернет FALSE.

Если поток завершился, не освободив мьютекс, то этот мьютекс все равно будет освобожден. При этом поток, ожидающий мьютекс, получит от функции WaitForSingleObject (или другой функции ожидания) значение WAIT_ABANDONED, которое свидетельствует о том, что мьютекс был освобожден некорректно.

Семафоры

Семафор (semaphore) – объект ядра, предназначенный для того, чтобы ограничить максимальное число потоков, одновременно работающих с неким разделяемым ресурсом.

В каждом объекте ядра «семафор» есть специальный счетчик. Когда значение этого счетчика больше нуля, семафор находится в свободном состоянии, иначе – в занятом состоянии. Не следует путать специальный счетчик семафора и его счетчик объекта ядра.

Для создания объектов ядра «семафор» используют одну из двух функций – CreateSemaphore и CreateSemaphoreEx:

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

HANDLE CreateSemaphoreEx(
   LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
   LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName,
   DWORD dwFlags, DWORD dwDesiredAccess);
```

Первый параметр, *lpSemaphoreAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «семафор». Если этот параметр установлен в NULL, объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Параметры *lInitialCount* и *lMaximumCount* задают, соответственно, начальное и максимальное значение счетчика семафора. Допускается присвоение параметру *lInitialCount* отрицательных значений.

Четвертый параметр *lpName* указывает на строку, которая определяет имя создаваемого объекта ядра. Если этот параметр установлен в NULL, будет создан безымянный объект ядра.

Параметр *dwFlags* не используется и должен принимать значение равное нулю. Параметр *dwDesiredAccess* позволяет указать уровень доступа для возвращаемого дескриптора объекта ядра «семафор». Возможные значения этого параметра перечислены в табл. 3.12.

Таблица 3.12. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
SEMAPHORE_ALL_ACCESS	Все возможные права доступа
SEMAPHORE_MODIFY_STATE	Разрешает использование дескриптора для изменения состояния семафора
SYNCHRONIZE	Разрешает использование дескриптора семафора в функциях ожидания

Если функция CreateSemaphore(Ex) выполняется успешно, возвращается дескриптор созданного объекта ядра «семафор» с полными правами доступа или правами, указанными в параметре *dwDesiredAccess*, иначе возвращается NULL.

Открыть уже созданный объект ядра «семафор» можно с помощью функции OpenSemaphore:

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

Как всегда, не нужный объект ядра «семафор» следует закрыть вызовом функции CloseHandle.

Поток получает доступ к разделяемому ресурсу, который «охраняется» семафором, с помощью функции WaitForSingleObject (или другой функции ожидания), передавая ей дескриптор этого семафора. Если значение счетчика семафора больше нуля (семафор свободен), значение этого счетчика уменьшается на 1. Если же значение счетчика семафора равно нулю (семафор занят), вызывающий поток ждет, пока другой поток не увеличит значение этого счетчика. Когда это случится, ждущий поток возобновит свою работу (при этом уменьшит значение счетчика семафора на 1). Увеличить значение счетчика семафора можно с помощью функции ReleaseSemaphore:

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

Первый параметр, *hSemaphore*, – дескриптор семафора, счетчик которого нужно увеличить. Этот дескриптор должен обладать правом доступа SEMAPHORE_MODIFY_STATE.

Второй параметр, *lReleaseCount*, определяет величину, на которую будет увеличено текущее значение счетчика семафора.

Третий параметр, *lpPreviousCount*, указывает на переменную типа LONG, в которую будет записано предыдущее значение счетчика. Этот параметр может быть установлен в NULL.

В случае успешного завершения функция ReleaseSemaphore возвращает значение отличное от FALSE.

События

Событие (event) – объект ядра, используемый потоком для того, чтобы сигнализировать другим потокам о наступлении какого-либо события. Объекты ядра «события» бывают двух типов: *сбрасываемые вручную события* (manual-reset events) и *автоматически сбрасываемые события* (auto-reset events). Первые позволяют возобновить работу сразу нескольких ждущих потоков, вторые – только одного.

Для создания объектов ядра «событие» применяют две функции – CreateEvent и CreateEventEx:

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

HANDLE CreateEventEx(LPSECURITY_ATTRIBUTES lpEventAttributes,
    LPCTSTR lpName, DWORD dwFlags, DWORD dwDesiredAccess);
```

Первый параметр, *lpEventAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «событие». Если этот параметр установлен в NULL, объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Параметр *lpName* указывает на строку, которая определяет имя создаваемого объекта ядра. Если этот параметр установлен в NULL, будет создан безымянный объект ядра.

Параметр *bManualReset* определяет, создаваемое событие будет сбрасываться вручную (TRUE) или автоматически (FALSE). Параметр *bInitialState* определяет начальное состояние создаваемого объекта ядра «события» – свободное (TRUE) или занятое (FALSE).

Параметр *dwFlags* в функции CreateEventEx заменяет параметры *bManualReset* и *bInitialState* из функции CreateEvent. Этот параметр может принимать одно или несколько значений из табл. 3.13.

Параметр *dwDesiredAccess* позволяет указать уровень доступа для возвращаемого дескриптора объекта ядра «событие». Возможные значения этого параметра перечислены в табл. 3.14.

Если функция CreateEvent(Ex) выполняется успешно, возвращается дескриптор созданного объекта ядра «событие» с полными правами

доступа или правами, указанными в параметре *dwDesiredAccess*, иначе возвращается NULL.

Таблица 3.13. Флаги создания события (значения параметра *dwFlags*)

Флаг	**Описание**
CREATE_EVENT_INITIAL_SET	Если этот флаг установлен, событие создается как свободное, иначе – как занятое
CREATE_EVENT_MANUAL_RESET	Если этот флаг установлен, событие создается как сбрасываемое вручную событие, иначе – как автоматически сбрасываемое событие

Таблица 3.14. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
EVENT_ALL_ACCESS	Все возможные права доступа
EVENT_MODIFY_STATE	Разрешает использование дескриптора для изменения состояния события
SYNCHRONIZE	Разрешает использование дескриптора события в функциях ожидания

Открыть уже созданный объект ядра «событие» можно с помощью функции OpenEvent:

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

Как всегда, не нужный объект ядра «событие» следует закрыть вызовом функции CloseHandle.

Создав событие, можно управлять его состоянием с помощью трех функций – SetEvent, ResetEvent и PulseEvent:

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

Параметр *hEvent* – дескриптор объект ядра «событие», состояние которого нужно изменить. Этот дескриптор должен обладать правом доступа EVENT_MODIFY_STATE. В случае успешного завершения эти функции возвращают значение отличное от FALSE.

Функция SetEvent переводит объект ядра «событие» в свободное состояние, а функция ResetEvent переводит его в занятое состояние.

Важно отметить, что автоматически сбрасываемое событие, когда его ожидание потоком успешно завершается, автоматически сбрасыва-

ется в занятое состояние функцией `WaitForSingleObject` (или другой функцией ожидания). Таким образом, для событий этого типа обычно не требуется вызывать функцию `ResetEvent`.

Функция `PulseEvent` переводит объект ядра «событие» в свободное состояние и тут же возвращает его обратно в занятое состояние. Вызов этой функции равнозначен последовательному вызову функций `SetEvent` и `ResetEvent`.

Задание к работе

1. Разработать в Visual C++ оконное приложение Win32, которое:
   - должно выводить в своем окне список всех процессов;
   - должно выводить список модулей, загруженных выбранным процессом;
   - должно приостанавливать свою работу, до завершения работы выбранного процесса или до истечения задаваемого времени;
   - должно иметь возможность принудительно завершать работу выбранного процесса;
   - должно иметь возможность создавать несколько процессов и группировать их в задание;
   - должно проверять включено ли оно в какое-либо задание, если да, то выводить список всех процессов, включенных в тоже задание;
   - должно иметь возможность изменять свой класс приоритета и относительный приоритет своего главного потока.
2. Разработать в Visual C++ приложение Win32, которое будет формировать матрицу, заполненную случайными числами от 0 и 999, и определять в ней количество двухзначных чисел, а также максимальное и минимальное значения. При этом для каждой строки матрицы следует использовать отдельные потоки.

   Синхронизация потоков должна выполняться с помощью механизма, который указан в варианте задания.
3. Разработать в Visual C++ приложение Win32, которое будет порождать несколько своих экземпляров, а затем ожидать завершения их работы. Все порожденные экземпляры разработанного приложения должны по очереди выполнять счет до 10, после чего начинать считать заново. Работа приложения завершается после того, как оно три раза сосчитает до 10.

Синхронизация должна выполняться с помощью объекта ядра, указанного в варианте задания.

4. Протестировать работу разработанных приложений на компьютере под управлением Windows XP (или выше). Результаты тестирования отразить в отчете.

5. Исследовать работу разработанных приложений с помощью утилиты Process Explorer. Результаты исследования отразить в отчете.

   Обязательно необходимо убедиться в том, что отсутствует утечка дескрипторов объектов ядра.

6. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Механизм синхронизации	Объект ядра (*механизм совместного использования*)
1,16	Функции взаимоблокировки	Мьютекс (*наследование*)
2,17	Критические секции	Семафор (*дублирование*)
3,18	Тонкая блокировка чтения и записи	Событие (*именование*)
4,19	Функции взаимоблокировки	Семафор (*именование*)
5,20	Критические секции	Мьютекс (*дублирование*)
6,21	Тонкая блокировка чтения и записи	Событие (*наследование*)
7,22	Функции взаимоблокировки	Семафор (*наследование*)
8,23	Критические секции	Событие (*дублирование*)
9,24	Тонкая блокировка чтения и записи	Мьютекс (*именование*)
10,25	Функции взаимоблокировки	Событие (*именование*)
11,26	Критические секции	Семафор (*дублирование*)
12,27	Тонкая блокировка чтения и записи	Мьютекс (*наследование*)
13,28	Функции взаимоблокировки	Событие (*наследование*)
14,29	Критические секции	Мьютекс (*дублирование*)
15,30	Тонкая блокировка чтения и записи	Семафор (*именование*)

Контрольные вопросы

1. Что в Windows называют объектом ядра? Какие объекты ядра позволяет создавать Windows?
2. Какие объекты ядра могут находиться в свободном или занятом состоянии?
3. Для чего предназначена утилита WinObj?
4. Как создаются и удаляются объекты ядра? Что такое счетчик использования объекта ядра?
5. Для чего используется функция `CloseHandle`?
6. Для чего применяют наследование дескрипторов объектов ядра? Как создать наследуемый дескриптор объекта ядра?
7. Для чего применяют дублирование дескрипторов объектов ядра? Какую функцию Win32 API следует использовать для дублирования дескриптора объекта ядра?
8. Что такое именованные объекты ядра? Для чего применяют именованные объекты ядра?
9. Что в Windows называют процессом?
10. Для чего предназначены утилиты Task Manager и Process Explorer?
11. Какую функцию Win32 API следует использовать для того, чтобы создать процесс? Что такое обособленный процесс?
12. Каким образом завершается работа процесса?
13. Как определить код завершения процесса?
14. Какие функции Win32 API применяются для принудительного завершения работы процесса?
15. Что такое идентификатор процесса? Как определить идентификатор текущего процесса?
16. Что такое псевдодескриптор процесса? Как определить псевдодескриптор текущего процесса?
17. Какие функции Win32 API следует использовать для того, чтобы получить дескриптор процесса, зная его идентификатор или псевдодескриптор?
18. Что такое класс приоритета процесса? Какие классы приоритета существуют в Windows?
19. Какие функции Win32 API следует использовать для работы с классом приоритета процесса?
20. Какую функцию Win32 API следует использовать для того, чтобы получить список идентификаторов процессов?

21. Какие функции Win32 API можно использовать для того, чтобы получить список модулей, загруженных процессом?
22. Какие функции Win32 API следует использовать для того, чтобы получить имя модуля, загруженного процессом?
23. Что в Windows называют заданием?
24. Какую функцию Win32 API следует использовать для того, чтобы создать задание?
25. Какие функции Win32 API используются для работы с ограничениями, накладываемыми на процессы в задании?
26. Как включить процесс в задание? Какую функцию Win32 API для этого следует использовать?
27. Можно ли включить процесс одновременно в несколько заданий?
28. Можно ли исключить процесс из задания?
29. Всегда ли дочерние процессы принадлежат к тому же заданию, что и родительский процесс?
30. Какую функцию Win32 API следует использовать для того, чтобы принудительно завершить работу всех процессов, включенных в одно задание?
31. Как получить список идентификаторов процессов, включенных в задание?
32. Что в Windows называют потоком? Что такое главный поток?
33. Какая функция Win32 API используется для создания потока?
34. Каким образом завершается работа потока? Как определить код завершения потока?
35. Какие функции Win32 API применяются для принудительного завершения работы потока?
36. Для чего предназначена функция `RtlUserThreadStart`?
37. Какие функции Visual C++ следует использовать для создания и завершения потока?
38. Что такое идентификатор потока? Как определить идентификатор текущего потока?
39. Что такое псевдодескриптор потока? Как определить псевдодескриптор текущего потока?
40. Какие функции Win32 API следует использовать для того, чтобы получить дескриптор потока, зная его идентификатор или псевдодескриптор?
41. Что такое счетчик приостановок потока?

42. Какие функции Win32 API применяются для приостановки, возобновления и переключения потоков?
43. Что такое относительный приоритет потока? Каким образом вычисляется уровень приоритета потока?
44. Какие функции Win32 API следует использовать для работы с относительным приоритетом потока?
45. Для чего применяется синхронизация потоков?
46. Что такое функции взаимоблокировки? Как и для чего применяют функции взаимоблокировки?
47. Что такое критические секции? Как и для чего применяют критические секции?
48. Что такое тонкая блокировка чтения и записи? Как и для чего применяют тонкую блокировку чтения и записи?
49. Что такое функции ожидания? Как и для чего применяют функции ожидания?
50. Что такое мьютекс? Как и для чего применяют мьютекс?
51. Что такое семафор? Как и для чего применяют семафор?
52. Что такое событие? Как и для чего применяют событие?

ЛАБОРАТОРНАЯ РАБОТА № 4
ФАЙЛЫ И КАТАЛОГИ. СИСТЕМНЫЙ РЕЕСТР WINDOWS

Цель работы

Получение практических навыков работы с файлами, каталогами и системным реестром в Windows с применением функций Win32 API.

Основные понятия

Файлы выполняют очень важную функцию в организации долговременного хранения данных, с которыми работают различные приложения. Помимо этого файлы обеспечивают простейшую форму взаимодействия между приложениями. Файлы также применяются для хранения параметров конфигурации операционной системы и различных приложений.

Практически всегда файлы организуют в каталоги. Каталоги также кроме файлов могут содержать вложенные каталоги.

Файлы и каталоги

Для работы с файлами и каталогами в Win32 API имеется целый ряд функций, с помощью которых можно создавать, удалять, копировать, перемещать и переименовывать файлы и каталоги. Некоторые из этих функций работают с атрибутами файлов и каталогов. Кроме того Win32 API предоставляет функции для поиска файлов и каталогов.

Правила именования файлов и каталогов

В Windows поддерживается иерархическая система имен файлов и каталогов, которая основана на следующих правилах:

- При указании пути к файлу или каталогу в качестве разделителя обычно используется символ «\», но в параметрах функций Win32 API для этого можно также использовать символ «/».
- Полное имя файла или каталога, содержащее путь к нему, начинается с указания буквы диска с двоеточием (`:`), например, `A:` или `C:`. Таким образом, полное имя файла или каталога иметь вид:

  диск`:\[`*путь*`]`*имя*

- Для задания полного имени файла или каталога можно использовать *универсальную кодировку имен* (Universal Naming Code, UNC),

в соответствии с которой указание пути начинается с глобального корневого каталога, обозначаемого «\\», с последующим указанием имени компьютера и имени общего ресурса (share name). Таким образом, полное имя файла или каталога в данном случае будет иметь вид:

```
\\имя_компьютера\имя_общего_ресурса\[путь]имя
```

- В именах файлов и каталогов не должны встречаться следующие символы:

```
< > : " | ? * \ /
```

- В именах файлов и каталогов разрешается использовать пробелы.
- Для отделения расширения от основной части имени файла используется точка (.). Кроме того точки могут использоваться и в основной части имени файла.
- Строчные и прописные буквы в именах файлов и каталогов не различаются, то есть имена *не чувствительны к регистру* (case-insensitive), но в тоже время они *запоминают регистр* (case-retaining). Например, если файл был создан с именем «Текстовый файл.txt», то это имя будет использоваться при его отображении, а для доступа к файлу может также использоваться имя «ТЕКСТОВЫЙ ФАЙЛ.TXT».
- Одиночный символ точки (.) и два символа точки (..), используемый в качестве имен каталогов, обозначают, соответственно текущий каталог и его родительский каталог.
- Длина полного имени файла и каталога ограничивается значением MAX_PATH (260 символов, согласно определению в заголовочном файле WinDef.h). Однако, это ограничение можно обойти, используя Unicode-версии функций Win32 API, работающих с файлами и каталогами, предваряя полное имя префиксом «\\?\» следующим образом:

```
\\?\диск:\[путь]имя
```

  При вызове этих функций префикс удаляется, и максимальная длина полного имени файла и каталога составляет 32 767 символов. При этом длина отдельных компонентов полного имени по-прежнему ограничена значением MAX_PATH. Если для задания имения файла или каталога используется UNC, префикс «\\?\» применяться следующим образом:

```
\\?\UNC\имя_компьютера\имя_общего_ресурса\[путь]имя
```

Работа с файлами

В этом разделе рассматриваются основные функции Win32 API, предназначенные для работы с файлами.

Создание и открытие файлов

Чтобы работать с файлом, следует его создать, либо открыть, если он был создан ранее. Эти операции выполняются с помощью одной функции – `CreateFile`. Вызов этой функции заставляет операционную систему создать объект ядра «файл», который используется для работы с файлом. Функция `CreateFile` имеет следующий прототип:

```
HANDLE CreateFile(LPCTSTR lpFileName,
    DWORD dwDesiredAccess, DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);
```

Первый параметр, *`lpFileName`*, указывает на строку, содержащую имя файла, который нужно создать или открыть.

Второй параметр, *`dwDesiredAccess`*, позволяет указать уровень доступа для возвращаемого дескриптора объекта ядра «файл». Возможные значения этого параметра перечислены в табл. 4.1. Отметим, что эти значения можно комбинировать.

Таблица 4.1. Значения параметра *`dwDesiredAccess`*

Значение	**Описание**
`GENERIC_READ`	Разрешает чтение данных из файла
`GENERIC_WRITE`	Разрешает запись данных в файл

Третий параметр, *`dwShareMode`*, управляет совместным доступом к файлу. Этот параметр может принимать одно (или комбинацию) из значений, перечисленных в табл. 4.2. Если параметр *`dwShareMode`* принимает значение равное нулю, совместный доступ к файлу запрещается, но при этом, если файл уже открыт, функция `CreateFile` завершится ошибкой.

Четвертый параметр, *`lpSecurityAttributes`*, указывает на структуру `SECURITY_ATTRIBUTES`, которая позволяет определить атрибуты безопасности создаваемого файла и наследование возвращаемого дескриптора. Дескриптор безопасности, который содержится в этой структуре, используется только при создании файлов в файловой системе NTFS, и игнорируется в остальных случаях. Если параметр *`lpSecurityAttributes`* установить в `NULL`, файл будет создан с атрибутами безопасности по умолчанию, а возвращаемый дескриптор не будет наследуемым.

Таблица 4.2. Значения параметра *dwShareMode*

Значение	Описание
FILE_SHARE_DELETE	Другим процессам разрешается удалять или перемещать файл, даже если он еще не закрыт
FILE_SHARE_READ	Другим процессам разрешается открывать файл для чтения из него данных. Если файл уже открыт с доступом только для записи, функция CreateFile завершится ошибкой
FILE_SHARE_WRITE	Другим процессам разрешается открывать файл для записи в него данных. Если файл уже открыт с доступом только для чтения, функция CreateFile завершится ошибкой

Пятый параметр, *dwCreationDisposition*, задает выполняемые действия с файлом: создать новый файл, открыть существующий файл и т.п. Возможные значения этого параметра приведены в табл. 4.3.

Таблица 4.3. Значения параметра *dwCreationDisposition*

Значение	Описание
CREATE_NEW	Создается новый файл. Если файл с таким именем уже существует, функция CreateFile завершается ошибкой
CREATE_ALWAYS	Создается новый файл, либо перезаписывает уже существующий файл
OPEN_EXISTING	Отрывается файл. Если файла с таким именем не существует, функция CreateFile завершается ошибкой
OPEN_ALWAYS	Отрывается файл, либо создается новый, если файла с таким именем не существует
TRUNCATE_EXISTING	Отрывается файл, при этом его размер усекается до 0 байт. Если файла с таким именем не существует, функция CreateFile завершается ошибкой

Шестой параметр, *dwFlagsAndAttributes*, позволяет указать флаги и атрибуты файла. Флаги позволяют уточнить способ обработки файла. Некоторые из возможных флагов приводятся в табл. 4.4.

Таблица 4.4. Флаги (значения параметра *dwFlagsAndAttributes*)

Флаг	Описание
FILE_FLAG_DELETE_ON_CLOSE	Если этот флаг установлен, после закрытия дескриптора файл будет удален
FILE_FLAG_OVERLAPPED	Этот флаг разрешает асинхронные операции чтения и записи данных

Окончание табл. 4.4.

Флаг	Описание
FILE_FLAG_WRITE_THROUGH	Этот флаг запрещает использование буферизации данных, предназначенных для записи в файл
FILE_FLAG_NO_BUFFERING	Этот флаг запрещает использование буферизации данных при работе с файлом
FILE_FLAG_SEQUENTIAL_SCAN	Если этот флаг установлен, будет использоваться упреждающее чтение данных из файла. Этот флаг следует использовать только при последовательном чтении файла
FILE_FLAG_RANDOM_ACCESS	Этот флаг запрещает упреждающее чтение данных

В табл. 4.5 перечислены наиболее часто применяемые атрибуты файла. При открытии существующего файла функция `CreateFile` игнорирует все атрибуты, заданные в параметре *`dwFlagsAndAttributes`*.

Таблица 4.5. Значения параметра *`dwFlagsAndAttributes`*

Атрибут	Описание
FILE_ATTRIBUTE_ARCHIVE	Файл является архивным. Для новых файлов функция `CreateFile` устанавливает этот флаг автоматически
FILE_ATTRIBUTE_COMPRESSED	Содержимое файла сжато для экономии места на диске
FILE_ATTRIBUTE_ENCRYPTED	Содержимое файла зашифровано для защиты данных
FILE_ATTRIBUTE_HIDDEN	Скрытый файл
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	Содержимое файла не будет индексироваться службой индексирования Windows
FILE_ATTRIBUTE_READONLY	Файл доступен только для чтения
FILE_ATTRIBUTE_SYSTEM	Системный файл
FILE_ATTRIBUTE_TEMPORARY	Временный файл

Подробное описание всех флагов и атрибутов файла можно найти в документации Platform SDK.

Последний параметр, *`hTemplateFile`*, – дескриптор открытого файла с правом доступа `GENERIC_READ`. Если этот параметр не установлен в `NULL`, функция `CreateFile` игнорирует все атрибуты, заданные в параметре *`dwFlagsAndAttributes`*, используя вместо них атрибуты файла, заданного параметром *`hTemplateFile`*. При открытии существующего файла параметр *`hTemplateFile`* игнорируется.

Если функция `CreateFile` выполняется успешно, она возвращает дескриптор открытого файла (точнее объекта ядра «файл»), в случае ошибки возвращается значение `INVALID_HANDLE_VALUE`.

В следующем примере продемонстрированно создание временного файла, который будет удален после закрытия:

Листинг 4.1. Создание временного файла

```
1   // определим флаги и атрибуты файла
2   DWORD dwFlagsAndAttributes = FILE_FLAG_DELETE_ON_CLOSE |
    FILE_ATTRIBUTE_TEMPORARY | FILE_ATTRIBUTE_HIDDEN;
3   // создадим новый файл
4   HANDLE hFile = CreateFile(TEXT("C:\\ProgramData\\temp.txt"),
5       GENERIC_READ | GENERIC_WRITE, // доступ для чтения и записи
6       0, // совместный доступ запрещен
7       NULL, // атрибуты безопасности по умолчанию
8       CREATE_NEW, // создать новый файл
9       dwFlagsAndAttributes, NULL);
10
11  if (INVALID_HANDLE_VALUE != hFile)
12  {
13      /* файл был успешно создан */
14  } // if
```

Закрытие файлов

Каждый открытый файл по окончании работы с ним должен быть закрыт вызовом функции `CloseHandle`. Закрытие файла сопровождается уменьшением на 1 счетчика объекта ядра «файл», что делает возможным (при необходимости) удаление временных файлов, когда этот счетчик обнулится.

Следует отметить, что при завершении процесса гарантировано, закрываются все открытые файлы. Однако все же лучше закрывать открытые файлы самостоятельно.

Операции ввода/вывода

При работе с файлом, для чтения данных используется функция `ReadFile`, а для записи – `WriteFile`:

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped);

BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped);
```

Первый параметр, *hFile*, – дескриптор открытого файла. Этот дескриптор должен обладать правами GENERIC_READ и/или GENERIC_WRITE.

Второй параметр, *lpBuffer*, указывает на буфер для хранения прочитанных данных либо данных, подлежащих записи.

Параметры *nNumberOfBytesToRead* и *nNumberOfBytesToWrite* сообщают функциям ReadFile и WriteFile, соответственно, сколько байт следует прочитать из файла либо записать в него.

Параметр *lpNumberOfBytesRead* (*lpNumberOfBytesWritten*) указывает адрес переменной типа DWORD, в которую будет сохранено число байт, считаных из файла либо записанных в него.

Последний параметр, *lpOverlapped*, указывает на структуру OVERLAPPED, которая используется для асинхронных операций чтения и записи. Для синхронных операций чтения и записи этот параметр должен быть установлен в NULL.

При успешном завершении функции ReadFile и WriteFile возвращают TRUE, иначе – FALSE.

Для повышения быстродействия операционная система буферизирует данные при чтении и записи. При буферизации данных может использоваться упреждающее чтение, то есть чтение из файла большего количества данных, чем реально требуется программе. Например, при последовательном чтении из файла следующей порции данных, эти данные, скорее всего, будут уже загружены в оперативную память. В итоге удастся сократить число обращений к диску, что сильно повышает скорость работы с файлом. Однако если программе не потребуются следующие порции данных из файла, оперативная память будет потрачена зря.

При работе с очень большими файлами может не хватить памяти для буферизации данных, в результате не удастся открыть такой файл. Поэтому при работе с очень большими файлами следует отключить буферизацию данных.

Если при записи данных используется буферизация, то непосредственная запись данных в файл будет выполнена только после закрытия этого файла. Чтобы заставить операционную систему записать данные в файл, можно воспользоваться функцией FlushFileBuffers:

```
BOOL FlushFileBuffers(HANDLE hFile);
```

Параметр *hFile* – дескриптор открытого файла. Этот дескриптор должен обладать правом доступа GENERIC_WRITE. При успешном завершении функция FlushFileBuffers возвращает значение отличное от FALSE.

Асинхронные операции ввода/вывода

Операциям ввода и вывода свойственная медленная скорость выполнения. Поэтому поток, который вызывает функции `ReadFile` и `WriteFile`, приостанавливается до завершения операции ввода/вывода. При этом страдает производительность, поскольку в данном состоянии поток простаивает. Чтобы потоки не простаивали можно использовать асинхронные операции ввода/вывода.

Первое, что необходимо сделать – это создать или открыть файл вызовом функции `CreateFile` с флагом `FILE_FLAG_OVERLAPPED` в составе параметра *`dwFlagsAndAttributes`*. Следует отметить, что асинхронные операции ввода/вывода для каждого файла выполняются по очереди.

Для выполнения асинхронных операций ввода/вывода в функции `ReadFile` и `WriteFile` необходимо в параметре *`lpOverlapped`* передавать адрес структуры `OVERLAPPED`, которая определена следующим образом:

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal; // код ошибки
  ULONG_PTR InternalHigh; // число байт
  DWORD Offset; // смещение в файле (младшие 32 бита)
  DWORD OffsetHigh; // смещение в файле (старшие 32 бита)
  HANDLE hEvent; // дескриптор объекта ядра «событие»
} OVERLAPPED, *LPOVERLAPPED;
```

Первые два поля (*`Internal`* и *`InternalHigh`*) этой структуры заполняют функции `ReadFile` и `WriteFile` по завершении асинхронной операции ввода/вывода. В поле *`Internal`* сохраняется код ошибки. Если операция ввода/вывода завершается успешно поле *`Internal`* содержит значение равное `ERROR_SUCCESS`. В поле *`InternalHigh`* сохраняется число байт, считаных из файла или записанных в него.

Следующие три поля структуры `OVERLAPPED` должны быть инициализированы перед вызовом функции `ReadFile` или `WriteFile`. Поля *`Offset`* и *`OffsetHigh`* определяют позицию в файле, с которой начнется операция ввода/вывода. Пара 32-разрядных значений, которые хранятся в этих полях, образует одно 64-разрядное значение, которое представляет собой смещение относительно начала файла. Поле *`hEvent`* содержит дескриптор объекта ядра «событие», который создается вызовом функции `CreateEvent` или `CreateEventEx`. Функции `ReadFile` и `WriteFile` переводят объекта ядра «событие» в занятое состояние. По завершении асинхронной операции ввод/вывода этот объект ядра переходит в свободное состояние.

В листингах 4.2,4.3 показаны две функции, которые начинают соответственно операции асинхронного чтения и записи.

Листинг 4.2. Функция асинхронного чтения данных из файла

```
1   BOOL ReadAsync(HANDLE hFile, LPVOID lpBuffer, DWORD dwOffset,
    DWORD dwSize, LPOVERLAPPED lpOverlapped)
2   {
3       // инициализируем структуру OVERLAPPED ...
4
5       ZeroMemory(lpOverlapped, sizeof(OVERLAPPED));
6
7       lpOverlapped->Offset = dwOffset;
8       lpOverlapped->hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
9
10      // начинаем асинхронную операцию чтения данных из файла
11      BOOL bRet = ReadFile(hFile, lpBuffer, dwSize, NULL,
    lpOverlapped);
12
13      if (FALSE == bRet && ERROR_IO_PENDING != GetLastError())
14      {
15          CloseHandle(lpOverlapped->hEvent), lpOverlapped->hEvent =
    NULL;
16          return FALSE;
17      } // if
18
19      return TRUE;
20  } // ReadAsync
```

Листинг 4.3. Функция асинхронной записи данных в файл

```
1   BOOL WriteAsync(HANDLE hFile, LPCVOID lpBuffer, DWORD dwOffset,
    DWORD dwSize, LPOVERLAPPED lpOverlapped)
2   {
3       // инициализируем структуру OVERLAPPED ...
4
5       ZeroMemory(lpOverlapped, sizeof(OVERLAPPED));
6
7       lpOverlapped->Offset = dwOffset;
8       lpOverlapped->hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
9
10      // начинаем асинхронную операцию записи данных в файл
11      BOOL bRet = WriteFile(hFile, lpBuffer, dwSize, NULL,
    lpOverlapped);
12
13      if (FALSE == bRet && ERROR_IO_PENDING != GetLastError())
14      {
15          CloseHandle(lpOverlapped->hEvent), lpOverlapped->hEvent =
    NULL;
16          return FALSE;
```

```
17      } // if
18
19      return TRUE;
20  } // WriteAsync
```

Поток может узнать, завершена ли операция ввода/вывода, вызвав функцию `WaitForSingleObject` (или другую функцию ожидания). В листингах 4.4,4.5 приводятся две функции для проверки завершения асинхронной операции ввода/вывода. Функция `FinishIo` ожидает завершения операции ввода/вывода, а функция `TryFinishIo` выполняет проверку, была ли завершена операция ввода/вывода. Если операция ввода/вывода завершена, эти функции возвращают `TRUE`; в противном случае – `FALSE`.

Листинг 4.4. Функция, ожидающая завершения операции ввода/вывода

```
1   BOOL FinishIo(LPOVERLAPPED lpOverlapped)
2   {
3       if (NULL != lpOverlapped->hEvent)
4       {
5           // ожидаем завершения операции ввода/вывода
6           DWORD dwResult = WaitForSingleObject(lpOverlapped->
    hEvent, INFINITE);
7
8           if (WAIT_OBJECT_0 == dwResult) // операция завершена
9           {
10              CloseHandle(lpOverlapped->hEvent), lpOverlapped->
    hEvent = NULL;
11              return TRUE;
12          } // if
13      } // if
14
15      return FALSE;
16  } // FinishIo
```

Листинг 4.5. Функция проверки завершения операции ввода/вывода

```
1   BOOL TryFinishIo(LPOVERLAPPED lpOverlapped)
2   {
3       if (NULL != lpOverlapped->hEvent)
4       {
5           // определяем состояние операции ввода/вывода
6           DWORD dwResult = WaitForSingleObject(lpOverlapped->
    hEvent, 0);
7
8           if (WAIT_OBJECT_0 == dwResult) // операция завершена
9           {
```

```
10              CloseHandle(lpOverlapped->hEvent), lpOverlapped->
    hEvent = NULL;
11              return TRUE;
12          } // if
13      } // if
14
15      return FALSE;
16  } // TryFinishIo
```

Для организации асинхронных операций ввода/вывода в оконных приложениях Win32, функцию `TryFinishIo` можно использовать в функции `OnIdle`, которая вызывается в цикле обработки оконных сообщений, пока в очереди нет сообщений (см. листинг 2.17 из лабораторной работы №2). Например, как показано в следующем примере.

```
1   void OnIdle(HWND hwnd)
2   {
3       if (TryFinishIo(&oRead) != FALSE)
4       {
5           if (ERROR_SUCCESS == oRead.Internal)
6           {
7               /* чтение данных из файла завершилось успешно */
8           } // if
9       } // if
10
11      if (TryFinishIo(&oWrite) != FALSE)
12      {
13          if (ERROR_SUCCESS == oWrite.Internal)
14          {
15              /* запись данных в файл завершилась успешно */
16          } // if
17      } // if
18  } // OnIdle
```

Здесь `oRead` и `oWrite` – глобальные переменные `OVERLAPPED`, которые использовались соответственно при вызове функций `ReadAsync` и `WriteAsync`.

Иногда требуется отменить ожидающие выполнения асинхронные операции ввода/вывода для определенного файла. Это можно сделать с помощью функции `CancelIoEx`:

```
BOOL CancelIoEx(HANDLE hFile, LPOVERLAPPED lpOverlapped);
```

Первый параметр, *`hFile`*, – дескриптор открытого файла, для которого необходимо отменить операцию ввода/вывода. Второй параметр, *`lpOverlapped`*, указывает на структуру `OVERLAPPED`, которая соответ-

ствует незавершенной операции ввода/вывода. Если же параметр *lpOverlapped* установлен в NULL, отменяются все незавершенные операции ввода/вывода для указанного файла. В случае успеха функция CancelIoEx возвращает значение отличное от FALSE.

Кроме того отменить все незавершенные асинхронные операции ввода/вывода файла можно закрыв его.

Текстовые файлы Unicode

Операционная система Windows никогда не станет преобразовывать данные внутри файлов (как это происходит при вызове функций Win32 API, в которых используются ANSI-строки). Решение о том, в какой кодировке хранить данные (в ANSI или Unicode) принимается самим приложением.

Для определения формата текстового файла в кодировке Unicode в начало этого файла записывается *сигнатура* – значение 0xFEFF, также именуемое *маркером последовательности байтов* (Byte Order Mark, BOM). Это позволяет различать текстовые файлы ANSI и Unicode, поскольку символа с кодом 0xFEFF не существует.

Указатели файлов

В каждом объекте ядра «файл» содержится так называемый *указатель файла* (file pointer), определяющий смещение внутри файла, по которому будет выполнена следующая операция чтения или записи. Изначально этот указатель установлен на ноль. Например, если вызвать функцию ReadFile сразу после функции CreateFile, файл будет прочитан с начала. Если при этом было считано 10 байт, указатель файла будет перемещен на 10 байт.

Для изменения указателя файла можно также использовать функцию SetFilePointerEx:

```
BOOL SetFilePointerEx(HANDLE hFile,
    LARGE_INTEGER liDistanceToMove,
    PLARGE_INTEGER lpNewFilePointer, DWORD dwMoveMethod);
```

Первый параметр, *hFile*, – дескриптор открытого файла, указатель которого нужно изменить.

Второй параметр, *liDistanceToMove*, задает число байт, на которое следует сместить указатель файла. Для задания этого параметра допускаются как положительные, так и отрицательные значения.

Третий параметр, *lpNewFilePointer*, указывает на переменную, в которую будет записано новое значение указателя файла. Этот параметр может быть установлен в NULL.

Последний параметр, *dwMoveMethod*, определяет начальную позицию смещения указателя. Этот параметр может принимать одно из значений, перечисленных в табл. 4.6.

При успешном завершении функция SetFilePointerEx возвращает значение отличное от FALSE.

Таблица 4.6. Значения параметра *dwMoveMethod*

Значение	Описание
FILE_BEGIN	Указатель файла перемещается относительно начала файла
FILE_CURRENT	Указатель файла перемещается относительно текущей позиции
FILE_END	Указатель файла перемещается относительно конца файла

Нужно отметить, что если с объектом ядра «файл» связано сразу несколько дескрипторов, положение указателя файла будет изменяться независимо от того, какой из этих дескрипторов используется.

Установка конца файла

Как правило, конец файла устанавливает автоматически при закрытии файла. Однако могут возникать ситуации, когда нужно принудительно сделать файл больше или меньше. Для этого используют функцию SetEndOfFile, которая уменьшает или увеличивает файл до размера, заданного текущим положением указателя файла.

Функция SetEndOfFile имеет следующий прототип:

```
BOOL SetEndOfFile(HANDLE hFile);
```

Параметр *hFile* – дескриптор открытого файла. Этот дескриптор должен обладать правом доступа GENERIC_WRITE. Если выполнение функции SetEndOfFile завершается успешно, возвращаемое значение отлично от FALSE.

В следующем примере приводится функция, которая изменяет размер файла с помощью функции SetEndOfFile.

Листинг 4.6. Функция, изменяющая размер файла

```
1  BOOL ResizeFile(LPCTSTR lpszFileName, LARGE_INTEGER liFileSize)
2  {
3      // открываем существующий файл
4      HANDLE hFile = CreateFile(lpszFileName,
5          GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
6
7      if (INVALID_HANDLE_VALUE != hFile)
8      {
9          // изменяем положение указателя файла
```

```
10          BOOL bRet = SetFilePointerEx(hFile, liFileSize, NULL,
    FILE_BEGIN);
11
12          // устанавливаем конец файла
13          if (FALSE != bRet)
14              bRet = SetEndOfFile(hFile);
15
16          CloseHandle(hFile); // закрываем файл
17          return bRet;
18      } // if
19
20      return FALSE;
21  } // ResizeFile
```

Например, чтобы назначить файлу размер 1 Кбайт, можно вызвать функцию из примера в листинге 4.4 следующим образом:

```
1   LARGE_INTEGER liFileSize = { 1024 }; // 1 Кбайт
2   ResizeFile(TEXT("C:\\ProgramData\\temp.txt"), liFileSize);
```

Удаление файлов

Удаление файла осуществляется при помощи функции `DeleteFile`:

```
BOOL DeleteFile(LPCTSTR lpFileName);
```

Параметр *`lpFileName`* указывает на строку, содержащую имя удаляемого файла. Если функция `DeleteFile` завершается успешно, возвращаемое значение отлично от `FALSE`.

Если файл уже открыт с доступом для удаления, то функция `DeleteFile` помечает этот файл как удаленный. После того как все дескрипторы этого файла будут закрыты, он будет удален. Если же файл открыт, но удаление не разрешено, функция `DeleteFile` завершается ошибкой.

Копирование файлов

Копирование файла выполняется с помощью функции `CopyFile`:

```
BOOL CopyFile(LPCTSTR lpExistingFileName,
    LPCTSTR lpNewFileName, BOOL bFailIfExists);
```

Первый параметр, *`lpExistingFileName`*, указывает на строку, содержащую имя существующего файла, который нужно скопировать. Второй параметр, *`lpNewFileName`*, указывает на строку, содержащую имя нового файла.

Последний параметр, *`bFailIfExists`*, определяет поведение функции `CopyFile` в случае существования файла с тем же именем, что в

параметре *lpNewFileName*. Если параметр *bFailIfExists* принимает значение FALSE, этот файл будет заменен новым файлом. Если же этот параметр принимает значение TRUE, функция завершится ошибкой.

При успешном завершении функция CopyFile возвращает значение отличное от FALSE.

Работа с каталогами

В этом разделе рассматриваются основные функции Win32 API, предназначенные для работы с каталогами.

Создание каталогов

Создание каталога осуществляется при помощи одной из двух функций – CreateDirectory и CreateDirectoryEx:

```
BOOL CreateDirectory(LPCTSTR lpPathName,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);

BOOL CreateDirectoryEx(LPCTSTR lpTemplateDirectory,
    LPCTSTR lpNewDirectory,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Параметр *lpPathName* (*lpNewDirectory*) указывает на строку, содержащую имя создаваемого каталога.

Параметр *lpSecurityAttributes* позволяет задать атрибуты безопасности для создаваемого каталога. Если этот параметр установлен в NULL, каталог будет создан с атрибутами безопасности по умолчанию, при этом дескриптор безопасности для каталога наследуется от родительского каталога.

Параметр *lpTemplateDirectory* указывает на строку, которая определяет каталог, используемый в качестве шаблона для атрибутов при создании нового каталога. Если этот параметр установлен в NULL, каталог будет создан с атрибутами по умолчанию.

Если функция CreateDirectory(Ex) завершается успешно, она возвращает значение отличное от FALSE.

Открытие и закрытие каталогов

Функция CreateFile кроме создания и открытия файлов позволяет открывать каталоги и логические диски. Открыв каталог или логический диск можно определить или изменить его атрибуты.

Каждый открытый каталог и логический диск по окончании работы с ним должен быть закрыт вызовом функции CloseHandle.

Для открытия каталога при вызове функции CreateFile нужно указать имя каталога и установить флаг FILE_FLAG_BACKUP_SEMANTICS следующим образом:

```
HANDLE hDirectory = CreateFile(TEXT("C:\\ProgramData\\Temp"),
    0, 0, NULL, OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS, NULL);

if (INVALID_HANDLE_VALUE != hDirectory)
{
    /* каталог успешно открыт */

    CloseHandle(hDirectory); // закрываем каталог
} // if
```

Для открытия логического диска при вызове функции `CreateFile` нужно указать имя в формате «\\.\X:», где «X» – буква диска. Например, открыть диск C можно следующим образом:

```
HANDLE hDevice = CreateFile(TEXT("\\\\.\\C:"), 0, 0, NULL,
OPEN_EXISTING, 0, NULL);

if (INVALID_HANDLE_VALUE != hDevice)
{
    /* логический диск успешно открыт */

    CloseHandle(hDevice); // закрываем логический диск
} // if
```

Удаление каталогов

Для удаления каталога используется функция `RemoveDirectory`:

```
BOOL RemoveDirectory(LPCTSTR lpPathName);
```

Параметр *lpPathName* указывает на строку, содержащую путь к удаляемому каталогу. Если функция `RemoveDirectory` завершается успешно, она возвращает значение отличное от `FALSE`.

Текущий каталог

Текущим каталогом (current directory) называется рабочий каталог процесса, который используется для нахождения файлов и каталогов, указанных либо только по имени, либо по относительному пути.

При создании процесса функция `CreateProcess` позволяет установить рабочий каталог для нового процесса. Однако, в большинстве случаев, новый процесс наследует рабочий каталог родительского процесса.

Изменить рабочий каталог для текущего процесса можно с помощью функции `SetCurrentDirectory`:

```
BOOL SetCurrentDirectory(LPCTSTR lpPathName);
```

Параметр *lpPathName* указывает на строку, содержащую путь к новому текущему каталогу. Последний символ в этой строке должен быть «\». Если это не так, этот символ будет добавлен автоматически.

Если функция SetCurrentDirectory завершается успешно, возвращаемое значение отлично от FALSE.

Чтобы определить рабочий каталог текущего процесса следует использовать функцию GetCurrentDirectory:

```
DWORD GetCurrentDirectory(DWORD nBufferLength,
     LPTSTR lpBuffer);
```

Первый параметр, *nBufferLength*, задает размер буфера (в символах), а второй параметр, *lpBuffer*, указывает на буфер, в который будет записан путь к текущему каталогу.

Если функция GetCurrentDirectory выполняется успешно, она возвращает число символов, записанных в буфер (нуль-символ не учитывается); при ошибке возвращаемое значение равно нулю.

Если путь к текущему каталогу не поместился полностью в буфере, то функция GetCurrentDirectory возвращает необходимый размер буфера (в символах), включая нуль-символ в конце строки. Поэтому для того, чтобы определить требуемый размер буфера, параметр *nBufferLength* должен принимать значение равное нулю.

Перемещение файлов и каталогов

Перемещение файла или каталога осуществляется при помощи одной из двух функций – MoveFile и MoveFileEx:

```
BOOL MoveFile(LPCTSTR lpExistingFileName,
    LPCTSTR lpNewFileName);

BOOL MoveFileEx(LPCTSTR lpExistingFileName,
    LPCTSTR lpNewFileName, DWORD dwFlags);
```

Первый параметр, *lpExistingFileName*, указывает на строку, содержащую имя существующего файла или каталога, который нужно переместить. Второй параметр, *lpNewFileName*, указывает на строку, содержащую имя нового файла или каталога.

Параметр *dwFlags* функции MoveFileEx определяет флаги, задающие поведение этой функции. Флаги, которые можно задавать в этом параметре, перечислены в табл. 4.7.

При успешном завершении функция MoveFile(Ex) возвращает значение отлично от FALSE.

Следует отметить, что если параметр *lpNewFileName* установлен в NULL и задан флаг MOVEFILE_DELAY_UNTIL_REBOOT, то файл, на который указывает параметр *lpExistingFileName*, будет удален после переза-

грузки операционной системы. Если при этом параметр *lpExisting-FileName* указывает на каталог, то этот каталог будет удален после перезагрузки только, если он пустой.

Таблица 4.7. Флаги (значения параметра *dwFlags*)

Флаг	Описание
MOVEFILE_COPY_ALLOWED	Этот флаг должен быть установлен, если необходимо переместить файл на другой диск. При этом перемещение осуществляется путем последовательного вызова функций CopyFile и DeleteFile, поэтому этот флаг нельзя применять для каталогов
MOVEFILE_DELAY_UNTIL_REBOOT	Если этот флаг установлен, фактическое перемещение файла или каталога будет осуществлено только после перезагрузки операционной системы. Этот флаг нельзя использовать совместно с MOVEFILE_COPY_ALLOWED
MOVEFILE_REPLACE_EXISTING	Разрешает замену существующего файла. Этот флаг нельзя применять для каталогов
MOVEFILE_WRITE_THROUGH	Если этот флаг установлен, функция вернет управление только после того, как файл или каталог будет фактически перемещен. Этот флаг игнорируется, если был установлен флаг MOVEFILE_DELAY_UNTIL_REBOOT

Функции MoveFile и MoveFileEx также можно применять для переименования файла или каталога. В этом случае параметры *lpExisting-FileName* и *lpNewFileName* должны содержать одинаковый путь.

Атрибуты файлов и каталогов

После создания файла или каталога можно изменять его атрибуты с помощью функции SetFileAttributes:

```
BOOL SetFileAttributes(LPCTSTR lpFileName,
    DWORD dwFileAttributes);
```

Первый параметр, *lpFileName*, указывает на строку, содержащую имя файла или каталога, а второй параметр, *dwFileAttributes*, задает новые атрибуты (см. табл. 4.5). При успешном завершении функция SetFileAttributes возвращает значение отличное от FALSE.

При работе с файлами или каталогами часто возникает необходимость получения их атрибутов. Получить атрибуты файла и каталога можно при помощи функции GetFileAttributes:

```
DWORD GetFileAttributes(LPCTSTR lpFileName);
```

Параметр *lpFileName* указывает на строку, содержащую имя файла или каталога. Если функция GetFileAttributes завершается ошибкой, она возвращает значение INVALID_FILE_ATTRIBUTES.

Например, определить является ли файл скрытым или нет, можно следующим образом:

```
1  DWORD dwAttr = GetFileAttributes(TEXT("C:\\Windows\\win.ini"));
2
3  if (INVALID_FILE_ATTRIBUTES != dwAttr && (dwAttr &
   FILE_ATTRIBUTE_HIDDEN))
4  {
5      /* файл является скрытым */
6  } // if
```

Следует отметить, что каталоги, помимо обычных атрибутов (см. табл. 4.5), имеют еще атрибут FILE_ATTRIBUTE_DIRECTORY.

Атрибуты файла и каталога включают в себя также дополнительную информацию, например дату и время создания, чтения и изменения, а также размер. Для получения дополнительной атрибутивной информации следует использовать функцию GetFileAttributesEx:

```
BOOL GetFileAttributesEx(LPCTSTR lpFileName,
    GET_FILEEX_INFO_LEVELS fInfoLevelId, LPVOID
    lpFileInformation);
```

Первый параметр, *lpFileName*, указывает на строку, содержащую имя файла или каталога.

Второй параметр, *fInfoLevelId*, определяет класс получаемой атрибутивной информации. Этот параметр должен принимать значение GetFileExInfoStandard.

Третий параметр, *lpFileInformation*, должен указывать на структуру WIN32_FILE_ATTRIBUTE_DATA, в которую будет записана полученная атрибутивная информация.

При успешном завершении функция GetFileAttributesEx возвращает значение отличное от FALSE.

Структура WIN32_FILE_ATTRIBUTE_DATA имеет следующее описание:

```
typedef struct _WIN32_FILE_ATTRIBUTE_DATA {
  DWORD dwFileAttributes; // атрибуты
  FILETIME ftCreationTime; // дата и время создания
  FILETIME ftLastAccessTime; // дата и время открытия
  FILETIME ftLastWriteTime; // дата и время изменения
  DWORD nFileSizeHigh; // размер (старшие 32 бита)
  DWORD nFileSizeLow; // размер (младшие 32 бита)
} WIN32_FILE_ATTRIBUTE_DATA, *LPWIN32_FILE_ATTRIBUTE_DATA;
```

Первое поле, *dwFileAttributes*, содержит рассмотренные ранее атрибуты файла или каталога.

Следующие три поля содержат различные даты и время в формате UTC. Поле *ftCreationTime* представляет собой дату и время создания файла или каталога, поле *ftLastAccessTime* – дату и время последнего обращения к нему, а поле *ftLastWriteTime* содержит дату и время его последнего изменения.

Последние два поля, *nFileSizeHigh* и *nFileSizeLow*, содержат размер файла (для каталога эти два поля принимают значения равные нулю). Пара 32-разрядных значений, которые хранятся в этих полях, образует одно 64-разрядное значение, которое представляет собой размер файла. Использование 64-разрядных значений для представления размера позволяет работать с очень большими файлами, размер которых теоретически может достигать 16 Эбайт. Однако чаще всего приходится работать с файлами, размер которых не превышает 4 Гбайт. В этом случае поле *nFileSizeHigh* будет нулевым.

Заметим, что поля *ftCreationTime*, *ftLastAccessTime* и *ftLastWriteTime* хранят дату и время в структуре FILETIME, которая хранит дату и время в виде 64-разрядной величины, содержащей количество 100 наносекундных интервалов с 1 января 1601 г. Структура FILETIME определена следующим образом:

```
typedef struct _FILETIME {
    DWORD dwLowDateTime; // младшее 32-разрядное значение
    DWORD dwHighDateTime; // старшее 32-разрядное значение
} FILETIME, *PFILETIME, *LPFILETIME;
```

Из определения структуры FILETIME можно сделать вывод, что она совершенно не подходит для работы с датой и временем в отличие от структуры SYSTEMTIME.

Для преобразования из FILETIME в SYSTEMTIME и наоборот применяются функции FileTimeToSystemTime и SystemTimeToFileTime:

```
BOOL FileTimeToSystemTime(const FILETIME *lpFileTime,
    LPSYSTEMTIME lpSystemTime);

BOOL SystemTimeToFileTime(const SYSTEMTIME *lpSystemTime,
    LPFILETIME lpFileTime);
```

Если преобразование выполнено успешно эти функции возвращают значение отличное от FALSE.

В следующем примере приводится реализации функции, которая возвращает дату и время, хранящиеся в структуре FILETIME, в виде строки, форматируя ее в определенном формате с учетом языковых особенностей.

Листинг 4.7. Функция форматирования даты и времени

```
1   BOOL GetFileTimeFormat(const LPFILETIME lpFileTime, LPTSTR
    lpszFileTime, DWORD cchFileTime)
2   {
3       SYSTEMTIME st;
4       // преобразуем дату и время из FILETIME в SYSTEMTIME
5       BOOL bRet = FileTimeToSystemTime(lpFileTime, &st);
6
7       // приведем дату и время к текущему часовому поясу
8       if (FALSE != bRet)
9           bRet = SystemTimeToTzSpecificLocalTime(NULL, &st, &st);
10
11      if (FALSE != bRet)
12      {
13          // скопируем дату в результирующую строку
14          GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE, &st,
    NULL, lpszFileTime, cchFileTime);
15
16          // добавим время в результирующую строку
17
18          StringCchCat(lpszFileTime, cchFileTime, TEXT(", "));
19          DWORD len = _tcslen(lpszFileTime);
20
21          if (len < cchFileTime)
22              GetTimeFormat(LOCALE_USER_DEFAULT,
    TIME_FORCE24HOURFORMAT, &st, NULL, lpszFileTime+len, cchFileTime-
    len);
23      } // if
24
25      return bRet;
26  } // GetFileTimeFormat
```

Кроме уже рассмотренных функций для работы с атрибутами в Win32 API имеется еще ряд функций (табл. 4.8), которые также работают с атрибутами, но вместо имени они используют дескриптор открытого файла или каталога. Подробное описание этих функций приведено в документации Platform SDK.

Таблица 4.8. Функции для работы атрибутами файла или каталога

Функция	**Описание**
`GetCompressedFileSize`	Возвращает фактический размер сжатого файла
`GetFileInformationByHandle` `GetFileInformationByHandleEx`	Возвращает атрибуты открытого файла или каталога

Окончание табл. 4.8.

Функция	Описание
GetFileSizeEx	Возвращает размер открытого файла
GetFileTime	Возвращает время создания, обращения и изменения открытого файла или каталога
SetFileInformationByHandle	Устанавливает атрибуты открытого файла
SetFileTime	Устанавливает время создания, обращения и изменения открытого файла или каталога

Поиск файлов и каталогов

В Windows поиск файлов и каталогов выполняется по *шаблону поиска*, который может содержать специальные символы «*» и «?». Символ «*» означает ноль или более символов, а «?» – ноль или один символ. Например, шаблон поиска «D:*.???» означает, что будет выполняться поиск файлов на диске D с любым именем и расширением не более трех символов. Тогда, как шаблон поиска «D:*.txt» означает поиск только текстовых файлов с любым именем, а шаблон «D:*» – вообще всех файлов и каталогов на диске D.

Поиск файлов и каталогов по шаблону поиска начинается с вызова функции FindFirstFile:

```
HANDLE FindFirstFile(LPCTSTR lpFileName,
    LPWIN32_FIND_DATA lpFindFileData);
```

Первый параметр, *lpFileName*, указывает на строку, содержащую шаблон поиска. Второй параметр, *lpFindFileData*, указывает на структуру WIN32_FIND_DATA, в которую сохраняется результат поиска.

Структура WIN32_FIND_DATA имеет следующее описание:

```
typedef struct _WIN32_FIND_DATA {
  DWORD dwFileAttributes; // атрибуты
  FILETIME ftCreationTime; // дата и время создания
  FILETIME ftLastAccessTime; // дата и время открытия
  FILETIME ftLastWriteTime; // дата и время изменения
  DWORD nFileSizeHigh; // размер (старшие 32 бита)
  DWORD nFileSizeLow; // размер (младшие 32 бита)
  DWORD dwReserved0; // зарезервировано
  DWORD dwReserved1; // зарезервировано
  TCHAR cFileName[MAX_PATH]; // имя файла или каталога
  TCHAR cAlternateFileName[14]; // альтернативное имя файла
} WIN32_FIND_DATA, *PWIN32_FIND_DATA, *LPWIN32_FIND_DATA;
```

Первые шесть полей структуры WIN32_FIND_DATA идентичны полям структуры WIN32_FILE_ATTRIBUTE_DATA. Следующие два поля, *dwReserved0* и *dwReserved1*, зарезервированы и не используются.

Девятое поле, *cFileName*, содержит имя найденного файла или каталога. Это поле также может содержать одно из двух специальных значений – «.» или «..», которые соответствуют текущему каталогу и его родительскому каталогу.

Последнее поле, *cAlternateFileName*, содержит имя файла в устаревшем «коротком» формате «8.3», где основная часть имени состоит из восьми символов, а расширение – из трех. Для каталогов это поле не используется.

Если функция FindFirstFile завершается успешно, она возвращает дескриптор поиска. Если же функция завершается ошибкой, возвращается значение INVALID_HANDLE_VALUE.

Для того чтобы продолжить поиск следует использовать функцию FindNextFile, а для того чтобы завершить поиск – функцию FindClose.

```
BOOL FindNextFile(HANDLE hFindFile,
    LPWIN32_FIND_DATA lpFindFileData);

BOOL FindClose(HANDLE hFindFile);
```

Параметр *hFindFile* – дескриптор поиска, который вернула функция FindFirstFile, а параметр *lpFindFileData* указывает на структуру WIN32_FIND_DATA, в которую сохраняется результат поиска. При успешном выполнении, возвращаемое значение отлично от FALSE.

В листинге 4.8 приведен пример того, как организовать поиск внутри каталога с помощью приведенных функций Win32 API. Нужно отметить, что в этом примере используется указатель на функцию, которая вызывается всякий раз для возврата результата поиска. Этот указатель следует определить следующим образом:

```
typedef BOOL (__stdcall *LPSEARCHFUNC)(LPCTSTR lpszFileName,
    const LPWIN32_FILE_ATTRIBUTE_DATA lpFileAttributeData,
    LPVOID lpvParam);
```

Листинг 4.8. Функция, которая выполняет поиск внутри каталога

```
1   BOOL FileSearch(LPCTSTR lpszFileName, LPCTSTR lpszDirName,
    LPSEARCHFUNC lpSearchFunc, LPVOID lpvParam)
2   {
3       WIN32_FIND_DATA fd;
4       TCHAR szFileName[MAX_PATH];
5
6       // формируем шаблон поиска
```

```
7       StringCchPrintf(szFileName, MAX_PATH, TEXT("%s\\%s"),
    lpszDirName, lpszFileName);
8
9       // начинаем поиск
10      HANDLE hFindFile = FindFirstFile(szFileName, &fd);
11      if (INVALID_HANDLE_VALUE == hFindFile) return FALSE;
12
13      BOOL bRet = TRUE;
14
15      for (BOOL bFindNext = TRUE; FALSE != bFindNext; bFindNext =
    FindNextFile(hFindFile, &fd))
16      {
17          if (_tcscmp(fd.cFileName, TEXT(".")) == 0 ||
    _tcscmp(fd.cFileName, TEXT("..")) == 0)
18          {
19              /* пропускаем текущий и родительский каталог */
20              continue;
21          } // if
22
23          // формируем полный путь к файлу
24          StringCchPrintf(szFileName, MAX_PATH, TEXT("%s\\%s"),
    lpszDirName, fd.cFileName);
25
26          bRet = lpSearchFunc(szFileName, (LPWIN32_FILE_ATTRIBUTE_
    DATA)&fd, lpvParam);
27          if (FALSE == bRet) break; // прерываем поиск
28      } // for
29
30      FindClose(hFindFile); // завершаем поиск
31      return bRet;
32  } // FileSearch
```

Функцию `FileSearch` можно использовать не только для простого поиска, но и, например, для вычисления размера каталога:

```
1   ULARGE_INTEGER size = { 0 };
2   FileSearch(TEXT("*"), TEXT("C:\\ProgramData"), CalculateSize,
    &size);
```

Функция `CalculateSize`, которая используется для вычисления размера каталога, может быть определена таким образом:

Листинг 4.9. Функция, используемая при поиске, для вычисления размера каталога

```
1   BOOL __stdcall CalculateSize(LPCTSTR lpszFileName, const
    LPWIN32_FILE_ATTRIBUTE_DATA lpFileAttributeData, LPVOID lpvParam)
```

```
2   {
3       if (lpFileAttributeData->dwFileAttributes &
    FILE_ATTRIBUTE_DIRECTORY)
4       {
5           // продолжим поиск внутри каталога
6           return FileSearch(TEXT("*"), lpszFileName, CalculateSize,
    lpvParam);
7       } // if
8
9       // прибавим полученный размер к результату
10
11      ULARGE_INTEGER size = { lpFileAttributeData->nFileSizeLow,
    lpFileAttributeData->nFileSizeHigh };
12      ((ULARGE_INTEGER *)lpvParam)->QuadPart += size.QuadPart;
13
14      return TRUE; // возвращаем TRUE, чтобы продолжить поиск
15  } // CalculateSize
```

Файлы инициализации

Файл инициализации (initialization file) представляет собой текстовый файл с расширением .ini, который содержит записи конфигурации операционной системы и некоторых приложений. Файл инициализации можно просматривать и редактировать при помощи любого текстового редактора.

Файлы инициализации содержат разделы. Имя каждого раздела заключено в квадратные скобки «[ ]». В каждом разделе содержатся записи, которые представляют собой строки вида «ключ=значение». Также в файле инициализации могут присутствовать однострочные комментарии, которые начинаются с символа «;».

В качестве примера файла инициализации можно рассмотреть файл odbc.ini, содержимое которого выглядит следующим образом:

Листинг 4.10. Пример файла odbc.ini

```
1   ; Список источников данных
2   [ODBC Data Sources]
3   MS Access=Microsoft Access Driver (*.mdb)
4   Excel=Microsoft Excel Driver (*.xls)
5   dBASE=Microsoft dBase Driver (*.dbf)
6
7   ; База данных MS Access
8   [MS Access]
9   Driver=C:\Windows\system32\odbcjt32.dll
10
11  ; Файлы Excel
```

```
[Excel]
Driver=C:\Windows\system32\odbcjt32.dll

; Файлы dBASE
[dBASE]
Driver=C:\Windows\system32\odbcjt32.dll
```

В Win32 API имеется ряд функций для работы с файлами инициализации. Например, для того чтобы добавить новую запись в файл инициализации или изменить уже существующую запись следует использовать функцию `WritePrivateProfileString`:

```
BOOL WritePrivateProfileString(LPCTSTR lpAppName,
    LPCTSTR lpKeyName, LPCTSTR lpString, LPCTSTR lpFileName);
```

Первый параметр, *lpAppName*, указывает на строку, содержащую имя раздела. Если такого раздела не существует, он будет создан.

Второй параметр, *lpKeyName*, указывает на строку, содержащую ключ, идентифицирующий запись. Если такой ключ не существует в указанном разделе, он будет создан. Если этот параметр установлен в `NULL`, раздел и все его записи будут удалены.

Третий параметр, *lpString*, указывает на строку, содержащую значение записи. Если этот параметр установлен в `NULL`, запись, на которую указывает параметр *lpKeyName*, будет удалена.

Последний параметр, *lpFileName*, указывает на строку, содержащую имя файла инициализации.

Если функция `WritePrivateProfileString` завершается успешно, она возвращает значение отличное от `FALSE`.

В следующем примере демонстрируется изменение файла инициализации:

```
// внесем в раздел General файла app.ini
// запись Title=Sample Application
WritePrivateProfileString(TEXT("General"), TEXT("Title"),
TEXT("Sample Application"), TEXT("C:\\Program Data\\app.ini"));
```

Для того чтобы прочитать запись из файла инициализации следует использовать функцию `GetPrivateProfileString`:

```
DWORD GetPrivateProfileString(LPCTSTR lpAppName,
    LPCTSTR lpKeyName, LPCTSTR lpDefault,
    LPTSTR lpReturnedString, DWORD nSize, LPCTSTR lpFileName);
```

Первый параметр, *lpAppName*, указывает на строку, содержащую имя раздела. Если этот параметр установлен в `NULL`, в выходной буфер будут скопированы имена всех разделов в указанном файле инициали-

зации. При этом имена разделов будут разделены нуль-символами, а в конце будет сразу два нуль-символа, как показано на рис. 4.1.

szBuffer	0x003bf830 "Раздел 1"	wchar_t [100]
[0]	1056 L'Р'	wchar_t
[1]	1072 L'а'	wchar_t
[2]	1079 L'з'	wchar_t
[3]	1076 L'д'	wchar_t
[4]	1077 L'е'	wchar_t
[5]	1083 L'л'	wchar_t
[6]	32 L' '	wchar_t
[7]	49 L'1'	wchar_t
[8]	0	wchar_t
[9]	1056 L'Р'	wchar_t
[10]	1072 L'а'	wchar_t
[11]	1079 L'з'	wchar_t
[12]	1076 L'д'	wchar_t
[13]	1077 L'е'	wchar_t
[14]	1083 L'л'	wchar_t
[15]	32 L' '	wchar_t
[16]	50 L'2'	wchar_t
[17]	0	wchar_t
[18]	0	wchar_t

Рис. 4.1. Отладчик среды Visual C++

Второй параметр, *lpKeyName*, указывает на строку, содержащую ключ, идентифицирующий запись. Если этот параметр установлен в NULL, в выходной буфер будут скопированы ключи всех записей в указанном разделе. При этом ключи будут разделяться также как и в случае с именами разделов.

Третий параметр, *lpDefault*, указывает на строку, которая будет скопирована в выходной буфер, если искомая запись не найдется. Если этот параметр установлен в NULL, в выходной буфер будет скопирована пустая строка.

Четвертый параметр, *lpReturnedString*, представляет собой указатель на выходной буфер, в который сохраняется результат работы функции. Пятый параметр, *nSize*, указывает на размер (в символах) выходного буфера. Если размер выходного буфера слишком мал, возвращаемый результат обрезается.

Последний параметр, *lpFileName*, указывает на строку, содержащую имя файла инициализации.

Функции GetPrivateProfileString в случае успеха возвращает число символов, скопированных в выходной буфер (нуль-символ в конце не учитывается).

Если необходимо считать из файла инициализации целочисленное значение, следует использовать функцию GetPrivateProfileInt:

```
UINT GetPrivateProfileInt(LPCTSTR lpAppName,
    LPCTSTR lpKeyName, INT nDefault, LPCTSTR lpFileName);
```

Первый параметр, *lpAppName*, указывает на строку, содержащую имя раздела, а второй параметр, *lpKeyName*, указывает на строку, содержащую ключ, идентифицирующий запись в этом разделе.

Третий параметр, *nDefault*, определяет значение, которое вернет функция, если искомая запись не будет найдена.

Последний параметр, *lpFileName*, указывает на строку, содержащую имя файла инициализации.

Функция GetPrivateProfileInt возвращает значение найденной записи или значение, заданное параметром *nDefault*, если искомая запись не будет найдена.

Системный реестр

Системный реестр Windows (Windows registry) – иерархическая централизованная база данных, используемая для хранения данных, необходимых для правильного функционирования Windows. В системном реестре хранятся такие данные, как профили пользователей, сведения об установленном программном обеспечении и типах файлов, а также информация об используемом оборудовании.

По замыслу Microsoft, системный реестр является заменой большинству файлов инициализации, которые использовались в Windows 3.1, а также файлам конфигурации MS-DOS, таким как Autoexec.bat и Config.sys. Следует отметить, что системный реестр для разных версий операционных систем семейства Windows имеет различия.

Структура системного реестра

Системный реестр имеет иерархическую структуру, состоящую из различных разделов, которые называют *ключами* (keys). Каждый ключ содержит *параметры* (values) и может содержать *вложенные ключи* (sub keys). Ключи, находящиеся на верхнем уровне этой иерархической структуры, называют *корневыми ключами* (root keys). Системный реестр Windows (начиная с Windows XP) состоит из шести корневых ключей (добавить к ним новые ключи или удалить существующие невозможно), описанных в табл. 4.9.

Параметры, расположенные в ключах реестра, содержат различные данные. Каждый параметр характеризуются именем, типом данных и значением. В табл. 4.10 перечислены основные типы данных, определенные и используемые в системном реестре.

Таблица 4.9. Корневые ключи системного реестра

Имя корневого ключа	Описание
HKEY_CURRENT_USER	Данный раздел содержит данные конфигурации пользователя, вошедшего в операционную систему. Вместо имени раздела иногда используется аббревиатура HKCU. На самом деле, данный раздел является ссылкой на один из вложенных разделов из HKEY_USERS
HKEY_USERS	Данный раздел содержит профили всех пользователей операционной системы. Вместо имени раздела иногда используется аббревиатура HKU
HKEY_LOCAL_MACHINE	Данный раздел содержит параметры конфигурации, относящиеся к данному компьютеру (для всех пользователей). Вместо имени раздела иногда используется аббревиатура HKLM
HKEY_CLASSES_ROOT	Данный раздел содержит ассоциации между приложениями и типами файлов (по расширениям файлов). Кроме того в этом разделе содержится информация, связанная с объектами COM. Вместо имени раздела иногда используется аббревиатура HKCR. Данный раздел является ссылкой на раздел HKEY_LOCAL_MACHINE\SOFTWARE\Classes
HKEY_CURRENT_CONFIG	Данный раздел содержит сведения об оборудовании, используемом компьютером при запуске операционной системы. Вместо имени раздела иногда используется аббревиатура HKCC. Данный раздел является ссылкой на раздел HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current
HKEY_PERFORMANCE_DATA	Данный раздел содержит показания счетчиков производительности. Вместо имени раздела иногда используется аббревиатура HKPD. Данный раздел не виден в редакторе реестра, он доступен только программным способом, через имеющиеся функции Win32 API

Таблица 4.10. Основные типы данных параметров системного реестра

Тип данных	Описание
REG_BINARY	Двоичный параметр. Значение этого параметра представляет собой последовательность байтов
REG_DWORD	Параметр DWORD. Значение этого параметра представляет собой 32-разрядное целое число без знака

Окончание табл. 4.10.

Тип данных	Описание
REG_EXPAND_SZ	Расширяемый строковый параметр. Значение этого параметра представляет собой Unicode-строку переменной длины
REG_MULTI_SZ	Многострочный параметр. Значение этого параметра представляет собой список Unicode-строк, разделенных нуль-символами
REG_QWORD	Параметр QWORD. Значение этого параметра представляет собой 64-разрядное целое число без знака
REG_SZ	Строковый параметр. Значение этого параметра представляет собой Unicode-строку фиксированной длины

Файлы системного реестра

При описании системного реестра, среди прочих, используется термин *куст реестра* (registry hive) – это подмножество ключей и параметров реестра, с которым ассоциирован определенный набор файлов. В табл. 4.11 перечислены кусты системного реестра Windows (начиная с Windows XP) и поддерживающие их файлы, которые бывают следующих типов:

- файлы без расширения или с расширением .dat содержат данные соответствующего куста реестра;
- файлы с расширением .log содержат журнал транзакций, в котором регистрируются все изменения, которые вносились в ключи соответствующего куста реестра;
- файлы с расширением .sav содержат копию данных соответствующего куста реестра;
- файл с расширением .alt содержит резервную копию куста HKEY_LOCAL_MACHINE\SYSTEM.

Таблица 4.11. Файлы, обеспечивающие поддержку кустов реестра

Куст реестра	Файлы
HKEY_CURRENT_USER	Ntuser.dat, Ntuser.dat.log
HKEY_USERS\.DEFAULT	Default, Default.log, Default.sav
HKEY_LOCAL_MACHINE\SAM	Sam, Sam.log, Sam.sav
HKEY_LOCAL_MACHINE\SECURITY	Security, Security.log, Security.sav
HKEY_LOCAL_MACHINE\SOFTWARE	Software, Software.log, Software.sav
HKEY_LOCAL_MACHINE\SYSTEM	System, System.alt, System.log, System.sav

Все файлы кустов реестра, за исключением HKEY_CURRENT_USER, находятся в каталоге %WINDIR%\system32\config. Файлы куста HKEY_CURRENT_USER находятся в каталоге %USERPROFILE%.

Нужно отметить, что не все разделы реестра хранятся в файлах кустов реестра. Например, раздел HKEY_LOCAL_MACHINE\HARDWARE строится динамически при запуске операционной системы на основании данных, хранящихся в файлах различных кустов реестра.

Редактор реестра

Для просмотра и редактирования содержимого системного реестра в Windows предусмотрена специальная программа – regedit.exe, которая находится в каталоге %WINDIR%.

При запуске программы regedit.exe открывается окно «Редактор реестра» (Register Editor). Как показано на рис. 4.2 в левой области этого окна отображаются ключи реестра, ответвляющиеся от элемента «Компьютер» (My Computer), а в правой области – параметры реестра.

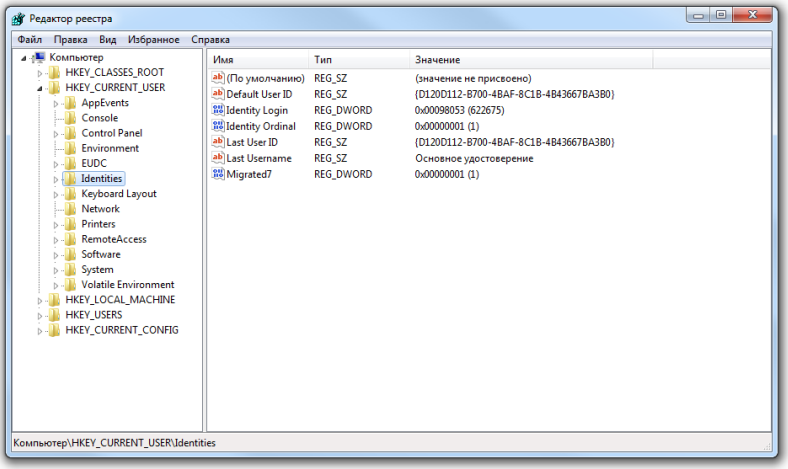

Рис. 4.2. Окно редактора реестра Windows

Создание и удаление ключей

Для создания нового ключа реестра необходимо выполнить следующие действия:

1. Выберите раздел, в котором необходимо создать новый ключ.
2. В меню **Правка** (**Edit**) выберите **Создать** (**New**) → **Раздел** (**Key**).
3. Введите имя нового ключа и нажмите клавишу «Enter».

Для удаления ключа реестра необходимо выполнить следующие действия:

1. Выберите ключ, который необходимо удалить.
2. В меню **Правка** (**Edit**) выберите **Удалить** (**Delete**).
3. В появившемся диалоговом окне (рис. 4.3) подтвердите удаление, нажав кнопку **Да** (**Yes**).

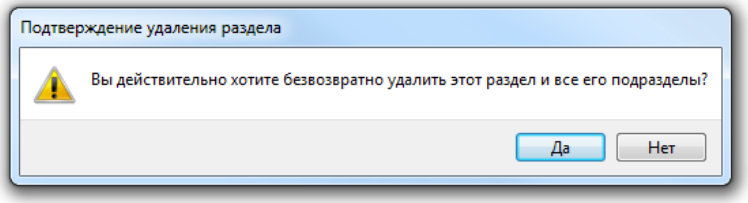

Рис. 4.3. Подтверждение удаления ключа реестра

Создание, изменение и удаление параметров

Для создания нового параметра в ключе реестра необходимо выполнить следующие действия:

1. Выберите раздел, в котором необходимо создать параметр.
2. В меню **Правка** (**Edit**) выберите **Создать** (**New**), а затем один из следующих пунктов:
   - Строковый параметр (String Value);
   - Двоичный параметр (Binary Value);
   - Параметр DWORD (DWORD Value);
   - Параметр QWORD (QWORD Value);
   - Мультистроковый параметр (Multi-String Value);
   - Расширяемый строковый параметр (Expandable String Value);
3. Введите имя нового параметра и нажмите клавишу «Enter».

Для изменения параметра в ключе реестра необходимо выполнить следующие действия:

1. Выберите раздел, в котором необходимо изменить параметр.
2. Выберите параметр, который необходимо изменить.
3. В меню **Правка** (**Edit**) выберите **Изменить** (**Modify**).
4. В появившемся диалоговом окне измените значение выбранного параметра.

Следует отметить, что для параметров различных типов предусмотрены различные диалоговые окна для их изменения. Например, на рис. 4.4 показано окно для изменения параметра DWORD.

5. Нажмите кнопку **OK**, чтобы сохранить изменения.

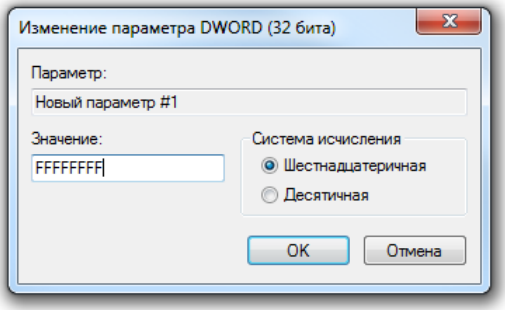

Рис. 4.4. Окно изменения параметра DWORD в редакторе реестра

Для удаления параметра в ключе реестра необходимо выполнить следующие действия:

1. Выберите раздел, в котором необходимо удалить параметр.
2. Выберите параметр, который необходимо удалить.
3. В меню **Правка** (**Edit**) выберите **Удалить** (**Delete**).
4. В появившемся диалоговом окне (рис. 4.5) подтвердите удаление, нажав кнопку **Да** (**Yes**).

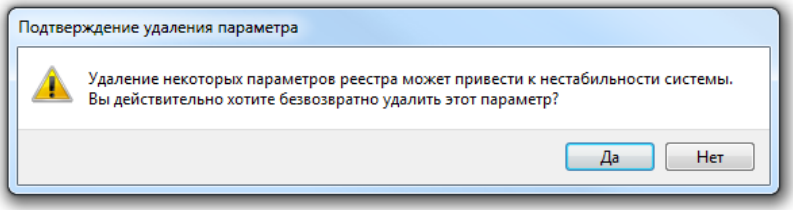

Рис. 4.5. Подтверждение удаления параметра ключа реестра

Работа с системным реестром

В этом разделе рассматриваются основные функции Win32 API, предназначенные для работы с системным реестром Windows. Следует отметить, что в функциях Win32 API, работающих с реестром, для указания ключей используются дескрипторы типа `HKEY`.

Создание и открытие ключей

Для создания ключа системного реестра необходимо использовать функцию RegCreateKeyEx. Если ключ уже существует, то эта функция открывает его без изменений. Вызов функции RegCreateKeyEx заставляет операционную систему создать объект ядра «ключ реестра», который используется для работы с соответствующим разделом реестра.

Функция RegCreateKeyEx имеет следующий прототип:

```
LSTATUS RegCreateKeyEx(HKEY hKey, LPCTSTR lpSubKey,
    DWORD Reserved, LPTSTR lpClass, DWORD dwOptions,
    REGSAM samDesired,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    PHKEY phkResult, LPDWORD lpdwDisposition);
```

Первый параметр, *hKey*, – дескриптор открытого ключа реестра с правом доступа KEY_CREATE_SUB_KEY или один из предопределенных дескрипторов корневых узлов системного реестра (см. табл. 4.9).

Второй параметр, *lpSubKey*, указывает на строку, содержащую имя нового ключа, создаваемого в разделе, на который указывает дескриптор *hKey*. В этой строке разрешается использование символа «\», но она не должна начинаться или заканчиваться этим символом.

Параметры *Reserved* и *lpClass* игнорируются. Параметр *Reserved* должен принимать значение равное нулю, а параметр *lpClass* должен быть установлен в NULL.

Пятый параметр, *dwOptions*, определяет опции ключа. Возможные значения этого параметра перечислены в табл. 4.12.

Таблица 4.12. Значения параметра *dwOptions*

Значение	**Описание**
REG_OPTION_NON_VOLATILE	Создается постоянно хранимый раздел реестра, который не исчезает после перезагрузки операционной системы. Это значение используется по умолчанию
REG_OPTION_VOLATILE	Создается временный раздел реестра, который удаляется при завершении работы операционной системы. Это значение игнорируется, если указанный ключ уже существует

Шестой параметр, *samDesired*, позволяет указать уровень доступа для возвращаемого дескриптора объекта ядра «ключ реестра». Некоторые значения этого параметра перечислены в табл. 4.13. Полный перечень значений этого параметра можно найти в документации Platform SDK.

Таблица 4.13. Права доступа (значения параметра *samDesired*)

Значение	Описание
KEY_ALL_ACCESS	Все возможные права доступа
KEY_CREATE_SUB_KEY	Разрешает создание вложенных ключей
KEY_ENUMERATE_SUB_KEYS	Необходимо для получения информации о вложенных ключах
KEY_QUERY_VALUE	Необходимо для получения значений параметров ключа
KEY_SET_VALUE	Разрешает изменение значений параметров ключа

Седьмой параметр, *lpSecurityAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить атрибуты безопасности создаваемого ключа реестра и наследование возвращаемого дескриптора. Если параметр *lpSecurityAttributes* установить в NULL, ключ реестра будет создан с атрибутами безопасности по умолчанию, а возвращаемый дескриптор не будет наследуемым.

Восьмой параметр, *phkResult*, указывает на переменную типа HKEY, через которую функция RegCreateKeyEx возвращает дескриптор открытого ключа реестра (точнее объекта ядра «ключ реестра»).

Последний параметр, *lpdwDisposition*, указывает на переменную типа DWORD, в которую будет сохранено одно из следующих значений:

- REG_CREATED_NEW_KEY – был создан новый ключ;
- REG_OPENED_EXISTING_KEY – был открыт существующий ключ.

При успешном завершении функция RegCreateKeyEx возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Для того чтобы просто открыть уже созданный ключ системного реестра необходимо использовать функцию RegOpenKeyEx. В отличие от функции RegCreateKeyEx, функция RegOpenKeyEx не создает указанный ключ, если он не существует.

В следующем примере продемонстрированно создание ключа с именем SampleApplication в разделе HKEY_CURRENT_USER\Software.

Листинг 4.11. Создание ключа в системном реестре

```
1   HKEY hKey; // дескриптор ключа реестра
2   DWORD dwDisposition;
3
4   // создание ключа HKEY_CURRENT_USER\Software\SampleApplication
5   LSTATUS lStatus = RegCreateKeyEx(HKEY_CURRENT_USER,
        TEXT("Software\\SampleApplication"),
```

```
6       0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL,
    &hKey, &dwDisposition);
7
8   if (ERROR_SUCCESS != lStatus)
9   {
10      /* произошла ошибка */
11  } // if
12  else if (REG_CREATED_NEW_KEY == dwDisposition)
13  {
14      /* был создан новый ключ */
15  } // if
16  else if (REG_OPENED_EXISTING_KEY == dwDisposition)
17  {
18      /* был открыт существующий ключ */
19  } // if
```

Функция RegOpenKeyEx имеет следующий прототип:

```
LSTATUS RegOpenKeyEx(HKEY hKey, LPCTSTR lpSubKey,
    DWORD ulOptions, REGSAM samDesired, PHKEY phkResult);
```

Параметры *hKey*, *lpSubKey*, *samDesired* и *phkResult* этой функции аналогичны одноименным параметрам функции RegCreateKeyEx. Параметр *ulOptions* не используется и поэтому должен быть равен нулю.

Если функция RegOpenKeyEx завершается успешно, она возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Закрытие ключей

Каждый открытый дескриптор объекта ядра «ключ реестра» должен быть закрыт вызовом функции RegCloseKey:

```
LSTATUS RegCloseKey(HKEY hKey);
```

Параметр *hKey* – дескриптор открытого ключа реестра, который необходимо закрыть. При успешном завершении функция RegCloseKey возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Следует отметить, что при завершении процесса гарантировано, закрываются все открытые ключи реестра. Однако все же лучше закрывать их самостоятельно.

Удаление ключей

Для удаления ключа реестра используется функция RegDeleteKey:

```
LSTATUS RegDeleteKey(HKEY hKey, LPCTSTR lpSubKey);
```

Первый параметр, *hKey*, представляет собой дескриптор открытого ключа реестра или один из предопределенных дескрипторов корневых узлов системного реестра.

Второй параметр, *lpSubKey*, указывает на строку, содержащую имя вложенного ключа, который нужно удалить.

Функция RegDeleteKey завершится ошибкой, если удаляемый ключ содержится вложенные ключи. Если функция RegDeleteKey завершается успешно, она возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Для того чтобы удалить ключ вместе с его вложенными ключами следует использовать функцию RegDeleteTree:

```
LSTATUS RegDeleteTree(HKEY hKey, LPCTSTR lpSubKey);
```

Первый параметр, *hKey*, представляет собой дескриптор открытого ключа реестра с правами доступа DELETE, KEY_ENUMERATE_SUB_KEYS и KEY_QUERY_VALUE или один из предопределенных дескрипторов корневых узлов системного реестра.

Второй параметр, *lpSubKey*, указывает на строку, содержащую имя вложенного ключа, который нужно удалить. Если параметр *lpSubKey* установлен в NULL, будут удалены все вложенные ключи раздела, на который указывает дескриптор *hKey*.

Если функция RegDeleteTree завершается успешно, она возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Перечисление вложенных ключей и параметров

Для получения имен вложенных ключей любого открытого ключа реестра следует использовать функцию RegEnumKeyEx:

```
LSTATUS RegEnumKeyEx(HKEY hKey, DWORD dwIndex, LPTSTR lpName,
   LPDWORD lpcName, LPDWORD lpReserved, LPTSTR lpClass,
   LPDWORD lpcClass, PFILETIME lpftLastWriteTime);
```

Первый параметр, *hKey*, – дескриптор открытого ключа реестра с правом доступа KEY_ENUMERATE_SUB_KEYS или один из предопределенных дескрипторов корневых узлов системного реестра.

Второй параметр, *dwIndex*, задает индекс вложенного ключа. Для первого вложенного ключа этот параметр должен принимать значение равное нулю.

Третий параметр, *lpName*, указывает на буфер, в который будет записано имя вложенного ключа.

Четвертый параметр, *lpcName*, указывает на переменную типа DWORD, которая задает размер (в символах) буфера, указываемого параметром *lpName*. Если функция завершается успешно, в переменную, на которую указывает параметр *lpcName*, будет записано количество символов, сохраненных в буфере (нуль-символ в конце не учитывается).

Пятый параметр, *lpReserved*, не используется и должен быть установлен в NULL.

Шестой параметр, *lpClass*, указывает на буфер, в который будет записан класс вложенного ключа. Этот параметр может быть установлен в NULL.

Седьмой параметр, *lpcClass*, указывает на переменную типа DWORD, которая задает размер (в символах) буфера, указываемого параметром *lpClass*. Если функция завершается успешно, в переменную, на которую указывает параметр *lpcClass*, будет записано количество символов, сохраненных в буфере (нуль-символ в конце не учитывается). Если параметр *lpClass* установлен в NULL, параметр *lpcClass* также должен быть установлен в NULL.

Последний параметр, *lpftLastWriteTime*, указывает на переменную типа FILETIME, в которую будут сохранены даты и время (в формате UTC) последнего изменения вложенного ключа. Этот параметр может быть установлен в NULL.

При успешном завершении функция RegEnumKeyEx возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Для получения имен и значений параметров в любом открытом ключе реестра следует использовать функцию RegEnumValue:

```
LSTATUS RegEnumValue(HKEY hKey, DWORD dwIndex,
    LPTSTR lpValueName, LPDWORD lpcchValueName,
    LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData,
    LPDWORD lpcbData);
```

Первый параметр, *hKey*, – дескриптор открытого ключа реестра с правом доступа KEY_QUERY_VALUE или один из предопределенных дескрипторов корневых узлов системного реестра.

Второй параметр, *dwIndex*, задает индекс параметра в ключе. Индекс первого параметра в ключе равен нулю.

Третий параметр, *lpValueName*, указывает на буфер, в который будет записано имя параметра в ключе.

Четвертый параметр, *lpcchValueName*, указывает на переменную типа DWORD, которая задает размер (в символах) буфера, указываемого параметром *lpValueName*. Если функция завершается успешно, в переменную, на которую указывает параметр *lpcchValueName*, будет записано количество символов, сохраненных в буфере (нуль-символ в конце не учитывается).

Пятый параметр, *lpReserved*, не используется и должен быть установлен в NULL.

Шестой параметр, *lpType*, указывает на переменную типа DWORD, в которую будет записан тип данных (см. табл. 4.10) значения параметра в ключе. Этот параметр может быть установлен в NULL.

Седьмой параметр, *lpData*, указывает на буфер, в который будет записано значение параметра в ключе. Этот параметр может быть установлен в NULL.

Последний параметр, *lpcbData*, указывает на переменную типа DWORD, которая задает размер (в байтах) буфера, указываемого параметром *lpData*. Если функция завершается успешно, в переменную, на которую указывает параметр *lpcbData*, будет записано количество байтов, сохраненных в буфере. Параметр *lpcbData* может быть установлен в NULL, только если параметр *lpData* тоже установлен в NULL.

При успешном завершении функция RegEnumValue возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки. Если буфер, на который указывает параметр *lpData*, не является достаточно большим, функция вернет значение ERROR_MORE_DATA и сохранит необходимый размер в переменную, на которую указывает параметр *lpcbData*.

Для того чтобы в открытом ключе реестра определить число параметров и вложенных ключей, а также максимальные размеры определенных буферов, следует воспользоваться функцией RegQueryInfoKey:

```
LSTATUS RegQueryInfoKey(HKEY hKey, LPTSTR lpClass,
    LPDWORD lpcClass, LPDWORD lpReserved, LPDWORD lpcSubKeys,
    LPDWORD lpcMaxSubKeyLen, LPDWORD lpcMaxClassLen,
    LPDWORD lpcValues, LPDWORD lpcMaxValueNameLen,
    LPDWORD lpcMaxValueLen, LPDWORD lpcbSecurityDescriptor,
    PFILETIME lpftLastWriteTime);
```

Первый параметр, *hKey*, – дескриптор открытого ключа реестра с правом доступа KEY_QUERY_VALUE или один из предопределенных дескрипторов корневых узлов системного реестра.

Второй параметр, *lpClass*, указывает на буфер, в который будет записан класс ключа. Этот параметр может быть установлен в NULL.

Третий параметр, *lpcClass*, указывает на переменную типа DWORD, которая задает размер (в символах) буфера, указываемого параметром *lpClass*. Если функция завершается успешно, в переменную, на которую указывает параметр *lpcClass*, будет записано количество символов, сохраненных в буфере (нуль-символ в конце не учитывается). Если параметр *lpClass* установлен в NULL, параметр *lpcClass* также должен быть установлен в NULL.

Четвертый параметр, *lpReserved*, не используется и должен быть установлен в NULL.

Пятый параметр, *lpcSubKeys*, указывает на переменную типа DWORD, в которую будет записано число вложенных ключей. Этот параметр может быть установлен в NULL.

Шестой параметр, *lpcMaxSubKeyLen*, указывает на переменную типа DWORD, в которую будет записан максимальный размер (в символах Unicode) буфера для имени вложенного ключа (нуль-символ в конце не учитывается). Этот параметр может быть установлен в NULL.

Седьмой параметр, *lpcMaxClassLen*, указывает на переменную типа DWORD, в которую будет записан максимальный размер (в символах Unicode) буфера для класса вложенного ключа (нуль-символ в конце не учитывается). Этот параметр может быть установлен в NULL.

Восьмой параметр, *lpcValues*, указывает на переменную типа DWORD, в которую будет записано число параметров в ключе. Этот параметр может быть установлен в NULL.

Девятый параметр, *lpcMaxValueNameLen*, указывает на переменную типа DWORD, в которую будет записан максимальный размер (в символах Unicode) буфера для имени параметра в ключе (нуль-символ в конце не учитывается). Этот параметр может быть установлен в NULL.

Десятый параметр, *lpcMaxValueLen*, указывает на переменную типа DWORD, в которую будет записан максимальный размер (в байтах) буфера для значения параметра в ключе. Этот параметр может быть установлен в NULL.

Одиннадцатый параметр, *lpcbSecurityDescriptor*, указывает на переменную типа DWORD, в которую будет записан размер дескриптора безопасности ключа. Этот параметр может быть установлен в NULL.

Последний параметр, *lpftLastWriteTime*, указывает на переменную типа FILETIME, в которую будут сохранены даты и время (в формате UTC) последнего изменения вложенного ключа. Этот параметр может быть установлен в NULL.

При успешном завершении функция RegQueryInfoKey возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

В листингах 4.12,4.13 представлены примеры перечисления вложенных ключей и параметров с помощью функций RegQueryInfoKey, RegEnumKeyEx и RegEnumValue. Заметим, что в этих примерах для указания дескриптора ключа используется переменная hKey.

Листинг 4.12. Перечисление вложенных ключей

```
1  DWORD cSubKeys, cMaxName;
2
3  // определим число вложенных ключей и
4  // максимальный размер буфера для имени вложенного ключа
5  LSTATUS lStatus = RegQueryInfoKey(hKey, NULL, NULL, NULL,
   &cSubKeys, &cMaxName, NULL, NULL, NULL, NULL, NULL, NULL);
6
7  if (ERROR_SUCCESS == lStatus && cSubKeys > 0)
```

```
{
    // выделим память под буфер
    LPTSTR szName = new TCHAR[cMaxName + 1];

    for (DWORD dwIndex = 0; dwIndex < cSubKeys; ++dwIndex)
    {
        // получим имя вложенного ключа с индексом dwIndex
        DWORD cchName = cMaxName + 1;
        lStatus = RegEnumKeyEx(hKey, dwIndex, szName, &cchName,
NULL, NULL, NULL, NULL);

        if (ERROR_SUCCESS == lStatus)
        {
            /* имя вложенного ключа успешно получено */
        } // if
    } // for

    // освободим память, выделенную под буфер
    delete[]szName;
} // if
```

Листинг 4.13. Перечисление параметров ключа

```
DWORD cValues, cMaxValueNameLen, cbMaxValueLen;

// определим число параметров ключа и
// максимальный размер буферов для имени и значения параметра
LSTATUS lStatus = RegQueryInfoKey(hKey, NULL, NULL, NULL, NULL,
NULL, NULL, &cValues, &cMaxValueNameLen, &cbMaxValueLen, NULL,
NULL);

if (ERROR_SUCCESS == lStatus && cValues > 0)
{
    // выделим память под буферы
    LPTSTR szValueName = new TCHAR[cMaxValueNameLen + 1];
    LPBYTE lpValueData = new BYTE[cbMaxValueLen];

    for (DWORD dwIndex = 0; dwIndex < cValues; ++dwIndex)
    {
        // получим имя, тип и значение параметра
        DWORD cchValueName = cMaxValueNameLen + 1, cbData =
cbMaxValueLen, dwType;
        lStatus = RegEnumValue(hKey, dwIndex, szValueName,
&cchValueName, NULL, &dwType, lpValueData, &cbData);

        if (ERROR_SUCCESS == lStatus)
        {
```

```
21              /* имя, тип и значение параметра успешно получены */
22          } // if
23      } // for
24
25      // освободим память, выделенную под буферы
26      delete[]lpValueData;
27      delete[]szValueName;
28  } // if
```

Получение и изменение значений параметров

Чтобы получить значение параметра в открытом ключе реестра следует воспользоваться функцией `RegQueryValueEx`:

```
LSTATUS RegQueryValueEx(HKEY hKey, LPCTSTR lpValueName,
    LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData,
    LPDWORD lpcbData);
```

Функция `RegQueryValueEx` работает аналогично функции `RegEnumValue`, за исключением того, что требует указание имени параметра, а не его индекса. Имя параметра передается в строке, на которую указывает параметр *`lpValueName`*. Если этот параметр установлен в `NULL` или указывает на пустую строку, функция вернет значение параметра «По умолчанию».

Для изменения значения параметра в открытом ключе реестра используется функция `RegSetValueEx`:

```
LSTATUS RegSetValueEx(HKEY hKey, LPCTSTR lpValueName,
    DWORD Reserved, DWORD dwType, const BYTE *lpData,
    DWORD cbData);
```

Первый параметр, *`hKey`*, – дескриптор открытого ключа реестра с правом доступа `KEY_SET_VALUE` или один из предопределенных дескрипторов корневых узлов системного реестра.

Второй параметр, *`lpValueName`*, указывает на строку, содержащую имя параметра, значение которого нужно изменить. Если этот параметр установлен в `NULL` или указывает на пустую строку, функция изменит значение параметра «По умолчанию». Если в указанном ключе параметр с таким именем не существует, функция добавит его в ключ.

Третий параметр, *`Reserved`*, не используется и должен принимать значение равное нулю.

Четвертый параметр, *`dwType`*, задает тип данных (см. табл. 4.10) нового значения изменяемого параметра в ключе.

Пятый параметр, *`lpData`*, указывает на буфер, в котором хранится новое значение изменяемого параметра в ключе.

Последний параметр, *cbData*, задает размер (в байтах) буфера, на который указывает параметр *lpData*.

Если функция RegSetValueEx завершается успешно, она возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

В листингах 4.14,4.15 приводятся функции для получения и изменения значений параметров типа REG_DWORD и REG_SZ. Функции для получения и изменения значений параметров других типов данных реализуются аналогичным образом.

Листинг 4.14. Функции получения значений параметров реестра

```
1   LSTATUS RegGetValueDWORD(HKEY hKey, LPCTSTR lpValueName,
    LPDWORD lpdwData)
2   {
3       DWORD dwType;
4
5       // определяем тип получаемого значения параметра
6       LSTATUS lStatus = RegQueryValueEx(hKey, lpValueName, NULL,
    &dwType, NULL, NULL);
7
8       if (ERROR_SUCCESS == lStatus && REG_DWORD == dwType)
9       {
10          // вычисляем размер буфера (в байтах)
11          DWORD cb = sizeof(DWORD);
12          // получаем значение параметра
13          lStatus = RegQueryValueEx(hKey, lpValueName, NULL, NULL,
    (LPBYTE)lpdwData, &cb);
14      } // if
15      else if (ERROR_SUCCESS == lStatus)
16      {
17          lStatus = ERROR_UNSUPPORTED_TYPE; // неверный тип данных
18      } // if
19
20      return lStatus;
21  } // RegGetValueDWORD
22
23  LSTATUS RegGetValueSZ(HKEY hKey, LPCTSTR lpValueName,
    LPTSTR lpszData, DWORD cch, LPDWORD lpcchNeeded)
24  {
25      DWORD dwType;
26
27      // определяем тип получаемого значения параметра
28      LSTATUS lStatus = RegQueryValueEx(hKey, lpValueName, NULL,
    &dwType, NULL, NULL);
29
30      if (ERROR_SUCCESS == lStatus && REG_SZ == dwType)
```

```
31      {
32          // вычисляем размер буфера (в байтах)
33          DWORD cb = cch * sizeof(TCHAR);
34          // получаем значение параметра
35          lStatus = RegQueryValueEx(hKey, lpValueName, NULL, NULL,
    (LPBYTE)lpszData, &cb);
36
37          if (NULL != lpcchNeeded)
38              *lpcchNeeded = cb / sizeof(TCHAR);
39      } // if
40      else if (ERROR_SUCCESS == lStatus)
41      {
42          lStatus = ERROR_UNSUPPORTED_TYPE; // неверный тип данных
43      } // if
44
45      return lStatus;
46  } // RegGetValueSZ
```

Листинг 4.15. Функции изменения значений параметров реестра

```
1   LSTATUS RegSetValueDWORD(HKEY hKey, LPCTSTR lpValueName,
    DWORD dwData)
2   {
3       // изменяем значение параметра
4       return RegSetValueEx(hKey, lpValueName, 0, REG_DWORD,
    (LPCBYTE)&dwData, sizeof(DWORD));
5   } // RegSetValueDWORD
6
7   LSTATUS RegSetValueSZ(HKEY hKey, LPCTSTR lpValueName,
    LPCTSTR lpszData)
8   {
9       // вычисляем размер строкового значения (в байтах)
10      DWORD cb = (_tcslen(lpszData) + 1) * sizeof(TCHAR);
11      // изменяем значение параметра
12      return RegSetValueEx(hKey, lpValueName, 0, REG_SZ,
    (LPCBYTE)lpszData, cb);
13  } // RegSetValueSZ
```

Удаление параметров

Удаление параметра в ключе может выполняться при помощи одной из двух функций – `RegDeleteValue` и `RegDeleteKeyValue`:

```
LSTATUS RegDeleteValue(HKEY hKey, LPCTSTR lpValueName);

LSTATUS RegDeleteKeyValue(HKEY hKey, LPCTSTR lpSubKey,
    LPCTSTR lpValueName);
```

Первый параметр, *hKey*, – дескриптор открытого ключа реестра с правом доступа KEY_SET_VALUE или один из предопределенных дескрипторов корневых узлов системного реестра.

Параметр *lpSubKey* указывает на строку, содержащую имя вложенного ключа, в котором нужно удалить параметр.

Последний параметр, *lpValueName*, указывает на строку, содержащую имя удаляемого параметра. Если этот параметр установлен в NULL или указывает на пустую строку, функция удалит только значение параметра «По умолчанию».

При успешном завершении функции RegDeleteValue и RegDeleteKeyValue возвращают значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Задание к работе

1. Разработать в Visual C++ оконное приложение Win32, которое:
   - должно иметь возможность создания, чтения и редактирования текстовых файлов;

     чтение и редактирование файлов должно осуществляться с помощью асинхронных операций ввода/вывода
   - должно сохранять в файле инициализации размер и положение окна, а также имя последнего редактируемого текстового фала, чтобы использовать их при повторном запуске.

     можно также сохранять в файле инициализации и другие параметры приложения (например, параметры шрифта)
2. Разработать в Visual C++ оконное приложение Win32, которое:
   - должно для выбранного файла или каталога выводить его имя, атрибуты (см. табл. 4.5) и размер, а также время его создания, изменения и последнего обращения;
   - должно иметь возможность переименования выбранного файла или каталога;
   - должно иметь возможность изменять атрибуты выбранного файла или каталога (кроме атрибутов системный и временный файл, а также атрибутов сжатия и шифрования);
   - должно сохранять в системном реестре размер и положение окна, а также имя последнего выбранного файла или каталога, чтобы использовать их при повторном запуске.

     сохранение в системном реестре должно осуществляться в разделе HKCU\Software\IT-3(#), где # – номер варианта.

3. Разработать в Visual C++ приложение Win32, которое должно выполнять указанную в варианте задания операцию с файлами и каталогами.

   Каталоги должны перемещаться, копироваться и удаляться вместе с вложенными файлами и каталогами.

4. Разработать в Visual C++ приложение Win32, которое должно выводить следующую информацию из системного реестра:
   - список установленных программ;

     `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`
   - список программ автозапуска.

     `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`
     `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`

5. Протестировать работу разработанных приложений на компьютере под управлением Windows XP (или выше). Результаты тестирования отразить в отчете.

6. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Операция с файлами и каталогами (п. 3 задания)
1,16	Перемещение файлов и каталогов
2,17	Копирование файлов и каталогов
3,18	Удаление файлов и каталогов
4,19	Перемещение файлов и каталогов
5,20	Копирование файлов и каталогов
6,21	Удаление файлов и каталогов
7,22	Перемещение файлов и каталогов
8,23	Копирование файлов и каталогов
9,24	Удаление файлов и каталогов
10,25	Перемещение файлов и каталогов
11,26	Копирование файлов и каталогов
12,27	Удаление файлов и каталогов
13,28	Перемещение файлов и каталогов
14,29	Копирование файлов и каталогов

№	Операция с файлами и каталогами (п. 3 задания)
15,30	Удаление файлов и каталогов

Контрольные вопросы

1. Какие существуют правила именования файлов и каталогов в Windows?
2. Какую функцию Win32 API следует использовать для создания/открытия файла?
3. Как открыть файл только для чтения или только для записи? Как открыть файл для чтения и записи?
4. Как организовать совместный доступ к открытому файлу? Для чего нужен совместный доступ к файлу?
5. Какую функцию Win32 API следует использовать для закрытия файла?
6. Какие функции Win32 API следует использовать для чтения и записи данных в файлах?
7. Что такое буферизация данных при чтении и записи данных? Как отключить буферизацию данных?
8. Для чего предназначено упреждающее чтение данных? Как отключить упреждающее чтение данных?
9. Что такое асинхронные операции чтении и записи данных? Как разрешить асинхронные операции чтения и записи данных?
10. Какие функции Win32 API следует использовать для асинхронных операций чтения и записи данных?
11. Какую функцию Win32 API следует использовать для отмены асинхронных операций чтения и записи данных?
12. Какую функцию Win32 API следует использовать для изменения указателя в файле?
13. Какую функцию Win32 API следует использовать для принудительного указания конца файла?
14. Какие функции Win32 API следует использовать для создания каталогов?
15. Какие функции Win32 API следует использовать для удаления файлов и каталогов?
16. Какие функции Win32 API следует использовать для перемещения или переименования файлов и каталогов?

17. Какую функцию Win32 API следует использовать для копирования файлов?
18. Какую функцию Win32 API следует использовать для открытия каталогов и логических дисков?
19. Что такое текущий каталог? Какие функции Win32 API следует использовать для определения и изменения текущего каталога?
20. Какими атрибутами обладают файлы и каталоги? Какие функции Win32 API следует использовать для определения и изменения атрибутов файлов и каталогов?
21. Какие функции Win32 API следует использовать для поиска файлов и каталогов? Что такое шаблон поиска?
22. Что такое файл инициализации? Какие функции Win32 API следует использовать для работы с файлами инициализации?
23. Что такое системный реестр Windows?
24. Какие функции Win32 API следует использовать для создания/открытия ключей реестра?
25. Какую функцию Win32 API следует использовать для закрытия ключей реестра?
26. Какие функции Win32 API следует использовать для удаления ключей реестра?
27. Какие функции Win32 API следует использовать для перечисления вложенных ключей и параметров в ключе реестра?
28. Какие функции Win32 API следует использовать для работы с параметрами ключа реестра?

ЛАБОРАТОРНАЯ РАБОТА № 5
БЕЗОПАСНОСТЬ В WINDOWS

Цель работы

Изучение модели безопасности Windows. Получение практических навыков применения функций Win32 API для управления безопасностью в Windows.

Основные понятия

В Windows (начиная с Windows NT) реализована эффективная модель безопасности, которая исключает возможность несанкционированного доступа к *защищаемым объектам* (securable objects) операционной системы таким, как файлы и каталоги (только в файловой системе NTFS), ключи системного реестра, различные объекты ядра и т.п. В этой модели безопасности каждая учетная запись является *субъектом безопасности* (security subject), которому может быть предоставлен доступ к защищаемому объекту или отказано в нем на основании разрешений, заданных для данного объекта.

Учетные записи

Учетные записи являются одним из ключевых элементов в модели безопасности Windows. Существует следующие типы учетных записей:

- *учетная запись пользователя* (user account) – нужна для входа в операционную систему и получения доступа к защищаемым объектам;
- *учетная запись компьютера* (computer account) – используется для регистрации компьютеров в домене и создается при подключении компьютера к домену;
- *учетная запись группы* (group account) или просто *группа* – состоит из учетных записей пользователей, компьютеров и других групп и может управляться как единое целое.

Кроме того учетные записи разделяются на локальные и доменные. *Локальные учетные записи* создаются на компьютере и хранятся в базе данных менеджера учетных записей (Security Account Manager, SAM) этого компьютера.

Доменные учетные записи создаются на контроллере домена и хранятся в базе данных домена (Domain Directory Database). При этом именно контроллеры домена отвечают за аутентификацию таких учетных записей.

Добавлять, удалять или редактировать учетные записи можно с помощью таких инструментальных средств, как «Управление компьютером» (Computer Management) (%WINDIR%\system32\compmgmt .msc /s) для локальных учетных записей или «Пользователи и группы Active Directory» (Active Directory Users and Groups) для доменных учетных записей. На рис. 5.1 показано окно «Управление компьютером», где в разделе «Локальные пользователи и группы» (Local Users and Groups) выведен список учетных записей пользователей локального компьютера.

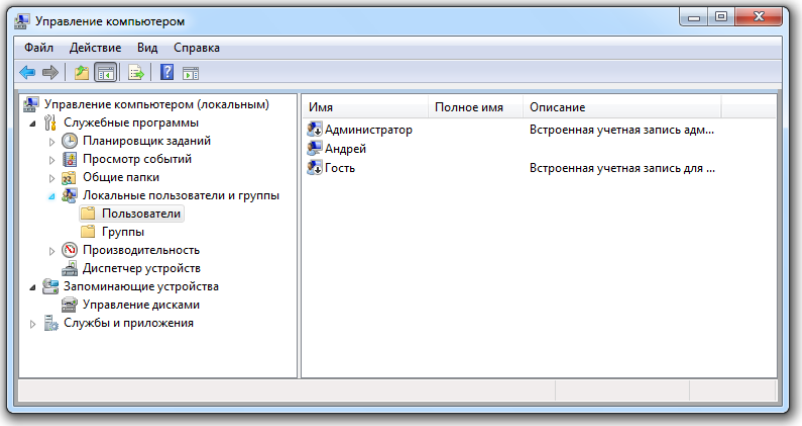

Рис. 5.1. Список учетных записей пользователей

Учетные записи пользователей

Прежде чем пользователь сможет работать в Windows, он должен быть зарегистрирован. При регистрации пользователя для него создается учетная запись. Учетная запись пользователя позволяет ему входить на компьютер или в домен, где ему предоставляется возможность доступа и аудита к защищаемым объектам.

Встроенные учетные записи

В Windows существуют предопределенные встроенные учетные записи. Каждая встроенная учетная запись имеет разную комбинацию прав и разрешений. В табл. 5.1 описаны все встроенные учетные записи пользователей.

Таблица 5.1. Встроенные учетные записи

Название	Описание
Администратор (Administrator)	Учетная запись администратора предоставляет полный доступ и позволяет при необходимости назначать права и разрешения для пользователей. Эта учетная запись создается автоматически, при установке операционной системы
Гость (Guest)	Учетная запись гостя создается автоматически, при установке операционной системы и используется теми, кто не имеет фактической учетной записи
Помощник (HelpAssistant)	Учетная запись помощника используется для установки сеанса удаленного помощника (remote assistance). Эта учетная запись создается при запросе сеанса удаленного помощника и удаляется после его завершения

Служебные учетные записи

Учетная запись, от имени которой, запускаются службы Windows, называется *служебной учетной записью* (service account). Все служебные учетные записи рассмотрены в табл. 5.2.

Таблица 5.2. Служебные учетные записи

Название	Описание
Система (SYSTEM или LocalSystem)	Учетная запись локальной системы имеет безграничные права доступа и является учетной записью, под которой запускаются все основные компоненты операционной системы Windows, работающие в пользовательском режиме
LOCAL SERVICE или LocalService	Учетная запись локальной службы имеет минимальные права доступа и предназначена для тех служб, которым нужно подключиться к компьютерам в сети, допускающим анонимный доступ
NETWORK SERVICE или NetworkService	Учетная запись сетевой службы имеет минимальные права доступа и предназначена для тех служб, которым нужно подключиться к компьютерам в сети, используя учетную запись локального компьютера

Как правило, большинство служб Windows запускаются под учетной записью локальной системы. Две другие служебные учетные записи используются службами Windows, не требующих всех прав, которые имеются у учетной записи локальной системы.

Учетные записи компьютеров

Каждый компьютер, работающий под управлением Windows, который присоединяется к домену, имеет учетную запись компьютера,

которая хранится в базе данных домена. Так же, как и учетные записи пользователей, учетные записи компьютеров предоставляют возможность доступа и аудита к защищаемым объектам. Дополнительные сведения об учетных записях компьютеров и доменов можно найти в документации Platform SDK.

Группы

Группы обеспечивают эффективное управление доступом к защищаемым объектам. Использование групп позволяет предоставлять набор прав и разрешений сразу для нескольких учетных записей, которые являются членами одной группы. При добавлении учетной записи в группу она получает все права, назначенные для этой группы, а также разрешения, назначенные группе на всех защищаемых объектах.

Группы различаются областью действия, которая определяет пределы применения группы. Например, существуют локальные группы, глобальные и универсальные:

- *локальные группы* могут включать учетные записи как из текущего домена, так и из других доменов, но могут быть использованы только в текущем домене;
- *глобальные группы* могут включать учетные записи только из текущего домена, но могут быть использованы в любом домене;
- *универсальные группы* могут включать учетные записи из любого домена и могут быть использованы также в любом домене.

Встроенные группы

В Windows есть встроенные (built-in) группы, с заранее определенным набором прав, дающих членам группы возможность выполнять определенные действия на компьютере или в домене. Как правило, встроенные группы создаются автоматически при установке операционной системы или при создании домена. В табл. 5.3,5.4 описаны некоторые из них.

Таблица 5.3. Некоторые встроенные группы локального компьютера

Название	**Описание**
Администраторы (Administrators)	Члены этой группы имеют полные ничем неограниченные права доступа
Гости (Guests)	Члены этой группы получают временный профиль, который создается при входе пользователя в систему и удаляется при выходе из нее
Операторы архива (Backup Operators)	Члены этой группы могут создавать резервные копии файлов операционной системы

Окончание табл. 5.3.

Название	**Описание**
Операторы учета (Account Operators)	Члены этой группы могут создавать, изменять и удалять учетные записи пользователей, групп и компьютеров
Операторы печати (Print Optrators)	Члены этой группы могут управлять, создавать, открывать для общего доступа и удалять принтеры в домене
Операторы настройки сети (Network Configuration Optrators)	Члены этой группы имеют права для управления настройкой сетевых параметров
Опытные пользователи (Power Users)	Члены этой группы по умолчанию имеют те же права, что и обычные пользователи. Изначально эта группа была создана для того, чтобы назначать пользователям особые административные права и разрешения для выполнения распространенных системных задач
Пользователи (Users)	Члены этой группы являются зарегистрированными пользователями, они могут запускать большинство приложений, но не имеют прав на изменение системы
Пользователи удаленного рабочего стола (Remote Desctop Connection)	Члены этой группы обеспечивают подключение к удаленному рабочему столу (Remote Desktop Connection)

Таблица 5.4. Некоторые встроенные группы домена

Название	**Описание**
Администраторы домена (Domain Admins)	Члены этой группы полностью контролируют домен. По умолчанию, эта группа является членом группы «Администраторы» на всех контроллерах домена и всех входящих в домен компьютерах на то время, пока они присоединены к домену. По умолчанию членом этой группы является учетная запись «Администратор»
Гости домена (Domain Guests)	Члены этой группы получают временный профиль, который создается при входе пользователя в домен и удаляется при выходе из него
Компьютеры домена (Domain Computers)	Членами этой группы являются все компьютеры, присоединенные к домену. По умолчанию любая созданная учетная запись компьютера автоматически становится членом этой группы

Окончание табл. 5.4.

Название	Описание
Контроллеры домена (Domain Controllers)	Членами этой группы являются все контроллеры домена в домене
Пользователи домена (Domain Users)	Членами этой группы являются все зарегистрированные пользователи в домене

Полный список и подробное описание всех встроенных групп Windows можно найти в документации Platform SDK.

Специальные группы

В Windows существуют специальные группы (табл. 5.5), которые в зависимости от обстоятельств представляют разных пользователей в различное время. Хотя специальным группам могут назначаться права и разрешения для доступа к защищаемым объектам, членство в этих группах нельзя изменить или просмотреть. Также специальные группы не обладают областью действия.

Таблица 5.5. Некоторые специальные группы

Название	Описание
Анонимный вход (Anonymous Logon)	Членами этой группы являются пользователи, которые в данный момент имеют доступ к компьютеру и его защищаемым объектам по сети без использования имени учетной записи, пароля и имени домена.
Все (Everyone)	Членами этой группы являются все текущие пользователи, включая гостей и пользователей из других доменов
Интерактивные (Interactive)	Членами этой группы являются пользователи, которые вошли на компьютер и получили доступ к находящимся на нем защищаемым объектам
Консольный вход (Console Logon)	Членами этой группы являются пользователи, которые подключились к существующему сеансу физической консоли на компьютере (сеанс 0)
Пакетные файлы (Batch)	Членами этой группы являются пользователи, которые вошли на компьютер с помощью средства обработки пакетных заданий
Прошедшие проверку (Authenticated Users)	Членами этой группы являются пользователи с действительной учетной записью
Сеть (Network)	Членами этой группы являются пользователи, которые в данный момент имеют доступ к компьютеру и его защищаемым объектам по сети

Окончание табл. 5.5.

Название	Описание
Служба (Service)	Членами этой группы являются все служебные учетные записи
Удаленный доступ (Dialup)	Членами этой группы являются пользователи, которые вошли на компьютер с использованием удаленного доступа

Пользователи становятся членами специальных групп автоматически, когда входят на компьютер или получают доступ к его защищаемым объектам. Полный список всех специальных групп и их подробное описание см. в документации Platform SDK.

Идентификаторы безопасности

Для идентификации учетных записей используется *идентификатор безопасности* (security identifier, SID), который представляет собой структуру переменной длины, состоящую из следующих компонентов (рис. 5.2):

- 8-битный *номер версии* (revision level) SID. На сегодняшний день используется версия 1;
- 8-битное значение определяющее *количество субъектов полномочий* (number of sub-authorities);
- 48-битный *идентификатор полномочий* (identifier authority). Некоторые из возможных значений идентификатора полномочий перечислены в табл. 5.6 (полный список можно найти в документации Platform SDK);
- 32-битные *субъекты полномочий* (sub-authorities) и/или 32-битный *относительный идентификатор* (relative identifier, RID), который используется для создания уникальных SID на основе *общего базового SID* (common-based SID).

Рис. 5.2. Структура SID

Таблица 5.6. Идентификаторы полномочий SID

Значение	Описание
0	Используется только для SID с неизвестными полномочиями (NULL Authority)
1	Используется только для SID с глобальными полномочиями (World Authority)
2	Используется только для SID с локальными полномочиями (Local Authority)
5	Используется подсистемой безопасности Windows NT (NT Authority) для создания SID учетных записей

SID может быть представлен в виде символьной строки, которая начинается с префикса S за которым следуют группы цифр, разделяемые дефисами, например:

```
S-1-5-21-3111953614-2450355651-2090111572-1007
```

В этом SID номер версии равен 1, идентификатор полномочий – 5, далее идут значения четырех субъектов полномочий и в конце RID. Данный SID относится к учетной записи пользователя на компьютере или в домене, который имеет SID с тем же номером версии, идентификатором полномочий и значениями субъектов полномочий:

```
S-1-5-21-3111953614-2450355651-2090111572
```

При установке операционной системы компьютеру (или домену при его создании) назначается уникальный SID. При этом маловероятно появление двух одинаковых SID. Далее для каждой учетной записи компьютера/домена создается свой SID, который формируется на основе SID компьютера/домена с добавлением RID учетной записи. RID учетной записи пользователя начинается с 1000 и увеличивается на 1 для каждой новой учетной записи. При этом для встроенных учетных записей RID начинается с 500. Так, например, RID учетной записи «Администратор» равен 500, а RID учетной записи «Гость» – 501:

```
S-1-5-21-3111953614-2450355651-2090111572-500
S-1-5-21-3111953614-2450355651-2090111572-501
```

Хорошо известные идентификаторы безопасности

Под хорошо известными (well known) понимаются идентификаторы безопасности общих пользователей или общих групп. Их значения остаются постоянными во всех операционных системах. Некоторые из них приведены в табл. 5.7 (полный список см. в документации Platform SDK).

Таблица 5.7. Некоторые хорошо известные SID

SID	Описание
S-1-0-0	Нет участника безопасности (NULL SID)
S-1-1-0	Группа «Все»
S-1-2-1	Группа «Консольный вход»
S-1-5-1	Группа «Удаленный доступ»
S-1-5-2	Группа «Сеть»
S-1-5-3	Группа «Пакетные файлы»
S-1-5-4	Группа «Интерактивные»
S-1-5-6	Группа «Служба»
S-1-5-7	Группа «Анонимный вход»
S-1-5-11	Группа «Прошедшие проверку»
S-1-5-18	Учетная запись локальной системы
S-1-5-19	Учетная запись локальной службы
S-1-5-20	Учетная запись сетевой службы
S-1-5-21-X-X-X-500	Учетная запись «Администратор»
S-1-5-21-X-X-X-501	Учетная запись «Гость»
S-1-5-21-X-X-X-512	Группа «Администраторы домена»
S-1-5-21-X-X-X-513	Группа «Пользователи домена»
S-1-5-21-X-X-X-514	Группа «Гости домена»
S-1-5-21-X-X-X-515	Группа «Компьютеры домена»
S-1-5-21-X-X-X-516	Группа «Контроллеры домена»
S-1-5-32-544	Группа «Администраторы»
S-1-5-32-545	Группа «Пользователи»
S-1-5-32-546	Группа «Гости»
S-1-5-32-547	Группа «Опытные пользователи»
S-1-5-32-548	Группа «Операторы учета»
S-1-5-32-550	Группа «Операторы печати»
S-1-5-32-555	Группа «Пользователи удаленного рабочего стола»
S-1-5-32-556	Группа «Операторы настройки сети»

Работа с идентификаторами безопасности

Чтобы определить SID учетной записи пользователя, группы или компьютера используется функция LookupAccountName:

```
BOOL LookupAccountName(LPCTSTR lpSystemName,
    LPCTSTR lpAccountName, PSID Sid, LPDWORD cbSid,
    LPTSTR lpReferencedDomainName,
    LPDWORD cchReferencedDomainName, PSID_NAME_USE peUse);
```

Первый параметр, *lpSystemName*, указывает на строку, содержащую имя компьютера или домена, на котором будет вестись поиск SID. Если этот параметр установлен в NULL, поиск ведется на локальном компьютере.

Второй параметр, *lpAccountName*, указывает на строку, содержащую имя учетной записи пользователя, группы или компьютера.

Третий параметр, *Sid*, указывает на буфер, в который будет сохранен SID. Размер этого буфера (в байтах) указывается в параметре *cbSid*. В переменную, на которую указывает параметр *cbSid*, будет записано число сохраненных в буфере байт.

Пятый параметр, *lpReferencedDomainName*, представляет собой указатель на буфер, в который сохраняется имя домена, к которому относится указанная учетная запись, а в случае со встроенными и специальными группами – значение «BUILTIN» или «NT AUTHORITY», или пустая строка. Шестой параметр, *cchReferencedDomainName*, указывает на размер (в символах) этого буфера. В переменную, на которую указывает этот параметр, сохраняется число символов записанных в буфер (нуль-символ в конце не учитывается).

Последний параметр, *peUse*, указывает на переменную перечисляемого типа SID_NAME_USE, в которую будет сохранен тип полученного SID. В табл. 5.8 перечислены основные типы SID.

Таблица 5.8. Основные типы SID

Тип	**Описание**
SidTypeAlias	Псевдоним
SidTypeDomain	Компьютер (домен)
SidTypeGroup	Группа
SidTypeUser	Учетная запись пользователя
SidTypeWellKnownGroup	Общеизвестная группа

При успешном завершении функция LookupAccountName возвращает значение отличное от FALSE. Выполнение этой функции завершится ошибкой, если размеры буферов, на которые указывают параметры *Sid*

и *lpReferencedDomainName*, будут меньше, чем требуется для сохранения результата. Для того чтобы определить необходимые размеры следует установить параметры *Sid* и *lpReferencedDomainName* в NULL, а значение переменных, на которые указывают параметры *cbSid* и *cchReferencedDomainName*, равным нулю. При этом параметр *cbSid* вернет размер буфера в байтах, а параметр *cchReferencedDomainName* – в символах (с учетом нуль-символа в конце). Также можно использовать константу SECURITY_MAX_SID_SIZE, которая соответствует максимально возможному размеру буфера для SID.

В листинге 5.1 представлена функция, которая определяет SID по имени учетной записи. Функция вернет адрес созданного буфера для SID. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции LocalFree.

Листинг 5.1. Функция, которая возвращает SID учетной записи

```
1   BOOL GetAccountSID(LPCTSTR lpAccountName, PSID *ppSid)
2   {
3       PSID pSid = NULL;
4       LPTSTR lpszDomainName = NULL;
5
6       DWORD cbSid = 0, cchDomainName = 0;
7
8       // определяем размеры буферов
9       LookupAccountName(NULL, lpAccountName, NULL, &cbSid, NULL,
    &cchDomainName, NULL);
10
11      if (cbSid > 0)
12      {
13          // выделяем блок памяти под буфер для SID
14          pSid = (PSID)LocalAlloc(LMEM_FIXED, cbSid);
15      } // if
16
17      if (cchDomainName > 0)
18      {
19          // выделяем блок памяти под буфер для доменного имени
20          lpszDomainName = (LPTSTR)LocalAlloc(LMEM_FIXED,
    cchDomainName*sizeof(TCHAR));
21      } // if
22
23      BOOL bRet = FALSE;
24
25      if (NULL != pSid && NULL != lpszDomainName)
26      {
27          SID_NAME_USE SidType;
28
```

```
29          // определяем SID указанной учетной записи
30          bRet = LookupAccountName(NULL, lpAccountName, pSid,
    &cbSid, lpszDomainName, &cchDomainName, &SidType);
31      } // if
32
33      if (FALSE != bRet)
34      {
35          *ppSid = pSid; // возвращаем полученный SID
36      } // if
37      else if (NULL != pSid)
38      {
39          LocalFree(pSid); // освобождаем выделенную память
40      } // if
41
42      // освобождаем выделенную память
43      if (NULL != lpszDomainName) LocalFree(lpszDomainName);
44
45      return bRet;
46  } // GetAccountSID
```

Чтобы получить хорошо известный SID можно использовать функцию `CreateWellKnownSid`:

```
BOOL CreateWellKnownSid(WELL_KNOWN_SID_TYPE WellKnownSidType,
    PSID DomainSid, PSID pSid, DWORD *cbSid);
```

Первый параметр, *`WellKnownSidType`*, определяет какой SID необходимо получить. Некоторые из значений, которые может принимать этот параметр, перечислены в табл. 5.9 (полный перечень см. в документации Platform SDK).

Второй параметр, *`DomainSid`*, указывает на буфер, в котором содержится SID компьютера (домена). Для большинства хорошо известных SID этот параметр игнорируется и может быть установлен в `NULL`.

Таблица 5.9. Некоторые значения параметра *`WellKnownSidType`*

Значение	**SID**
`WinNullSid`	`S-1-0-0`
`WinWorldSid`	`S-1-1-0`
`WinConsoleLogonSid`	`S-1-2-1`
`WinDialupSid`	`S-1-5-1`
`WinNetworkSid`	`S-1-5-2`
`WinBatchSid`	`S-1-5-3`

Окончание табл. 5.9.

Значение	**SID**
WinInteractiveSid	S-1-5-4
WinServiceSid	S-1-5-6
WinAnonymousSid	S-1-5-7
WinAuthenticatedUserSid	S-1-5-11
WinLocalSystemSid	S-1-5-18
WinLocalServiceSid	S-1-5-19
WinNetworkServiceSid	S-1-5-20
WinAccountAdministratorSid	S-1-5-21-X-X-X-500
WinAccountGuestSid	S-1-5-21-X-X-X-501
WinAccountDomainAdminsSid	S-1-5-21-X-X-X-512
WinAccountDomainUsersSid	S-1-5-21-X-X-X-513
WinAccountDomainGuestsSid	S-1-5-21-X-X-X-514
WinAccountComputersSid	S-1-5-21-X-X-X-515
WinAccountControllersSid	S-1-5-21-X-X-X-516
WinBuiltinAdministratorsSid	S-1-5-32-544
WinBuiltinUsersSid	S-1-5-32-545
WinBuiltinGuestsSid	S-1-5-32-546
WinBuiltinPowerUsersSid	S-1-5-32-547
WinBuiltinAccountOperatorsSid	S-1-5-32-548
WinBuiltinPrintOperatorsSid	S-1-5-32-550
WinBuiltinRemoteDesktopUsersSid	S-1-5-32-555
WinBuiltinNetworkConfigurationOperatorsSid	S-1-5-32-556

Третий параметр, *pSid*, указывает на буфер, в который будет сохранен SID. Размер этого буфера (в байтах) указывается в параметре *cbSid*. В переменную, на которую указывает параметр *cbSid*, будет записано число сохраненных в буфере байт.

Также как и функция LookupAccountName выполнение функции CreateWellKnownSid завершится ошибкой, если размер буфера, на который указывает параметр *Sid*, будет меньше, чем требуется для сохранения результата. При этом параметр *cbSid* вернет нужный размер буфера. Можно также использовать константу SECURITY_MAX_SID_SIZE.

При успешном завершении функция `CreateWellKnownSid` возвращает значение отличное от `FALSE`.

В листинге 5.2 представлена функция, которая определяет встроенный общеизвестный SID. Функция вернет адрес созданного буфера для SID. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`.

Листинг 5.2. Функция, которая возвращает хорошо известный SID

```
1   BOOL GetWellKnownSID(WELL_KNOWN_SID_TYPE WellKnownSidType,
    PSID pDomainSid, PSID *ppSid)
2   {
3       DWORD cbSid = SECURITY_MAX_SID_SIZE;
4
5       // выделяем блок памяти под буфер для SID
6       PSID pSid = (PSID)LocalAlloc(LMEM_FIXED, cbSid);
7       if (NULL == pSid) return FALSE;
8
9       // определяем SID
10      BOOL bRet = CreateWellKnownSid(WellKnownSidType, pDomainSid,
    pSid, &cbSid);
11
12      if (FALSE != bRet)
13      {
14          *ppSid = pSid; // возвращаем полученный SID
15      } // if
16      else
17      {
18          LocalFree(pSid); // освобождаем выделенную память
19      } // else
20
21      return bRet;
22  } // GetWellKnownSID
```

В Win32 API имеется еще множество других функций, предназначенных для работы с SID. Некоторые из них приведены в табл. 5.10. Описание этих и других функций для работы с SID можно найти в документации Platform SDK.

Таблица 5.10. Функции для работы с SID

Функция	**Описание**
`AllocateAndInitializeSid`	Создает и инициализирует SID
`ConvertSidToStringSid`	Преобразует SID в строку
`ConvertStringSidToSid`	Преобразует строку в SID

Окончание табл. 5.10.

Функция	Описание
CopySid	Копирует SID из одного буфера в другой
EqualSid	Сравнивает два SID
GetLengthSid	Определяет длину SID
IsValidSid	Проверяет корректность SID
IsWellKnownSid	Сравнивает SID с хорошо известным SID
LookupAccountSid	Возвращает имя учетной записи пользователя, группы или компьютера для указанного SID

В следующем примере продемонстрировано использование нескольких функций для работы с SID.

Листинг 5.3. Функция, которая возвращает имя учетной записи по SID

```
1   BOOL GetAccountName(PSID pSid, LPTSTR *lpName)
2   {
3       // проверяем корректность SID
4       if (FALSE == IsValidSid(pSid)) return FALSE;
5
6       LPTSTR lpszName = NULL;
7       DWORD cch = 0, cchDomainName = 0;
8
9       // определяем размер буфера
10      LookupAccountSid(NULL, pSid, NULL, &cch, NULL,
    &cchDomainName, NULL);
11      DWORD cb = (cch + cchDomainName) * sizeof(TCHAR);
12
13      // выделяем блок памяти под буфер
14      if (cb > 0) lpszName = (LPTSTR)LocalAlloc(LMEM_FIXED, cb);
15
16      BOOL bRet = FALSE;
17      SID_NAME_USE SidType;
18
19      if (NULL != lpszName)
20      {
21          // определяем имя учетной записи
22          bRet = LookupAccountSid(NULL, pSid, lpszName +
    cchDomainName, &cch, lpszName, &cchDomainName, &SidType);
23      } // if
24
25      if (FALSE != bRet)
26      {
27          if (SidTypeDomain != SidType)
```

```
28          {
29              if (cchDomainName > 0) lpszName[cchDomainName] =
    TEXT('\\');
30              else StringCbCopy(lpszName, cb, lpszName + 1);
31          } // if
32
33          *lpName = lpszName; // возвращаем полученную строку
34      } // if
35      else
36      {
37          // если не удалось получить имя, преобразуем SID в строку
38          ConvertSidToStringSid(pSid, lpName);
39
40          // освобождаем выделенную память
41          if (NULL != lpszName) LocalFree(lpszName);
42      } // else
43
44      return bRet;
45  } // GetAccountName
```

Функция `GetAccountName` из представленного примера возвращает адрес созданного буфера для имени учетной записи. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`.

Привилегии и права учетной записи

Многие выполняемые операции не могут быть авторизованы через защиту доступа к объекту, поскольку они не касаются взаимодействия с защищаемым объектом. Например, возможность изменения системного времени является свойством учетной записи, а не защищаемого объекта. Для управления связанными с безопасностью возможностями учетных записей, в Windows используются привилегии и права учетной записи.

Привилегии (privileges) являются правом выполнять под той или иной учетной записью конкретную операцию, связанную с операционной системой, например выключение компьютера или изменение системного времени. *Права учетной записи* (account rights) разрешают или запрещают учетной записи, которой они назначены, выполнять конкретный тип входа на компьютер или в домен, например локальный или интерактивный.

Назначать привилегии и права учетным записям можно с помощью таких инструментальных средств, как «Пользователи и группы Active Directory» (Active Directory Users and Groups) для доменных учетных записей или редактор локальной политики безопасности

(%WINDIR%\system32\secpol.msc). На рис. 5.3 показывается раздел «Назначение прав пользователя» (User Rights Assignment) в окне «Локальная политика безопасности» (Local Security Policy Editor), где выведен полный список привилегий и прав учетных записей. Заметим, что в этом списке не делается никаких различий между привилегиями и правами учетных записей. Однако отличить их друг от друга можно достаточно просто, поскольку привилегии учетных записей не содержат в названии слов «вход в».

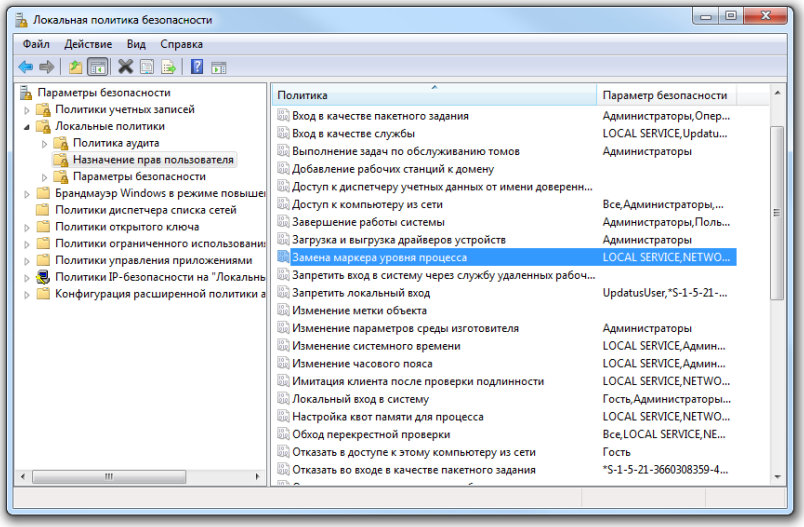

Рис. 5.3. Список привилегий и прав учетных записей Windows

Права учетной записи

При попытке пользователя войти на компьютер *локальная система безопасности* (Local Security Authority, LSA) сверяет тип входа с правами учетной записи (табл. 5.11), назначенными учетной записи этого пользователя. Если у учетной записи нет прав, разрешающих данный тип входа, LSA отклоняет вход.

Таблица 5.11. Права учетной записи

Название	Описание
`SeInteractiveLogonRight`	*Локальный вход в систему.* Данное право позволяет пользователю входить на компьютер

Окончание табл. 5.11.

Название	Описание
SeDenyInteractiveLogonRight	*Запретить локальный вход*. Данное право используется для запрета пользователю входить на компьютер
SeNetworkLogonRight	*Доступ к компьютеру из сети*. Данное право позволяет пользователю подключаться к компьютеру по сети
SeDenyNetworkLogonRight	*Отказать в доступе к этому компьютеру из сети*. Данное право используется для запрета пользователю подключаться к компьютеру по сети
SeBatchLogonRight	*Вход в качестве пакетного задания*. Данное право позволяет пользователю входить на компьютер при помощи средств обработки пакетных заданий и предоставляется только для совместимости со старыми версиями Windows
SeDenyBatchLogonRight	*Отказать во входе в качестве пакетного задания*. Данное право используется для запрета пользователю входить на компьютер при помощи средств обработки пакетных заданий
SeServiceLogonRight	*Вход в качестве службы*. Данное право позволяет пользователю запускать от своего имени службы Windows
SeDenyServiceLogonRight	*Отказать во входе в качестве службы*. Данное право используется для запрета пользователю запускать от своего имени службы Windows
SeRemoteInteractiveLogonRight	*Разрешать вход в систему через службы удаленных рабочих столов*. Данное право позволяет пользователю входить на компьютер с использованием удаленного доступа
SeDenyRemoteInteractiveLogonRight	*Запретить вход в систему через службы удаленных рабочих столов*. Данное право используется для запрета пользователю входить на компьютер с использованием удаленного доступа

Следует отметить, что запрещающее право подавляет соответствующее разрешающее право, если к учетной записи применяются оба.

Привилегии

В отличие от прав учетной записи, которые задаются LSA в одном месте, различные привилегии определяются разными компонентами Windows, которые их же и применяют. В табл. 5.12 представлен полный список привилегий и дано их краткое описание.

Таблица 5.12. Привилегии

Название	Описание
`SeAssignPrimaryTokenPrivilege`	*Замена маркера уровня процесса*. Данная привилегия позволяет вызывать функцию `CreateProcessAsUser` для того, чтобы одна служба могла запускать другую
`SeAuditPrivilege`	*Создание аудитов безопасности*. Данная привилегия позволяет добавление записей в журнал безопасности
`SeBackupPrivilege`	*Архивация файлов и каталогов*. Данная привилегия позволяет игнорировать разрешения для файлов, каталогов, реестра и других постоянных объектов с целью архивации системы
`SeChangeNotifyPrivilege`	*Обход перекрестной проверки*. Данная привилегия позволяет производить обзор деревьев каталога, даже если для этого отсутствуют разрешения на каталог
`SeCreateGlobalPrivilege`	*Создание глобальных объектов*. Данная привилегия позволяет создавать глобальные объекты, доступные для всех пользовательских сеансов
`SeCreatePagefilePrivilege`	*Создание файла подкачки*. Данная привилегия позволяет создание и изменение размера файла подкачки
`SeCreatePermanentPrivilege`	*Создание постоянных общих объектов*. Данная привилегия позволяет создание объекта каталога при помощи диспетчера объектов
`SeCreateSymbolicLinkPrivilege`	*Создание символических ссылок*. Данная привилегия определяет для пользователя возможность создавать символические ссылки с компьютера, на который он вошел
`SeCreateTokenPrivilege`	*Создание маркерного объекта*. Данная привилегия позволяет создание маркеров доступа, которые затем могут быть использованы для получения доступа к любым локальным защищаемым объектам

Продолжение табл. 5.12.

Название	Описание
SeDebugPrivilege	*Отладка программ*. Данная привилегия позволяет подключать отладчик к любому процессу или ядру. Эту привилегию не нужно назначать разработчикам, выполняющим отладку собственных приложений
SeEnableDelegationPrivilege	*Разрешение доверия к учетным записям компьютеров и пользователей при делегировании*. Данная привилегия позволяет устанавливать параметр «Делегирование разрешено» для пользователя или объекта компьютера
SeImpersonatePrivilege	*Имитация клиента после проверки подлинности*. Данная привилегия позволяет программам, выполняемым от имени локального пользователя, олицетворять клиента
SeIncreaseBasePriorityPrivilege	*Увеличение приоритета выполнения*. Данная привилегия позволяет повышение приоритета выполнения другого процесса
SeIncreaseQuotaPrivilege	*Настройка квот памяти для процесса*. Данная привилегия позволяет изменять максимальный объем памяти, используемый процессом
SeIncreaseWorkingSetPrivilege	*Увеличение рабочего набора процесса*. Данная привилегия позволяет увеличивать и уменьшать размер рабочего набора процесса
SeLoadDriverPrivilege	*Загрузка и выгрузка драйверов устройств*. Данная привилегия позволяет динамически загружать и выгружать драйверы устройств или другой код в режиме ядра
SeLockMemoryPrivilege	*Блокировка страниц в памяти*. Данная привилегия позволяет процессам использовать для сохранения данных в физической памяти для предотвращения сброса этих данных в виртуальную память
SeMachineAccountPrivilege	*Добавление рабочих станций к домену*. Данная привилегия позволяет добавлять компьютеры в домен
SeManageVolumePrivilege	*Выполнение задач по обслуживанию томов*. Данная привилегия позволяет выполнять задачи по обслуживанию томов, например, удаленную дефрагментацию

Продолжение табл. 5.12.

Название	Описание
SeProfileSingleProcessPrivilege	*Профилирование одного процесса*. Данная привилегия позволяет использовать средства мониторинга производительности для отслеживания производительности несистемных процессов
SeRelabelPrivilege	*Изменение метки объекта*. Данная привилегия позволяет изменять метки целостности защищаемых объектов, владельцами которых являются другие пользователи
SeRemoteShutdownPrivilege	*Принудительное удаленное завершение работы*. Данная привилегия позволяет удаленное завершение работы компьютера
SeRestorePrivilege	*Восстановление файлов и каталогов*. Данная привилегия позволяет обойти разрешения на файлы, каталоги, реестр и другие постоянные объекты при восстановлении архивных копий файлов и каталогов
SeSecurityPrivilege	*Управление аудитом и журналом безопасности*. Данная привилегия позволяет указывать параметры аудита доступа к защищаемым объектам
SeShutdownPrivilege	*Завершение работы системы*. Данная привилегия позволяет завершить работу операционной системы
SeSyncAgentPrivilege	*Синхронизация данных службы каталогов*. Данная привилегия позволяет синхронизировать все данные службы каталогов
SeSystemEnvironmentPrivilege	*Изменение параметров среды изготовителя*. Данная привилегия позволяет изменять значения параметров аппаратной среды
SeSystemProfilePrivilege	*Профилирование производительности системы*. Данная привилегия позволяет использовать средства мониторинга производительности для отслеживания производительности системных процессов
SeSystemtimePrivilege	*Изменение системного времени*. Данная привилегия позволяет изменять время и дату внутренних часов компьютера
SeTakeOwnershipPrivilege	*Смена владельцев файлов и других объектов*. Данная привилегия позволяет тать владельцем любого защищаемого объекта

Окончание табл. 5.12.

Название	Описание
SeTcbPrivilege	*Работа в режиме операционной системы*. Данная привилегия позволяет процессу олицетворять любого пользователя без проверки подлинности. Процесс, таким образом, может получать доступ к тем же локальным ресурсам, что и пользователь
SeTimeZonePrivilege	*Изменение часового пояса*. Данная привилегия позволяет изменять часовой пояс, используемый компьютером для отображения местного времени
SeTrustedCredManAccessPrivilege	*Доступ к диспетчеру учетных данных от имени доверенного вызывающего*. Данная привилегия не должна предоставляться учетным записям, так как она используется диспетчером учетных данных в ходе архивации и восстановления
SeUndockPrivilege	*Отключение компьютера от стыковочного узла*. Данная привилегия позволяет отстыковать портативный компьютер от стыковочного узла без входа в систему

Заметим, что некоторые привилегии настолько влиятельны, что учетная запись, которой они назначены, фактически имеет полный контроль над компьютером. Эти привилегии могут использоваться для получения несанкционированного доступа к защищаемым объектам и для выполнения несанкционированных операций.

Информацию о том, какими привилегиями обладают по умолчанию те или иные учетные записи, а также подробное описание всех привилегий можно найти в документации Platform SDK.

Работа с привилегиями и правами учетных записей

Для просмотра и редактирования привилегий и прав учетных записей используются функции Win32 API, которые работают с базой данных системы LSA. Прототипы этих функций определены в заголовочном файле NTSecAPI.h.

Следует отметить, что в функциях из NTSecAPI.h для работы со строками используется структура `LSA_UNICODE_STRING`, которая определяет строку, состоящую из символов Unicode. В отличие от обычных строк языка C/C++, где конец обозначается нуль-символом, длина строки, определяемой структурой `LSA_UNICODE_STRING`, задается явно.

```
typedef struct _LSA_UNICODE_STRING {
  USHORT Length; // длина строки (в байтах)
  USHORT MaximumLength; // максимальная длина строки (в байтах)
  PWSTR Buffer; // Unicode-строка
} LSA_UNICODE_STRING, *PLSA_UNICODE_STRING;
```

Заметим, что длина строки задается в байтах, а не в символах, как это принято в C/C++.

В приводимых примерах будет использоваться функция `InitUnicodeString` (листинг 5.4), которая инициализирует структуру `LSA_UNICODE_STRING` на основе обычной строки C/C++.

Листинг 5.4. Функция инициализации структуры `LSA_UNICODE_STRING`

```
1  void InitUnicodeString(LPCWSTR lpString, PLSA_UNICODE_STRING
   pUnicodeString)
2  {
3      size_t cch = wcslen(lpString); // определяем длину строки
4
5      pUnicodeString->Length = cch * sizeof(WCHAR);
6      pUnicodeString->MaximumLength = (cch + 1) * sizeof(WCHAR);
7      pUnicodeString->Buffer = (PWSTR)lpString;
8  } // InitUnicodeString
```

Также следует отметить, что подавляющее количество функций из NTSecAPI.h возвращают значение типа `NTSTATUS`. Для проверки значения типа `NTSTATUS` служит макрос `LSA_SUCCESS`. Можно также использовать функцию `LsaNtStatusToWinError`, которая преобразует значение `NTSTATUS` в привычный код ошибки. Функция `LsaNtStatusToWinError` имеет следующий прототип:

```
ULONG LsaNtStatusToWinError(NTSTATUS Status);
```

Если соответствующего кода ошибки не существует, функция `LsaNtStatusToWinError` вернет значение `ERROR_MR_MID_NOT_FOUND`.

Открытие и закрытие объекта политики безопасности

Для того чтобы работать с привилегиями правами учетных записей сперва необходимо открыть объект политики безопасности вызовом функции `LsaOpenPolicy`:

```
NTSTATUS LsaOpenPolicy(PLSA_UNICODE_STRING SystemName,
    PLSA_OBJECT_ATTRIBUTES ObjectAttributes,
    ACCESS_MASK DesiredAccess, PLSA_HANDLE PolicyHandle);
```

Первый параметр, *SystemName*, указывает на структуру `LSA_UNICODE_STRING`, содержащую имя компьютера, на котором открывается объект

политики безопасности. Имя компьютера может быть задано, например, так «Андрей-ПК» или так «\\Андрей-ПК». Если этот параметр установлен в NULL, открывается объект политики безопасности локального компьютера.

Второй параметр, *ObjectAttributes*, указывает на структуру LSA_OBJECT_ATTRIBUTES, задающую атрибуты объекта. На самом деле функция LsaOpenPolicy не использует этот параметр, так что структуру LSA_OBJECT_ATTRIBUTES надо просто заполнить нулями.

Третий параметр, *DesiredAccess*, определяет запрашиваемые права доступа к объекту политики безопасности. Некоторые из возможных значений этого параметра перечислены в табл. 5.13. Полный перечень и подробное описание значений параметра *DesiredAccess* можно найти в документации Platform SDK.

Таблица 5.13. Права доступа к объекту политики безопасности

Значение	**Описание**
POLICY_ALL_ACCESS	Все возможные права доступа
POLICY_AUDIT_LOG_ADMIN	Необходимо для изменения параметров аудита
POLICY_CREATE_ACCOUNT	Необходимо для создания новой учетной записи
POLICY_GET_PRIVATE_INFORMATION	Необходимо для просмотра конфиденциальной информации
POLICY_LOOKUP_NAMES	Необходимо для преобразования между именами и SID учетных записей
POLICY_TRUST_ADMIN	Необходимо для изменения первичной информации о домене
POLICY_VIEW_AUDIT_INFORMATION	Необходимо для просмотра информации из журнала аудита
POLICY_VIEW_LOCAL_INFORMATION	Необходимо для просмотра информации о политике безопасности

Последний параметр, *PolicyHandle*, указывает на переменную типа LSA_HANDLE, через которую функция LsaOpenPolicy возвращает дескриптор объекта политики безопасности.

Функция LsaOpenPolicy завершится ошибкой ERROR_ACCESS_DENIED (*отказано в доступе*), если вызывающий процесс не будет запущен от имени «Администратора».

В листинге 5.5 представлен пример реализации функций, которая возвращает дескриптор объекта политики безопасности для локального компьютера.

Листинг 5.5. Функция, открывающая объект политики безопасности локального компьютера

```
1   LSA_HANDLE OpenLocalPolicy(ACCESS_MASK DesiredAccess)
2   {
3       LSA_HANDLE lsahPolicy;
4       LSA_OBJECT_ATTRIBUTES ObjectAttributes;
5
6       // заполняем структуру LSA_OBJECT_ATTRIBUTES нулями
7       ZeroMemory(&ObjectAttributes, sizeof(ObjectAttributes));
8
9       // получаем дескриптор объекта политики безопасности
10      NTSTATUS ntsResult = LsaOpenPolicy(NULL, &ObjectAttributes,
    DesiredAccess, &lsahPolicy);
11
12      // сохраняем код последней ошибки
13      SetLastError(LsaNtStatusToWinError(ntsResult));
14
15      // возвращаем дескриптор объекта политики безопасности
16      return LSA_SUCCESS(ntsResult) ? lsahPolicy : NULL;
17  } // OpenLocalPolicy
```

Когда дескриптор объекта политики безопасности станет более не нужен, его следует закрыть с помощью функции `LsaClose`.

```
NTSTATUS LsaClose(LSA_HANDLE ObjectHandle);
```

Настройка привилегий и прав учетных записей

В Win32 API есть две функции, `LsaAddAccountRights` и `LsaRemoveAccountRights`, с помощью которых можно соответственно добавлять и удалять привилегии и права из учетной записи. Эти функции имеют следующие прототипы:

```
NTSTATUS LsaAddAccountRights(LSA_HANDLE PolicyHandle,
    PSID AccountSid, PLSA_UNICODE_STRING UserRights,
    ULONG CountOfRights);

NTSTATUS LsaRemoveAccountRights(LSA_HANDLE PolicyHandle,
    PSID AccountSid, BOOLEAN AllRights,
    PLSA_UNICODE_STRING UserRights, ULONG CountOfRights);
```

Первый параметр, *PolicyHandle*, – дескриптор объекта политики безопасности, который должен обладать правом доступа `POLICY_LOOKUP_NAMES`.

Второй параметр, *AccountSid*, указывает на буфер, который содержит SID учетной записи пользователя или группы, для которой необходимо настроить права и привилегии.

Последние два параметра, *UserRights* и *CountOfRights*, задают соответственно список привилегий или прав учетной записи, которые нужно добавить/удалить, и количество элементов в этом списке.

Параметр *AllRights* в функции LsaRemoveAccountRights используется для удаления всех прав из учетной записи. Для этого данный параметр должен принимать значение TRUE. При этом список, заданный параметром *UserRights*, будет игнорироваться.

В листингах 5.6,5.7 представлены примеры реализаций функций, которые можно использовать для настройки привилегий и прав учетной записи.

Листинг 5.6. Добавление привилегии или права учетной записи

```
1   BOOL AddAccountRight(LSA_HANDLE lsahPolicy, PSID pSid,
    LPCWSTR UserRight)
2   {
3       LSA_UNICODE_STRING right;
4       InitUnicodeString(UserRight, &right);
5
6       // добавляем привилегию/право учетной записи
7       NTSTATUS ntsResult = LsaAddAccountRights(lsahPolicy, pSid,
    &right, 1);
8
9       // сохраняем код последней ошибки
10      SetLastError(LsaNtStatusToWinError(ntsResult));
11
12      return LSA_SUCCESS(ntsResult) ? TRUE : FALSE;
13  } // AddAccountRight
```

Листинг 5.7. Удаление привилегии или права учетной записи

```
1   BOOL RemoveAccountRight(LSA_HANDLE lsahPolicy, PSID pSid,
    LPCWSTR UserRight)
2   {
3       LSA_UNICODE_STRING right;
4       InitUnicodeString(UserRight, &right);
5
6       // удаляем привилегию/право учетной записи
7       NTSTATUS ntsResult = LsaRemoveAccountRights(lsahPolicy, pSid,
    FALSE, &right, 1);
8
9       // сохраняем код последней ошибки
10      SetLastError(LsaNtStatusToWinError(ntsResult));
11
12      return LSA_SUCCESS(ntsResult) ? TRUE : FALSE;
13  } // RemoveAccountRight
```

Для указания имен привилегий и прав учетных записей можно использовать специальные константы (табл. 5.14,5.15), определенные в заголовочных файлах NTSecAPI.h и WinNT.h. Как правило, в документации Platform SDK используются именно константы.

Таблица 5.14. Константы прав учетных записей

Название	**Константа**
SeBatchLogonRight	SE_BATCH_LOGON_NAME
SeDenyBatchLogonRight	SE_DENY_BATCH_LOGON_NAME
SeDenyInteractiveLogonRight	SE_DENY_INTERACTIVE_LOGON_NAME
SeDenyNetworkLogonRight	SE_DENY_NETWORK_LOGON_NAME
SeDenyRemoteInteractiveLogonRight	SE_DENY_REMOTE_INTERACTIVE_LOGON_NAME
SeDenyServiceLogonRight	SE_DENY_SERVICE_LOGON_NAME
SeInteractiveLogonRight	SE_INTERACTIVE_LOGON_NAME
SeNetworkLogonRight	SE_NETWORK_LOGON_NAME
SeRemoteInteractiveLogonRight	SE_REMOTE_INTERACTIVE_LOGON_NAME
SeServiceLogonRight	SE_SERVICE_LOGON_NAME

Таблица 5.15. Константы привилегий

Название	**Константа**
SeAssignPrimaryTokenPrivilege	SE_ASSIGNPRIMARYTOKEN_NAME
SeAuditPrivilege	SE_AUDIT_NAME
SeBackupPrivilege	SE_BACKUP_NAME
SeChangeNotifyPrivilege	SE_CHANGE_NOTIFY_NAME
SeCreateGlobalPrivilege	SE_CREATE_GLOBAL_NAME
SeCreatePagefilePrivilege	SE_CREATE_PAGEFILE_NAME
SeCreatePermanentPrivilege	SE_CREATE_PERMANENT_NAME
SeCreateSymbolicLinkPrivilege	SE_CREATE_SYMBOLIC_LINK_NAME
SeCreateTokenPrivilege	SE_CREATE_TOKEN_NAME
SeDebugPrivilege	SE_DEBUG_NAME
SeEnableDelegationPrivilege	SE_ENABLE_DELEGATION_NAME
SeImpersonatePrivilege	SE_IMPERSONATE_NAME
SeIncreaseBasePriorityPrivilege	SE_INC_BASE_PRIORITY_NAME
SeIncreaseQuotaPrivilege	SE_INCREASE_QUOTA_NAME
SeIncreaseWorkingSetPrivilege	SE_INC_WORKING_SET_NAME

Окончание табл. 5.15.

Название	Константа
SeLoadDriverPrivilege	SE_LOAD_DRIVER_NAME
SeLockMemoryPrivilege	SE_LOCK_MEMORY_NAME
SeMachineAccountPrivilege	SE_MACHINE_ACCOUNT_NAME
SeManageVolumePrivilege	SE_MANAGE_VOLUME_NAME
SeProfileSingleProcessPrivilege	SE_PROF_SINGLE_PROCESS_NAME
SeRelabelPrivilege	SE_RELABEL_NAME
SeRemoteShutdownPrivilege	SE_REMOTE_SHUTDOWN_NAME
SeRestorePrivilege	SE_RESTORE_NAME
SeSecurityPrivilege	SE_SECURITY_NAME
SeShutdownPrivilege	SE_SHUTDOWN_NAME
SeSyncAgentPrivilege	SE_SYNC_AGENT_NAME
SeSystemEnvironmentPrivilege	SE_SYSTEM_ENVIRONMENT_NAME
SeSystemProfilePrivilege	SE_SYSTEM_PROFILE_NAME
SeSystemtimePrivilege	SE_SYSTEMTIME_NAME
SeTakeOwnershipPrivilege	SE_TAKE_OWNERSHIP_NAME
SeTcbPrivilege	SE_TCB_NAME
SeTimeZonePrivilege	SE_TIME_ZONE_NAME
SeTrustedCredManAccessPrivilege	SE_TRUSTED_CREDMAN_ACCESS_NAME
SeUndockPrivilege	SE_UNDOCK_NAME

Перечисление привилегий и прав учетной записи

Чтобы получить список привилегий и прав, которыми обладает учетная запись, следует использовать функцию `LsaEnumerateAccount-Rights`:

```
NTSTATUS LsaEnumerateAccountRights(LSA_HANDLE PolicyHandle,
    PSID AccountSid, PLSA_UNICODE_STRING *UserRights,
    PULONG CountOfRights);
```

Первый параметр, *`PolicyHandle`*, – дескриптор объекта политики безопасности, который должен обладать правом доступа `POLICY_LOOKUP_NAMES`.

Второй параметр, *`AccountSid`*, указывает на буфер, который содержит SID учетной записи пользователя или группы, для которой необходимо получить список привилегий и прав.

Третий параметр, *UserRights*, указывает на переменную типа PLSA_UNICODE_STRING, в которую сохраняется список привилегий и прав учетной записи.

Последний параметр, *CountOfRights*, указывает на переменную типа ULONG, через которую функция LsaEnumerateAccountRights возвращает количество элементов в списке привилегий и прав учетной записи.

После того как полученный список прав учетной записи станет более не нужен, следует освободить память выделенную для буфера, который содержит данный список. Для этого нужно вызвать функцию LsaFreeMemory:

```
NTSTATUS LsaFreeMemory(PVOID Buffer);
```

Параметр *Buffer* указывает на буфер, для которого нужно освободить выделенную память.

В следующем пример показано, как использовать функцию LsaEnumerateAccountRights для перечисления привилегий и прав учетной записи. В данном примере SID учетной записи указывается переменной pSid.

Листинг 5.8. Перечисление привилегий и прав учетной записи

```
1   // открываем дескриптор объекта политики безопасности локального
    компьютера
2   LSA_HANDLE lsahPolicy = OpenLocalPolicy(POLICY_LOOKUP_NAMES);
3
4   if (NULL != lsahPolicy)
5   {
6       PLSA_UNICODE_STRING UserRights; // список привилегий и прав
7       ULONG nCountOfRights; // количество элементов в списке
8
9       // получаем список привилегий и прав учетной записи
10      NTSTATUS ntsResult = LsaEnumerateAccountRights(lsahPolicy,
    pSid, &UserRights, &nCountOfRights);
11
12      if (LSA_SUCCESS(ntsResult))
13      {
14          for (ULONG i = 0; i < nCountOfRights; ++i)
15          {
16              // определяем имя привилегии/права учетной записи
17              LPCWSTR sUserRight = UserRights[i].Buffer;
18
19              /* здесь можно работать с полученным именем ... */
20          } // for
21
```

```
22          // освобождаем память выделенную для списка
23          LsaFreeMemory(UserRights);
24      } // if
25
26      // закрываем дескриптор объекта политики безопасности
27      LsaClose(lsahPolicy);
28  } // if
```

Для преобразования полученного имени привилегии в дружественное имя (display name), которое повсеместно используется в пользовательском интерфейсе Windows (см. рис. 5.3), можно использовать функция `LookupPrivilegeDisplayName`:

```
BOOL LookupPrivilegeDisplayName(LPCTSTR lpSystemName,
    LPCTSTR lpName, LPTSTR lpDisplayName,
    LPDWORD cchDisplayName, LPDWORD lpLanguageId);
```

Первый параметр, *`lpSystemName`*, указывает на строку, содержащую имя компьютера, на котором будет вестись поиск дружественного имени. Если этот параметр установлен в `NULL`, поиск ведется на локальном компьютере.

Второй параметр, *`lpName`*, указывает на строку, которая содержит имя привилегии.

Третий параметр, *`lpDisplayName`*, указывает на строку, в которую будет сохранено дружественное имя привилегии. Четвертый параметр, *`cchDisplayName`*, указывает на переменную типа `DWORD`, которая определяет размер результирующей строки (в символах). Функция запишет в эту переменную число символов записанных в результирующую строку (нуль-символ в конце не учитывается). Если необходимо определить размер результирующей строки, значение переменной, на которую указывает параметр *`cchDisplayName`*, должно быть равным нулю. В этом случае в переменную будет сохранен необходимый размер буфера в символах (с учетом нуль-символа в конце).

Последний параметр, *`lpLanguageId`*, указывает на переменную типа `DWORD`, в которую будет сохранен идентификатор языка (language identifier) для возвращаемого дружественного имени.

При успешном выполнении функция `LookupPrivilegeDisplayName` возвращает значение отличное от `FALSE`.

Идентификаторы привилегий

Несмотря на то, что каждая привилегия имеет уникальное имя, все же при вызове некоторых функций Win32 API может использоваться *локально уникальный идентификатор* (locally unique identifier, LUID), который представляет собой уникальное 64-битное значение.

Для преобразования имен привилегий в LUID и обратно используются функции LookupPrivilegeValue и LookupPrivilegeName:

```
BOOL LookupPrivilegeValue(LPCTSTR lpSystemName,
    LPCTSTR lpName, PLUID lpLuid);

BOOL LookupPrivilegeName(LPCTSTR lpSystemName, PLUID lpLuid,
    LPTSTR lpName, LPDWORD cchName);
```

Первый параметр, *lpSystemName*, указывает на строку, содержащую имя компьютера, на котором будет вестись поиск LUID. Если этот параметр установлен в NULL, поиск ведется на локальном компьютере.

Параметр *lpName* указывает на строку, которая содержит имя привилегии или же в которую будет сохранено имя привилегии.

Параметр *lpLuid* указывает на переменную типа LUID, которая содержит LUID или же в которую будет сохранен LUID.

Параметр *cchName* указывает на переменную типа DWORD, которая определяет размер результирующей строки (в символах). Функция запишет в эту переменную число символов записанных в результирующую строку (нуль-символ в конце не учитывается). При необходимости определить размер результирующей строки, значение переменной, на которую указывает параметр *cchName*, должно быть равным нулю. В этом случае в переменную будет сохранен необходимый размер буфера в символах (с учетом нуль-символа в конце).

Если функции LookupPrivilegeValue и LookupPrivilegeName завершаются успешно, возвращаемое значение отлично от FALSE.

Дескриптор безопасности

Все защищаемые объекты имеют *дескриптор безопасности* (security descriptor), который позволяет обеспечивать контроль доступа к таким объектам. При запросе доступа к защищаемому объекту операционная система просматривает дескриптор безопасности этого объекта, чтобы убедиться в дозволенности запрашиваемого доступа.

Дескриптор безопасности представляет собой структуру, которая состоит из следующих компонентов:

- *Владелец* (owner) – учетная запись пользователя, которому можно изменять разрешения на доступ к объекту даже при отсутствии доступа к нему. По умолчанию владельцем объекта становится его создатель. Владельцем объекта может стать любой пользователь из группы «Администраторы» или пользователи, обладающий разрешением «смена владельца» для данного объекта.
- *Основная группа* (primary group) – группа, члены которой могут изменять разрешения на доступ к объекту даже при отсутствии до-

ступа к нему. Только одна группа пользователей может являться основной.

- *Список разграниченного контроля доступа* (Discretionary Access Control List, DACL) – список, используемый для принятия решений при контроле доступа к защищаемому объекту.
- *Системный список контроля доступа* (System Access Control List, SACL) – список, используемый для аудита успешных и неудачных попыток доступа к защищаемому объекту.

На рис. 5.4 с помощью утилиты WinObj показан дескриптор безопасности для объекта ядра «мьютекс» в сеансе пользователя. Аналогичный графический интерфейс для просмотра и редактирования дескриптора безопасности используется также проводником Windows для файлов и каталогов в файловой системе NTFS (рис. 5.5) и редактором реестра Windows (рис. 5.6).

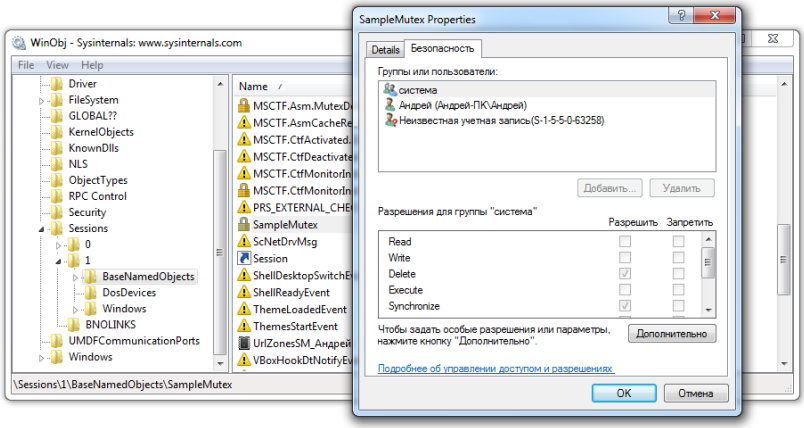

Рис. 5.4. Объект ядра «мьютекс» и его дескриптор безопасности, просматриваемый с помощью утилиты WinObj

Заметим, то, что показано на рис. 5.4,5.6, на самом деле является принадлежащим защищаемому объекту списком разграниченного контроля доступа (DACL). Чтобы просмотреть и редактировать все компоненты дескриптора безопасности нужно нажать кнопку **Дополнительно** (**Advanced**). Появится окно «Дополнительные параметры безопасности» (Advanced Security Settings) (рис. 5.7), где компоненты дескриптора безопасности распределены по разным вкладкам.

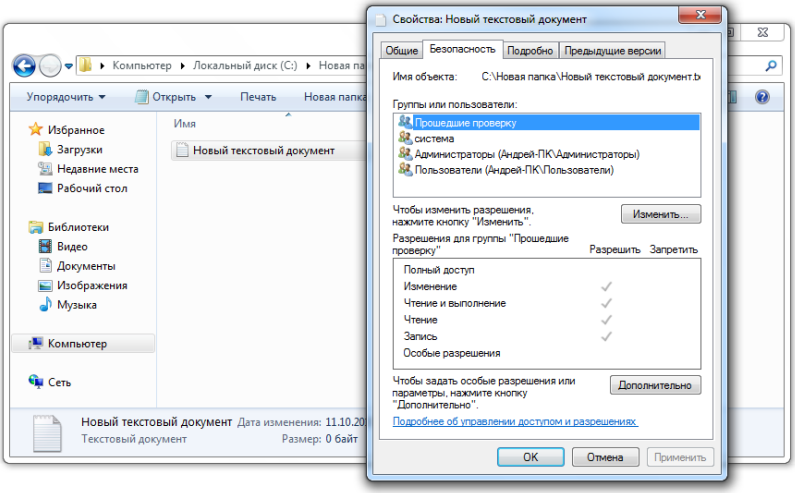

Рис. 5.5. Дескриптор безопасности файла, просматриваемый с помощью проводника Windows

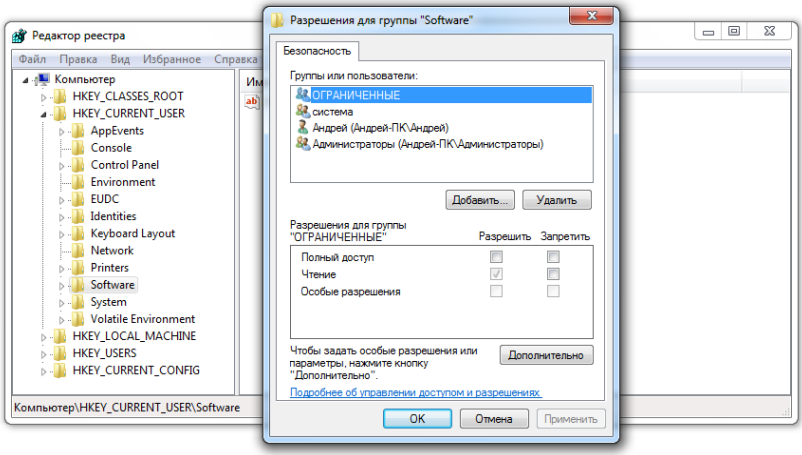

Рис. 5.6. Дескриптор безопасности ключа реестра, просматриваемый с помощью редактора реестра Windows

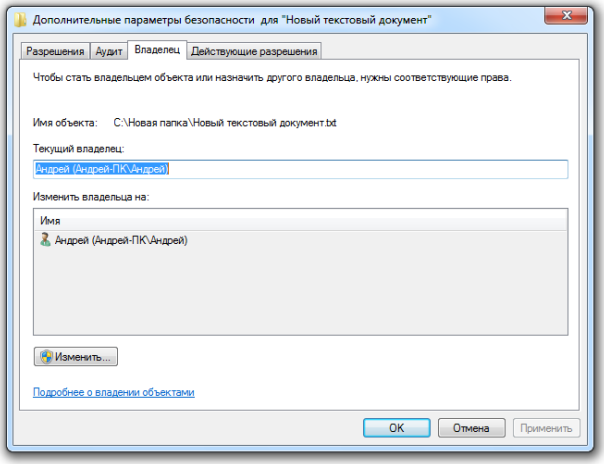

Рис. 5.7. Дополнительные параметры безопасности

Списки контроля доступа

Каждый DACL и SACL состоит из набора *элементов контроля доступа* (Access Control Entry, ACE) различных типов. В DACL используются два типа ACE: для *разрешения доступа* (allowed access) и для *запрета доступа* (denied access). Важно отметить, что все ACE в DACL должны располагаться в определенном порядке:

1. ACE, запрещающие доступ определенным пользователям.
2. ACE, запрещающие доступ пользователям, которые входят в определенные группы.
3. ACE, разрешающие доступ определенным пользователям.
4. ACE, разрешающие доступ пользователям, которые входят в определенные группы.

Если ACE в DACL будут располагаться в произвольном порядке, возможны непредвиденные ситуации при принятии решения предоставления доступа к защищаемому объекту.

В SACL также используются два типа ACE: для *аудита успеха* (audit success) и для *аудита отказа* (audit failure) при запросе доступа к защищаемому объекту. При этом порядок ACE в SACL не имеет значения.

Создание списка контроля доступа

Новый DACL или SACL можно создать с помощью функции `SetEntriesInAcl`, которая реализована в библиотеке Advapi32.dll. Прото-

тип этой функции определен в заголовочном файле AclAPI.h следующим образом:

```
DWORD SetEntriesInAcl(ULONG cCountOfExplicitEntries,
    PEXPLICIT_ACCESS pListOfExplicitEntries, PACL OldAcl,
    PACL *NewAcl);
```

Первый параметр, *cCountOfExplicitEntries*, задает количество структур EXPLICIT_ACCESS в массиве, на который указывает второй параметр, *pListOfExplicitEntries*. Этот массив содержит ACE, которые будут занесены в новый список. Следует отметить, что элементы в этом массиве могут быть расположены в произвольном порядке, так как функция SetEntriesInAcl при необходимости сортирует созданный список должным образом.

Третий параметр, *OldAcl*, указывает на буфер, в котором хранится список контроля доступа. Элементы этого списка будут скопированы в новый список. Этот параметр может быть установлен в NULL.

Последний параметр, *NewAcl*, указывает на переменную типа PACL, через которую функция SetEntriesInAcl вернет адрес созданного буфера для нового списка. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции LocalFree.

При успешном завершении функция SetEntriesInAcl возвращает значение ERROR_SUCCESS, иначе – код возникшей ошибки.

Структура EXPLICIT_ACCESS определена в заголовочном файле AccCtrl.h следующим образом:

```
typedef struct _EXPLICIT_ACCESS {
  DWORD grfAccessPermissions; // права доступа
  ACCESS_MODE grfAccessMode; // режим доступа
  DWORD grfInheritance; // применение
  TRUSTEE Trustee; // доверенный объект (субъект безопасности)
} EXPLICIT_ACCESS, *PEXPLICIT_ACCESS;
```

Первое поле, *grfAccessPermissions*, определяет права доступа, которые должны быть добавлены в ACE. Различные объекты имеют свой набор прав доступа. Например, в случае с файлами и каталогами можно использовать одно из значений (или их комбинацию), перечисленных в табл. 5.16.

Таблица 5.16. Возможные значения поля *grfAccessPermissions*

Значение	**Описание**
DELETE	Удаление
FILE_ADD_FILE	Создание файлов

Окончание табл. 5.16.

Значение	Описание
FILE_ADD_SUBDIRECTORY	Создание каталогов
FILE_ALL_ACCESS	Полный доступ
FILE_APPEND_DATA	Дозапись данных
FILE_DELETE_CHILD	Удаление файлов и каталогов
FILE_EXECUTE	Выполнение файлов
FILE_GENERIC_EXECUTE	Набор из READ_CONTROL, FILE_READ_ATTRIBUTES и FILE_EXECUTE
FILE_GENERIC_READ	Набор из READ_CONTROL, FILE_READ_ATTRIBUTES, FILE_READ_EA и FILE_READ_DATA
FILE_GENERIC_WRITE	Набор из READ_CONTROL, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA, FILE_WRITE_DATA и FILE_APPEND_DATA
FILE_LIST_DIRECTORY	Просмотр списка содержимого в каталоге
FILE_READ_ATTRIBUTES	Чтение атрибутов
FILE_READ_DATA	Чтение данных
FILE_READ_EA	Чтение дополнительных атрибутов
FILE_TRAVERSE	Траверс каталогов
FILE_WRITE_ATTRIBUTES	Запись атрибутов
FILE_WRITE_DATA	Запись данных
FILE_WRITE_EA	Запись дополнительных атрибутов
READ_CONTROL	Чтение разрешений, определение владельца и дескриптора безопасности
WRITE_DAC	Смена разрешений
WRITE_OWNER	Смена владельца

Второе поле, *grfAccessMode*, используется для добавления ACE в список. Кроме того это поле можно использоваться для удаления всех ACE из списка для указанного SID. Основные значения, которые может принимать поле *grfAccessMode*, перечислены в табл. 5.17.

Таблица 5.17. Возможные значения поля *grfAccessMode*

Значение	Описание
GRANT_ACCESS	Добавляет новый ACE, разрешающий доступ для указанного SID, сочетая с уже существующими ACE
SET_ACCESS	Добавляет новый ACE, разрешающий доступ для указанного SID. При этом все другие ACE для указанного SID удаляются

Окончание табл. 5.17.

Значение	Описание
DENY_ACCESS	Добавляет новый ACE, запрещающий доступ для указанного SID, сочетая с уже существующими ACE
REVOKE_ACCESS	Удаляет из списка все существующие ACE, разрешающие доступ или аудит, для указанного SID
SET_AUDIT_SUCCESS	Добавляет новый ACE аудита успеха доступа к защищаемому объекту
SET_AUDIT_FAILURE	Добавляет новый ACE аудита отказа в доступе к защищаемому объекту

Третье поле, *grfInheritance*, определяет, на какие объекты будут распространяться задаваемые права доступа. Это поле может принимать значения, перечисленные в табл. 5.18. Следует отметить, что некоторые из этих значений можно комбинировать.

Таблица 5.18. Возможные значения поля *grfInheritance*

Значение	Описание
NO_INHERITANCE	Задаваемые права доступа распространяются только на объект, к которому применяется созданный список. Это значение не может использоваться совместно с другими значениями
SUB_OBJECTS_ONLY_INHERIT	Задаваемые права доступа распространяются на объект и на все его дочерние не контейнерные объекты
SUB_CONTAINERS_ONLY_INHERIT	Задаваемые права доступа распространяются на объект и на все его дочерние контейнерные объекты
SUB_CONTAINERS_AND_OBJECTS_INHERIT	Комбинация значений SUB_OBJECTS_ONLY_INHERIT и SUB_CONTAINERS_ONLY_INHERIT
INHERIT_NO_PROPAGATE	Задаваемые права доступа распространяются на объект и на дочерние объекты, содержащиеся только в нем
INHERIT_ONLY	Задаваемые права доступа распространяются только на дочерние объекты. Это значение должно использоваться совместно с другими значениями

Последнее поле, *Trustee*, позволяет указать доверенный объект, для которого задаются права доступа к защищаемому объекту. Подробное описание этого поля см. в документации Platform SDK.

В листинге 5.9 продемонстрировано, как создать новый DACL для файла (или каталога) с помощью функции SetEntriesInAcl. Отметим, что в этом примере для получения SID учетной записи «Андрей» ис-

пользуется функция GetAccountSID (см. листинг 5.1), а для получения SID группы «Все» – функция GetWellKnownSID (см. листинг 5.2).

Листинг 5.9. Пример создания нового ACL

```
PSID pSidUser = NULL, pSidEveryone = NULL;

// получаем SID учетной записи «Андрей»
BOOL bRet = GetAccountSID(TEXT("Андрей"), &pSidUser);

if (FALSE != bRet) // получаем SID группы «Все»
    bRet = GetWellKnownSID(WinWorldSid, NULL, &pSidEveryone);

if (FALSE != bRet)
{
    EXPLICIT_ACCESS ea[2] = { };

    // зададим первый ACE
    ea[0].grfAccessPermissions = FILE_ALL_ACCESS; // полный доступ
    ea[0].grfAccessMode = GRANT_ACCESS; // разрешить доступ
    ea[0].grfInheritance= NO_INHERITANCE; // не наследовать
    ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID; // доверенный объект задается с помощью SID
    ea[0].Trustee.ptstrName  = (LPTSTR)pSidEveryone; // SID группы «Все»

    // зададим второй ACE
    ea[1].grfAccessPermissions = DELETE | WRITE_DAC | WRITE_OWNER; // удаление, смена разрешений и владельца
    ea[1].grfAccessMode = DENY_ACCESS; // запретить доступ
    ea[1].grfInheritance= NO_INHERITANCE; // не наследовать
    ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID; // доверенный объект задается с помощью SID
    ea[1].Trustee.ptstrName  = (LPTSTR)pSidUser; // SID учетной записи «Андрей»

    // создаем новый DACL
    PACL pDacl = NULL;
    DWORD dwResult = SetEntriesInAcl(2, ea, NULL, &pDacl);

    if (ERROR_SUCCESS == dwResult) // новый DACL успешно создан
    {
        /* здесь можно работать с новым DACL */

        LocalFree(pDacl); // освобождаем память
    } // if
```

```
37  } // if
38
39  // освобождаем память, выделенную под буферы для SID
40  if (NULL != pSidUser) LocalFree(pSidUser);
41  if (NULL != pSidEveryone) LocalFree(pSidEveryone);
```

Извлечение элементов из списка контроля доступа

Для того чтобы извлечь ACE из DACL или SACL можно использовать функцию `GetExplicitEntriesFromAcl`, которая создает массив структур `EXPLICIT_ACCESS`, описывающих ACE. Эта функция имеет следующий прототип:

```
DWORD GetExplicitEntriesFromAcl(PACL pacl,
    PULONG pcCountOfExplicitEntries,
    PEXPLICIT_ACCESS *pListOfExplicitEntries);
```

Первый параметр, *pacl*, указывает на буфер, в котором храниться список контроля доступа. Элементы этого списка будут скопированы в результирующий массив.

Второй параметр, *pcCountOfExplicitEntries*, указывает на переменную типа `ULONG`, через которую функция вернет количество элементов в результирующем массиве.

Третий параметр, *pListOfExplicitEntries*, указывает на переменную типа `PEXPLICIT_ACCESS`, через которую функция вернет адрес созданного буфера для результирующего массива. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`.

Каждый элемент в результирующем массиве содержит набор прав доступа, тип ACE и SID. При этом тип ACE указан в поле *grfAccessMode*: для DACL используются значения `GRANT_ACCESS` и `DENY_ACCESS`, а для SACL – `SET_AUDIT_SUCCESS` и `SET_AUDIT_FAILURE`. Дополнительное описание этих значений см. в табл. 5.17.

При успешном завершении функция `GetExplicitEntriesFromAcl` возвращает значение `ERROR_SUCCESS`, иначе – код возникшей ошибки.

В следующем примере продемонстрировано, как использовать функцию `GetExplicitEntriesFromAcl` для извлечения ACE:

Листинг 5.10. Чтение списка контроля доступа

```
1   ULONG uCount = 0; // количество элементов в массиве ACE
2   PEXPLICIT_ACCESS pEA = NULL; // массив ACE
3
4   // извлекаем элементы из списка контроля доступа
5   DWORD dwResult = GetExplicitEntriesFromAcl(pAcl, &uCount, &pEA);
6
```

```
7   if (ERROR_SUCCESS == dwResult)
8   {
9       for (ULONG i = 0; i < uCount; ++i)
10      {
11          LPTSTR lpszName = NULL; // имя учетной записи
12
13          // определим имя учетной записи
14          if (TRUSTEE_IS_SID == pEA[i].Trustee.TrusteeForm)
15              GetAccountName(pEA[i].Trustee.ptstrName, &lpszName);
16
17          // определим права доступа
18          DWORD dwAccessPermissions = pEA[i].grfAccessPermissions;
19
20          // определим тип ACE
21          switch (pEA[i].grfAccessMode)
22          {
23          case GRANT_ACCESS:
24              /* ACE, разрешающий доступ */
25              break;
26          case DENY_ACCESS:
27              /* ACE, запрещающий доступ */
28              break;
29          case SET_AUDIT_SUCCESS:
30              /* ACE аудита успеха доступа */
31              break;
32          case SET_AUDIT_FAILURE:
33              /* ACE аудита отказа в доступе */
34              break;
35          } // switch
36
37          // освобождаем выделенную память
38          if (NULL != lpszName) LocalFree(lpszName);
39      } // for
40
41      LocalFree(pEA); // освобождаем выделенную память
42  } // if
```

Удаление элементов из списка контроля доступа

Для того чтобы удалить ACE из DACL или SACL нужно использовать функцию `DeleteAce`, которая имеет следующий прототип:

```
BOOL DeleteAce(PACL pAcl, DWORD dwAceIndex);
```

Первый параметр, *pAcl*, указывает на буфер, в котором хранится список контроля доступа. Второй параметр, *dwAceIndex*, задает поряд-

ковый номер элемента, который необходимо удалить из списка контроля доступа.

Если функция `DeleteAce` завершается успешно, она возвращает значение отличное от `FALSE`.

Работа с дескриптором безопасности

Для работы с дескриптором безопасности используются различные функции Win32 API, которым в качестве параметра передается указатель на буфер, в котором хранится дескриптор безопасности. Различают два формата дескриптора безопасности – *абсолютный* (absolute) и *относительный* (self-relative).

Дескриптор безопасности в абсолютном формате (рис. 5.8, *а*) содержит указатели на буферы, в которых хранятся его компоненты. В относительном формате (рис. 5.8, *б*) дескриптора безопасности и его компоненты хранятся в одном буфере. Абсолютный формат используется при формировании нового дескриптора безопасности, а относительный – при получении существующего дескриптора безопасности.

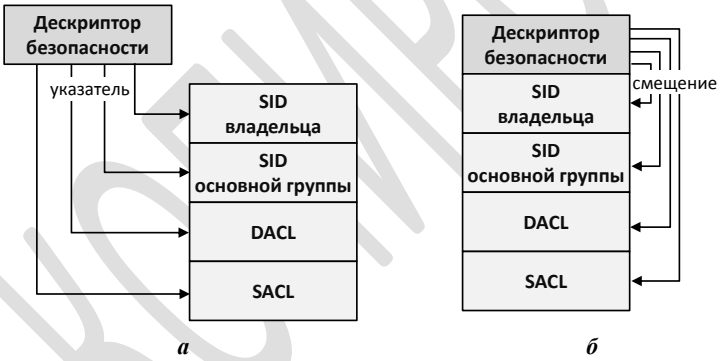

Рис. 5.8. Формат дескриптора безопасности:
а – абсолютный; *б* – относительный

Инициализация дескриптора безопасности

Прежде чем начать работу с дескриптором безопасности в абсолютном формате его нужно инициализировать. Для этого необходимо использовать функцию `InitializeSecurityDescriptor`:

```
BOOL InitializeSecurityDescriptor(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    DWORD dwRevision);
```

Первый параметр, *`pSecurityDescriptor`*, указывает на буфер, в котором хранится дескриптор безопасности в абсолютном формате.

Второй параметр, *dwRevision*, определяет номер версии дескриптора безопасности (на сегодняшний день это 1). Для задания этого параметра можно использовать константу SECURITY_DESCRIPTOR_REVISION.

Если функция InitializeSecurityDescriptor завершается успешно, она возвращает значение отличное от FALSE.

Управляющие флаги дескриптора безопасности

Управляющие флаги дескриптора безопасности, перечисленные в табл. 5.19, определяют, какой смысл присваивается дескриптору безопасности. Доступ к управляющим флагам обеспечивают функции GetSecurityDescriptorControl и SetSecurityDescriptorControl.

Таблица 5.19. Управляющие флаги дескриптора безопасности

Флаг	**Описание**
SE_DACL_AUTO_INHERITED	Если этот флаг установлен, наследственные ACE в DACL из дескриптора безопасности объекта будут распространены на его дочерние объекты
SE_DACL_DEFAULTED	Если этот флаг установлен, дескриптор безопасности содержит используемый по умолчанию DACL
SE_DACL_PRESENT	Если этот флаг установлен, дескриптор безопасности содержит DACL. Если этот флаг не установлен или если этот флаг установлен, а DACL имеет значение NULL, дескриптор безопасности разрешает полный доступ для всех
SE_DACL_PROTECTED	Если этот флаг не установлен, дескриптор безопасности наследует DACL из дескриптора безопасности родительского объекта
SE_GROUP_DEFAULTED	Если этот флаг установлен, дескриптор безопасности содержит SID основной группы, используемым по умолчанию
SE_OWNER_DEFAULTED	Если этот флаг установлен, дескриптор безопасности содержит SID владельца, используемым по умолчанию
SE_SACL_AUTO_INHERITED	Если этот флаг установлен, наследственные ACE в SACL из дескриптора безопасности объекта будут распространены на его дочерние объекты
SE_SACL_DEFAULTED	Если этот флаг установлен, дескриптор безопасности содержит используемый по умолчанию SACL
SE_SACL_PRESENT	Если этот флаг установлен, дескриптор безопасности содержит SACL

Окончание табл. 5.19.

Флаг	Описание
SE_SACL_PROTECTED	Если этот флаг не установлен, дескриптор безопасности наследует SACL из дескриптора безопасности родительского объекта
SE_SELF_RELATIVE	Если этот флаг установлен, дескриптор безопасности находится в относительном формате

Функция GetSecurityDescriptorControl используется для того, чтобы определить какие управляющие флаги используются в дескрипторе безопасности. Функция GetSecurityDescriptorControl имеет следующий прототип:

```
BOOL GetSecurityDescriptorControl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    PSECURITY_DESCRIPTOR_CONTROL pControl,
    LPDWORD lpdwRevision);
```

Первый параметр, *pSecurityDescriptor*, указывает на буфер, в котором хранится дескриптор безопасности.

Второй параметр, *pControl*, указывает на переменную типа SECURITY_DESCRIPTOR_CONTROL, через которую функция вернет набор управляющих флагов указанного дескриптора безопасности.

Последний параметр, *lpdwRevision*, указывает на переменную типа DWORD, в которую функция сохранит номер версии указанного дескриптора безопасности.

Если функция GetSecurityDescriptorControl завершается успешно, она возвращает значение отличное от FALSE.

Следующий небольшой пример демонстрирует, как определить формат дескриптора безопасности:

```
1   SECURITY_DESCRIPTOR_CONTROL wControl;
2   DWORD dwRevision;
3
4   // определим управляющие флаги дескриптора безопасности
5   BOOL bRet = GetSecurityDescriptorControl(pSD, &wControl,
    &dwRevision);
6
7   if (FALSE != bRet && (wControl & SE_SELF_RELATIVE))
8   {
9       /* дескриптор безопасности в относительном формате */
10  } // if
11  else if (FALSE != bRet)
12  {
```

```
13      /* дескриптор безопасности в абсолютном формате */
14  } // if
```

Функция SetSecurityDescriptorControl предназначена для изменения управляющих флагов дескриптора безопасности. Эта функция имеет следующий прототип:

```
BOOL SetSecurityDescriptorControl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    SECURITY_DESCRIPTOR_CONTROL ControlBitsOfInterest,
    SECURITY_DESCRIPTOR_CONTROL ControlBitsToSet);
```

Первый параметр, *pSecurityDescriptor*, указывает на буфер, в котором хранится дескриптор безопасности.

Второй параметр, *ControlBitsOfInterest*, сообщает функции какой флаг (или флаги) указанного дескриптора безопасности необходимо изменить. Третий параметр, *ControlBitsToSet*, устанавливает или сбрасывает флаги.

Если функция SetSecurityDescriptorControl завершается успешно, она возвращает значение отличное от FALSE.

Например, чтобы добавить в DACL дескриптора безопасности ACE, наследуемые от родительских объектов, нужно установить флаг SE_DACL_PROTECTED, а чтобы не добавлять – сбросить этот флаг:

```
1   // установка флага SE_DACL_PROTECTED
2   SetSecurityDescriptorControl(pSD, SE_DACL_PROTECTED,
    SE_DACL_PROTECTED);
3
4   // сброс флага SE_DACL_PROTECTED
5   SetSecurityDescriptorControl(pSD, SE_DACL_PROTECTED, 0);
```

Идентификаторы безопасности и дескриптор безопасности

Полученный SID можно использовать, чтобы изменить в дескрипторе безопасности владельца или основную группу объекта. Для этого нужно вызвать одну из двух функций – SetSecurityDescriptorOwner или SetSecurityDescriptorGroup:

```
BOOL SetSecurityDescriptorOwner(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    PSID pOwner, BOOL bOwnerDefaulted);
BOOL SetSecurityDescriptorGroup(
    PSECURITY_DESCRIPTOR pSecurityDescriptor, PSID pGroup,
    BOOL bGroupDefaulted);
```

Первый параметр, *pSecurityDescriptor*, указывает на буфер, в котором хранится дескриптор безопасности в абсолютном формате.

Второй параметр, *pOwner* (*pGroup*), указывает на буфер, который содержит SID учетной записи пользователя (группы).

Если третий параметр, *bOwnerDefaulted* (*bGroupDefaulted*), принимает значение равное TRUE, функция установит управляющий флаг SE_OWNER_DEFAULTED (SE_GROUP_DEFAULTED) в дескрипторе безопасности.

При успешном завершении функции SetSecurityDescriptorOwner и SetSecurityDescriptorGroup вернут значение отличное от FALSE.

Определить владельца или основную группу объекта можно с помощью функции GetSecurityDescriptorOwner и GetSecurityDescriptorGroup, которые извлекают из дескриптора безопасности SID учетной записи владельца или основной группы. Эти функции имеют следующие прототипы:

```
BOOL GetSecurityDescriptorOwner(
    PSECURITY_DESCRIPTOR pSecurityDescriptor, PSID *pOwner,
    LPBOOL lpbOwnerDefaulted);
BOOL GetSecurityDescriptorGroup(
    PSECURITY_DESCRIPTOR pSecurityDescriptor, PSID *pGroup,
    LPBOOL lpbGroupDefaulted);
```

Первый параметр, *pSecurityDescriptor*, указывает на буфер, в котором хранится дескриптор безопасности.

Второй параметр, *pOwner* (*pGroup*), указывает на переменную типа PSID, через которую функция вернет указатель на буфер, содержащий SID владельца (основной группы).

Третий параметр, *lpbOwnerDefaulted* (*lpbGroupDefaulted*), указывает на переменную типа BOOL, которой будет присвоено значение TRUE, если в указанном дескрипторе безопасности установлен управляющий флаг SE_OWNER_DEFAULTED (SE_GROUP_DEFAULTED); иначе – FALSE.

При успешном завершении функции GetSecurityDescriptorOwner и GetSecurityDescriptorGroup вернут значение отличное от FALSE.

В листинге 5.11 показано, как определить имя учетной записи владельца. Заметим, что в этом примере для получения имени учетной записи используется функция GetAccountName (см. листинг 5.3).

Листинг 5.11. Функция, возвращающая имя учетной записи владельца

```
1  BOOL GetOwnerName(PSECURITY_DESCRIPTOR pSD, LPTSTR *lpName)
2  {
3      PSID pSid;
4      BOOL bDefaulted;
5
6      // получаем SID владельца
7      BOOL bRet = GetSecurityDescriptorOwner(pSD, &pSid,
   &bDefaulted);
```

```
8
9       if (FALSE != bRet)
10      {
11          // определяем имя учетной записи владельца
12          bRet = GetAccountName(pSid, lpName);
13      } // if
14
15      return bRet;
16  } // GetOwnerName
```

Списки контроля доступа и дескриптор безопасности

Чтобы связать DACL и SACL с дескриптором безопасности соответственно нужно использовать функции `SetSecurityDescriptorDacl` и `SetSecurityDescriptorSacl`:

```
BOOL SetSecurityDescriptorDacl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    BOOL bDaclPresent, PACL pDacl, BOOL bDaclDefaulted);
BOOL SetSecurityDescriptorSacl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    BOOL bSaclPresent, PACL pSacl, BOOL bSaclDefaulted);
```

Первый параметр, *pSecurityDescriptor*, указывает на буфер, в котором хранится дескриптор безопасности в абсолютном формате.

Второй параметр, *bDaclPresent* (*bSaclPresent*), определяет наличие списка контроля доступа в дескрипторе безопасности. При этом если этот параметр имеет значение равное `TRUE`, функция установит управляющий флаг `SE_DACL_PRESENT` (`SE_SACL_PRESENT`) в дескрипторе безопасности. Если же значение этого параметра равно `FALSE`, то последующие два параметра игнорируются.

Третий параметр, *pDacl* (*pSacl*), указывает на буфер, который содержит список контроля доступа. Этот параметр игнорируется, если второй параметр принимает значение равное `FALSE`.

Если четвертый параметр, *bDaclPresent* (*bSaclDefaulted*), принимает значение равное `TRUE`, функция установит флаг `SE_DACL_DEFAULTED` (`SE_SACL_DEFAULTED`) в дескрипторе безопасности. Этот параметр игнорируется, если второй параметр принимает значение равное `FALSE`.

При успешном завершении функции `SetSecurityDescriptorDacl` и `SetSecurityDescriptorSacl` вернут значение отличное от `FALSE`.

Для получения DACL и SACL следует воспользоваться функциями `GetSecurityDescriptorDacl` и `GetSecurityDescriptorSacl`:

```
BOOL GetSecurityDescriptorDacl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
```

```
    LPBOOL lpbDaclPresent, PACL *pDacl,
    LPBOOL lpbDaclDefaulted);
BOOL GetSecurityDescriptorSacl(
    PSECURITY_DESCRIPTOR pSecurityDescriptor,
    LPBOOL lpbSaclPresent, PACL *pSacl,
    LPBOOL lpbSaclDefaulted);
```

Первый параметр, *pSecurityDescriptor*, указывает на буфер, в котором хранится дескриптор безопасности.

Второй параметр, *lpbDaclPresent* (*lpbSaclPresent*), указывает на переменную типа BOOL, которой будет присвоено значение TRUE, если в указанном дескрипторе безопасности установлен управляющий флаг SE_DACL_PRESENT (SE_SACL_PRESENT); иначе – FALSE.

Третий параметр, *pDacl* (*pSacl*), указывает на переменную типа PACL, через которую функция вернет указатель на буфер, содержащий список контроля доступа.

Последний параметр, *lpbDaclDefaulted* (*lpbSaclDefaulted*), указывает на переменную типа BOOL, которой будет присвоено значение TRUE, если в указанном дескрипторе безопасности установлен управляющий флаг SE_DACL_DEFAULTED (SE_SACL_DEFAULTED); иначе – FALSE.

При успешном завершении функции GetSecurityDescriptorDacl и GetSecurityDescriptorSacl вернут значение отличное от FALSE.

В листинге 5.12 продемонстрирован пример того, как можно получить DACL и извлечь из него элементы.

Листинг 5.12. Функция, которая извлекает элементы из DACL

```
1   BOOL GetEntriesFromDacl(PSECURITY_DESCRIPTOR pSD,
2       PULONG pcCountOfEntries, PEXPLICIT_ACCESS *pListOfEntries)
3   {
4       PACL pDacl;
5       BOOL bDaclPresent, bDaclDefaulted;
6
7       // получаем DACL
8       BOOL bRet = GetSecurityDescriptorDacl(pSD, &bDaclPresent,
    &pDacl, &bDaclDefaulted);
9
10      if (FALSE != bRet && FALSE != bDaclPresent)
11      {
12          // извлекаем элементы из DACL
13          DWORD dwResult = GetExplicitEntriesFromAcl(pDacl,
    pcCountOfEntries, pListOfEntries);
14          bRet = (ERROR_SUCCESS == dwResult) ? TRUE : FALSE;
15      } // if
16      else
```

```
    {
        *pcCountOfEntries = 0; // возвращаем 0 элементов
    } // else

    return bRet;
} // GetExplicitEntriesFromDacl
```

В листинге 5.13 представлен еще один пример получения DACL. В этом примере показано, как удалить элемент из DACL.

Листинг 5.13. Функция, которая удаляет элемент из DACL

```
BOOL DeleteEntryFromDalc(PSECURITY_DESCRIPTOR pSD, DWORD dwIndex)
{
    PACL pDacl;
    BOOL bDaclPresent, bDaclDefaulted;

    // получаем DACL
    BOOL bRet = GetSecurityDescriptorDacl(pSD, &bDaclPresent,
&pDacl, &bDaclDefaulted);

    if (FALSE != bRet)
    {
        // удаляем элемент из DACL
        bRet = DeleteAce(pDacl, dwIndex);
    } // if

    return bRet;
} // DeleteEntryFromDalc
```

Задание дескриптора безопасности при создании объектов

При создании файлов, каталогов, ключей реестра, различных объектов ядра и т.п., для них можно задать дескриптор безопасности. Для этого используется структура `SECURITY_ATTRIBUTES`, в соответствующем поле которой указывается дескриптор безопасности.

В листинге 5.14 приведен пример создания общедоступного (полный доступ для всех) объекта ядра «мьютекс». Общедоступным этот объект является потому, что в его дескрипторе безопасности установлен управляющий флаг `SE_DACL_PRESENT`, а DACL имеет значение `NULL`.

Листинг 5.14. Создание общедоступного объекта ядра «мьютекс»

```
SECURITY_DESCRIPTOR sd; // дескриптор безопасности

// инициализируем дескриптор безопасности
```

```
4   BOOL bRet = InitializeSecurityDescriptor(&sd,
    SECURITY_DESCRIPTOR_REVISION);
5
6   if (FALSE != bRet)
7       bRet = SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE);
8
9   HANDLE hMutex = NULL; // дескриптор объекта ядра «мьютекс»
10
11  if (FALSE != bRet)
12  {
13      SECURITY_ATTRIBUTES sa = { sizeof(SECURITY_ATTRIBUTES) };
14      sa.lpSecurityDescriptor = &sd; // укажем дескриптор
    безопасности
15
16      // создаем мьютекс в глобальном пространстве имен
17      hMutex = CreateMutexEx(&sa, TEXT("Global\\SampleMutex"), 0,
    MUTEX_ALL_ACCESS);
18  } // if
19
20  if (NULL != hMutex)
21  {
22      /* мьютекс был успешно создан */
23  } // if
```

Заметим, что в этом примере объект ядра создается в глобальном пространстве имен, о чем свидетельствует префикс «Global\» в его имени. Таким образом, созданный объект ядра будет виден всем пользователям операционной системы.

Получение и изменение дескриптора безопасности

Для получения дескриптора безопасности файла или каталога можно использовать функцию `GetFileSecurity`:

```
BOOL GetFileSecurity(LPCTSTR lpFileName,
    SECURITY_INFORMATION RequestedInformation,
    PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD nLength,
    LPDWORD lpnLengthNeeded);
```

Первый параметр, *`lpFileName`*, указывает на строку, содержащую имя файла или каталога, дескриптор безопасности которого необходимо получить.

Второй параметр, *`RequestedInformation`*, определяет, какие компоненты дескриптора безопасности следует получить. Этот параметр может принимать одно (или комбинацию) из значений, перечисленных в табл. 5.20. В этой таблице перечислены не все возможные значения. Полный список значений см. в документации Platform SDK.

Таблица 5.20. Значения параметра *RequestedInformation*

Значение	Описание
OWNER_SECURITY_INFORMATION	Информация о владельце
GROUP_SECURITY_INFORMATION	Информация об основной группе
DACL_SECURITY_INFORMATION	Информация о DACL
SACL_SECURITY_INFORMATION	Информация о SACL

Третий параметр, *pSecurityDescriptor*, указывает на буфер, в который будет скопирован полученный дескриптор безопасности в относительном формате. Размер этого буфера (в байтах) задается параметром *nLength*.

Последний параметр, *lpnLengthNeeded*, указывает на переменную типа DWORD, в которую будет записан необходимый размер буфера (в байтах) для дескриптора безопасности. При этом если необходимо определить размер буфера, параметр *nLength* должен принимать значение равное нулю.

Если функция GetFileSecurity завершается успешно, она возвращает значение отличное от FALSE.

В листинге 5.15 представлена функция, которая возвращает дескриптор безопасности файла или каталога. Функция вернет адрес созданного буфера для дескриптора безопасности. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции LocalFree.

Листинг 5.15. Функция, возвращающая дескриптор безопасности

```
1   BOOL GetFileSecurityDescriptor(LPCTSTR lpFileName,
    SECURITY_INFORMATION RequestedInformation,
    PSECURITY_DESCRIPTOR *ppSD)
1   {
2       DWORD cb = 0;
3       // определим размер дескриптора безопасности
4       GetFileSecurity(lpFileName, RequestedInformation, NULL, 0,
    &cb);
5
6       // выделим память для дескриптора безопасности
7       PSECURITY_DESCRIPTOR pSD = (PSECURITY_DESCRIPTOR)
    LocalAlloc(LMEM_FIXED, cb);
8       if (NULL == pSD) return FALSE;
9
10      // получим дескриптор безопасности
11      BOOL bRet = GetFileSecurity(lpFileName, RequestedInformation,
    pSD, cb, &cb);
```

```
12
13      if (FALSE != bRet)
14      {
15          *ppSD = pSD; // возвращаем полученный дескриптор
    безопасности
16      } // if
17      else
18      {
19          LocalFree(pSD); // освобождаем выделенную память
20      } // else
21
22      return bRet;
23  } // GetFileSecurityDescriptor
```

Для изменения дескриптора безопасности в файле или каталоге можно использовать функцию `SetFileSecurity`:

```
BOOL SetFileSecurity(LPCTSTR lpFileName,
    SECURITY_INFORMATION SecurityInformation,
    PSECURITY_DESCRIPTOR pSecurityDescriptor);
```

Первый параметр, *`lpFileName`*, указывает на строку, содержащую имя файла или каталога, для которого необходимо изменить дескриптор безопасности.

Второй параметр, *`SecurityInformation`*, определяет, какие компоненты дескриптора безопасности следует изменить (см. табл. 5.20).

Последний параметр, *`SecurityDescriptor`*, указывает на буфер, в котором хранится дескриптор безопасности, содержащий новую информацию о безопасности объекта.

При успешном завершении функция `SetFileSecurity` возвращает значение отличное от `FALSE`.

В листинге 5.16 показано, как изменять информацию в дескрипторе безопасности. Заметим, что в этом примере при изменении DACL учитываются уже имеющиеся ACE.

Листинг 5.16. Функция изменения информации в дескрипторе безопасности

```
1   BOOL SetFileSecurityInfo(LPCTSTR lpFileName,
    LPCTSTR lpAccountOwner, ULONG cCountOfEntries,
    PEXPLICIT_ACCESS pListOfEntries, BOOL bMergeEntries)
2   {
3       SECURITY_DESCRIPTOR sd; // дескриптор безопасности
4
5       PSID pOwner = NULL; // буфер для SID владельца
6       PACL pNewDacl = NULL; // буфер для нового DACL
7
```

```
8       // инициализируем дескриптор безопасности
9       BOOL bRet = InitializeSecurityDescriptor(&sd,
    SECURITY_DESCRIPTOR_REVISION);
10
11      // ///// //
12
13      if (FALSE != bRet && NULL != lpAccountOwner)
14      {
15          // получаем SID указанной учетной записи
16          bRet = GetAccountSID(lpAccountOwner, &pOwner);
17
18          // связываем полученный SID с дескриптором безопасности
19          if (FALSE != bRet)
20              bRet = SetSecurityDescriptorOwner(&sd, pOwner,
    FALSE);
21      } // if
22
23      // ///// //
24
25      if (FALSE != bRet && cCountOfEntries > 0 && NULL !=
    pListOfEntries)
26      {
27          PSECURITY_DESCRIPTOR pOldSD = NULL;
28          PACL pOldDacl = NULL; // указатель на буфер для DACL
29
30          BOOL bDaclDefaulted = FALSE;
31
32          if (FALSE != bMergeEntries)
33          {
34              bRet = GetFileSecurityDescriptor(lpFileName,
    DACL_SECURITY_INFORMATION, &pOldSD);
35
36              if (FALSE != bRet)
37              {
38                  BOOL bDaclPresent;
39                  // получаем DACL
40                  GetSecurityDescriptorDacl(pOldSD, &bDaclPresent,
    &pOldDacl, &bDaclDefaulted);
41              } // if
42          } // if
43
44          // создаем новый DACL
45          DWORD dwResult = SetEntriesInAcl(cCountOfEntries,
    pListOfEntries, pOldDacl, &pNewDacl);
46          bRet = (ERROR_SUCCESS == dwResult) ? TRUE : FALSE;
47
```

```
48          // связываем новый DACL с дескриптором безопасности
49          if (FALSE != bRet)
50              bRet = SetSecurityDescriptorDacl(&sd, TRUE, pNewDacl,
    bDaclDefaulted);
51
52          // освобождаем память
53          if (NULL != pOldSD) LocalFree(pOldSD);
54      } // if
55
56      // ///// //
57
58      if (FALSE != bRet)
59      {
60          SECURITY_INFORMATION si = 0;
61          if (NULL != pOwner) si |= OWNER_SECURITY_INFORMATION;
62          if (NULL != pNewDacl) si |= DACL_SECURITY_INFORMATION;
63
64          // изменяем дескриптор безопасности
65          bRet = SetFileSecurity(lpFileName, si, &sd);
66      } // if
67
68      // освобождаем память, выделенную под буфер для SID
69      if (NULL != pOwner) LocalFree(pOwner);
70      // освобождаем память, выделенную под буфер для DACL
71      if (NULL != pNewDacl) LocalFree(pNewDacl);
72
73      return bRet;
74  } // SetFileSecurityInfo
```

Функцию SetFileSecurityInfo из листинга 5.16 можно использовать для добавления ACE следующим образом:

```
1   PSID pSid = NULL;
2   // получаем SID учетной записи «Андрей»
3   BOOL bRet = GetAccountSID(TEXT("Андрей"), &pSid);
4
5   if (FALSE != bRet)
6   {
7       EXPLICIT_ACCESS ea;
8
9       ea.grfAccessPermissions = FILE_ALL_ACCESS; // полный доступ
10      ea.grfAccessMode = GRANT_ACCESS; // разрешить доступ
11      ea.grfInheritance= NO_INHERITANCE; // не наследовать
12      ea.Trustee.TrusteeForm = TRUSTEE_IS_SID; // доверенный объект
    задается с помощью SID
13      ea.Trustee.ptstrName = (LPTSTR)pSid; // SID учетной записи
14
```

```
15      // добавляем ACE в дескриптор безопасности файла
16      bRet = SetFileSecurityInfo(TEXT("D:\\test.txt"), NULL, 1,
    &ea, TRUE);
17
18      if (FALSE != bRet)
19      {
20          /* ACE успешно добавлен */
21      } // if
22
23      LocalFree(pSid); // освобождаем память, выделенную для SID
24  } // if
```

Функцию SetFileSecurityInfo также можно использовать для изменения владельца:

```
25  BOOL bRet = SetFileSecurityInfo(TEXT("C:\\Temp"), TEXT("Андрей"),
    0, NULL, FALSE);
26
27  if (FALSE != bRet)
28  {
29      /* владелец успешно изменен */
30  } // if
```

В Win32 API имеются еще две функции для чтения и изменения дескриптора безопасности объекта ядра – GetKernelObjectSecurity и SetKernelObjectSecurity:

```
BOOL GetKernelObjectSecurity(HANDLE Handle,
    SECURITY_INFORMATION RequestedInformation,
    PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD nLength,
    LPDWORD lpnLengthNeeded);
BOOL SetKernelObjectSecurity(HANDLE Handle,
    SECURITY_INFORMATION SecurityInformation,
    PSECURITY_DESCRIPTOR SecurityDescriptor);
```

Эти функции работают также как и другая пара функций – GetFileSecurity и SetFileSecurity. Основным отличием является то, что здесь в качестве первого параметра вместо имени файла (или каталога) указывается дескриптор объекта ядра, который должен обладать определенным уровнем доступа. Для получения информации о владельце, основной группе и DACL, дескриптор объекта ядра должен обладать правом доступа READ_CONTROL. Для изменения информации о владельце или основной группе дескриптор объекта ядра должен обладать правом доступа WRITE_OWNER, а для изменения DACL – WRITE_DAC. Для получения и изменения SACL дескриптор объекта ядра должен обладать правом доступа ACCESS_SYSTEM_SECURITY.

Можно изменить функции из листингов 5.15,5.16 так, чтобы они работали с функциями `GetKernelObjectSecurity` и `SetKernelObjectSecurity`. Например, как показано в листинге 5.17.

Листинг 5.17. Функция, возвращающая дескриптор безопасности объекта ядра

```
BOOL GetObjectSecurityDescriptor(HANDLE hObject,
SECURITY_INFORMATION RequestedInformation,
PSECURITY_DESCRIPTOR *ppSD)
{
    DWORD cb = 0;
    // определим размер дескриптора безопасности
    GetKernelObjectSecurity(hObject, RequestedInformation, NULL,
0, &cb);

    // выделим память для дескриптора безопасности
    PSECURITY_DESCRIPTOR pSD = (PSECURITY_DESCRIPTOR)
LocalAlloc(LMEM_FIXED, cb);
    if (NULL == pSD) return FALSE;

    // получим дескриптор безопасности
    BOOL bRet = GetKernelObjectSecurity(hObject,
RequestedInformation, pSD, cb, &cb);

    if (FALSE != bRet)
    {
        *ppSD = pSD; // возвращаем полученный дескриптор
безопасности
    } // if
    else
    {
        LocalFree(pSD); // освобождаем выделенную память
    } // else

    return bRet;
} // GetObjectSecurityDescriptor
```

Кроме того в Win32 API для работы с дескриптором безопасности имеется еще несколько функций (табл. 5.21). Описание этих функций см. в документации Platform SDK.

Таблица 5.21. Функции для работы с дескриптором безопасности

Функция	Описание
`GetNamedSecurityInfo`	Возвращает информацию из дескриптора безопасности для объекта, указанного именем

Окончание табл. 5.21.

Функция	Описание
`GetSecurityInfo`	Возвращает информацию из дескриптора безопасности для объекта, указанного дескриптором
`SetNamedSecurityInfo`	Изменяет информацию в дескрипторе безопасности для объекта, указанного именем
`SetSecurityInfo`	Изменяет информацию в дескрипторе безопасности для объекта, указанного дескриптором

Маркер доступа

Каждый процесс и его потоки исполняются в контексте учетной записи пользователя. Этот контекст также называют *контекстом безопасности* (security context). Для работы с контекстом безопасности используется объект ядра, называемый *маркером доступа* (access token). Операционная система использует маркер доступа, когда процесс (вернее какой-то из его потоков) взаимодействует с защищаемым объектом или же пытается выполнить определенную операцию, которая требует наличия соответствующих привилегий. При этом если процесс обращается к защищаемым объектам удаленного компьютера, происходит передача данных контекста безопасности на удаленный компьютер для доступа к его защищаемым объектам.

Когда пользователь входит в систему создается *первичный маркер доступа* (primary access token), который сопоставляется с первым процессом данного пользователя – обычно `userinit.exe`. Далее при создании дочерних процессов, каждый из них по умолчанию получает копию маркера доступа родительского процесса. Таким образом все процессы в сеансе одного пользователя, как правило, выполняются с копией одного и того же первичного маркера доступа.

Кроме того, потоку может быть задан *замещающий маркер доступа* (impersonation access token), который будет отличаться от маркера доступа процесса. Замещающий маркер доступа позволяет потоку выполняться в контексте безопасности отличном от контекста безопасности процесса, к которому данный поток относится.

Маркер доступа содержит различную информацию, включая следующую:

- SID учетной записи пользователя, в контексте безопасности которого выполняется процесс/поток;
- список групп;
- идентификатор сеанса пользователя;

- список привилегий;
- SID учетной записи пользователя, который используется в качестве владельца по умолчанию при создании защищаемого объекта;
- SID группы, которая используется в качестве основной группы по умолчанию при создании защищаемого объекта;
- DACL, который используется по умолчанию при создании защищаемого объекта.

Более подробное описание содержимого маркера доступа можно найти в документации Platform SDK.

Если запустить утилиту Process Explorer и посмотреть на содержимое вкладки **Security** (Безопасность) диалогового окна Properties (Свойства), открытого для выбранного процесса (рис. 5.9), то можно увидеть содержимое маркера доступа этого процесса.

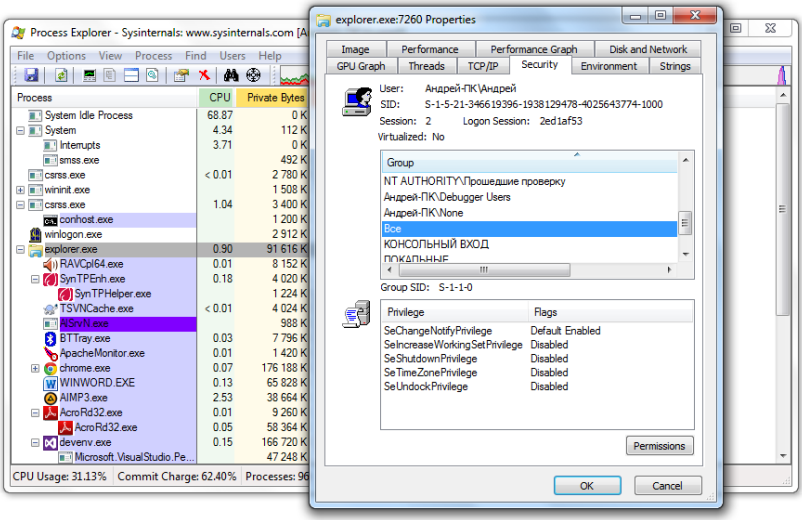

Рис. 5.9. Содержимое маркера доступа процесса, просматриваемое с помощью утилиты Process Explorer

Открытие маркера доступа

Прежде чем работать с маркером доступа его необходимо открыть. Для открытия маркера доступа процесса следует использовать функцию `OpenProcessToken`, которая возвращает дескриптор объекта ядра «маркер доступа». Этот объект ядра используется для работы с маркером доступа.

Функция OpenProcessToken имеет следующий прототип:

```
BOOL OpenProcessToken(HANDLE ProcessHandle,
    DWORD DesiredAccess, PHANDLE TokenHandle);
```

Первый параметр, *ProcessHandle*, – дескриптор процесса, для которого нужно получить маркер доступа. Этот дескриптор должен обладать правом доступа PROCESS_QUERY_INFORMATION.

Второй параметр, *DesiredAccess*, определяет запрашиваемые права доступа к объекту ядра «маркер доступа». Некоторые из возможных значений этого параметра перечислены в табл. 5.22. Полный перечень значений параметра *DesiredAccess* можно найти в документации Platform SDK.

Таблица 5.22. Права доступа к маркеру доступа

Значение	**Описание**
TOKEN_ADJUST_DEFAULT	Разрешает изменение информации, которая используется в дескрипторе безопасности по умолчанию при создании защищаемого объекта
TOKEN_ADJUST_GROUPS	Разрешает изменение групп в маркере доступа
TOKEN_ADJUST_PRIVILEGES	Разрешает изменение состояния привилегий в маркере доступа
TOKEN_ADJUST_SESSIONID	Разрешает изменение идентификатора сеанса пользователя в маркере доступа
TOKEN_ALL_ACCESS	Все возможные права доступа
TOKEN_ASSIGN_PRIMARY	Разрешает присоединение первичного маркера доступа к процессу
TOKEN_DUPLICATE	Разрешает дублирование маркера доступа
TOKEN_IMPERSONATE	Разрешает замещать маркер доступа процесса
TOKEN_QUERY	Необходимо для получения информации о содержимом маркера доступа
TOKEN_QUERY_SOURCE	Необходимо для получения информации об источнике маркера доступа

Последний параметр, *TokenHandle*, указывает на переменную типа HANDLE, через которую функция OpenProcessToken возвращает дескриптор маркера доступа (точнее объекта ядра «маркер доступа»).

При успешном завершении функция OpenProcessToken возвращает значение отличное от FALSE.

В следующем примере показано, как открыть маркер доступ процесса с помощью функции OpenProcessToken.

Листинг 5.18. Функция, открывающая маркер доступа процесса

```
1   HANDLE OpenProcessTokenByProcessId(DWORD dwProcessId,
    DWORD dwDesiredAccess)
2   {
3       HANDLE hToken = NULL;
4
5       // получаем дескриптор процесса
6       HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION,
    FALSE, dwProcessId);
7
8       if (NULL != hProcess)
9       {
10          // открываем маркер доступа процесса
11          OpenProcessToken(hProcess, dwDesiredAccess, &hToken);
12
13          // закрываем дескриптор процесса
14          CloseHandle(hProcess);
15      } // if
16
17      return hToken;
18  } // OpenProcessTokenByProcessId
```

Для получения маркера доступа потока нужно использовать функцию `OpenThreadToken`, которая имеет следующий прототип:

```
BOOL OpenThreadToken(HANDLE ThreadHandle, DWORD DesiredAccess,
    BOOL OpenAsSelf, PHANDLE TokenHandle);
```

Первый параметр, *`ThreadHandle`*, – дескриптор потока, для которого нужно получить маркер доступа.

Второй параметр, *`DesiredAccess`*, определяет запрашиваемые права доступа к объекту ядра «маркер доступа» (см. табл. 5.22).

Третий параметр, *`OpenAsSelf`*, указывает на необходимость выполнения функции `OpenThreadToken` в контексте безопасности вызывающего процесса. Если параметр принимает значение `FALSE`, функция будет выполняться в контексте безопасности указанного потока.

Последний параметр, *`TokenHandle`*, указывает на переменную типа `HANDLE`, через которую функция `OpenThreadToken` возвращает дескриптор маркера доступа.

При успешном завершении функция `OpenThreadToken` возвращает значение отличное от `FALSE`.

В листинге 5.19 продемонстрированно, как использовать функцию `OpenThreadToken` для получения маркера доступа вызывающего потока.

Листинг 5.19. Функция, открывающая маркер доступа вызывающего потока

```
1   HANDLE OpenToken(DWORD dwDesiredAccess)
2   {
3       HANDLE hToken = NULL;
4
5       // получаем маркер доступа текущего потока
6       BOOL bRet = OpenThreadToken(GetCurrentThread(),
    dwDesiredAccess, FALSE, &hToken);
7
8       if (FALSE == bRet &&
            GetLastError() == ERROR_NO_TOKEN) // Ошибка: Попытка
    обращения к несуществующему маркеру доступа
9       {
10          // получаем маркер доступа процесса
11          OpenProcessToken(GetCurrentProcess(), dwDesiredAccess,
    &hToken);
12      } // if
13
14      return hToken;
15  } // OpenToken
```

После того как объект ядра «маркер доступа» станет более не нужен его следует закрыть вызовом функции `CloseHandle`.

Получение маркера доступа пользователя

Получить маркер доступа определенного пользователя, зарегистрированного в операционной системе, можно с помощью функция `LogonUser`, которая имеет следующий прототип:

```
BOOL LogonUser(LPTSTR lpszUsername, LPTSTR lpszDomain,
    LPTSTR lpszPassword, DWORD dwLogonType,
    DWORD dwLogonProvider, PHANDLE phToken);
```

Первый параметр, *`lpszUsername`*, указывает на строку, содержащую имя учетной записи пользователя.

Второй параметр, *`lpszDomain`*, указывает на строку, содержащую имя домена, на котором будет вестись поиск учетной записи. Если этот параметр установлен в `NULL`, имя пользователя должно указываться в UPN (User Principal Name) формате (например, UserName@Example.Microsoft.com). Если этот параметр принимает значение «.», поиск будет осуществляться на локальном компьютере.

Третий параметр, *`lpszPassword`*, указывает на строку, содержащую пароль пользователя. Пароль указывается без шифрования.

Четвертый параметр, *dwLogonType*, указывает тип входа в систему. Некоторые из возможных значений этого параметра перечислены в табл. 5.23. Подробное описание всех возможных значений параметра *dwLogonType* можно найти в документации Platform SDK.

Таблица 5.23. Значения параметра *dwLogonType*

Значение	**Описание**
LOGON32_LOGON_BATCH	Этот тип входа предназначен для пакетной обработки заданий от имени пользователя без его прямого вмешательства
LOGON32_LOGON_INTERACTIVE	Этот тип входа предназначен для пользователей, которые будут работать на компьютере в интерактивном режиме
LOGON32_LOGON_NETWORK_CLEARTEXT	Этот тип входа используется для получения доступа к компьютеру по сети
LOGON32_LOGON_NEW_CREDENTIALS	Этот тип входа позволяет вызывающему потоку клонировать свой дескриптор безопасности и указать новые учетные данные для доступа к компьютеру по сети
LOGON32_LOGON_SERVICE	Этот тип входа используется службами Window

Пятый параметр, *dwLogonProvider*, определяет поставщика информации об учетных записях пользователей (подробнее см. в документации Platform SDK). Для использования стандартного поставщика следует использовать константу LOGON32_PROVIDER_DEFAULT.

Последний параметр, *phToken*, указывает на переменную типа HANDLE, через которую функция LogonUser возвращает дескриптор маркера доступа.

При успешном завершении функция LogonUser возвращает значение отличное от FALSE. После того как полученный дескриптор маркера доступа станет более не нужен его следует закрыть вызовом функции CloseHandle.

Дублирование маркера доступа осуществляется с помощью функции DuplicateTokenEx, которая имеет следующий прототип:

```
BOOL DuplicateTokenEx(HANDLE hExistingToken,
    DWORD dwDesiredAccess,
    LPSECURITY_ATTRIBUTES lpTokenAttributes,
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
    TOKEN_TYPE TokenType, PHANDLE phNewToken);
```

Первый параметр, *hExistingToken*, – дескриптор объекта ядра «маркер доступа», для которого нужно получить дубликат. Этот дескриптор должен обладать правом доступа TOKEN_DUPLICATE.

Второй параметр, *DesiredAccess*, определяет запрашиваемые права доступа к объекту ядра «маркер доступа» (см. табл. 5.22).

Третий параметр, *lpTokenAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «маркер доступ». Этот параметр можно установить в NULL, тогда объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Четвертый параметр, *ImpersonationLevel*, задает уровень замещения в новом маркере доступа. Этот параметр принимает значения перечисляемого типа SECURITY_IMPERSONATION_LEVEL (табл. 5.24). Уровень замещения регулирует степень, в которой процесс удаленного компьютера (сервера) может действовать в контексте безопасности пользователя локального компьютера (клиента).

Таблица 5.24. Значения перечисляемого типа SECURITY_IMPERSONATION_LEVEL

Значение	**Описание**
SecurityAnonymous	Анонимный уровень, на котором процесс сервера не может идентифицировать клиента
SecurityIdentification	Идентифицирующий уровень, на котором процесс сервера может получать любую идентификационную информацию о клиенте, но не может выполнять никаких действий в контексте безопасности клиента
SecurityImpersonation	Уровень подмены контекста безопасности, на котором процесс сервера может работать в контексте безопасности клиента. При этом процесс, используя контекст безопасности клиента, может обращаться к защищаемым объектам сервера, но не может обращаться к удаленного компьютера по сети
SecurityDelegation	Уровень делегирования, на этом уровне клиент полностью делегирует свои полномочия процессу сервера. При этом процесс, используя контекст безопасности клиента, может обращаться к защищаемым объектам сервера и к удаленному компьютеру по сети

Пятый параметр, *TokenType*, задает тип создаваемого маркера доступа. Этот параметр принимает одно из двух значений перечисляемого типа TOKEN_TYPE:

– `TokenPrimary` – первичный маркер доступа;
– `TokenImpersonation` – замещающий маркер доступа.

Последний параметр, *`phNewToken`*, указывает на переменную типа `HANDLE`, через которую функция `DuplicateTokenEx` возвращает дескриптор маркера доступа.

При успешном завершении функция `DuplicateTokenEx` возвращает значение отличное от `FALSE`.

В листинге 5.20 представлен пример функции, которая получает маркер доступа для указанного пользователя. Заметим, что в этом примере для того, чтобы получить дескриптор объекта ядра «маркер доступа» с определенным уровнем доступа создается дубликат полученного маркера доступа пользователя.

Листинг 5.20. Функция, возвращающая маркер доступа пользователя

```
1   HANDLE OpenUserToken(LPCTSTR lpszUsername, LPCTSTR lpszDomain,
    LPCTSTR lpszPassword, DWORD dwLogonType, DWORD dwDesiredAccess,
    PSECURITY_ATTRIBUTES lpTokenAttributes, TOKEN_TYPE TokenType,
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel)
2   {
3       HANDLE hToken = NULL;
4
5       // получаем маркер доступа указанного пользователя
6       BOOL bRet = LogonUser(lpszUsername, lpszDomain, lpszPassword,
    dwLogonType, LOGON32_PROVIDER_DEFAULT, &hToken);
7
8       if (FALSE != bRet)
9       {
10          HANDLE hNewToken = NULL;
11
12          // дублируем маркер доступа
13          bRet = DuplicateTokenEx(hToken, dwDesiredAccess,
    lpTokenAttributes, ImpersonationLevel, TokenType, &hNewToken);
14
15          // закрываем ранее полученный маркер доступа
16          CloseHandle(hToken);
17
18          // возвращаем результат
19          hToken = (FALSE != bRet) ? hNewToken : NULL;
20      } // if
21
22      return hToken;
23  } // OpenUserToken
```

В следующем небольшом примере продемонстрированно, как с помощью функции OpenUserToken из листинга 5.20 можно получить маркер доступа пользователя «Гость».

```
HANDLE hToken = OpenUserToken(TEXT("Гость"), // имя пользователя
    TEXT("."), // локальный компьютер
    TEXT(""), // пароль (отсутствует)
    LOGON32_LOGON_INTERACTIVE, // локальный вход
    TOKEN_ALL_ACCESS, // полные права к маркеру доступа
    NULL, // параметры безопасности по умолчанию
    TokenImpersonation, // замещающий маркер доступа
    SecurityImpersonation); // уровень замещения

if (NULL != hToken)
{
    /* маркер доступа пользователя «Гость» успешно получен */

    CloseHandle(hToken); // закрываем маркер доступа
} // if
```

Нужно отметить, что при использовании типа входа LOGON32_LOGON_INTERACTIVE функция OpenUserToken завершится ошибкой, если учетная запись «Гость» не будет обладать правом SeInteractiveLogonRight (*локальный вход в систему*).

Получение информации из маркера доступа

Чтобы получить информацию о содержимом маркера доступа нужно воспользоваться функцией GetTokenInformation:

```
BOOL GetTokenInformation(HANDLE TokenHandle,
    TOKEN_INFORMATION_CLASS TokenInformationClass,
    LPVOID TokenInformation, DWORD TokenInformationLength,
    PDWORD ReturnLength);
```

Первый параметр, *TokenHandle*, – дескриптор объекта ядра «маркер доступа», который должен обладать правом доступа TOKEN_QUERY.

Второй параметр, *TokenInformationClass*, определяет, какую информацию необходимо получить из маркера доступа. Этот параметр принимает значения перечисляемого типа TOKEN_INFORMATION_CLASS, основные значения которого перечислены в табл. 5.25.

Третий параметр, *TokenInformation*, указывает на буфер, в который будет сохранен результат. В зависимости от значения, заданного параметром *TokenInformationClass*, информация из маркера доступа записана в структурированном виде или имеет значение некоторого перечислимого типа. В табл. 5.26 приведено соответствие значений перечислимого типа TOKEN_INFORMATION_CLASS и типов данных, исполь-

зуемых для представления соответствующей информации в маркере доступа.

Таблица 5.25. Значения перечисляемого типа TOKEN_INFORMATION_CLASS

Значение	Описание
TokenUser	Информация о пользователе
TokenGroups	Информация о группах, связанных с маркером доступа
TokenPrivileges	Информация о привилегиях
TokenOwner	Информация о владельце объекта по умолчанию
TokenPrimaryGroup	Информация об основной группе объекта по умолчанию
TokenDefaultDacl	Информация о DACL объекта по умолчанию
TokenSource	Источник маркера доступа
TokenType	Тип маркера доступа
TokenImpersonationLevel	Уровень замещения маркера доступа
TokenStatistics	Статистическая информация
TokenRestrictedSids	Список ограничивающих идентификаторов безопасности
TokenSessionId	Идентификатор сеанса пользователя

Таблица 5.26. Типы данных, используемые в маркере доступа

Значение	Тип данных
TokenUser	TOKEN_USER
TokenGroups	TOKEN_GROUPS
TokenPrivileges	TOKEN_PRIVILEGES
TokenOwner	TOKEN_OWNER
TokenPrimaryGroup	TOKEN_PRIMARY_GROUP
TokenDefaultDacl	TOKEN_DEFAULT_DACL
TokenSource	TOKEN_SOURCE
TokenType	TOKEN_TYPE
TokenImpersonationLevel	SECURITY_IMPERSONATION_LEVEL
TokenStatistics	TOKEN_STATISTICS
TokenRestrictedSids	TOKEN_GROUPS
TokenSessionId	DWORD

Четвертый параметр, *TokenInformationLength*, задает размер буфера, на который указывает параметр *TokenInformation*. Этот параметр должен принимать нулевое значение, если параметр *TokenInformation* установлен в NULL.

Последний параметр, *ReturnLength*, указывает на переменную типа DWORD, в которую будет сохранен необходимый размер (в байтах) результирующего буфера.

При успешном завершении функция GetTokenInformation возвращает значение отличное от FALSE.

Определение типа маркера доступа и уровня замещения

В следующем примере демонстрируется использование функции GetTokenInformation для определения типа маркера доступа и уровня замещения.

Листинг 5.21. Определение типа маркера доступа и уровня замещения

```
1   TOKEN_TYPE type;
1   DWORD cb = sizeof(TOKEN_TYPE);
2
3   // определяем тип маркера доступа
4   BOOL bRet = GetTokenInformation(hToken, TokenType, &type, cb,
    &cb);
5
6   if (FALSE != bRet && TokenPrimary == type)
7   {
8       /* первичный маркер доступа */
9   } // if
10  else if (FALSE != bRet && TokenImpersonation == type)
11  {
12      /* замещающий маркер доступа */
13
14      SECURITY_IMPERSONATION_LEVEL level;
15      cb = sizeof(SECURITY_IMPERSONATION_LEVEL);
16
17      // определяем уровня замещения в маркере доступа
        bRet = GetTokenInformation(hToken, TokenImpersonationLevel,
    &level, cb, &cb);
18
19      if (FALSE != bRet)
20      {
21          switch (level)
22          {
23          case SecurityAnonymous:
24              /* анонимный уровень */
25              break;
26          case SecurityIdentification:
27              /* идентифицирующий уровень */
28              break;
29          case SecurityImpersonation:
```

```
            /* уровень подмены контекста безопасности */
            break;
        case SecurityDelegation:
            /* уровень делегирования */
            break;
        } // switch
    } // if
} // if
```

Заметим, что маркер доступа в этом примере указывается с помощью переменной `hToken`.

Получение учетной записи пользователя из маркера доступа

Для получения информации о пользователе, связанном с маркером доступа, используется структура `TOKEN_USER`, которая определена следующим образом:

```
typedef struct _TOKEN_USER {
  SID_AND_ATTRIBUTES User; // SID пользователя и атрибуты
} TOKEN_USER, *PTOKEN_USER;
```

Поле *`User`* представляет собой структуру `SID_AND_ATTRIBUTES`, которая определенна следующим образом:

```
typedef struct _SID_AND_ATTRIBUTES {
  PSID Sid; // идентификатор безопасности пользователя
  DWORD Attributes; // атрибуты идентификатора безопасности
} SID_AND_ATTRIBUTES, * PSID_AND_ATTRIBUTES;
```

Первое поле, *`Sid`*, указывает на буфер, в котором содержится SID учетной записи пользователя.

Второе поле, *`Attributes`*, содержит атрибуты идентификатора безопасности. В настоящее время для SID пользователя не определены атрибуты, поэтому это поле не используется.

В листинге 5.22 представлена функция, которая извлекает из маркера доступа SID пользователя. Функция вернет адрес созданного буфера для SID. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`.

Листинг 5.22. Функция, извлекающая из маркера доступа SID пользователя

```
BOOL GetTokenUser(HANDLE hToken, PSID *ppSid)
{
    DWORD cb;

    // определеям размер блока памяти (в байтах)
```

```
6       GetTokenInformation(hToken, TokenUser, NULL, 0, &cb);
7
8       // выделяем блок памяти
9       PTOKEN_USER pTokenUser = (PTOKEN_USER)LocalAlloc(LPTR, cb);
10      if (NULL == pTokenUser) return FALSE;
11
12      // получаем информацию о пользователе в маркере доступа
13      BOOL bRet = GetTokenInformation(hToken, TokenUser,
    pTokenUser, cb, &cb);
14
15      PSID pSid = NULL;
16
17      // копируем SID пользователя
18      if (FALSE != bRet)
19      {
20          DWORD cbSid = GetLengthSid(pTokenUser->User.Sid);
21          pSid = (PSID)LocalAlloc(LPTR, cbSid);
22
23          bRet = CopySid(cbSid, pSid, pTokenUser->User.Sid);
24      } // if
25
26      // освобождаем выделенную память
27      LocalFree(pTokenUser);
28
29      if (FALSE != bRet)
30      {
31          *ppSid = pSid; // возвращаем результат
32      } // if
33      else
34      {
35          LocalFree(pSid); // освобождаем выделенную память
36      } // else
37
38      return bRet;
39  } // GetTokenUser
```

Получение информации о группах из маркера доступа

Для получения информации о группах, связанных с маркером доступа, используется структура TOKEN_GROUPS, которая определена следующим образом:

```
typedef struct _TOKEN_GROUPS {
  DWORD GroupCount; // количество групп
  SID_AND_ATTRIBUTES Groups[ANYSIZE_ARRAY]; // список групп
} TOKEN_GROUPS, *PTOKEN_GROUPS;
```

Первое поле, *GroupCount*, содержит количество групп, связанных с маркером доступа.

Второе поле, *Groups*, указывает на список, каждый элемент которого представляет собой структуру SID_AND_ATTRIBUTES. При этом поле *Attributes* структуры SID_AND_ATTRIBUTES задается флагами, некоторые из которых перечислены в табл. 5.27 (подробнее см. документацию Platform SDK).

Таблица 5.27. Флаги для поля *Attributes*

Флаг	Описание
SE_GROUP_ENABLED	Группа проверяется при доступе к объекту
SE_GROUP_ENABLED_BY_DEFAULT	Флаг SE_GROUP_ENABLED установлен по умолчанию
SE_GROUP_LOGON_ID	SID идентифицирует сессию, выполнившую вход в систему для пользователя, связанного с группой
SE_GROUP_MANDATORY	Нельзя сбросить флаг SE_GROUP_ENABLED
SE_GROUP_OWNER	Пользователь, связанный с маркером доступа, является владельцем группы; или группа может быть владельцем объекта
SE_GROUP_RESOURCE	Локальная группа домена
SE_GROUP_USE_FOR_DENY_ONLY	Группа используется только при проверке запрещения доступа к объекту

В листинге 5.23 представлен пример использования функции GetTokenInformation для получения списка групп, связанных с маркером доступа.

Листинг 5.23. Функция извлечения из маркера доступа список групп

```
1   BOOL GetTokenGroups(HANDLE hToken, PTOKEN_GROUPS *pTokenGroups)
2   {
3       DWORD cb;
4
5       // определеям размер блока памяти (в байтах)
6       GetTokenInformation(hToken, TokenGroups, NULL, 0, &cb);
7
8       // выделяем блок памяти
9       PTOKEN_GROUPS Groups = (PTOKEN_GROUPS)LocalAlloc(LPTR, cb);
10      if (NULL == Groups) return FALSE;
11
12      // получаем список привилегий в маркере доступа
13      BOOL bRet = GetTokenInformation(hToken, TokenGroups, Groups,
    cb, &cb);
```

```
14
15      if (FALSE != bRet)
16      {
17          *pTokenGroups = Groups; // возвращаем результат
18      } // if
19      else
20      {
21          LocalFree(Groups); // освобождаем выделенную память
22      } // else
23
24      return bRet;
25  } // GetTokenGroups
```

Функция `GetTokenGroups` из представленного примера возвращает адрес созданного буфера для структуры `TOKEN_GROUPS`. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`.

Перечисление привилегий

Для получения информации о привилегиях, которыми обладает маркер доступа, используется структура `TOKEN_PRIVILEGES`:

```
typedef struct _TOKEN_PRIVILEGES {
  DWORD PrivilegeCount; // количество привелегий
  LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY]; // привилегии
} TOKEN_PRIVILEGES, *PTOKEN_PRIVILEGES;
```

Первое поле, *`PrivilegeCount`*, содержит количество привилегий, которыми обладает маркер доступа.

Второе поле, *`Privileges`*, указывает на список, каждый элемент которого представляет собой структуру `LUID_AND_ATTRIBUTES`.

Структура `LUID_AND_ATTRIBUTES` определена следующим образом:

```
typedef struct _LUID_AND_ATTRIBUTES {
  LUID Luid; // идентификатор привилегии
  DWORD Attributes; // атрибуты привилегии
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

Первое поле, *`Luid`*, указывает на буфер, в котором содержится LUID привилегии. Второе поле, *`Attributes`*, значение, которого задается флагами, перечисленными в табл. 5.28.

В листинге 5.24 продемонстрированно, как использовать функцию `GetTokenInformation` для получения списка привилегий.

Таблица 5.28. Флаги для поля `Attributes`

Флаг	Описание
SE_PRIVILEGE_ENABLED_BY_DEFAULT	Привилегия включена по умолчанию
SE_PRIVILEGE_ENABLED	Привилегия включена
SE_PRIVILEGE_USED_FOR_ACCESS	Привилегия используется для контроля доступа к защищаемым объектам

Листинг 5.24. Функция, возвращающая список привилегий маркера доступа

```
1   BOOL GetTokenPrivileges(HANDLE hToken, PTOKEN_PRIVILEGES
    *pPrivileges)
2   {
3       DWORD cb;
4
5       // определеям размер блока памяти (в байтах)
6       GetTokenInformation(hToken, TokenPrivileges, NULL, 0, &cb);
7
8       // выделяем блок памяти
9       PTOKEN_PRIVILEGES Privileges = (PTOKEN_PRIVILEGES)
    LocalAlloc(LPTR, cb);
10      if (NULL == Privileges) return FALSE;
11
12      // получаем список привилегий в маркере доступа
13      BOOL bRet = GetTokenInformation(hToken, TokenPrivileges,
    Privileges, cb, &cb);
14
15      if (FALSE != bRet)
16      {
17          *pPrivileges = Privileges; // возвращаем результат
18      } // if
19      else
20      {
21          LocalFree(Privileges); // освобождаем выделенную память
22      } // else
23
24      return bRet;
25  } // GetTokenPrivileges
```

Функция `GetTokenPrivileges` из представленного примера возвращает адрес созданного буфера для структуры `TOKEN_PRIVILEGES`. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`. Функцию `GetTokenPrivileges` можно использовать следующим образом.

```
PTOKEN_PRIVILEGES pTokenPrivileges = NULL;

// получаем список привилегий
BOOL bRet = GetTokenPrivileges(hToken, &pTokenPrivileges);

if (FALSE != bRet)
{
    TCHAR szName[256];

    for (DWORD i = 0; i < pTokenPrivileges->PrivilegeCount; ++i)
    {
        // получаем LUID привилегии
        LUID Luid = pTokenPrivileges->Privileges[i].Luid;

        // определяем имя привилегии
        DWORD cch = _countof(szName);
        LookupPrivilegeName(NULL, &Luid, szName, &cch);

        // получаем атрибуты привилегии
        DWORD dwAttributes = pTokenPrivileges->Privileges[i].
Attributes;

        if (dwAttributes & SE_PRIVILEGE_ENABLED)
        {
            /* привилегия включена */
        } // if
        else
        {
            /* привилегия выключена */
        } // else
    } // for

    // освобождаем выделенную память
    LocalFree(pTokenPrivileges);
} // if
```

Кроме того, получив список привилегий можно проверить наличие в маркере доступа конкретной привилегии. В листингах 5.25,5.26 представлены примеры функций, которые можно использовать для проверки наличия привилегии в маркере доступа. Заметим, что в случае возникновения ошибки эти функции возвращают значение -1.

Листинг 5.25. Функция, определяющая наличие привилегии

```
BOOL IsPrivilegeExists(HANDLE hToken, LPCTSTR lpszPrivilegeName)
{
```

```
3       LUID Luid;
4       PTOKEN_PRIVILEGES pTokenPrivileges = NULL;
5
6       // определяем LUID указанной привилегии
7       BOOL bRet = LookupPrivilegeValue(NULL, lpszPrivilegeName,
    &Luid);
8
9       // получаем список привилегий
10      if (FALSE != bRet)
11          bRet = GetTokenPrivileges(hToken, &pTokenPrivileges);
12
13      if (FALSE == bRet) return -1; // возникла ошибка
14
15      bRet = FALSE;
16      for (DWORD i = 0; i < pTokenPrivileges->PrivilegeCount; ++i)
17      {
18          // сравниваем два LUID
19          int res = memcmp(&pTokenPrivileges->Privileges[i].Luid,
    &Luid, sizeof(LUID));
20
21          if (0 == res)
22          {
23              bRet = TRUE;
24              break; // выходим из цикла
25          } // if
26      } // for
27
28      // освобождаем выделенную память
29      LocalFree(pTokenPrivileges);
30
31      return bRet;
32  } // IsPrivilegeExists
```

Листинг 5.26. Функция, определяющая наличие нескольких привилегий

```
1   BOOL IsPrivilegesExists(HANDLE hToken, LPCTSTR names[],
    DWORD dwCount, BOOL bAllPrivileges)
2   {
3       LUID Luid;
4       PTOKEN_PRIVILEGES pTokenPrivileges = NULL;
5
6       // получаем список привилегий
7       BOOL bRet = GetTokenPrivileges(hToken, &pTokenPrivileges);
8       if (FALSE == bRet) return -1; // возникла ошибка
9
10      for (DWORD i = 0; i < dwCount; ++i)
```

```
11      {
12          // определяем LUID указанной привилегии
13          bRet = LookupPrivilegeValue(NULL, names[i], &Luid);
14
15          if (FALSE == bRet) // возникла ошибка
16          {
17              bRet = -1;
18              break; // выходим из цикла
19          } // if
20
21          bRet = FALSE;
22
23          for (DWORD j = 0;
24              j < pTokenPrivileges->PrivilegeCount; ++j)
25          {
26              // сравниваем два LUID
27              int res = memcmp(&pTokenPrivileges->Privileges[j].
    Luid, &Luid, sizeof(LUID));
28
29              if (0 == res)
30              {
31                  bRet = TRUE;
32                  break; // выходим из цикла
33              } // if
34          } // for
35
36          if ((FALSE == bRet && FALSE != bAllPrivileges) ||
37              (FALSE != bRet && FALSE == bAllPrivileges))
38          {
39              break; // выходим из цикла
40          } // if
41      } // for
42
43      // освобождаем выделенную память
44      LocalFree(pTokenPrivileges);
45
46      return bRet;
47  } // IsPrivilegesExists
```

Состояния привилегий

Привилегии, которыми обладает маркер доступа, могут находиться в двух состояниях: включенном и выключенном. Чтобы процесс мог выполнить привилегированную операцию, соответствующая привилегия должна присутствовать в маркере доступа и быть включена. Привилегии должны быть включены только при их востребованности,

чтобы процесс не мог по недосмотру выполнить привилегированную небезопасную операцию.

Проверка состояния привилегии

Для проверки состояния привилегий в маркере доступа следует использовать функцию PrivilegeCheck:

```
BOOL PrivilegeCheck(HANDLE ClientToken,
    PPRIVILEGE_SET RequiredPrivileges, LPBOOL pfResult);
```

Первый параметр, *ClientToken*, представляет собой дескриптор объекта ядра «маркер доступа», который должен обладать правом доступа TOKEN_QUERY.

Второй параметр, *RequiredPrivileges*, указывает на структуру PRIVILEGE_SET, содержащую массив привилегий, которые необходимо проверить.

Третий параметр, *pfResult*, указывает на переменную типа BOOL, в которую сохраняется значение TRUE, если поле *Control* в структуре PRIVILEGE_SET принимает значение PRIVILEGE_SET_ALL_NECESSARY и все указанные привилегии включены, или если поле *Control* принимает нулевое значение и включена любая из привилегий; в противном случае – FALSE.

Если функция PrivilegeCheck завершается успешно, возвращаемое значение отлично от FALSE.

Структура PRIVILEGE_SET имеет следующее описание:

```
typedef struct _PRIVILEGE_SET {
  DWORD PrivilegeCount; // количество привилегий
  DWORD Control; // флаг
  LUID_AND_ATTRIBUTES Privilege[ANYSIZE_ARRAY]; // привилегии
} PRIVILEGE_SET, *PPRIVILEGE_SET;
```

Первое поле, *PrivilegeCount*, определяет количество элементов в массиве, на который указывает поле *Privilege*.

Второе поле, *Control*, определяет должны ли быть включены все привилегии, указанные в массиве *Privilege*. Следует присвоить этому полю значение PRIVILEGE_SET_ALL_NECESSARY, если необходимо чтобы все указанные привилегии были включены, или нулевое значение, если достаточно чтобы любая из привилегий была включена.

Третье поле, *Privilege*, указывает на список, каждый элемент которого представляет собой структуру LUID_AND_ATTRIBUTES. Если привилегия включена, функция PrivilegeCheck устанавливает флаг SE_PRIVILEGE_USED_FOR_ACCESS в поле *Attributes* структуры LUID_AND_ATTRIBUTES.

В следующем примере продемонстрированно, как использовать функцию PrivilegeCheck для определения состояния привилегии.

Листинг 5.27. Функция, проверяющая состояние привилегии

```
1  BOOL IsPrivilegeEnabled(HANDLE hToken, LPCTSTR lpszPrivilegeName)
2  {
3      BOOL fResult; // результат проверки
4      PRIVILEGE_SET PrivSet;
5
6      PrivSet.PrivilegeCount = 1; // количество привилегий
7      PrivSet.Control = PRIVILEGE_SET_ALL_NECESSARY;
8
9      // определяем LUID указанной привилегии
10     BOOL bRet = LookupPrivilegeValue(NULL, lpszPrivilegeName,
   &PrivSet.Privilege[0].Luid);
11
12     if (FALSE != bRet)
13     {
14         // определяем состояние указанной привилегии
15         bRet = PrivilegeCheck(hToken, &PrivSet, &fResult);
16     } // if
17
18     return (FALSE != bRet) ? fResult : -1;
19 } // IsPrivilegeEnabled
```

Нужно отметить, что функция PrivilegeCheck позволяет проверить состояние сразу нескольких привилегий, которыми обладает маркер доступа. В листинге 5.28 представлен пример функции для проверки состояний сразу нескольких привилегий.

Листинг 5.28. Функция проверки состояния нескольких привилегий

```
1  BOOL PrivilegesCheck(HANDLE hToken, LPCTSTR names[],
   DWORD dwCount, BOOL bSetAllNecessary, PPRIVILEGE_SET *pPrivSet)
2  {
3      // определеям размер блока памяти (в байтах)
4      // для сохранения результата
5      DWORD cb = sizeof(PRIVILEGE_SET) + (dwCount - 1) *
   sizeof(LUID_AND_ATTRIBUTES);
6
7      // выделяем блок памяти для хранения результата
8      PPRIVILEGE_SET PrivSet = (PPRIVILEGE_SET)LocalAlloc(LPTR,
   cb);
9      if (NULL == PrivSet) return -1; // возникла ошибка
10
11     BOOL fResult; // результат проверки
```

```
12
13      // задаем количество указанных привилегий
14      PrivSet->PrivilegeCount = dwCount;
15      // указываем должны ли быть включены все указанные привелегии
16      PrivSet->Control = (FALSE != bSetAllNecessary) ?
    PRIVILEGE_SET_ALL_NECESSARY : 0;
17
18      BOOL bRet = TRUE;
19
20      // определяем LUID для каждой указанной привилегии
21      for (DWORD i = 0; i < dwCount && FALSE != bRet; ++i)
22      {
23          bRet = LookupPrivilegeValue(NULL, names[i], &PrivSet-
    >Privilege[i].Luid);
24      } // for
25
26      if (FALSE != bRet)
27      {
28          // определяем состояние указанных привилегий
29          bRet = PrivilegeCheck(hToken, PrivSet, &fResult);
30      } // if
31
32      if (FALSE != bRet && NULL != pPrivSet)
33      {
34          *pPrivSet = PrivSet; // возвращаем результат
35      } // if
36      else
37      {
38          // освобождаем выделенную память
39          LocalFree(PrivSet);
40      } // else
41
42      return (FALSE != bRet) ? fResult : -1;
43  } // PrivilegesCheck
```

Функция `PrivilegesCheck` из представленного примера возвращает адрес созданного буфера для структуры `PRIVILEGE_SET`. Когда полученный буфер станет более не нужен, следует освободить выделенную для него память вызовом функции `LocalFree`. Функцию `PrivilegesCheck` можно использовать следующим образом:

```
1   // список привилегий
2   LPCTSTR names[] = {
3       SE_ASSIGNPRIMARYTOKEN_NAME,
4       SE_TIME_ZONE_NAME,
5       SE_SHUTDOWN_NAME,
```

```
6       SE_TAKE_OWNERSHIP_NAME,
7       SE_CREATE_GLOBAL_NAME,
8       SE_CREATE_TOKEN_NAME
9   };
10
11  // указатель на блок памяти, в котором будет сохранен результат
12  PPRIVILEGE_SET Privileges = NULL;
13
14  // проверяем состояние указанных привилегий
15  BOOL bRet = PrivilegesCheck(hToken, names, _countof(names), TRUE,
    &Privileges);
16
17  if (bRet == -1)
18  {
19      /* возникла ошибка */
20  } // if
21  else
22  {
23      if (FALSE != bRet)
24      {
25          /* все привилегии включены */
26      } // if
27      else
28      {
29          for (DWORD i = 0; i < Privileges->PrivilegeCount; ++i)
30          {
31              DWORD dwAttributes = Privileges-
    >Privilege[i].Attributes;
32
33              // проверяем полученные атрибуты
34              if (dwAttributes & SE_PRIVILEGE_USED_FOR_ACCESS)
35              {
36                  /* привилегия включена */
37              } // if
38              else
39              {
40                  /* привилегия выключена */
41              } // else
42          } // for
43      } // else
44
45      LocalFree(Privileges); // освобождаем выделенную память
46  } // else
```

Заметим, что в этом примере осуществляется проверка того, чтобы все указанные привилегии были включены.

Изменение состояния привилегии

Чтобы изменить состояние привилегий в маркере доступа можно использовать функцию AdjustTokenPrivileges:

```
BOOL AdjustTokenPrivileges(HANDLE TokenHandle,
    BOOL DisableAllPrivileges, PTOKEN_PRIVILEGES NewState,
    DWORD BufferLength, PTOKEN_PRIVILEGES PreviousState,
    PDWORD ReturnLength);
```

Первый параметр, *TokenHandle*, представляет собой дескриптор объекта ядра «маркер доступа», который должен обладать правом доступа TOKEN_ADJUST_PRIVILEGES. Нужно отметить, что если параметр *PreviousState* не установлен в NULL, дескриптор, который передается в этом параметре, должен также обладать правом доступа TOKEN_QUERY.

Второй параметр, *DisableAllPrivileges*, определяет, будут ли выключены все привилегии, которыми обладает маркер доступа. Если этот параметр принимает значение равное TRUE, функция выключит все привилегии.

Третий параметр, *NewState*, указывает структуру TOKEN_PRIVILEGES, содержащую список привилегий, состояние которых необходимо изменить. При этом для каждой привилегии поле *Attributes* структуры LUID_AND_ATTRIBUTES определяет новое состояние. Это поле должно принимать одно из следующих значений:

- 0 – выключить привилегию;
- SE_PRIVILEGE_ENABLED – включить привилегию;
- SE_PRIVILEGE_REMOVED – удалить привилегию из маркера доступа.

Следует отметить, что параметр *NewState* игнорируется, если параметр *DisableAllPrivileges* принимает значение равное TRUE.

Четвертый параметр, *BufferLength*, задает размер структуры (в байтах), на которую указывает параметр *PreviousState*. Это параметр должен принимать значение равное нулю, если параметр *PreviousState* установлен в NULL.

Пятый параметр, *PreviousState*, указывает на структуру TOKEN_PRIVILEGES, содержащую список привилегий с их предыдущими состояниями. Этот параметр может быть установлен в NULL.

Последний параметр, *ReturnLength*, указывает на переменную типа DWORD, в которую будет сохранен необходимый размер (в байтах) для структуры, на которую указывает параметр *PreviousState*. Этот параметр может быть установлен в NULL, если параметр *PreviousState* тоже установлен в NULL.

Если функция AdjustTokenPrivileges завершается успешно, она возвращает значение отличное от FALSE.

В листингах 5.29,5.31 показано, как изменять состояние привилегий, используя функцию `AdjustTokenPrivileges`.

Листинг 5.29. Функция, изменяющая состояние привилегии

```
1   BOOL EnablePrivilege(HANDLE hToken, LPCTSTR lpszPrivilegeName,
    BOOL bEnable)
2   {
3       TOKEN_PRIVILEGES Privileges;
4       Privileges.PrivilegeCount = 1; // количество привилегий
5
6       // установим новое состояние указанной привилегии
7       Privileges.Privileges[0].Attributes = (FALSE != bEnable) ?
    SE_PRIVILEGE_ENABLED : 0;
8
9       // определяем LUID указанной привилегии
10      BOOL bRet = LookupPrivilegeValue(NULL, lpszPrivilegeName,
    &Privileges.Privileges[0].Luid);
11
12      if (FALSE != bRet)
13      {
14          // изменяем состояние указанной привилегии
15          bRet = AdjustTokenPrivileges(hToken, FALSE, &Privileges,
    sizeof(TOKEN_PRIVILEGES), NULL, NULL);
16      } // if
17
18      return bRet;
19  } // EnablePrivilege
```

Листинг 5.30. Функция изменения состояния нескольких привилегий

```
1   BOOL AdjustPrivileges(HANDLE hToken, LPCTSTR names[],
    DWORD NewState[], DWORD dwCount)
2   {
3       // определеям размер блока памяти (в байтах)
4       DWORD cb = sizeof(TOKEN_PRIVILEGES) + (dwCount - 1) *
    sizeof(LUID_AND_ATTRIBUTES);
5
6       // выделяем блок памяти
7       PTOKEN_PRIVILEGES Privileges = (PTOKEN_PRIVILEGES)
    LocalAlloc(LPTR, cb);
8       if (NULL == Privileges) return FALSE;
9
10      BOOL bRet = TRUE;
11
12      // задаем количество указанных привилегий
13      Privileges->PrivilegeCount = dwCount;
```

```
14
15      // определяем LUID и установим новое состояние
16      // для каждой указанной привилегии
17      for (DWORD i = 0; i < dwCount && FALSE != bRet; ++i)
18      {
19          Privileges->Privileges[i].Attributes = NewState[i];
20          bRet = LookupPrivilegeValue(NULL, names[i], &Privileges-
    >Privileges[i].Luid);
21      } // for
22
23      if (FALSE != bRet)
24      {
25          // изменяем состояние указанных привилегий
26          bRet = AdjustTokenPrivileges(hToken, FALSE, Privileges,
    cb, NULL, NULL);
27      } // if
28
29      // освобождаем выделенную память
30      LocalFree(Privileges);
31
32      return bRet;
33  } // AdjustPrivileges
```

Листинг 5.31. Функция, выключающая все привилегии в маркере доступа

```
1   BOOL DisableAllPrivileges(HANDLE hToken)
2   {
3       return AdjustTokenPrivileges(hToken, TRUE, NULL, 0, NULL,
    NULL);
4   } // DisableAllPrivileges
```

Замещение маркера доступа потока

Для замещения маркера доступа потока используется функция `SetThreadToken`, которая имеет следующий прототип:

```
BOOL SetThreadToken(PHANDLE Thread, HANDLE Token);
```

Первый параметр, *`Thread`*, содержит адрес дескриптора потока, для которого будет замещен маркер доступа. Если это параметр установлен в `NULL`, маркер доступа будет замещен для вызывающего потока.

Второй параметр, *`Token`*, – дескриптор маркера доступа, который должен заместить маркер доступа в потоке. Этот дескриптор должен обладать правом доступа `TOKEN_IMPERSONATE`. При этом маркер доступа, который замещает исходный маркер доступа, должен иметь тип `TokenImpersonation`.

Если функция SetThreadToken завершается успешно, она возвращает значение отличное от FALSE.

В листинге 5.32 представлен пример создания потока, который будет выполняться в контексте безопасности пользователя «Администратор». Заметим, что в этом примере для получения маркера доступа пользователя используется функция OpenUserToken (см. листинг 5.20).

Листинг 5.32. Пример замещения маркера доступа потока

```
// получаем маркер доступа пользователя «Администратор»
HANDLE hToken = OpenUserToken(TEXT("Администратор"), TEXT("."),
TEXT("Qwerty123"), LOGON32_LOGON_INTERACTIVE, TOKEN_IMPERSONATE,
NULL, TokenImpersonation, SecurityImpersonation);

if (NULL != hToken)
{
    // создаем поток и приостанавливаем его работу
    HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0, ThreadFunc,
NULL, CREATE_SUSPENDED, NULL);

    if (NULL != hThread)
    {
        // замещаем маркер доступа потока
        BOOL bRet = SetThreadToken(&hThread, hToken);

        if (FALSE != bRet)
        {
            /* исходный маркер доступа успешно замещен */

            ResumeThread(hThread); // возобновляем работу потока
        } // if
        else
        {
            /* не удалось заместить исходный маркер доступа */

            TerminateThread(hThread, 0); // завершаем поток
        } // else

        CloseHandle(hThread); // закрываем дескриптор потока
    } // if

    CloseHandle(hToken); // закрываем дескриптор маркера доступа
} // if
```

Кроме того в Win32 API имеется функция ImpersonateLoggedOnUser, которая позволяет замещать исходный маркер доступа вызыва-

ющего потока маркером доступ пользователя. При этом маркер доступа может иметь любой тип. Функция `ImpersonateLoggedOnUser` имеет следующий прототип:

```
BOOL ImpersonateLoggedOnUser(HANDLE hToken);
```

Параметр *hToken* – дескриптор маркера доступа пользователя, который должен заместить маркер доступа в вызывающем потоке. Если маркер доступа имеет тип `TokenPrimary`, дескриптор должен обладать правами доступа `TOKEN_QUERY` и `TOKEN_DUPLICATE`. Если же маркер доступа имеет тип `TokenImpersonation`, дескриптор должен обладать правами доступа `TOKEN_QUERY` и `TOKEN_IMPERSONATE`.

При успешном завершении функция `ImpersonateLoggedOnUser` возвращает значение отличное от `FALSE`.

Если необходимо вернутся к исходному маркеру доступа, можно использовать функцию `RevertToSelf`, которая завершает замещение маркера доступа в вызывающем потоке. Функция `RevertToSelf` имеет следующий прототип:

```
BOOL RevertToSelf();
```

Если функция `RevertToSelf` завершается успешно, она возвращает значение отличное от `FALSE`.

В следующем примере показано, как использовать функции `ImpersonateLoggedOnUser` и `RevertToSelf` для получения доступа к защищаемому объекту (файлу) по сети.

Листинг 5.33. Пример замещения маркера доступа вызывающего потока

```
1   HANDLE hToken = NULL;
2
3   // получаем маркер доступа пользователя «Гость»
4   BOOL bRet = LogonUser(TEXT("Гость"), TEXT("."), TEXT(""),
    LOGON32_LOGON_NETWORK_CLEARTEXT, LOGON32_PROVIDER_DEFAULT,
    &hToken);
5
6   if (FALSE != bRet)
7       bRet = ImpersonateLoggedOnUser(hToken); // замещаем маркер
    доступа потока
8
9   if (FALSE != bRet)
10  {
11      // обращение к сетевому ресурсу \\Андрей-ПК\Share\Sample.txt
12      HANDLE hFile = CreateFile(TEXT("\\\\Андрей-ПК\\Share\\Sample.
    txt"), GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
```

```
13
14      if (INVALID_HANDLE_VALUE != hFile)
15      {
16          /* здесь можно работать с файлом в контексте безопасности
    пользователя «Гость» */
17
18          CloseHandle(hFile); // закрываем дескриптор файла
19      } // if
20
21      RevertToSelf(); // завершаем замещение маркера доступа
22  } // if
23
24  // закрываем дескриптор маркера доступа
25  if (NULL != hToken) CloseHandle(hToken);
```

Нужно отметить, что при использовании типа входа `LOGON32_LOGON_NETWORK_CLEARTEXT` функция `CreateFile` завершится ошибкой, если на компьютере «Андрей-ПК» учетная запись «Гость» не будет обладать правом `SeNetworkLogonRight` (*доступ к компьютеру из сети*).

Задание к работе

1. Разработать в Visual C++ приложение Win32, которое:
   - должно выводить SID локального компьютера;
   - должно выводить SID учетной записи текущего пользователя;
   - должно выводить имена учетных записей для хорошо известных SID, указанных в варианте задания;
   - должно выводить список привилегий и прав учетной записи текущего пользователя;
   - должно выводить списки привилегий и прав учетных записей хорошо известных SID, указанных в варианте задания.
2. Разработать в Visual C++ оконное приложение Win32, которое:
   - должно выводить в своем окне список всех процессов;
   - должно выводить имя и SID учетной записи пользователя, связанной с маркером доступа выбранного процесса;
   - должно выводить список групп, связанных с маркером доступа выбранного процесса;
   - должно выводить список привилегий, которыми обладает маркер доступа выбранного процесса;
   - должно для выбранного процесса проверять наличие в его маркере доступа привилегий, указанных в варианте задания;

для привилегий, которые имеются в маркере доступа, необходимо определить состояние

- должно иметь возможность изменять состояние привилегий, которыми обладает маркер доступа выбранного процесса.

3. Разработать в Visual C++ приложение Win32, которое для выбранного файла или каталога:
   - должно выводить список разрешений (DACL);
   - должно иметь возможность добавлять и удалять ACE;
   - должно выводить и изменять имя учетной записи владельца.
4. Изменить приложение Win32, разработанное в п. 3 задания лабораторной работы №4, добавив возможность выполнять соответствующую операцию с файлами и каталогами от имени задаваемой учетной записи пользователя.

   Имя и пароль учетной записи пользователя должны запрашиваться всякий раз, когда приложение не может выполнить операцию с файлами и каталогами от имени текущего пользователя.
5. Протестировать работу разработанных приложений на компьютере под управлением Windows XP (или выше). Результаты тестирования отразить в отчете.
6. Исследовать работу разработанных приложений с помощью утилиты Process Explorer. Результаты исследования отразить в отчете.
7. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Хорошо известные SID (п. 1 задания)	Привилегии (п. 2 задания)
1,16	S-1-1-0 S-1-5-7 S-1-5-18 S-1-5-21-X-X-X-500 S-1-5-32-545 S-1-5-32-547	SeAssignPrimaryTokenPrivilege SeAuditPrivilege SeBackupPrivilege SeChangeNotifyPrivilege SeCreateGlobalPrivilege
2,17	S-1-2-1 S-1-5-3 S-1-5-19 S-1-5-21-X-X-X-501 S-1-5-32-546 S-1-5-32-555	SeCreatePermanentPrivilege SeEnableDelegationPrivilege SeIncreaseBasePriorityPrivilege SeManageVolumePrivilege SeShutdownPrivilege

№	Хорошо известные SID (п. 1 задания)	Привилегии (п. 2 задания)
3,18	S-1-1-0 S-1-5-6 S-1-5-20 S-1-5-21-X-X-X-500 S-1-5-32-544 S-1-5-32-555	SeCreateSymbolicLinkPrivilege SeImpersonatePrivilege SeIncreaseQuotaPrivilege SeMachineAccountPrivilege SeProfileSingleProcessPrivilege
4,19	S-1-1-0 S-1-5-3 S-1-5-19 S-1-5-21-X-X-X-501 S-1-5-32-544 S-1-5-32-556	SeCreateTokenPrivilege SeIncreaseBasePriorityPrivilege SeIncreaseWorkingSetPrivilege SeManageVolumePrivilege SeShutdownPrivilege
5,20	S-1-1-0 S-1-2-1 S-1-5-20 S-1-5-21-X-X-X-500 S-1-5-32-545 S-1-5-32-555	SeCreatePagefilePrivilege SeCreatePermanentPrivilege SeCreateSymbolicLinkPrivilege SeCreateTokenPrivilege SeDebugPrivilege
6,21	S-1-2-1 S-1-5-6 S-1-5-18 S-1-5-21-X-X-X-501 S-1-5-32-547 S-1-5-32-555	SeImpersonatePrivilege SeLoadDriverPrivilege SeMachineAccountPrivilege SeSecurityPrivilege SeTakeOwnershipPrivilege
7,22	S-1-1-0 S-1-5-2 S-1-5-20 S-1-5-21-X-X-X-500 S-1-5-32-546 S-1-5-32-547	SeEnableDelegationPrivilege SeIncreaseBasePriorityPrivilege SeLockMemoryPrivilege SeManageVolumePrivilege SeShutdownPrivilege
8,23	S-1-2-1 S-1-5-2 S-1-5-18 S-1-5-21-X-X-X-501 S-1-5-32-544 S-1-5-32-545	SeEnableDelegationPrivilege SeIncreaseQuotaPrivilege SeLockMemoryPrivilege SeProfileSingleProcessPrivilege SeRestorePrivilege
9,24	S-1-2-1 S-1-5-4 S-1-5-19 S-1-5-21-X-X-X-500 S-1-5-32-547 S-1-5-32-556	SeIncreaseBasePriorityPrivilege SeIncreaseWorkingSetPrivilege SeLoadDriverPrivilege SeManageVolumePrivilege SeShutdownPrivilege

№	Хорошо известные SID (п. 1 задания)	Привилегии (п. 2 задания)
10,25	S-1-1-0 S-1-5-11 S-1-5-20 S-1-5-21-X-X-X-501 S-1-5-32-546 S-1-5-32-556	SeEnableDelegationPrivilege SeImpersonatePrivilege SeIncreaseBasePriorityPrivilege SeIncreaseQuotaPrivilege SeIncreaseWorkingSetPrivilege
11,26	S-1-2-1 S-1-5-11 S-1-5-18 S-1-5-21-X-X-X-500 S-1-5-32-545 S-1-5-32-556	SeLoadDriverPrivilege SeLockMemoryPrivilege SeMachineAccountPrivilege SeManageVolumePrivilege SeProfileSingleProcessPrivilege
12,27	S-1-1-0 S-1-5-4 S-1-5-19 S-1-5-21-X-X-X-501 S-1-5-32-545 S-1-5-32-546	SeRelabelPrivilege SeRemoteShutdownPrivilege SeRestorePrivilege SeSecurityPrivilege SeShutdownPrivilege
13,28	S-1-2-1 S-1-5-7 S-1-5-20 S-1-5-21-X-X-X-500 S-1-5-32-544 S-1-5-32-546	SeSyncAgentPrivilege SeSystemEnvironmentPrivilege SeSystemProfilePrivilege SeSystemtimePrivilege SeTakeOwnershipPrivilege
14,29	S-1-2-1 S-1-5-1 S-1-5-19 S-1-5-21-X-X-X-501 S-1-5-32-544 S-1-5-32-547	SeRestorePrivilege SeTcbPrivilege SeTimeZonePrivilege SeTrustedCredManAccessPrivilege SeUndockPrivilege
15,30	S-1-1-0 S-1-5-1 S-1-5-20 S-1-5-21-X-X-X-500 S-1-5-32-555 S-1-5-32-556	SeIncreaseBasePriorityPrivilege SeIncreaseWorkingSetPrivilege SeLockMemoryPrivilege SeMachineAccountPrivilege SeSystemEnvironmentPrivilege

Контрольные вопросы

1. Какие объекты Windows относятся к защищаемым объектам?
2. Что в модели безопасности Windows является субъектом безопасности?

3. Какие существуют типы учетных записей?
4. Чем отличаются локальные и доменные учетные записи?
5. Какие существуют встроенные учетные записи?
6. Какие учетные записи называются служебными?
7. Как различаются группы по области действия?
8. Какие существуют встроенные группы?
9. Какие существуют специальные группы?
10. Что такое идентификатор безопасности?
11. Какие существуют хорошо известные идентификаторы безопасности?
12. Какие функции Win32 API следует использовать для получения идентификатора безопасности учетной записи пользователя, компьютера или группы?
13. Что такое права учетной записи? Какие существую права учетной записи?
14. Что такое привилегии? Какие существую привилегии?
15. Для чего предназначен объект политики безопасности?
16. Какие функции Win32 API следует использовать для открытия и закрытия объекта политики безопасности?
17. Какие функции Win32 API следует использовать для настройки и перечисления привилегий и прав учетной записи пользователя?
18. Что такое дескриптор безопасности? Из каких компонентов состоит дескриптор безопасности?
19. Что такое владелец и основная группа защищаемого объекта?
20. Что такое списки контроля доступа? Какие бывают виды списков контроля доступа?
21. Что такое элемент контроля доступа? Какие различают типы элементов контроля доступа?
22. В каком порядке должны располагаться элементы в списках контроля доступа?
23. Что такое абсолютный и относительный формат дескриптора безопасности? В чем заключаются различия между ними?
24. Какие существуют управляющие флаги дескриптора безопасности?
25. Какие функции Win32 API следует использовать для работы с управляющими флагами дескриптора безопасности?

26. Какие функции Win32 API следует использовать для работы с владельцем и основной группой в дескрипторе безопасности?
27. Какие функции Win32 API следует использовать для работы со списками контроля доступа в дескрипторе безопасности?
28. Какие функции Win32 API следует использовать для работы с дескриптором безопасности?
29. Что такое маркер доступа? Какую информацию включает в себя маркер доступа?
30. Что такое первичный и замещающий маркер доступа? В чем заключаются различия между ними?
31. Какие функции Win32 API следует использовать для открытия маркера доступа?
32. Какую функцию Win32 API следует использовать для получения маркера доступа пользователя?
33. Какую функцию Win32 API следует использовать для дублирования маркера доступа?
34. Какую функцию Win32 API следует использовать для получения информации из маркера доступа?
35. В каких состояниях могут находиться привилегии, которые содержит маркер доступа?
36. Какие функции Win32 API следует использовать для работы привилегиями, которые содержит маркер доступа?
37. Какие функции Win32 API следует использовать для замещения маркера доступа потока?
38. Какую функцию Win32 API следует использовать для прекращения замещения маркера доступа потока?

ЛАБОРАТОРНАЯ РАБОТА № 6
МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ. СЛУЖБЫ WINDOWS

Цель работы

Изучение механизмов межпроцессного взаимодействия. Получение практических навыков обмена данными между несколькими приложениями Windows. Знакомство со службами Windows.

Основные понятия

Время от времени возникает необходимость обмена данными между различными приложениями. В Windows это возможно благодаря различным механизмам *межпроцессного взаимодействия* (Inter-Process Communication, IPC), которые обеспечивают обмен данными между различными процессами.

Буфер обмена

Буфер обмена (clipboard) – стандартный механизм обмена данными между приложениями Windows. Как правило, приложения используют буфер обмена для передачи текста, но могут предавать и так называемые пользовательские данные (custom data), формат которых специфичен для конкретного приложения.

Открытие и закрытие буфера обмена

Чтобы использовать буфер обмена его нужно открыть с помощью функции `OpenClipboard`:

```
BOOL OpenClipboard(HWND hWndNewOwner);
```

Параметр *hWndNewOwner* – дескриптор окна, который позволяет связать открытый буфер обмена с конкретным окном. Если этот параметр установлен в `NULL`, операционная система связывает открытый буфер обмена с вызывающим процессом.

Нужно отметить, что функция `OpenClipboard` блокирует буфер обмена. Пока буфер обмена открыт одним процессом, другие процессы не смогут его использовать. Если буфер обмена уже открыт, функция `OpenClipboard` завершится ошибкой и вернет значение `FALSE`. Если же функция `OpenClipboard` завершена успешно, возвращаемое значение отлично от `FALSE`.

Когда работа с буфером обмена завершена, его следует закрыть вызовом функции `CloseClipboard`:

```
BOOL CloseClipboard();
```

Если функция `CloseClipboard` завершена успешно, возвращается значение отличное от `FALSE`. После этого буфер обмена станет доступен другим процессам.

Копирование данных в буфер обмена

Перед тем как копировать данные в буфер обмена, следует удалить его содержимое, вызвав функцию `EmptyClipboard`:

```
BOOL EmptyClipboard();
```

Если функция `EmptyClipboard` завершена успешно, возвращается значение отличное от `FALSE`.

Для того чтобы разместить данные в открытом буфере обмена следует использовать функцию `SetClipboardData`:

```
HANDLE SetClipboardData(UINT uFormat, HANDLE hMem);
```

Первый параметр, *`uFormat`*, определяет формат размещаемых данных. Этот параметр может принимать значение `CF_UNICODETEXT` (текстовый формат Unicode) или `CF_TEXT` (текстовый формат ANSI). Кроме того доступны и другие стандартные форматы (см. в документации Platform SDK).

Второй параметр, *`hMem`*, – дескриптор блока памяти, в котором размещены данные. Этот дескриптор должен быть получен вызовом функции `GlobalAlloc` с флагом `GMEM_MOVEABLE`.

Если функция `SetClipboardData` завершена успешно, возвращается дескриптор, переданный в параметре *`hMem`*. Если же функция завершается ошибкой, возвращается `NULL`.

Важно отметить, что нельзя освобождать блок памяти, в котором размещены данные до тех пор, пока не будет вызвана функция `CloseClipboard`.

В листинге 6.1 представлен пример, демонстрирующий передачу текста в буфер обмена.

Листинг 6.1. Функция копирования текста в буфер обмена

```
1   BOOL WriteClipboard(LPCTSTR lpszText)
2   {
3       BOOL bRet = OpenClipboard(NULL); // открывем буфер обмена
4
5       if (FALSE != bRet)
6       {
```

```
7           EmptyClipboard(); // удаляем содержимое буфера обмена
8
9           // определяем размер (в байтах) блока памяти
10          DWORD cb = (_tcslen(lpszText) + 1) * sizeof(TCHAR);
11
12          // выделяем блок памяти под текст
13          HANDLE hMem = GlobalAlloc(GMEM_MOVEABLE, cb);
14          bRet = (NULL != hMem) ? TRUE : FALSE;
15
16          if (FALSE != bRet)
17          {
18              // копируем текст в блок памяти
19              StringCbCopy((LPTSTR)GlobalLock(hMem), cb, lpszText);
20              GlobalUnlock(hMem);
21
22  #ifdef UNICODE
23              UINT uFormat = CF_UNICODETEXT;
24  #else
25              UINT uFormat = CF_TEXT;
26  #endif /* UNICODE */
27
28              // передаем данные в буфер обмена
29              HANDLE hRet = SetClipboardData(uFormat, hMem);
30              bRet = (NULL != hRet) ? TRUE : FALSE;
31          } // if
32
33          CloseClipboard(); // закрываем буфер обмена
34
35          // освобождаем блок памяти
36          if (NULL != hMem) GlobalFree(hMem);
37      } // if
38
39      return bRet;
40  } // WriteClipboard
```

Чтение данных из буфера обмена

Перед тем как читать данные из буфера обмена, следует удостовериться в том, что нужные данные в нем есть. Это можно сделать с помощью функции `IsClipboardFormatAvailable`:

```
BOOL IsClipboardFormatAvailable(UINT uFormat);
```

Параметр *`uFormat`* указывает формат требуемых данных. Если данные указанного формата имеются в буфере обмена, функция `IsClipboardFormatAvailable` возвращает значение `TRUE`, иначе – `FALSE`.

Чтобы извлечь данные из открытого буфера обмена следует использовать функцию GetClipboardData:

```
HANDLE GetClipboardData(UINT uFormat);
```

Если функция GetClipboardData завершена успешно, возвращается дескриптор блока памяти, в котором размещены данные указанного формата. Если же функция завершается ошибкой, возвращается NULL.

Важно отметить, что не нужно освобождать блок памяти, на который указывает возвращаемый дескриптор, так как им управляет буфер обмена, а не вызывающий процесс. Кроме того нельзя обращаться к этому дескриптору после того, как была вызвана функция EmptyClipboard или CloseClipboard.

В следующем примере демонстрируется извлечение текста из буфера обмена.

Листинг 6.2. Функция копирования текста из буфера обмена

```
1   BOOL ReadClipboard(LPTSTR lpszText, DWORD cch)
2   {
3   #ifdef UNICODE
4       UINT uFormat = CF_UNICODETEXT;
5   #else
6       UINT uFormat = CF_TEXT;
7   #endif /* UNICODE */
8
9       // проверяем есть ли в буфере обмена данные нужного формата
10      BOOL bRet = IsClipboardFormatAvailable(uFormat);
11
12      if (FALSE != bRet) // если да
13          bRet = OpenClipboard(NULL); // открывем буфер обмена
14
15      if (FALSE != bRet)
16      {
17          // получаем дескриптор блока памяти
18          HANDLE hMem = GetClipboardData(uFormat);
19          bRet = (NULL != hMem) ? TRUE : FALSE;
20
21          if (FALSE != bRet)
22          {
23              // копируем текст из полученного блока памяти
24              StringCchCopy(lpszText, cch,
    (LPCTSTR)GlobalLock(hMem));
25              GlobalUnlock(hMem);
26          } // if
27
28          CloseClipboard(); // закрываем буфер обмена
```

```
    } // if

    return bRet;
} // ReadClipboard
```

Передача пользовательских данных

Буфер обмена также можно использовать для передачи данных, формат которых специфичен для конкретного приложения. Для этого приложение должно зарегистрировать свой собственный формат данных и затем использовать его при работе с буфером обмена.

Для регистрации нового формата данных в буфере обмена применяется функция `RegisterClipboardFormat`:

```
UINT RegisterClipboardFormat(LPCTSTR lpszFormat);
```

Параметр *`lpszFormat`* указывает на строку, которая определяет имя нового формата (регистр символов не учитывается).

Если функция `RegisterClipboardFormat` завершается успешно, она возвращает значение в диапазоне от `0xC000` до `0xFFFF`, которое идентифицирует новый формат данных. Далее это значение может использоваться при вызове функций `SetClipboardData`, `GetClipboardData` и `IsClipboardFormatAvailable`. Если же функция `RegisterClipboardFormat` завершается ошибкой, возвращаемое значение равно нулю.

Следует отметить важный момент, если формат с указанным именем уже существует, новый формат не регистрируется, а возвращаемое значение идентифицирует уже существующий формат. Это дает возможность разным приложениям обмениваться данными, используя один и тот же формат данных в буфере обмена.

В листингах 6.3,6.4 проиллюстрировано, как можно осуществлять передачу пользовательских данных в буфер обмена и из него.

Листинг 6.3. Функция копирования данных в буфер обмена

```
BOOL WriteClipboard(LPCTSTR lpszFormat, LPCVOID lpData, DWORD cb)
{
    // регистрируем формат данных
    UINT uFormat = RegisterClipboardFormat(lpszFormat);
    // открывем буфер обмена
    BOOL bRet = (0 != uFormat) ? OpenClipboard(NULL) : FALSE;

    if (FALSE != bRet)
    {
        EmptyClipboard(); // удаляем содержимое буфера обмена

        // выделяем блок памяти под данные
```

```
13          HANDLE hMem = GlobalAlloc(GMEM_MOVEABLE, cb);
14          bRet = (NULL != hMem) ? TRUE : FALSE;
15
16          if (FALSE != bRet)
17          {
18              // копируем данные в блок памяти
19              CopyMemory(GlobalLock(hMem), lpData, cb);
20              GlobalUnlock(hMem);
21
22              // передаем данные в буфер обмена
23              HANDLE hRet = SetClipboardData(uFormat, hMem);
24              bRet = (NULL != hRet) ? TRUE : FALSE;
25          } // if
26
27          CloseClipboard(); // закрываем буфер обмена
28
29          // освобождаем блок памяти
30          if (NULL != hMem) GlobalFree(hMem);
31      } // if
32
33      return bRet;
34  } // WriteClipboard
```

Листинг 6.4. Функция копирования данных из буфера обмена

```
1   BOOL ReadClipboard(LPCTSTR lpszFormat, LPVOID lpData, DWORD cb,
    LPDWORD lpcbNeeded)
2   {
3       // регистрируем формат данных
4       UINT uFormat = RegisterClipboardFormat(lpszFormat);
5
6       // проверяем если в буфере обмена данные нужного формата
7       BOOL bRet = (0 != uFormat) ?
    IsClipboardFormatAvailable(uFormat) : FALSE;
8
9       if (FALSE != bRet) // если да
10          bRet = OpenClipboard(NULL); // открывем буфер обмена
11
12      if (FALSE != bRet)
13      {
14          // получаем дескриптор блока памяти
15          HANDLE hMem = GetClipboardData(uFormat);
16          bRet = (NULL != hMem) ? TRUE : FALSE;
17
18          if (FALSE != bRet)
19          {
20              // определяем размер (в байтах) данных
```

```
21              DWORD cbMem = GlobalSize(hMem);
22
23              if (NULL != lpData)
24              {
25                  // копируем данные из полученного блока памяти
26                  CopyMemory(lpData, GlobalLock(hMem), min(cb,
    cbMem));
27                  GlobalUnlock(hMem);
28              } // if
29
30              // возвращаем полученный размер данных
31              if (NULL != lpcbNeeded) *lpcbNeeded = cbMem;
32          } // if
33
34          CloseClipboard(); // закрываем буфер обмена
35      } // if
36
37      return bRet;
38  } // ReadClipboard
```

Проецируемые в память файлы

Проецируемые в память файлы (memory-mapped files) позволяют резервировать регион в адресном пространстве процесса и передавать ему данные файлов. Проецируемые файлы, кроме всего прочего, применяются для разделения данных между несколькими процессами.

Для того чтобы спроецировать файл на адресное пространство процесса нужно выполнить следующие операции:

1. Создать или открыть файл (объект ядра «файл»), который необходимо использовать как проецируемый в память.
2. Создать объект ядра «проекция файла».
3. Отобразить объект ядра «проекция файла» на адресное пространство процесса.

Когда работа с проецируемым в память файлом будет завершена, следует выполнить три операции:

1. Отменить отображение объекта ядра «проекция файла» на адресное пространство процесса.
2. Закрыть объект ядра «проекция файла».
3. Закрыть объект ядра «файл».

Создание и открытие объекта ядра «проекция файла»

Для создания объекта ядра «проекция файла» предназначена функция CreateFileMapping:

```
HANDLE CreateFileMapping(HANDLE hFile,
    LPSECURITY_ATTRIBUTES lpAttributes, DWORD flProtect,
    DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow,
    LPCTSTR lpName);
```

Первый параметр, *hFile*, – дескриптор открытого файла, который возвращает функция CreateFile. Этот параметр так же может принимать значение INVALID_HANDLE_VALUE. При этом операционная система создаст объект ядра «проекция файла», физическая память которого выделяется из страничного файла (файла подкачки).

Второй параметр, *lpAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра. Этот параметр можно установить в NULL, тогда объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Третий параметр, *flProtect*, указывает желаемые атрибуты защиты, присваиваемые страницам физической памяти, когда они отображаются на адресное пространство процесса. Для задания этого параметра используется одно из значений в табл. 6.1. Полный перечень значений параметра *flProtect* см. в документации Platform SDK.

Таблица 6.1. Значения параметра *flProtect*

Атрибут защиты	Описание
PAGE_READONLY	Позволяет считывать данные из проецируемого в память файла. При этом файл должен быть создан или открыт с помощью функции CreateFile с флагом GENERIC_READ
PAGE_READWRITE	Позволяет считывать данные из проецируемого в память файла и записывать их. При этом файл должен быть создан или открыт с помощью функции CreateFile с комбинацией флагов GENERIC_READ|GENERIC_WRITE

Следующие два параметра, *dwMaximumSizeHigh* и *dwMaximumSizeLow*, определяют максимальный размер региона в адресном пространстве процесса. Если оба этих параметра равняются нулю, максимальный размер равен текущему размеру файла, указанного параметром *hFile*.

Последний параметр, *lpName*, указывает на строку, которая определяет имя создаваемого объекта ядра. Если этот параметр установлен в NULL, будет создан безымянный объект ядра.

Если функция `CreateFileMapping` выполняется успешно, она возвращается дескриптор созданного объекта ядра «проекция файла», а в случае ошибки `NULL`.

В следующем примере продемонстрированно создание проецируемого в память файла.

```
// создадим файл, который будет спроецирован
HANDLE hFile = CreateFile(TEXT("C:\\ProgramData\\Sample.dat"),
    GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ|FILE_SHARE_WRITE,
    NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

if (INVALID_HANDLE_VALUE != hFile)
{
    // создадим для файла объект ядра "проеция файла"
    // максимального размера 1 КБт
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
PAGE_READWRITE, 0, 1024, NULL);

    if (NULL != hFileMapping)
    {
        /* объект ядра "проеция файла" был успешно создан */
    } // if
} // if
```

Кроме того, любой поток может открыть уже созданный объект ядра «проекция файла» с помощью функции `OpenFileMapping`:

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

Первый параметр, *`dwDesiredAccess`*, определяет запрашиваемые права доступа к объекту ядра «проекция файла». Некоторые из возможных значений этого параметра перечислены в табл. 6.2. Полный перечень значений параметра *`dwDesiredAccess`* можно найти в документации Platform SDK.

Второй параметр, *`bInheritHandle`*, позволяет указать, должен ли полученный дескриптор быть наследуемым.

Третий параметр, *`lpName`*, определяет имя объекта ядра, дескриптор которого нужно получить.

Если функция `OpenFileMapping` завершается успешно, она возвращает дескриптор объекта ядра «проекция файла», увеличивая на 1 значение его счетчика. В случае ошибки возвращается `NULL`.

Следует отметить, что объекты «проекция файла», так же как и все объекты ядра, можно совместно использовать несколькими процесса-

ми одним из трех способов: именование объектов ядра, наследование или дублирование дескрипторов объектов ядра.

Таблица 6.2. Права доступа к проекции файла

Значение	Описание
FILE_MAP_ALL_ACCESS	Все возможные права доступа
FILE_MAP_READ	Разрешает использование дескриптора объекта ядра «проекция файла» для чтения данных из проецируемого в память файла
FILE_MAP_WRITE	Разрешает использование дескриптора объекта ядра «проекция файла» для чтения данных из проецируемого в память файла и записи их. При этом объект ядра «проекция файла» должен быть создан с атрибутом защиты PAGE_READWRITE

Когда необходимость в объекте ядра «проекция файла» отпадет, необходимо его закрыть вызовом функции `CloseHandle`. Конечно, по завершении процесс автоматически закроет все открытые объекты. Однако все же лучше закрывать их самостоятельно.

Проецирование в адресное пространство

Когда объект «проекция файла» создан, нужно, чтобы операционная система зарезервировала регион в адресном пространстве процесса и передать ему данные из файла. Для этого используется функция `MapViewOfFile`:

```
LPVOID MapViewOfFile(HANDLE hFileMappingObject,
   DWORD dwDesiredAccess, DWORD dwFileOffsetHigh,
   DWORD dwFileOffsetLow, SIZE_T dwNumberOfBytesToMap);
```

Первый параметр, *hFileMappingObject*, – дескриптор объекта ядра «проекция файла», возвращаемый либо функцией `CreateFileMapping`, либо функцией `OpenFileMapping`.

Второй параметр, *dwDesiredAccess*, определяет уровень доступа к данным (см. табл. 6.2).

Параметры *dwFileOffsetHigh* и *dwFileOffsetLow* представляют собой 64-разрядное значение, которое представляет собой смещение в файле. Важно отметить, что смещение в файле должно быть кратно гранулярности выделения памяти в Windows (в настоящее время она составляет 64 Кб).

Последний параметр, *dwNumberOfBytesToMap*, указывает сколько байт данных из файла должно быть спроецировано на адресное про-

странство процесса. Если этот параметр равен нулю, будут спроецированы байты, начиная с указанного смещения и до конца файла.

Если функция `MapViewOfFile` завершается успешно она возвращает начальный адрес региона, созданного в адресном пространстве процесса. Если же функция завершается ошибкой, возвращаемое значение равно `NULL`.

Для повышения производительности при работе с объектом ядра «проекция файла» операционная система буферизирует данные спроецированного файла и не обновляет не медленно их на диске. При необходимости можно заставить операционную систему записать измененные данные (все или частично) на диск, с помощью вызова функции `FlushViewOfFile`:

```
BOOL FlushViewOfFile(LPCVOID lpBaseAddress,
    SIZE_T dwNumberOfBytesToFlush);
```

Первый параметр, *`lpBaseAddress`*, указывает на адрес блока байтов, которые записываются на диск.

Второй параметр, *`dwNumberOfBytesToFlush`*, указывает число копируемых байтов. Если этот параметр равен нулю, на диск будут записаны байты, начиная с адреса, указанного параметром *`lpBaseAddress`*. и до конца зарезервированного региона.

Если функция `FlushViewOfFile` завершается успешно, возвращаемое значение отлично от `FALSE`.

Когда более нет необходимости в данных спроецированного файла, следует освободить выделенный регион адресного пространства вызовом функции `UnmapViewOfFile`:

```
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress);
```

Параметр *`lpBaseAddress`* указывает на адрес возвращаемого региона. Значение этого параметра должно совпадать со значением, которое возвращает функция `MapViewOfFile`. Если функция `UnmapViewOfFile` завершается успешно, возвращаемое значение отлично от `FALSE`.

Использование проецируемых в память файлов

Проецируемые в память файлы могут применяться для обмена данными между различными процессами. При этом передаваемые данные имеют формат специфичный только для этих процессов.

Следует отметить, что, как правило, для обмена данными между различными процессами используется страничный файл (файл подкачки). Однако обычные файлы также могут использоваться.

В листингах 6.5,6.6 показаны функции передачи данных с помощью спроецированного в память файла.

Листинг 6.5. Функция записи данных в спроецированный файл

```
1   BOOL WriteFileMapping(HANDLE hFileMapping, LPCVOID lpData,
    DWORD cb)
2   {
3       // проецируем файл на регион в адресном пространстве процесса
4       PBYTE pbFile = (PBYTE)MapViewOfFile(hFileMapping,
    FILE_MAP_WRITE, 0, 0, cb);
5       if (NULL == pbFile) return FALSE;
6
7       // записываем в выделенный регион передаваемые данные
8       CopyMemory(pbFile, lpData, cb);
9
10      // принудительно записываем данные из региона на диск
11      BOOL bRet = FlushViewOfFile(pbFile, cb);
12
13      // освобождаем выделенный регион
14      UnmapViewOfFile(pbFile);
15
16      return bRet;
17  } // WriteFileMapping
```

Листинг 6.6. Функция чтения данных из спроецированного файла

```
1   BOOL ReadFileMapping(HANDLE hFileMapping, LPVOID lpData,
    DWORD cb)
2   {
3       // проецируем файл на регион в адресном пространстве процесса
4       PBYTE pbFile = (PBYTE)MapViewOfFile(hFileMapping,
    FILE_MAP_READ, 0, 0, cb);
5       if (NULL == pbFile) return FALSE;
6
7       // считываем из выделенного региона переданные данные
8       CopyMemory(lpData, pbFile, cb);
9
10      // освобождаем выделенный регион
11      UnmapViewOfFile(pbFile);
12
13      return TRUE;
14  } // ReadFileMapping
```

Оконные сообщения

Помимо уведомления о том или ином событии, оконные сообщения (windows messages) можно использовать для обмена данными между окнами различных приложений.

Передача текстовых данных

Для передачи текстовых данных другому окну используется сообщение WM_SETTEXT. В этом сообщении параметр *wParam* не используется, а параметр *lParam* используется как указатель на строку, содержащую текст.

```
1   // отправляем сообщение WM_SETTEXT
2   // с текстом «Примет мир!»
3   SendMessage(hWnd, WM_SETTEXT, 0, (LPARAM)TEXT("Привет мир!"));
```

Отметим, что при отправке сообщения WM_SETTEXT в параметре *lParam* передается адрес строки, расположенной в адресном пространстве вызывающего процесса. Поэтому, если окно-получатель было создано вызывающим процессом, в оконную процедуру этого окна передается адрес строки, который был указан при отправке сообщения. Если же окно-получатель было создано другим процессом, указанная строка копируется из адресного пространства вызывающего процесса в проекцию файла (см. раздел «Проецируемые в память файлы») и сообщение отправляется соответствующему потоку другого процесса, передавая адрес строки, расположенной в проекции файла. После обработки этого сообщения, проекция файла уничтожается. (Заметим, что описанный алгоритм выполняется при отправке любого сообщения параметры *wParam* или *lParam*, которого передают указатель на какую-либо структуру данных.)

Для получения текстовых данных от окна используется сообщение WM_GETTEXT. В этом сообщении параметр *wParam* задает максимальный размер буфера (в символах), включая нуль-символ в конце, а параметр *lParam* указывает на буфер, принимающий текст. Если оконная процедура обрабатывает это сообщение, она должна вернуть число символов, скопированных в буфер (нуль-символ в конце не учитывается).

```
1   TCHAR szText[255];
2   SendMessage(hWnd, WM_GETTEXT, _countof(szText), (LPARAM)szText);
```

Если окно-получатель было создано другим процессом, создается проекция файла, через которую выполняется обмен данными. После того, как текст будет скопирован, проекция файла будет удалена.

Чтобы определить размера текстового буфера, следует отправить сообщение WM_GETTEXTLENGTH. Если оконная процедура обрабатывает это сообщение, она должна вернуть длину текста, не учитывая нуль-символ в конце.

Обрабатывая сообщения WM_SETTEXT, WM_GETTEXTLENGTH и WM_GETTEXT окна могут обмениваться текстовыми данными (листинг 6.7).

Листинг 6.7. Обработка оконных сообщений для передачи текстовых данных

```
case WM_SETTEXT:
    {
        TCHAR szText[255];
        StringCchCopy(szText, _countof(szText), (LPCTSTR)lParam);
    }
    return 0;

case WM_GETTEXTLENGTH:
    return _tcslen(TEXT("Привет мир!"));

case WM_GETTEXT:
    StringCchCopy((LPTSTR)lParam, wParam, TEXT("Привет мир!"));
    return _tcslen((LPTSTR)lParam);
```

Передача пользовательских данных

Для передачи пользовательских данных окну используется сообщение WM_COPYDATA. В этом сообщении параметр *wParam* представляет собой дескриптор окна-отправителя, а параметр *lParam* – указатель на структуру COPYDATASTRUCT, которая описывается следующим образом:

```
typedef struct tagCOPYDATASTRUCT {
  ULONG_PTR dwData; // данные
  DWORD cbData; // размер данных (в байтах)
  PVOID lpData; // данные
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

Поле *dwData* содержит данные, которые будут переданы окну. В него разрешается записывать любые значение. Например, в этом поле можно указать тип передаваемых данных.

Поле *cbData* содержит размер буфера, на который указывает поле *lpData*. Поле *lpData* указывает на буфер, в котором хранятся передаваемые данные. Это поле может принимать значение NULL.

Перед тем, как передать данные через сообщение WM_COPYDATA, нужно инициализировать структуру COPYDATASTRUCT (листинг 6.8).

Листинг 6.8. Функция передачи данных через сообщение WM_COPYDATA

```
BOOL SendData(HWND hWnd, HWND hwndFrom, LPCVOID lpData,
DWORD cbData, ULONG_PTR dwData)
{
    // инициализируем структуру
    COPYDATASTRUCT cds;

```

```
6       cds.lpData = (PVOID)lpData;
7       cds.cbData = cbData;
8       cds.dwData = dwData;
9
10      // отправляем сообщение WM_COPYDATA
11      return SendMessage(hWnd, WM_COPYDATA, (WPARAM)hwndFrom,
    (LPARAM)&cds);
12  } // SendData
```

Если оконная процедура обрабатывает сообщение WM_COPYDATA, она должна вернуть значение TRUE, в противном случае она должна вернуть – FALSE.

Листинг 6.9. Обработка сообщения WM_COPYDATA

```
1   case WM_COPYDATA:
2       {
3           HWND hwndFrom = (HWND)wParam;
4           PCOPYDATASTRUCT pcds = (PCOPYDATASTRUCT)lParam;
5
6           /* обработка полученных данных */
7       }
8       return TRUE;
```

Отправка сообщений

Сообщения WM_SETTEXT, WM_GETTEXTLENGTH, WM_GETTEXT и WM_COPYDATA должны отправляться с помощью функции SendMessage. Если использовать функцию PostMessage, она завершится ошибкой.

Когда функция SendMessage отправляет сообщение окну, созданному другим потоком, она, как и функция PostMessage добавляет сообщение в очередь оконных сообщений потока-получателя. Функция SendMessage не вернет управление вызывающему потоку, пока поток-получатель не обработает полученное сообщение. Следует отметить, что приоритет сообщений, добавленных в очередь с помощью функции SendMessage, выше приоритета сообщений, добавленных с помощью функции PostMessage.

При использовании функции SendMessage возможно «зависание» вызывающего потока при неправильной обработке сообщений потоком-получателем. Для предотвращения такой ситуации в Win32 API имеются следующие функции:

```
LRESULT SendMessageTimeout(HWND hWnd, UINT Msg,
    WPARAM wParam, LPARAM lParam,
    UINT fuFlags, UINT uTimeout, PDWORD_PTR lpdwResult);
```

```
BOOL SendMessageCallback(HWND hWnd, UINT Msg,
    WPARAM wParam, LPARAM lParam,
    SENDASYNCPROC lpCallBack, ULONG_PTR dwData);

BOOL SendNotifyMessage(HWND hWnd, UINT Msg, WPARAM wParam,
    LPARAM lParam);
```

Первая функция, `SendMessageTimeout`, позволяет задавать отрезок времени, в течение которого вызывающий поток готов ждать ответа на отправленное сообщение. Первые четыре параметра этой функции идентичны параметрам функции `SendMessage`.

Пятый параметр, *`fuFlags`*, содержит один (или комбинацию) из флагов, перечисленных в табл. 6.3.

Таблица 6.3. Флаги функции `SendMessageTimeout`

Флаг	Описание
`SMTO_ABORTIFHUNG`	Если этот флаг установлен, функция первым делом проверит, завис ли поток-получатель, и, если да, немедленно вернет управление
`SMTO_BLOCK`	Если этот флаг установлен, вызывающий поток приостанавливает обработку любых оконных сообщений, пока функция не вернет управление
`SMTO_NORMAL`	Если этот флаг установлен, функция вернет управление после того, как время ожидания истекло
`SMTO_NOTIMEOUTIFNOTHUNG`	Если этот флаг установлен, функция игнорирует ограничение по времени, если поток-получатель не завис

Шестой параметр, *`uTimeout`*, определяет время (в миллисекундах), в течение которого вызывающий поток готов ждать ответа. Последний параметр, *`lpdwResult`*, указывает на переменную, в которую будет записан результат обработки сообщения.

При успешном выполнении функция `SendMessageTimeout` возвращает значение отличное от нуля.

Если поток вызывает функцию `SendMessageTimeout` для отправки сообщения окну, созданному этим же потоком, она не вернет управление до тех пор, пока оконная процедура не обработает сообщение (при этом ограничение по времени игнорируется).

Вторая функция, `SendMessageCallback`, отправляет сообщение в очередь оконных сообщений потока-получателя и сразу же возвращает управление вызывающему потоку. Закончив обработку сообщения, поток-получатель отправит ответ потоку-отправителю.

Первые четыре параметра функции `SendMessageCallback` идентичны параметрам функции `SendMessage`. Пятый параметр, *`lpCallBack`*,

указывает на функцию, которая вызывается после получения ответа на оправленное сообщение. Шестой параметр, *dwData*, будет передан при вызове функции, на которую указывает параметр *lpCallBack*.

При успешном выполнении функция SendMessageCallback возвращает значение отличное от FALSE.

Функция, на которую указывает параметр *lpCallBack*, имеет следующую сигнатуру (имя может быть любым):

```
VOID CALLBACK SendAsyncProc(HWND hwnd, UINT uMsg,
    ULONG_PTR dwData, LRESULT lResult);
```

Здесь параметр *hwnd* – дескриптор окна, а параметр *uMsg* – код сообщения. Параметр *dwData* всегда получает значение, переданное функции SendMessageCallback в одноименном параметре. Последний параметр, *lResult*, содержит значение, полученное от оконной процедуры.

Если функция SendMessageCallback используется для отправки сообщения окну, созданному вызывающим потоком, она вызывает оконную процедуру, а после обработки сообщения – функцию, на которую указывает параметр *lpCallBack*. Когда эта функция вернет управление в функцию SendMessageCallback, та в свою очередь вернет управление вызывающему потоку.

Третья функция, SendNotifyMessage, отправляет сообщение в очередь оконных сообщений потока-получателя и тут же возвращает управление вызывающему потоку. Однако если сообщение отправляется окну, созданному вызывающим потоком, функция SendNotifyMessage работает точно так же, как и функция SendMessage, то есть. не возвращает управление до тех пор, пока оконная процедура не обработает сообщение.

Параметры функции SendNotifyMessage идентичны параметрам функции SendMessage. Если функция SendNotifyMessage выполняется успешно, она возвращает значение отличное от FALSE.

Почтовые ящики

Почтовый ящик (mailslot) – объект ядра, который обеспечивает симплексное (однонаправленное) межпроцессное взаимодействие. В зависимости от того, как был получен дескриптор объекта ядра «почтовый ящик», поток может либо читать данные из соответствующего почтового ящика, либо записывать данные в этот почтовый ящик.

Данные, записываемые в почтовый ящик, образуют сообщения. В почтовый ящик можно записать несколько сообщений. После прочтения сообщения удаляются из почтового ящика.

Создание и открытие почтового ящика

Для создания объекта ядра «почтовый ящик» используют функцию CreateMailslot:

```
HANDLE CreateMailslot(LPCTSTR lpName,
    DWORD nMaxMessageSize, DWORD lReadTimeout,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Первый параметр, *lpName*, указывает на строку, содержащую имя почтового ящика. Если почтовый ящик с таким же именем уже существует, функция завершится ошибкой. Имя почтового ящика должно иметь следующий вид:

```
\\.\mailslot\[путь]имя
```

Имя должно быть уникальным. Точка (.) указывает на то, что почтовый ящик создается на локальном компьютере. Имя может включать в себя несколько уровней псевдо каталогов, разделенных «\». Например, имя почтового ящика может быть таким

```
\\.\mailslot\SampleMailslot
```

или таким

```
\\.\mailslot\app\def\SampleMailslot
```

Второй параметр, *nMaxMessageSize*, задает максимальный размер сообщения (в байтах), которое может быть записано в почтовый ящик. Если этот параметр принимает значение равное нулю, сообщения могут быть любого размера.

Третий параметр, *lReadTimeout*, определяет длительность интервала ожидания (в миллисекундах) для операции чтения. Этот параметр также может принимать одно из следующих значение:

- 0 – функция чтения немедленно возвращает управление, если в почтовом ящике нет сообщений;
- MAILSLOT_WAIT_FOREVER – функция чтения ждет, пока в почтовом ящике появится сообщение.

Последний параметр, *lpSecurityAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «почтовый ящик». Если этот параметр установлен в NULL, объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

При успешном завершении функция CreateMailslot возвращает дескриптор созданного объекта ядра, иначе – INVALID_HANDLE_VALUE.

Полученный дескриптор обладает правами доступа для чтения и изменения почтового ящика, но не обладает правом доступа для записи.

Следующий пример иллюстрирует создание объекта ядра «почтовый ящик».

Листинг 6.10. Создание почтового ящика

```
1   HANDLE hMailslot = CreateMailslot(
2       TEXT("\\\\.\\mailslot\\SampleMailslot"), // указываем имя
3       0, // указываем, что сообщения не имеют ограничений в размере
4       MAILSLOT_WAIT_FOREVER, // указываем, что функции чтения будут
    ждать нового сообщения
5       NULL);
6
7   if (INVALID_HANDLE_VALUE != hMailslot)
8   {
9       /* объект ядра «почтовый ящик» успешно создан */
10  } // if
```

Открыть уже созданный объект ядра «почтовый ящик» можно с помощью функции `CreateFile`, в которой параметр *`lpFileName`* должен указывать на имя почтового ящика в следующем виде:

- `\\.\mailslot\[`*`путь`*`]`*`имя`* – определяет почтовый ящик, созданный на локальном компьютере;
- `\\`*`имя_компьютера`*`\mailslot\[`*`путь`*`]`*`имя`* – определяет почтовый ящик, созданный на компьютере с заданным именем;
- `\\`*`имя_домена`*`\mailslot\[`*`путь`*`]`*`имя`* – определяет все почтовые ящики, созданные на компьютерах, принадлежащих заданному домену. В этом случае максимальный размер сообщения всегда составляет 424 байта;
- `\\*\mailslot\[`*`путь`*`]`*`имя`* – определяет все почтовые ящики, созданные на компьютерах, принадлежащих одному домену. В этом случае максимальный размер сообщения всегда составляет 424 байта.

При вызове функции `CreateFile` для открытия объекта ядра «почтовый ящик» параметр *`dwShareMode`* должен принимать значение `FILE_SHARE_READ`, а параметр *`dwCreationDisposition`* – `OPEN_EXISTING`.

Если функция `CreateFile` выполняется успешно, она возвращает дескриптор открытого объекта ядра «почтовый ящик», в случае ошибки возвращается значение `INVALID_HANDLE_VALUE`. Возвращаемый дескриптор может обладать только правом доступа для записи.

В следующем примере продемонстрировано открытие объекта ядра «почтовый ящик».

Листинг 6.11. Открытие почтового ящика

```
HANDLE hMailslot = CreateFile(
    TEXT("\\\\.\\mailslot\\SampleMailslot"), // указываем имя
    GENERIC_WRITE, // требуем право доступа для записи
    FILE_SHARE_READ, // обязательный флаг
    NULL,
    OPEN_EXISTING, // обязательный флаг
    0, NULL);

if (INVALID_HANDLE_VALUE != hMailslot)
{
    /* объект ядра "почтовый ящик" успешно открыт */
} // if
```

Закрытие почтового ящика

Почтовый ящик считается закрытым, если процесс, который его создал, закрыл с помощью функции `CloseHandle` все соответствующие ему дескрипторы. После того, как почтовый ящик был закрыт, уже нельзя его использовать, и поэтому следует закрыть все его дескрипторы, используемые в других процессах.

Объект ядра «почтовый ящик» уничтожается, когда будут закрыты все его дескрипторы.

Использование почтовых ящиков

Для записи сообщений в почтовый ящик используется функция `WriteFile`, а для чтения – `ReadFile`.

Для того чтобы организовать двунаправленный обмен данными между двумя процессами при помощи почтовых ящиков, необходимо, чтобы каждый из этих процессов создал свой собственный объект ядра «почтовый ящик». В листинге 6.12 продемонстрирована функция, которая объединяет операции записи и чтения при работе с двумя почтовыми ящиками.

Листинг 6.12. Функция обмена данными с помощью почтовых ящиков

```
BOOL TransactMailslot(HANDLE hWriteMailslot, LPCVOID lpInBuffer,
DWORD cbInBuffer, HANDLE hReadMailslot, LPVOID lpOutBuffer,
DWORD cbOutBuffer, LPDWORD lpBytesRead)
{
    DWORD nBytes;

```

```
4       // записываем данные из буфера lpInBuffer
5       // в почтовый ящик hWriteMailslot
6       BOOL bRet = WriteFile(hWriteMailslot, lpInBuffer, cbInBuffer,
    &nBytes, NULL);
7
8       if (FALSE != bRet)
9       {
10          // ждем появления данных в почтовом ящике hReadMailslot
11          // после этого считываем данные в буфер lpOutBuffer
12          bRet = ReadFile(hReadMailslot, lpOutBuffer, cbOutBuffer,
    lpBytesRead, NULL);
13      } // if
14
15      return bRet;
16  } // TransactMailslot
```

Для обработки входящих сообщений можно организовать цикл их обработки, как показано в следующем примере:

Листинг 6.13. Цикл обработки сообщений из почтового ящика

```
1   for (;;)
2   {
3       // читаем сообщение из почтового ящика
4       BOOL bRet = ReadFile(hMailslot, ...);
5
6       if (FALSE != bRet)
7       {
8           /* здесь выполняется обработка полученного сообщения */
9       } // if
10      else
11      {
12          DWORD dwError = GetLastError();
13
14          if (ERROR_INVALID_HANDLE == dwError)
15          {
16              /* почтовый ящик был закрыт */
17              break; // выходим из цикла
18          } // if
19          else if (ERROR_INSUFFICIENT_BUFFER == dwError)
20          {
21              /* размер буфера слишком мал */
22          } // if
23      } // else
24  } // for
```

Кроме этого, для работы с почтовыми ящиками в Win32 API имеются еще две функции: GetMailslotInfo и SetMailslotInfo.

Функция GetMailslotInfo извлекает информацию, связанную с почтовым ящиком:

```
BOOL GetMailslotInfo(HANDLE hMailslot,
    LPDWORD lpMaxMessageSize, LPDWORD lpNextSize,
    LPDWORD lpMessageCount, LPDWORD lpReadTimeout);
```

Первый параметр, *hMailslot*, – дескриптор объекта ядра «почтовый ящик», информацию о котором нужно получить.

Второй параметр, *lpMaxMessageSize*, указывает на переменную типа DWORD, в которую будет записан максимальный размер сообщения (в байтах), установленный для указанного почтового ящика. Этот параметр может быть установлен в NULL.

Третий параметр, *lpNextSize*, указывает на переменную типа DWORD, в которую будет записан размер следующего сообщения (в байтах). Если следующее сообщение отсутствует, в эту переменную будет записано значение MAILSLOT_NO_MESSAGE. Параметр *lpNextSize* может быть установлен в NULL.

Четвертый параметр, *lpMessageCount*, указывает на переменную типа DWORD, в которую будет записано количество сообщений, ждущих прочтения. Этот параметр может быть установлен в NULL.

Последний параметр, *lpReadTimeout*, указывает на переменную типа DWORD, в которую будет записано время ожидания для операции чтения, установленное для указанного почтового ящика. Этот параметр может быть установлен в NULL.

При успешном завершении функция GetMailslotInfo возвращает значение отличное от FALSE.

Функция SetMailslotInfo устанавливает время ожидания для операции чтения:

```
BOOL SetMailslotInfo(HANDLE hMailslot, DWORD lReadTimeout);
```

Первый параметр, *hMailslot*, – дескриптор объекта ядра «почтовый ящик», а второй параметр, *lReadTimeout*, задает длительность интервала ожидания (в миллисекундах) для операции чтения. Параметр *lReadTimeout* может также принимать значение MAILSLOT_WAIT_FOREVER или 0. При успешном завершении функция SetMailslotInfo возвращает значение отличное от FALSE.

Важно отметить, что вызовы функций GetMailslotInfo и SetMailslotInfo завершаются ошибкой, если дескриптор объекта ядра «почтовый ящик» был получен с помощью функции CreateFile.

Каналы

Канал (pipe) – объект ядра, который обеспечивает двунаправленное межпроцессное взаимодействие. Различают два вида канала: *анонимный канал* (anonymous pipe) и *именованный канал* (named pipe).

Анонимные каналы

Анонимные каналы обеспечивают полудуплексное межпроцессное взаимодействие, для чего используются два дескриптора – один для записи данных в канал, а второй для чтения данных из этого канала.

Создание анонимного канала

Для создания объекта ядра «анонимный канал» используют функцию `CreatePipe`:

```
BOOL CreatePipe(PHANDLE hReadPipe, PHANDLE hWritePipe,
    LPSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize);
```

Первые два параметра, *`hReadPipe`* и *`hWritePipe`*, указывают на переменные типа `HANDLE`, через которые функция `CreatePipe` вернет дескрипторы, соответственно, с правами только для чтения данных из канала и только для записи данных в этот канал.

Третий параметр, *`lpPipeAttributes`*, указывает на структуру `SECURITY_ATTRIBUTES`, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра «анонимный канал». Если этот параметр установлен в `NULL`, объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Последний параметр, *`nSize`*, задает размер буфера канала (в байтах). Значение, которое передает этот параметр, является только рекомендуемым, так как операционная система сама определяет размер буфера канала. Если параметр *`nSize`* принимает значение равное нулю, будет использоваться размер по умолчанию.

При успешном выполнении функция `CreatePipe` возвращает значение отличное от `FALSE`, а в переменные, на которые указывают параметры *`hReadPipe`* и *`hWritePipe`*, будут записаны соответствующие дескрипторы.

В листинге 6.14 показан пример создания объекта ядра «анонимный канал».

Листинг 6.14. Создание анонимного канала

```
1  HANDLE hReadPipe, hWritePipe;
2
3  // создание объекта ядра «анонимный канал»
4  BOOL bRet = CreatePipe(&hReadPipe, &hWritePipe, NULL, 0);
```

```
5
6   if (FALSE != bRet)
7   {
8       /* объект ядра «анонимный канал» успешно создан */
9   } // if
```

Закрытие анонимного канала

Если процесс, в котором был создан анонимный канал, закрыл с помощью функции `CloseHandle` его дескриптор для записи, канал считается закрытым. После того, как анонимный канал был закрыт, уже нельзя его использовать, и поэтому следует закрыть все его дескрипторы с помощью функции `CloseHandle`.

Объект ядра «анонимный канал» уничтожается, когда будут закрыты все его дескрипторы.

Использование анонимного канала

Для записи данных в анонимный канал используется функция `WriteFile`, а для чтения – `ReadFile`. В следующем примере продемонстрирована функция, которая записывает данные в анонимный канал:

Листинг 6.15. Функция записи данных в анонимный канал

```
1   BOOL SendToPipe(HANDLE hWritePipe, LPCVOID lpData, DWORD cbData)
2   {
3       DWORD nBytes;
4       // записываем данные из буфера lpData в канал
5       return WriteFile(hWritePipe, lpData, cbData, &nBytes, NULL);
6   } // SendToPipe
```

Для обработки данных, поступающих из канала, можно организовать цикл их обработки (как и в случае с почтовыми ящиками):

Листинг 6.16. Цикл обработки сообщений из анонимного канала

```
1   for (;;)
2   {
3       BOOL bRet = ReadFile(hReadPipe, ...);
4
5       if (FALSE != bRet)
6       {
7           /* здесь выполняется обработка полученных данных */
8       } // if
9       else if (ERROR_BROKEN_PIPE == GetLastError())
10      {
11          /* канал был закрыт */
12          break; // выходим из цикла
```

```
13      } // if
14  } // for
```

Именованные каналы

Именованные каналы обеспечивают дуплексное межпроцессное взаимодействие между процессом-сервером и одним или несколькими процессами-клиентами. Они предоставляют больше функциональных возможностей, чем анонимные каналы.

Создание и открытие именованного канала

Для создания объекта ядра «именованный канал» используют функцию `CreateNamedPipe`:

```
HANDLE CreateNamedPipe(LPCTSTR lpName, DWORD dwOpenMode,
    DWORD dwPipeMode, DWORD nMaxInstances,
    DWORD nOutBufferSize, DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Первый параметр, *lpName*, указывает на строку, содержащую имя канала. Если канал с таким же именем уже существует, функция завершится ошибкой. Имя канала должно иметь следующий формат:

```
\\.\pipe\[путь]имя
```

Точка (.) указывает на то, что канал создается на локальном компьютере. Имя канала может включать в себя несколько уровней псевдо каталогов, разделенных «\». Например, имя канала может быть таким

```
\\.\pipe\SamplePipe
```

Второй параметр, *dwOpenMode*, определяет режим доступа к именованному каналу. Этот параметр может принимать одно из значений, перечисленных в табл. 6.4.

Таблица 6.4. Значения параметра *dwOpenMode*

Значение	Описание
PIPE_ACCESS_DUPLEX	Канал доступен для чтения и записи данных
PIPE_ACCESS_INBOUND	Канал доступен только для чтения данных
PIPE_ACCESS_OUTBOUND	Канал доступен только для записи данных

Кроме значений, перечисленных в этой таблице, параметр *dwOpenMode* может содержать флаг `FILE_FLAG_OVERLAPPED`, который позволяет выполнять асинхронные операции ввода/вывода. Также этот параметр позволяет задать уровень доступа для возвращаемого дескриптора

объекта ядра «именованный канал» (подробнее см. в документации Platform SDK).

Третий параметр, *dwPipeMode*, определяет режим работы именованного канала. Этот параметр принимает комбинацию значений, перечисленных в табл. 6.5.

Таблица 6.5. Значения параметра *dwPipeMode*

Значение	**Описание**
PIPE_TYPE_BYTE	Режим работы канала, при котором данные записываются в канал как поток отдельных байтов. Это значение не может использоваться вместе со значением PIPE_READMODE_MESSAGE
PIPE_TYPE_MESSAGE	Режим работы канала, при котором данные записываются в канал в виде отдельных сообщений
PIPE_READMODE_BYTE	Режим работы канала, при котором данные считываются из канала как поток отдельных байтов
PIPE_READMODE_MESSAGE	Режим работы канала, при котором данные считываются из канала в виде отдельных сообщений Это значение может использоваться только вместе со значением PIPE_TYPE_MESSAGE
PIPE_WAIT	Блокирующий режим работы канала. При этом режиме вызывающий процесс переводится в состояние ожидания до завершения операций в канале
PIPE_NOWAIT	Неблокирующий режим работы канала. Если операция не может быть выполнена немедленно, в неблокирующем режиме функция, работающая с таким каналом, завершается с ошибкой

Четвертый параметр, *nMaxInstances*, задает максимальное количество экземпляров создаваемого канала, а, следовательно, и количество одновременно поддерживаемых процессов-клиентов. Этот параметр может принимать значения от 1 до PIPE_UNLIMITED_INSTANCES. В последнем случае максимальное количество экземпляров ограничивается только наличием свободных системных ресурсов.

Следующие два параметра, *nOutBufferSize* и *nInBufferSize*, позволяют задать размеры (в байтах) выходного и входного буферов именованного канала. Чтобы использовать размер буфера по умолчанию, следует задать значение соответствующего параметра равным нулю.

Седьмой параметр, *nDefaultTimeOut*, определяет длительность интервала ожидания (в миллисекундах), который будет использоваться по умолчанию процессом-клиентом при ожидании готовности процесса-сервера установить с ним соединение по создаваемому каналу. Если

этот параметр принимает значение равное нулю, будет использоваться значение по умолчанию (обычно это 50 миллисекунд).

Последний параметр, *lpSecurityAttributes*, указывает на структуру SECURITY_ATTRIBUTES, которая позволяет определить параметры безопасности и наследования для создаваемого объекта ядра. Если этот параметр установлен в NULL, объект ядра будет создан с параметрами безопасности и наследования по умолчанию.

Если функция CreateNamedPipe завершается успешно, она возвращает дескриптор созданного объекта ядра «именованный канал», иначе – INVALID_HANDLE_VALUE.

Следующий пример иллюстрирует создание объекта ядра «именованный канал».

Листинг 6.17. Создание именованного канала

```
1   HANDLE hNamedPipe = CreateNamedPipe(
2       TEXT("\\\\.\\pipe\\SamplePipe"),
3       PIPE_ACCESS_DUPLEX|FILE_FLAG_OVERLAPPED,
4       PIPE_TYPE_MESSAGE|PIPE_READMODE_MESSAGE|PIPE_WAIT,
5       PIPE_UNLIMITED_INSTANCES, 0, 0, 0, NULL);
6
7   if (INVALID_HANDLE_VALUE != hNamedPipe)
8   {
9       /* объект ядра «именованный канал» успешно создан */
10  } // if
```

Как правило, именованный канал создается процессом-сервером, а процессы-клиенты открывают этот канал (образуя соединение с процессом-сервером по этому каналу). Открыть уже созданный именованный канал можно с помощью функции CreateFile, в которой параметр *lpFileName* должен указывать на имя канала в следующем виде:

– \\.\pipe\[*путь*]*имя* – определяет канал, созданный на локальном компьютере;

– *имя_компьютера*\pipe\[*путь*]*имя* – определяет канал, созданный на указанном компьютере.

При вызове функции CreateFile для открытия объекта ядра «именованный канал» параметр *dwCreationDisposition* должен принимать значение OPEN_EXISTING.

Если функция CreateFile выполняется успешно, она возвращает дескриптор открытого объекта ядра «именованный канал», в случае ошибки возвращается значение INVALID_HANDLE_VALUE. Следует отметить, что вызов функции CreateFile завершает ошибкой, если уже достигнуто максимальное количество экземпляров открываемого канала.

В следующем примере продемонстрировано открытие объекта ядра «именованный канал».

Листинг 6.18. Открытие именованного канала

```
1   HANDLE hNamedPipe = CreateFile(
2       TEXT("\\\\.\\pipe\\SamplePipe"), // указываем имя канала
3       GENERIC_READ|GENERIC_WRITE, // требуем права доступа для
    чтения и записи
4       0, NULL,
5       OPEN_EXISTING, // обязательный флаг
6       0, NULL);
7
8   if (INVALID_HANDLE_VALUE != hNamedPipe)
9   {
10      /* объект ядра "именованный канал" успешно открыт */
11  } // if
```

Закрытие именованного канала

Именованный канал считается закрытым, если процесс-сервер, который его создал, закрыл с помощью функции `CloseHandle` все соответствующие ему дескрипторы. После того, как именованный канал был закрыт, уже нельзя его использовать, и поэтому следует закрыть все его дескрипторы, используемые в процессах-клиентах.

Объект ядра «именованный канал» уничтожается, когда будут закрыты все его дескрипторы.

Использование именованного канала

После создания именованного канала процесс-сервер может ожидать, когда процесс-клиент установит с ним соединение. Для этого используется функция `ConnectNamedPipe`:

```
BOOL ConnectNamedPipe(HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped);
```

Первый параметр, *`hNamedPipe`*, – дескриптор созданного именованного канала. Второй параметр, *`lpOverlapped`*, указывает на структуру `OVERLAPPED`, которая используется, только если именованный канал был создан с флагом `FILE_FLAG_OVERLAPPED`; если это не так, этот параметр должен быть установлен в `NULL`.

Если именованный канал был создан без указания флага `FILE_FLAG_OVERLAPPED`, функция `ConnectNamedPipe` при успешном завершении возвращает значение отличное от `FALSE`. При этом функция `ConnectNamedPipe` не вернет управление, пока процесс-клиент не откроет указанный канал или не произойдет ошибка.

Если именованный канал был создан с флагом FILE_FLAG_OVERLAPPED, функция ConnectNamedPipe сразу вернет управление. При этом возвращаемое значение всегда равно FALSE. Поэтому узнать, что функция ConnectNamedPipe была завершена успешно, можно вызвав функцию GetLastError, которая вернет одно из следующих значений:

- ERROR_IO_PENDING – функция ConnectNamedPipe завершена успешно, но процесс-клиент еще не открыл указанный канал. При этом узнать, когда процесс-клиент откроет канал можно с помощью объекта ядра «событие», дескриптор которого необходимо указать в поле *hEvent* структуры OVERLAPPED, передаваемой в качестве аргумента в функцию ConnectNamedPipe. Когда процесс-клиент откроет канал, указанный объект ядра «событие» перейдет в свободное состояние;
- ERROR_PIPE_CONNECTED – функция ConnectNamedPipe завершена успешно и процесс-клиент уже открыл указанный канал. Это возможно, если процесс-клиент открыл канал между вызовами процессом-сервером функций CreateNamedPipe и ConnectNamedPipe.

Если же функция ConnectNamedPipe завершается ошибкой, функции GetLastError возвращает код ошибки, значение которого отлично от ERROR_IO_PENDING и ERROR_PIPE_CONNECTED.

После того, как процесс-сервер и процесс-клиент установили соединение по именованному каналу, они могут выполнять чтение данных из канала посредством вызова функции ReadFile и запись данных в канал с помощью функции WriteFile.

Процесс-сервер может разорвать соединение, вызвав функцию DisconnectNamedPipe, чтобы освободить экземпляр канала для соединения с другим возможным клиентом.

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

Параметр *hNamedPipe* – дескриптор именованного канала, экземпляр которого нужно освободить. Если функция DisconnectNamedPipe завершается успешно, возвращаемое значение отлично от FALSE.

В листинге 6.19 представлен пример, в котором процесс-сервер создает соединение с процессом-клиентом, взаимодействует с ним, пока тот не разорвет соединение, разрывает соединение на своей стороне, а затем образует соединение с другим процессом-клиентом.

Листинг 6.19. Цикл обработки данных от клиента

```
1   for (;;)
2   {
3       OVERLAPPED oConnect = { 0 };
```

```
4       oConnect.hEvent = hEvent;
5
6       // ожидаем процесс-клиент и образуем с ним соединение
7       ConnectNamedPipe(hNamedPipe, &oConnect);
8       DWORD dwResult = WaitForSingleObject(hEvent, INFINITE);
9
10      if (WAIT_OBJECT_0 != dwResult ||
11          ERROR_SUCCESS != oConnect.Internal)
12      {
13          break; // выходим из цикла
14      } // if
15
16      for (;;)
17      {
18          // чтение данных из канала
19          BOOL bRet = ReadFile(hNamedPipe, ...);
20          if (FALSE == bRet) break; // выходим из цикла
21
22          /* здесь выполняется обработка полученных данных */
23
24          WriteFile(hNamedPipe, ...); // запись данных в канал
25      } // for
26
27      DisconnectNamedPipe(hNamedPipe); // разрываем соединение
28  } // for
```

Процесс-клиент имеет возможность убедиться в том, что процесс-сервер готов к образованию соединения. Для этого используется функция `WaitNamedPipe`:

```
BOOL WaitNamedPipe(LPCTSTR lpNamedPipeName, DWORD nTimeOut);
```

Первый параметр, *`lpNamedPipeName`*, указывает на строку, содержащую имя канала.

Второй параметр, *`nTimeOut`*, определяет интервал времени (в миллисекундах) ожидания готовности процесса-сервера. Также этот параметр может принимать одно из следующих значение:

- `NMPWAIT_USE_DEFAULT_WAIT` – используется интервал ожидания по умолчанию, заданный в вызове функции `CreateNamedPipe` при создании именованного канала;
- `NMPWAIT_WAIT_FOREVER` – функция `WaitNamedPipe` не возвращает управление до тех пор, пока процесс-сервер не будет готов.

Если процесс-сервер будет готов к образованию соединения до истечения указанного времени, функция `WaitNamedPipe` вернет значение

отличное от FALSE. Если же время ожидания истекло или произошла ошибка, функция WaitNamedPipe возвращает значение FALSE.

Следует обратить внимание, что вызов функции WaitNamedPipe завершится ошибкой, если именованный канал к моменту ее вызова еще не был создан.

В следующем примере рассмотрены операции, выполняемые процессом-клиентом при работе с именованным каналом.

Листинг 6.20. Передача данных клиентом через именованный канал

```
1   // ожидаем готовности процесса-сервера к образованию соединения
2   BOOL bRet = WaitNamedPipe(TEXT("\\\\.\\pipe\\SamplePipe"),
    NMPWAIT_WAIT_FOREVER);
3
4   HANDLE hNPipe = INVALID_HANDLE_VALUE;
5
6   if (FALSE != bRet) // открываем именованный канал
7       hNPipe = CreateFile(TEXT("\\\\.\\pipe\\SamplePipe"), ...);
8
9   if (INVALID_HANDLE_VALUE != hNPipe)
10  {
11      // запись данных в канал
12      bRet = WriteFile(hNPipe, ...);
13
14      // чтение данных из канала
15      if (FALSE != bRet) ReadFile(hNPipe, ...);
16
17      CloseHandle(hNPipe); // разрываем соединение
18  } // if
```

Встречающуюся в этом примере последовательность вызовов функций WriteFile и ReadFile, которые выполняет процесс-клиент, можно рассматривать как единую транзакцию. В Win32 API для этого существует функция TransactNamedPipe, которая сочетает в себе функции WriteFile и ReadFile, применяемые к дескриптору канала.

```
BOOL TransactNamedPipe(HANDLE hNamedPipe, LPVOID lpInBuffer,
    DWORD nInBufferSize, LPVOID lpOutBuffer,
    DWORD nOutBufferSize, LPDWORD lpBytesRead,
    LPOVERLAPPED lpOverlapped);
```

Первый параметр, *hNamedPipe*, – дескриптор именованного канала, который возвращает функция CreateNamedPipe или CreateFile.

Второй параметр, *lpInBuffer*, указывает на буфер данных, подлежащих записи в канал. Параметр *nInBufferSize* указывает на размер этого буфера (в байтах).

Четвертый параметр, *lpOutBuffer*, указывает на буфер для хранения прочитанных данных. Параметр *nOutBufferSize* указывает на размер этого буфера (в байтах).

Шестой параметр, *lpBytesRead*, указывает адрес переменной типа DWORD, в которую будет сохранено число байт, считаных из канала. Для синхронных операций чтения и записи этот параметр не может быть установлен в NULL.

Последний параметр, *lpOverlapped*, указывает на структуру OVERLAPPED, которая используется для асинхронных операций ввода/вывода. Для синхронных операций ввода/вывода этот параметр должен быть установлен в NULL.

При успешном завершении функция TransactNamedPipe возвращает значение отличное от FALSE.

Следует отметить, что функцию TransactNamedPipe также можно использовать и для анонимных каналов.

В Win32 API есть еще функция CallNamedPipe, которая объединяет в себе выполнение операций WaitNamedPipe, CreateFile, WriteFile, ReadFile и CloseHandle (см. листинг 6.20). Функция CallNamedPipe имеет следующий прототип:

```
BOOL CallNamedPipe(LPCTSTR lpNamedPipeName,
    LPVOID lpInBuffer, DWORD nInBufferSize,
    LPVOID lpOutBuffer, DWORD nOutBufferSize,
    LPDWORD lpBytesRead, DWORD nTimeOut);
```

Первый параметр, *lpNamedPipeName*, указывает на строку, содержащую имя созданного канала.

Следующие пять параметров (*lpInBuffer*, *nInBufferSize*, *lpOutBuffer*, *nOutBufferSize* и *lpBytesRead*) функции CallNamedPipe аналогичны одноименным параметрам функции TransactNamedPipe.

Последний параметр, *nTimeOut*, определяет интервал времени (в миллисекундах) ожидания готовности процесса-сервера к образованию соединения. Также этот параметр может принимать одно из следующих значение:

- NMPWAIT_NOWAIT – ожидание не выполняется. При этом если канал не доступен, функции CallNamedPipe завершается ошибкой;
- NMPWAIT_USE_DEFAULT_WAIT – используется интервал ожидания по умолчанию, заданный в вызове функции CreateNamedPipe при создании именованного канала;
- NMPWAIT_WAIT_FOREVER – ожидание выполняется бесконечно долго.

Если функция CreateNamedPipe выполняется успешно, она возвращает значение отличное от FALSE.

Преимуществом использования функции TransactNamedPipe или функции CallNamedPipe является лучшее использование канала за счет снижения накладных расходов системных ресурсов на один запрос.

Службы Windows

Службы Windows (Windows Services) (далее просто службы) представляют собой приложения, работающие в фоновом режиме без пользовательского интерфейса. Службы могут быть остановлены и запущены повторно, а также способны автоматически запускаться при запуске операционной системы. Эти особенности делают службы идеальным вариантом при разработке серверных приложений.

В Windows имеется целый ряд различных служб, обеспечивающих ключевые возможности операционной системы и не только. Информация обо всех службах содержится в разделе системного реестра HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.

Диспетчер управления службами

Все службы Windows выполняются под контролем *диспетчера управления службами* (Service Control Manager, SCM), который запускается при загрузке операционной системы. Основными задачами SCM являются:

- управление службами;
- управление базой данных, содержащей информацию обо всех службах;
- запуск служб, которые должны быть запущены автоматически при запуске операционной системы.

Кроме того SCM с помощью функций Win32 API предоставляет возможность управления различными службами и получения информации о состоянии этих служб.

Открытие и закрытие диспетчера управления службами

Первым делом необходимо установить соединение с диспетчером управления службами, для чего служит функция OpenSCManager:

```
SC_HANDLE OpenSCManager(PCTSTR lpMachineName,
   PCTSTR lpDatabaseName, DWORD dwDesiredAccess);
```

Первый параметр, *lpMachineName*, указывает на строку, содержащую имя компьютера. Для локального компьютера этот параметр может быть установлен в NULL.

Второй параметр, *lpDatabaseName*, указывает на строку, содержащую имя базы данных, содержащей информацию о службах. Обычно этот параметр установлен в NULL.

Третий параметр, *dwDesiredAccess*, определяет запрашиваемые права доступа к SCM. Некоторые из возможных значений этого параметра перечислены в табл. 6.6. Полный перечень значений параметра *dwDesiredAccess* можно найти в документации Platform SDK.

Таблица 6.6. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
SC_MANAGER_ALL_ACCESS	Все возможные права доступа
SC_MANAGER_CREATE_SERVICE	Разрешает создание новой службы с помощью функции CreateService
SC_MANAGER_ENUMERATE_SERVICE	Разрешает перечисление служб с помощью функции EnumServicesStatusEx
SC_MANAGER_LOCK	Разрешает блокировку базы данных служб с помощью функции LockServiceDatabase
SC_MANAGER_QUERY_LOCK_STATUS	Необходимо для определения состояния блокировки базы данных служб с помощью функции QueryServiceLockStatus

В случае успешного выполнения функция OpenSCManager возвращает дескриптор объекта для работы с SCM, а в случае ошибки NULL.

Когда дескриптор SCM более не нужен, его следует закрыть с помощью функции CloseServiceHandle:

```
BOOL CloseServiceHandle(SC_HANDLE hSCObject);
```

Параметр *hSCObject* – дескриптор объекта, который необходимо закрыть. Если функция CloseServiceHandle завершается успешно, возвращаемое значение отлично от FALSE.

Создание, открытие и закрытие службы

В Win32 API имеется функция CreateService, которая создает объект службы и добавляет его в базу данных диспетчера управления службами. Функция CreateService имеет следующий прототип:

```
SC_HANDLE CreateService(SC_HANDLE hSCManager,
    LPCTSTR lpServiceName, LPCTSTR lpDisplayName,
    DWORD dwDesiredAccess, DWORD dwServiceType,
    DWORD dwStartType, DWORD dwErrorControl,
    LPCTSTR lpBinaryPathName, LPCTSTR lpLoadOrderGroup,
    LPDWORD lpdwTagId, LPCTSTR lpDependencies,
    LPCTSTR lpServiceStartName, LPCTSTR lpPassword);
```

Первый параметр, *hSCManager*, – дескриптор объекта SCM, полученный с помощью функции OpenSCManager.

Следующие два параметра, *lpServiceName* и *lpDisplayName*, указывают на строки, содержащие имена службы. Максимальная длина строки – 256 символов. Параметр *lpServiceName* указывает на строку, содержащую имя службы, которое указывается при запуске службы, а параметр *lpDisplayName* – дружественное имя службы, которое отображается в списке служб системной утилиты «Службы» (Services).

Четвертый параметр, *dwDesiredAccess*, определяет запрашиваемые права доступа к службе. Некоторые из возможных значений этого параметра перечислены в табл. 6.7. Полный перечень значений параметра *dwDesiredAccess* можно найти в документации Platform SDK.

Таблица 6.7. Права доступа (значения параметра *dwDesiredAccess*)

Значение	**Описание**
SERVICE_ALL_ACCESS	Все возможные права доступа
SERVICE_PAUSE_CONTINUE	Разрешает приостановить службу или продолжить ее работу
SERVICE_QUERY_CONFIG	Необходимо для получения конфигурации службы
SERVICE_QUERY_STATUS	Необходимо для получения текущего статуса службы
SERVICE_START	Разрешает запустить службу
SERVICE_STOP	Разрешает остановить службу

Пятый параметр, *dwServiceType*, задает тип новой службы. Возможные значения этого параметра перечислены в табл. 6.8.

Таблица 6.8. Типы служб

Значение	**Описание**
SERVICE_FILE_SYSTEM_DRIVER	Служба представляет собой драйвер файловой системы
SERVICE_INTERACTIVE_PROCESS	Интерактивная служба, которая может взаимодействовать с рабочим столом. Это значение может использоваться совместно со значением SERVICE_WIN32_SHARE_PROCESS или SERVICE_WIN32_OWN_PROCESS
SERVICE_KERNEL_DRIVER	Служба представляет собой драйвер операционной системы
SERVICE_WIN32_OWN_PROCESS	Служба, которая работает в собственном процессе
SERVICE_WIN32_SHARE_PROCESS	Служба, которая разделяет процесс с несколькими другими службами

Шестой параметр, *dwStartType*, определяет вариант запуска новой службы. Этот параметр может принимать одно из значений, перечисленных в табл. 6.9.

Таблица 6.9. Варианты запуска службы

Значение	**Описание**
SERVICE_AUTO_START	Служба запускается автоматически диспетчером управления службами при запуске операционной системы
SERVICE_BOOT_START	Служба запускается системным загрузчиком. Это значение используется только для драйверов устройств
SERVICE_DEMAND_START	Служба запускается «вручную» диспетчером управления службами
SERVICE_DISABLED	Служба не может быть запущена. Попытка запустить такую службу завершится ошибкой
SERVICE_SYSTEM_START	Служба запускается диспетчером управления службами. Это значение используется только для драйверов устройств

Седьмой параметр, *dwErrorControl*, определяет предпринимаемые меры, если новая служба будет не в состоянии запуститься. Этот параметр может принимать значения из табл. 6.10.

Таблица 6.10. Принимаемые меры при ошибке при запуске службы

Значение	**Описание**
SERVICE_ERROR_CRITICAL	Диспетчер управления службами регистрирует ошибку в журнале событий (если это возможно) и пытается запустить последний известный «рабочий» экземпляр этой службы. Если последний «рабочий» экземпляр службы не найден или в настоящее время он уже запущен, запуск не выполняется
SERVICE_ERROR_IGNORE	Диспетчер управления службами игнорирует ошибку и продолжает операцию запуска службы
SERVICE_ERROR_NORMAL	Диспетчер управления службами регистрирует ошибку в журнале событий (если это возможно), но продолжает операцию запуска службы
SERVICE_ERROR_SEVERE	Диспетчер управления службами регистрирует ошибку в журнале событий (если это возможно) и пытается запустить последний известный «рабочий» экземпляр этой службы (даже если он уже запущен)

Восьмой параметр, *lpBinaryPathName*, позволяет указать командную строку, используемую для запуска службы. Предполагается, что первый аргумент командной строки представляет собой путь и имя исполняемого файла. Эта строка может также включать в себя и аргументы, которые передаются службе при запуске.

Девятый параметр, *lpLoadOrderGroup*, указывает на строку, содержащую имя группы очередности загрузки, членом которой будет являться новая служба. Диспетчер управления службами использует группы очередности загрузки, чтобы загрузить службы в определенном порядке. Список групп очередности загрузки хранится в системном реестре в разделе HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder. Если новая служба не должна принадлежать никакой группе, этот параметр устанавливается в NULL.

Десятый параметр, *lpdwTagId*, указывает на переменную типа DWORD, в которую записывается значение признака, являющееся уникальным в группе, заданной в параметре *lpLoadOrderGroup*. Можно использовать этот параметр для того, чтобы изменить порядок запуска внутри группы (см. документацию Platform SDK). Если параметр *lpdwTagId* не используется, он должен быть установлен в NULL.

Одиннадцатый параметр, *lpDependencies*, указывает на строку, содержащую список имен служб, от которых зависит новая служба и которые диспетчер управления службами должен запустить перед запуском новой службы. Элементы этого списка должны разделятся нуль-символами, а сам список должен завершаться двумя нуль-символами. Если новая служба не имеет никаких зависимостей, параметр *lpDependencies* должен быть установлен в NULL.

Последние два параметра, *lpServiceStartName* и *lpPassword*, используются для указания имени учетной записи (под которой должна запускаться служба) и пароля к этой учетной записи. Для служб типа SERVICE_WIN32_OWN_PROCESS имя учетной записи задается в полном формате (например, Андрей-ПК\Андрей). Для служб типа SERVICE_FILE_SYSTEM_DRIVER и SERVICE_KERNEL_DRIVER параметры *lpServiceStartName* и *lpPassword* игнорируются.

Как показано в следующем примере, если функция CreateService завершается успешно, она возвращает дескриптор службы, а в случае ошибки NULL.

Листинг 6.21. Пример создания службы Windows

```
1   // создание службы SampleWinService
2   SC_HANDLE hService = CreateService(hSCManager,
3       TEXT("SampleWinService"), TEXT("Пример службы Windows"),
```

```
4       SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
    SERVICE_DEMAND_START, SERVICE_ERROR_IGNORE,
5       TEXT("C:\\SampleWinService.exe /runservice"),
6       NULL, NULL, NULL, NULL, NULL);
7
8   if (NULL != hService)
9   {
10      /* служба успешно создана */
11  } // if
```

Открыть уже созданную службу можно с помощью вызова функции `OpenService`:

```
SC_HANDLE OpenService(SC_HANDLE hSCManager,
    LPCTSTR lpServiceName, DWORD dwDesiredAccess);
```

Первый параметр, *hSCManager*, – дескриптор объекта SCM, полученный с помощью функции `OpenSCManager`.

Второй параметр, *lpServiceName*, указывает на строку, содержащую имя службы, которое указывается при запуске службы

Третий параметр, *dwDesiredAccess*, определяет запрашиваемые права доступа к службе (см. табл. 6.7).

При успешном завершении функция `OpenService` возвращает дескриптор открытой службы, а в случае ошибки `NULL`.

Наконец, когда дескриптор службы более не нужен, его следует закрыть с помощью функции `CloseServiceHandle`.

Управление службами

Созданная служба сразу не выполняется. Для этого ее следует запустить, вызовом функции `StartService`:

```
BOOL StartService(SC_HANDLE hService, DWORD dwNumServiceArgs,
    LPCTSTR *lpServiceArgVectors);
```

Первый параметр, *hService*, – дескриптор запускаемой службы. Этот дескриптор должен обладать правом `SERVICE_START`.

Следующие два параметра, *dwNumServiceArgs* и *lpServiceArgVectors*, позволяют передать аргументы в функцию, которая является точкой входа в службу (подробно эта функция рассматривается в разделе «Приложения, реализующие службы Windows»). Параметр *dwNumServiceArgs* задает количество аргументов, а параметр *lpServiceArgVectors* указывает на массив строк, которые представляют собой передаваемые в службу аргументы.

Если аргументы в службу не передаются, значение параметра *dwNumServiceArgs* должно равняться нулю, а параметр *lpServiceArgVectors* должен быть установлен в `NULL`. В противном случае, первый

элемент массива (на который указывает параметр *lpServiceArgVectors*) должен содержать имя запускаемой службы, а остальные элементы – передаваемые аргументы.

Если функция StartService завершается успешно, она возвращает значение отличное от FALSE.

Управление службой осуществляется посредством вызова функции ControlService, которая передает службе управляющий код.

```
BOOL ControlService(SC_HANDLE hService, DWORD dwControl,
    LPSERVICE_STATUS lpServiceStatus);
```

Первый параметр, *hService*, – дескриптор службы, которой необходимо передать управляющий код.

Второй параметр, *dwControl*, указывает код уведомления, которое передается службе. Основные значения этого параметра перечислены в табл. 6.11. Кроме того, параметр *dwControl* может использоваться и для передачи специфичных для определенных служб уведомлений. Коды таких уведомлений должны лежать в диапазоне от 128 до 255.

Таблица 6.11. Основные значения параметра *dwControl*

Значение	**Описание**
SERVICE_CONTROL_CONTINUE	Уведомляет временно приостановленную службу о том, что она должна возобновить свою работу. Дескриптор *hService* должен иметь право доступа SERVICE_PAUSE_CONTINUE
SERVICE_CONTROL_PAUSE	Уведомляет службу о том, что она должна приостановить свое выполнение. Дескриптор *hService* должен иметь право доступа SERVICE_PAUSE_CONTINUE
SERVICE_CONTROL_STOP	Уведомляет службу о том, что она должна завершить свою работу. Дескриптор *hService* должен иметь право доступа SERVICE_STOP

Последний параметр, *lpServiceStatus*, указывает на структуру SERVICE_STATUS, в которую сохраняется новое состояние службы.

При успешном завершении функции ControlService, возвращаемое значение отлично от FALSE.

Для определения текущего состояния службы используется функция QueryServiceStatus:

```
BOOL QueryServiceStatus(SC_HANDLE hService,
    LPSERVICE_STATUS lpServiceStatus);
```

Первый параметр, *hService*, – дескриптор службы, состояние которой нужно определить. Этот дескриптор должен обладать правом доступа SERVICE_QUERY_STATUS.

Второй параметр, *lpServiceStatus*, указывает на структуру SERVICE_STATUS, в которую сохраняется текущее состояние указанной службы.

При успешном завершении функции QueryServiceStatus, возвращаемое значение отлично от FALSE.

Структура SERVICE_STATUS имеет следующее описание:

```
typedef struct _SERVICE_STATUS {
  DWORD dwServiceType; // тип службы
  DWORD dwCurrentState; // состояние службы
  DWORD dwControlsAccepted; // управляющие команды
  DWORD dwWin32ExitCode; // код ошибки
  DWORD dwServiceSpecificExitCode; // код ошибки
  DWORD dwCheckPoint; // контрольная точка
  DWORD dwWaitHint; // время ожидания
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Первое поле, *dwServiceType*, содержит тип службы (см. табл. 6.8).

Второе поле, *dwCurrentState*, содержит текущее состояние службы. В табл. 6.12 перечислены возможные значения этого поля. Диаграмма возможных состояний службы представлена на рис. 6.1.

Таблица 6.12. Состояния службы

Значение	**Описание**
SERVICE_CONTINUE_PENDING	Ожидается продолжение работы службы
SERVICE_PAUSE_PENDING	Ожидается приостановка службы
SERVICE_PAUSED	Служба приостановлена
SERVICE_RUNNING	Служба выполняется
SERVICE_START_PENDING	Служба запускается
SERVICE_STOP_PENDING	Служба останавливается
SERVICE_STOPPED	Служба остановлена

Третье поле, *dwControlsAccepted*, определяет какие уведомления, служба может принимать и обрабатывать. Основные значения этого поля перечислены в табл. 6.13. Дополнительные значения поля *dwControlsAccepted* можно найти в документации Platform SDK.

Таблица 6.13. Основные значения поля *dwControlsAccepted*

Значение	Описание
SERVICE_ACCEPT_STOP	Позволяет службе получать уведомление SERVICE_CONTROL_STOP
SERVICE_ACCEPT_SHUTDOWN	Позволяет службе получать уведомление SERVICE_CONTROL_SHUTDOWN (о прекращении работы операционной системы). Важно отметить, что функция ControlService не может отправлять это уведомление
SERVICE_ACCEPT_PAUSE_CONTINUE	Позволяет службе получать уведомления SERVICE_CONTROL_PAUSE и SERVICE_CONTROL_CONTINUE

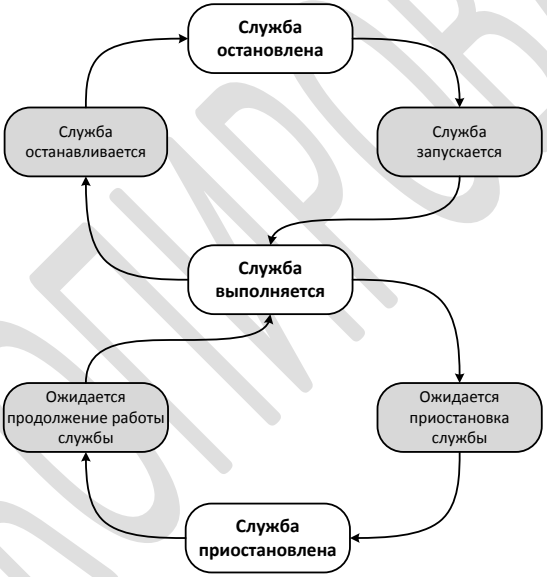

Рис. 6.1. Диаграмма состояний службы Windows

Поля *dwWin32ExitCode* и *dwServiceSpecificExitCode* используются для того, чтобы сообщить об ошибке, которая происходит, когда служба запускается или останавливается. Поле *dwWin32ExitCode* содержит код возникшей ошибки (см. в документации Platform SDK) или одно из следующих значений:

- NO_ERROR – служба запустилась/завершилась без ошибок;
- ERROR_SERVICE_SPECIFIC_ERROR – код ошибки содержится в поле *dwServiceSpecificExitCode*.

Следующее поле, *dwCheckPoint*, содержит значение контрольной точки службы, которое сообщает о ходе длительного запуска, остановки, паузы или продолжения операции. Это поле принимает значение равное нулю, если служба выполняется, приостановлена или остановлена.

Последнее поле, *dwWaitHint*, содержит предполагаемое время (в миллисекундах), требуемое для ожидания запуска, остановки, паузы или операции ожидания продолжения.

Удаление службы

Для удаления службы используется функция `DeleteService`, которая имеет следующий прототип:

```
BOOL DeleteService(SC_HANDLE hService);
```

Параметр *hService* – дескриптор удаляемой службы. Этот дескриптор должен обладать правом доступа `DELETE`.

Если функция `DeleteService` завершается успешно, возвращаемое значение отлично от `FALSE`. При этом если служба в данный момент запущена, она будет удалена сразу после того как будет остановлена.

Управление службами встроенными средствами Windows

Самым простым способом управления службами Windows является использование специальной программы services.msc, которая находится в каталоге %WINDIR%\system32. Важно отметить, что для использования программы services.msc пользователь должен быть членом группы «Администраторы» или иметь соответствующие привилегии и права учетной записи.

При запуске программы services.msc открывается окно «Службы» (Services). Как показано на рис. 6.2 в этом окне отображаются все созданные службы.

Управление службами

В главном окне программы services.msc выберите службу из списка, щелкните ее правой кнопкой мыши и выберите **Свойства** (**Properties**). В появившемся диалоговом окне (рис. 6.3) на вкладке **Общие** (**General**) содержится имя службы, описание службы, имя выполняемого файла службы, кнопки управления службой. Здесь также можно изменить тип запуска службы или вовсе отключить эту службу.

В поле **Состояние** (**Service status**) выводится текущее состояние службы. Чтобы изменить текущее состояние службы следует нажать соответствующую кнопку управления службой. Следует отметить, что

при запуске службы в поле **Параметры запуска** (**Start parameters**) можно указать (через пробел) передаваемые в службу аргументы.

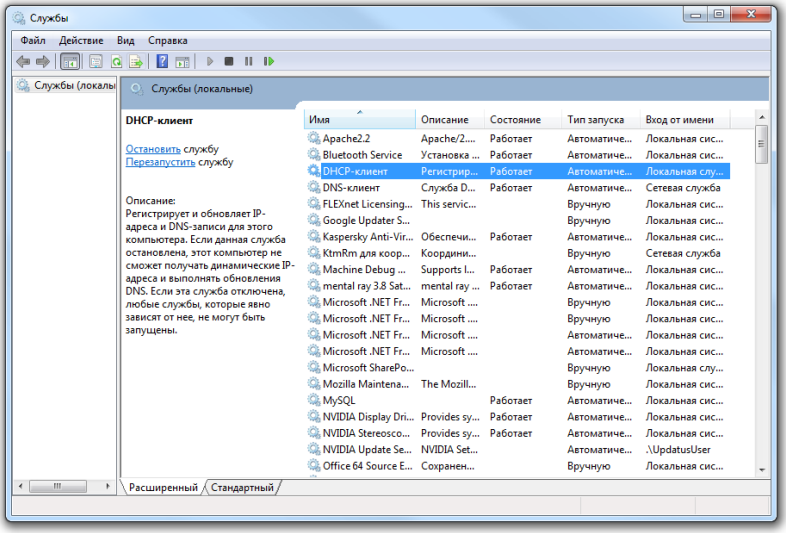

Рис. 6.2. Службы Windows

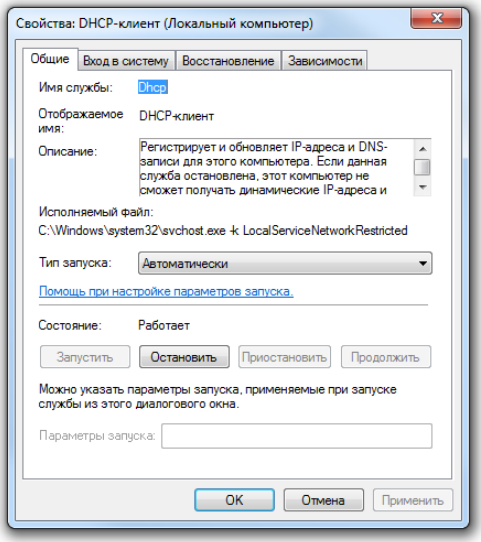

Рис. 6.3. Окно свойств службы Windows

На остальных вкладках (**Вход в систему** (**Log On**), **Восстановление** (**Recovery**) и **Зависимости** (**Dependencies**)) можно изменить имя учетной записи (под которой должна запускаться служба) и пароль к этой учетной записи, определять действия, выполняемые при сбое в работе службы, и получить информацию о зависимостях службы.

Управление службами с помощью командной строки

Службами также можно управлять с помощью следующих системных команд:

- `sc create` – создает службу;
- `sc delete` – удаляет службу;
- `sc pause` – приостанавливает работу службы;
- `sc continue` – возобновляет работу службы;
- `sc start` – запускает службу;
- `sc stop` – останавливает службу.

На рис. 6.4 продемонстрирован пример использования команды `sc delete` для удаления службы SampleWinService. При этом важным является тот факт, что команда `sc delete` выполняется от имени «Администратора».

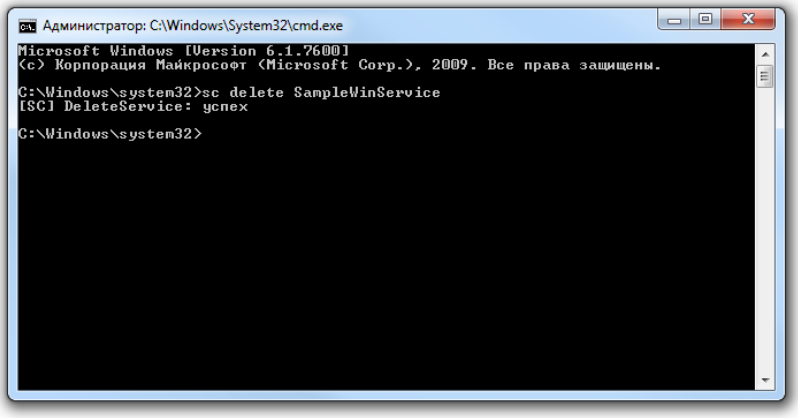

Рис. 6.4. Удаление службы Windows

Подробное описание перечисленных команд можно найти по адресу: http://go.microsoft.com/fwlink/?linkid=53528.

Приложения, реализующие службы Windows

Все приложения, реализующие службы Windows (независимо от ее назначения), обладают определенным сходством. В частности, они должны предусматривать возможности запуска и остановки службы. В этом разделе показано, как реализовать такие возможности.

Точка входа

Когда SCM запускает службу, он создает процесс (в котором будет выполняться запускаемая служба) и ждет пока, главный поток этого процесса вызовет функцию `StartServiceCtrlDispatcher`. Важно отметить, что вызов указанной функции должен осуществляться именно из главного потока и в течение первых 30 секунд. Иначе запуск службы завершится ошибкой.

Функция `StartServiceCtrlDispatcher` связывает SCM с главным потоком процесса запускаемой службы. После этого главный поток действует как диспетчер управления этой службой, выполняя следующие действия:

- создает новый поток, чтобы вызвать в нем функцию, которая является точкой входа в соответствующую службу;
- периодически вызывает функцию, которая обрабатывает уведомления, поступающие от SCM, для соответствующей службы.

Функция `StartServiceCtrlDispatcher` имеет следующий прототип:

```
BOOL StartServiceCtrlDispatcher(
    const SERVICE_TABLE_ENTRY *lpServiceTable);
```

Параметр *lpServiceTable* представляет собой массив структур `SERVICE_TABLE_ENTRY`, каждая из которых содержит имя запускаемой службы и точку входа в эту службу. Значения полей последней структуры в этом массиве должны быть установлены в `NULL`.

Если функция `StartServiceCtrlDispatcher` выполняется успешно, она не вернет управление до тех пор, пока все запущенные с ее помощью службы не завершат свою работу. При этом возвращаемое значение будет отлично от `FALSE`.

Структура `SERVICE_TABLE_ENTRY` имеет следующее описание:

```
typedef struct _SERVICE_TABLE_ENTRY {
  LPTSTR lpServiceName; // имя службы
  LPSERVICE_MAIN_FUNCTION lpServiceProc; // точка входа
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

Первое поле, *lpServiceName*, указывает на строку, содержащую имя запускаемой службы.

Второе поле, *lpServiceProc*, указывает на функцию, которая является точкой входа в службу. Имя этой функции может быть любым, но она обязательно должна иметь следующую сигнатуру:

```
void WINAPI ServiceMain(DWORD dwArgc, LPTSTR *lpszArgv);
```

Здесь параметр *dwArgc* – число аргументов командной строки, передаваемых при запуске службы, а параметр *lpszArgv* указывает на массив аргументов командной строки.

В следующем примере продемонстрирован фрагмент программного кода, в котором происходит запуск службы.

```
1   SERVICE_TABLE_ENTRY svcDispatchTable[] = {
2       { TEXT("SampleWinService"), ServiceMain },
3       { NULL, NULL }
4   };
5
6   BOOL bRet = StartServiceCtrlDispatcher(svcDispatchTable);
7
8   if (FALSE == bRet)
9   {
10      /* возникла ошибка при запуске службы */
11  } // if
```

Обработка уведомлений

Для того чтобы служба, могла обрабатывать уведомления, поступающие от SCM, необходимо для этой службы зарегистрировать функцию-обработчика поступающих уведомлений. Это можно сделать с помощью функции `RegisterServiceCtrlHandlerEx`.

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandlerEx(
    LPCTSTR lpServiceName, LPHANDLER_FUNCTION lpHandlerProc,
    LPVOID lpContext);
```

Первый параметр, *lpServiceName*, указывает на строку, содержащую имя службы.

Второй параметр, *lpHandlerProc*, указывает на функцию, которая является обработчиком поступающих уведомлений. Эта функция может иметь любое имя, но обязана иметь следующую сигнатуру:

```
DWORD WINAPI HandlerEx(DWORD fdwControl, DWORD dwEventType,
    LPVOID lpEventData, LPVOID lpContext);
```

Здесь параметр *fdwControl* представляет собой код уведомления (см. табл. 6.11), которое необходимо обработать. Параметр *dwEventType* обычно принимает значение равное нулю, а не нулевые значения

указывают на тип данных, передаваемых в параметре *lpEventData* (подробнее см. в документации Platform SDK).

Наконец параметр *lpContext* – данные, передаваемые в функцию RegisterServiceCtrlHandlerEx через одноименный параметр.

Если функция RegisterServiceCtrlHandlerEx завершается успешно, она возвращает дескриптор состояния службы, иначе – NULL. Полученный дескриптор состояния службы не должен быть закрыт.

Функция RegisterServiceCtrlHandlerEx должна вызываться в теле функции, которая является точкой входа в службу.

Нужно также отметить, что SCM отправляет уведомления главному потоку через именованный канал, который создается при регистрации функции-обработчика.

Изменение состояния службы

Когда функция-обработчик уведомлений будет зарегистрирована, необходимо перевести службу в состояние SERVICE_START_PENDING. Для этого следует использовать функцию SetServiceStatus:

```
BOOL SetServiceStatus(SERVICE_STATUS_HANDLE hServiceStatus,
    LPSERVICE_STATUS lpServiceStatus);
```

Первый параметр, *hServiceStatus*, – дескриптор состояния службы, который вернула функция RegisterServiceCtrlHandlerEx.

Второй параметр, *lpServiceStatus*, указывает на структуру SERVICE_STATUS, содержащую информацию о состоянии службы.

При успешном завершении функции SetServiceStatus, возвращаемое значение отличное от FALSE.

Когда служба будет готова, к работе она должна вновь изменить свое состояние, на этот раз на состояние SERVICE_RUNNING.

Кроме того функция-обработчик при обработке управляющих уведомлений должна устанавливать состояние службы при каждом ее вызове, даже если ее состояние не менялось.

Создание приложения, реализующего службу Windows

Среда разработки Visual C++ позволяет создавать приложения, которые будут работать как службы. В большинстве случаев это консольные приложения Win32. Для создания службы необходимо выполнить следующие действия:

1. Создайте пустой проект консольного приложения Win32 (например, SampleWinService).
2. Добавьте в проект файл исходного кода WinService.cpp.

3. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 6.22 и сохраните этот файл.

Листинг 6.22. Файл исходного кода WinService.cpp

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>
#include <locale.h>
#include <strsafe.h>

extern LPCTSTR gSvcName; // имя службы
extern LPCTSTR gSvcDisplayName; // отображаемое имя службы

SERVICE_STATUS gSvcStatus; // текущее состояние службы
SERVICE_STATUS_HANDLE gSvcStatusHandle; // дескриптор состояния службы

// эта функция вызывается при запуске службы
BOOL OnSvcInit(DWORD dwArgc, LPTSTR *lpszArgv);
// эта функция вызывается для остановки службы
void OnSvcStop();
// в этой функции реализован основной функционал
DWORD SvcMain(DWORD dwArgc, LPTSTR *lpszArgv);

// -----------------------------------------------------------
DWORD WINAPI SvcHandler(DWORD fdwControl, DWORD dwEventType, LPVOID lpEventData, LPVOID lpContext)
{
    if (SERVICE_CONTROL_STOP == fdwControl || SERVICE_CONTROL_SHUTDOWN == fdwControl)
    {
        OnSvcStop(); // останавливаем службу

        gSvcStatus.dwCurrentState = SERVICE_STOP_PENDING; // новое состояние службы
    } // if

    // изменяем текущее состояние службы
    SetServiceStatus(gSvcStatusHandle, &gSvcStatus);
    return NO_ERROR;
} // ServiceControlHandler

// -----------------------------------------------------------
void WINAPI ServiceMain(DWORD dwArgc, LPTSTR *lpszArgv)
{
    // регистрируем функцию-обработчик
```

```
39      gSvcStatusHandle = RegisterServiceCtrlHandlerEx(gSvcName,
    SvcHandler, NULL);
40
41      if (NULL != gSvcStatusHandle)
42      {
43          // начальное состояние
44          gSvcStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
45          gSvcStatus.dwCurrentState = SERVICE_START_PENDING;
46          gSvcStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP |
    SERVICE_ACCEPT_SHUTDOWN;
47          gSvcStatus.dwWin32ExitCode = NO_ERROR;
48          gSvcStatus.dwServiceSpecificExitCode = 0;
49          gSvcStatus.dwCheckPoint = 0;
50          gSvcStatus.dwWaitHint = 0;
51
52          // задаем начальное состояние службы
53          SetServiceStatus(gSvcStatusHandle, &gSvcStatus);
54
55          // //// //
56
57          if (OnSvcInit(dwArgc, lpszArgv) != FALSE)
58          {
59              gSvcStatus.dwCurrentState = SERVICE_RUNNING; // новое
    состояние
60              // изменяем текущее состояние службы
61              SetServiceStatus(gSvcStatusHandle, &gSvcStatus);
62
63
64              DWORD dwExitCode = SvcMain(dwArgc, lpszArgv);
65
66
67              if (dwExitCode != 0) // возвращаем код ошибки
68              {
69                  gSvcStatus.dwWin32ExitCode =
    ERROR_SERVICE_SPECIFIC_ERROR;
70                  gSvcStatus.dwServiceSpecificExitCode =
    dwExitCode;
71              } // if
72              else
73              {
74                  gSvcStatus.dwWin32ExitCode = NO_ERROR;
75              } // else
76          } // if
77
78          // //// //
79
```

```
80          gSvcStatus.dwCurrentState = SERVICE_STOPPED; // новое
    состояние
81          // задаем конечное состояние службы
82          SetServiceStatus(gSvcStatusHandle, &gSvcStatus);
83      } // if
84  } // ServiceMain
85
86  // ------------------------------------------------------------
87  int _tmain(int argc, LPTSTR argv[])
88  {
89      _tsetlocale(LC_ALL, TEXT(""));
90
91      if (argc < 2)
92      {
93          _tprintf(TEXT("> Не указан параметр.\n"));
94          return 0; // завершаем работу приложения
95      } // if
96
97      // //// //
98
99      if (_tcscmp(argv[1], TEXT("/runservice")) == 0) // начало
    работы службы
100     {
101         SERVICE_TABLE_ENTRY svcDispatchTable[] = {
102             { (LPTSTR)gSvcName, ServiceMain },
103             { NULL, NULL }
104         };
105
106         StartServiceCtrlDispatcher(svcDispatchTable);
107         return 0; // завершаем работу приложения
108     } // if
109
110     // //// //
111
112     if (_tcscmp(argv[1], TEXT("/create")) == 0) // создание
    службы
113     {
114         // открываем SCM
115         SC_HANDLE hSCM = OpenSCManager(NULL, NULL,
    SC_MANAGER_CREATE_SERVICE);
116
117         if (NULL == hSCM)
118         {
119             _tprintf(TEXT("> Не удалось открыть диспетчер
    управления службами Windows.\n"));
120             return -1; // завершаем работу приложения
```

```
121         } // if
122
123
124         TCHAR szCmdLine[MAX_PATH+13]; // командная строка
125
126         // определяем путь и имя исполняемого файла
127         GetModuleFileName(NULL, szCmdLine, _countof(szCmdLine));
128         // добавляем аргумент командной строки
129         StringCchCat(szCmdLine, _countof(szCmdLine),
    TEXT(" /runservice"));
130
131         // создаем службу
132         SC_HANDLE hSvc = CreateService(hSCM, gSvcName,
    gSvcDisplayName, 0,
133             SERVICE_WIN32_OWN_PROCESS, // служба, которая
    работает в собственном процессе
134             SERVICE_DEMAND_START, // служба запускается "вручную"
135             SERVICE_ERROR_NORMAL, szCmdLine, NULL, NULL, NULL,
    NULL, NULL);
136
137         if (NULL != hSvc)
138         {
139             _tprintf(TEXT("> Служба успешно создана!\n"));
140             CloseServiceHandle(hSvc); // закрываем дескриптор
141         } // if
142         else
143         {
144             _tprintf(TEXT("> Не удалось создать службу.\n"));
145         } // else
146
147         CloseServiceHandle(hSCM); // закрываем дескриптор
148         return 0; // завершаем работу приложения
149     } // if
150
151     // //// //
152
153     if (_tcscmp(argv[1], TEXT("/delete")) == 0) // удаление
    службы
154     {
155         // открываем SCM
156         SC_HANDLE hSCM = OpenSCManager(NULL, NULL,
    SC_MANAGER_CREATE_SERVICE);
157
158         if (NULL == hSCM)
159         {
```

```
160             _tprintf(TEXT("> Не удалось открыть диспетчер
    управления службами Windows.\n"));
161             return -1; // завершаем работу приложения
162         } // if
163 
164         // открываем службу для удаления
165         SC_HANDLE hSvc = OpenService(hSCM, gSvcName, DELETE);
166 
167         if (NULL != hSvc)
168         {
169             BOOL bRet = DeleteService(hSvc); // удаляем службу
170 
171             if (FALSE != bRet) _tprintf(TEXT("> Служба успешно
    удалена!\n"));
172             else _tprintf(TEXT("> Не удалось удалить
    службу.\n"));
173 
174             CloseServiceHandle(hSvc); // закрываем дескриптор
175         } // if
176         else
177         {
178             _tprintf(TEXT("> Не удалось удалить службу.\n"));
179         } // else
180 
181         CloseServiceHandle(hSCM); // закрываем дескриптор
182         return 0; // завершаем работу приложения
183     } // if
184 
185     // //// //
186 
187     _tprintf(TEXT("> Неизвестный параметр.\n"));
188 } // _tmain
```

4. Добавьте в проект файл исходного кода SampleWinService.cpp.
5. В файле, открывшемся в редакторе Visual Studio, введите программный код из примера в листинге 6.23 и сохраните этот файл.

Листинг 6.23. Файл исходного кода SampleWinService.cpp

```
1  #include <Windows.h>
2  #include <tchar.h>
3 
4  // задаем имя службы
5  LPCTSTR gSvcName = TEXT("SampleWinService");
6  // задаем отображаемое имя службы
7  LPCTSTR gSvcDisplayName = TEXT("Пример службы Windows");
```

```
8
9   // -------------------------------------------------------------
    BOOL OnSvcInit(DWORD dwArgc, LPTSTR *lpszArgv)
10  {
11      /* в этой функции можно выполнять
12         инициализацию различных глобальных переменных */
13
14      /* если эта функция вернет FALSE,
15         считается что произошла ошибка в инициализации */
16
17      return TRUE;
18  } // OnSvcInit
19
20  // -------------------------------------------------------------
21  void OnSvcStop()
22  {
23      /* в этой функции, нужно реализовать механизм
24         прекращения работы службы */
25  } // OnSvcStop
26
27  // -------------------------------------------------------------
28  DWORD SvcMain(DWORD dwArgc, LPTSTR *lpszArgv)
29  {
30      /* в этой функции должен
31         выполняться основной функционал службы */
32
33      /* когда эта функция вернет управление,
34         служба считается завершенной */
35
36      /* если возвращаемое значение не равно 0,
37         считается что служба завершилась с ошибкой */
38
39      return 0;
40  } // SvcMain
```

6. В меню **Построение** (**Build**) выберите команду **Построить решение** (**Build Solution**).

Нужно отметить, что приложение SampleWinService также осуществляет создание и удаление реализованной в нем службы. Для этого используются, соответственно, команды `/create` и `/delete`. В этом случае приложение должно запускаться от имени «Администратора».

Изоляция служб в Windows

Операционная система Windows обеспечивает определенную степень изоляции между службами и приложениями, запускаемыми в

различных сеансах. Не составляет труда организовать взаимодействие между процессами в одном сеансе, но обмен данными между сеансами ограничен (более подробно см. лабораторную работу № 5 раздел «Задание дескриптора безопасности при создании объектов»).

Первый сеанс, созданный при запуске операционной системы, именуется «Сеанс 0» (Session 0). В Windows XP и младше в сеансе 0 выполняются службы и приложения, запускаемые первым пользователем, вошедшим в операционную систему. Однако запускать пользовательские приложения в сеансе 0 рискованно, так как в этом случае злоумышленник может вмешаться в функционирование привилегированных служб Windows.

В Windows Vista (и более новых версиях) в сеансе 0 могут выполняться только службы, а приложения, запущенные первым пользователем, который вошел в операционную систему, будут находиться в другом сеансе. Поэтому приложения не могут воздействовать на службы. На рис. 6.5 показаны различия между архитектурой управления сеансами Windows XP и Windows Vista.

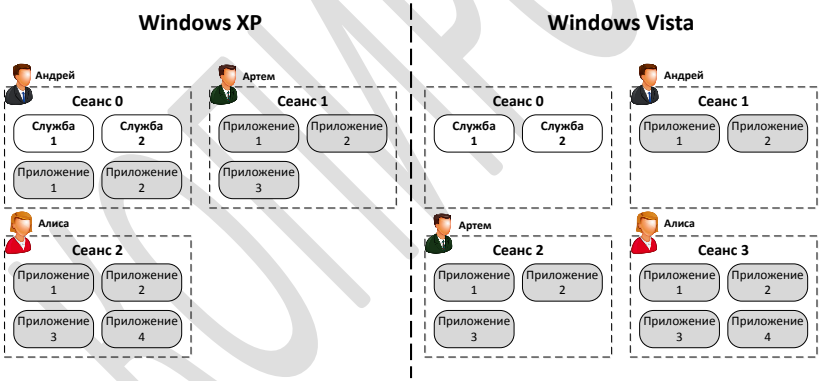

Рис. 6.5. Управление сеансами в Windows

Благодаря изоляции сеанса 0 в Windows Vista (и более новых версиях) службы (даже, если они являются интерактивными) не могут напрямую взаимодействовать с пользователями. Но это не означает, что службы совсем не могут взаимодействовать с пользователями. Для этого можно использовать различные механизмы межпроцессного взаимодействия, которые позволят установить связь между процессами в разных сеансах.

Задание к работе

1. Разработать в Visual C++ оконное приложение Win32, которое, будучи запущенно несколько раз, способно выполнять обмен данными между всеми своими экземплярами.

   Обмен данными должен выполняться с помощью механизма межпроцессного взаимодействия, указанного в варианте задания.

2. Разработать в Visual C++ два приложения:
   - серверное приложение (реализованное, как служба Windows), которое хранит список студентов и обрабатывает запросы на получение данных о них;
   - клиентское приложение (реализованное, как приложение Win-32), которое запрашивает данные о студентах у серверного приложения и выводит полученные данные.

   Обмен данными должен выполняться с помощью механизма межпроцессного взаимодействия, указанного в варианте задания.

3. Протестировать работу разработанных приложений на компьютере под управлением Windows XP (или выше). Результаты тестирования отразить в отчете.

4. Включить в отчет исходный программный код и выводы о проделанной работе.

Варианты заданий

№	Механизм межпроцессного взаимодействия (п. 1 задания)	Механизм межпроцессного взаимодействия (п. 2 задания)
1,16	Буфер обмена	Проецируемые в память файлы
2,17	Оконные сообщения	Почтовые ящики
3,18	Анонимные каналы	Именованные каналы
4,19	Буфер обмена	Почтовые ящики
5,20	Оконные сообщения	Именованные каналы
6,21	Анонимные каналы	Проецируемые в память файлы
7,22	Буфер обмена	Именованные каналы
8,23	Оконные сообщения	Проецируемые в память файлы
9,24	Анонимные каналы	Почтовые ящики
10,25	Буфер обмена	Проецируемые в память файлы

№	Механизм межпроцессного взаимодействия (п. 1 задания)	Механизм межпроцессного взаимодействия (п. 2 задания)
11,26	Оконные сообщения	Именованные каналы
12,27	Анонимные каналы	Почтовые ящики
13,28	Буфер обмена	Именованные каналы
14,29	Оконные сообщения	Проецируемые в память файлы
15,30	Анонимные каналы	Почтовые ящики

Контрольные вопросы

1. Что называют межпроцессным взаимодействием?
2. Какие механизмы межпроцессного взаимодействия поддерживаются в Windows?
3. Что такое буфер обмена Windows?
4. Какие функции Win32 API следует использовать для того, чтобы открыть или закрыть буфер обмена? Для чего необходимо открывать и закрывать буфер обмена?
5. Какие функции Win32 API следует использовать для передачи данных через буфер обмена?
6. Какие форматы текстовых данных поддерживаются в буфере обмена?
7. Как создать собственный формат данных в буфере обмена?
8. Что в Windows называют проецируемым в память файлом?
9. Как спроецировать файл на адресное пространство процесса? Какие функции Win32 API следует для этого использовать?
10. Как удалить спроецированный файл из адресного пространства процесса? Какие функции Win32 API следует для этого использовать?
11. Как использовать спроецированные в память файлы для передачи данных между процессами?
12. Какие оконные сообщения можно использовать для передачи данных между процессами? Какие функции Win32 API следует для этого использовать?
13. Каким образом осуществляется отправка оконных сообщений окнам, созданным в другом потоке или процессе?
14. Как осуществляется передача текстовых данных с помощью оконных сообщений `WM_SETTEXT` и `WM_GETTEXT`? Чем отличаются эти оконные сообщения?

15. Как осуществляется передача данных с помощью оконного сообщения `WM_COPYDATA`?
16. Что в Windows называют почтовым ящиком?
17. Какие функции Win32 API следует использовать для создания и открытия почтового ящика?
18. Какие существуют правила именования почтовых ящиков?
19. Какие функции Win32 API следует использовать для передачи данных через почтовый ящик?
20. Как организовать двунаправленный обмен данными между двумя процессами при помощи почтовых ящиков?
21. Что в Windows называют анонимным каналом?
22. Какую функцию Win32 API следует использовать для создания анонимного канала?
23. Какие функции Win32 API следует использовать для передачи данных через анонимный канал?
24. Что в Windows называют именованным каналом?
25. Какие функции Win32 API следует использовать для создания и открытия именованного канала?
26. Какие существуют правила именования каналов?
27. Какие функции Win32 API следует использовать для передачи данных через именованный канал?
28. Чем отличаются механизмы передачи данных через анонимный канал и через именованный канал?
29. Что такое службы Windows?
30. Для чего предназначен диспетчер управления службами?
31. Какие функции Win32 API следует использовать для создания, открытия и удаления служб Windows?
32. Какие функции Win32 API следует использовать для управления службами Windows?
33. Как в Visual C++ создать проект приложения, реализующего службу Windows?
34. Что такое точка входа в службу Windows?
35. Как осуществляется обработка уведомлений, поступающих в службу Windows?
36. Как и для чего применяется изоляция служб в Windows?

ЗАКЛЮЧЕНИЕ

В данном учебном пособии были даны определения ключевым понятиям и терминам, которые составляют необходимый базис для понимания устройства Windows. Мы получили представление о работе консольных и оконных приложений Windows. Мы также получили начальное представление о полезных инструментальных средствах, доступных для отладки таких приложений.

Была изучена работа процессов, потоков и заданий, рассмотрены способы их создания и основные механизмы синхронизация. Мы изучили основные механизмы работы с файлами и каталогами, а также общую структуру системного реестра Windows.

В учебном пособии также был дан краткий обзор внутренних компонентов, положенных в основу функций безопасности Windows, и механизмов межпроцессного взаимодействия. Мы ознакомились с общей архитектурой служб Windows

Итак, мы изучили базовые принципы разработки приложений для операционной системы Windows. Заложив основу, мы готовы к изучению более сложных механизмов, построенных на этом базисе.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Арчер, Т.* Visual C++ .NET. Библия пользователя / Т. Арчер, Э. Уайтчепел; пер. с англ. В. А. Коваленко. – М.: Издательский дом «Вильямс», 2005. – 1216 с.
2. *Глощапов, А. Л.* Microsoft Visual Studio 2010 / А.Л. Голощапов. – СПб.: БХВ-Петербург, 2011. – 544 с.
3. *Гордеев, А. В.* Операционные системы: Учебник для вузов / А. В. Гордеев. – 2-е изд. – СПб.: Питер, 2009. – 416 с.
4. *Давыдов, В. Г.* Visual C++. Разработка Windows-приложений с помощью MFC и API-функций / В. Г. Давыдов. – СПб.: БХВ-Петербург, 2008. – 576 с.
5. *Клименко Р. А.* Большая книга Windows Vista. Для профессионалов / Р.А. Клименко. – СПб.: Питер, 2009. – 688с.
6. *Королева О. И.* Реестр Windows 7 / О.И. Королева. – СПб.: БХВ-Петербург, 2010. – 704 с.
7. *Мартынов, Н. Н.* Программирование для Windows на C/C++. В 2 томах. Том 1 / Н.Н. Мартынов. – М.: Бином, 2012. – 528 с.
8. *Мартынов, Н. Н.* Программирование для Windows на C/C++. В 2 томах. Том 2 / Н.Н. Мартынов. – М.: Бином, 2013. – 480 с.
9. *Пирогов, В. Ю.* Программирование на Visual C++ .NET / В. Ю. Пирогов. – СПб.: БХВ-Петербург, 2003. – 800 с.
10. *Побегайло, А. П.* Системное программирование в Windows / А. П. Побегайло. – СПб.: БХВ-Петербург, 2006. – 1056 с.
11. Полный справочник по C++ / Г. Шилдт; пер. с англ. Д.А. Клюшина. – 4-е изд. – М.: Издательский дом «Вильямс», 2007. – 800 с.
12. *Рихтер, Д.* Windows via C/C++. Программирование на языке Visual C++ / Д. Рихтер, К. Назар; пер. с англ. – М.: Издательство «Русская Редакция»; СПб.: Питер, 2009. – 896 с.
13. *Руссинович, М.* Внутреннее устройство Microsoft Windows / М. Руссинович, Д. Соломон; пер. с англ. Н. Вильчинский. – 6-е изд. – СПб.: Питер, 2013. – 800 с.
14. *Таненбаум, Э.* Современные операционные системы / Э. Таненбаум; пер. с англ. Н. Вильчинский, А. Лашкевич. – 3-е изд. – СПб.: Питер, 2013. – 1120 с.

15. *Фленов, М. Е.* Программирование на C++ глазами хакера / М. Е. Фленов. – 2-е изд. – СПб.: БХВ-Петербург, 2009. – 352 с.

16. *Харт, М.* Системное программирование в среде Windows / М. Харт; пер. с англ. А. Г. Гузикевич. – 3-е изд. – М.: Издательский дом «Вильямс», 2005. – 592 с.

17. *Щупак, Ю. А.* Win32 API. Разработка приложений для Windows / Ю. А. Щупак. – СПб.: Питер, 2008. – 592 с.

18. Access Control Model [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/aa374876/.

19. Common Dialog Box Library [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms645524/.

20. Control Library [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/bb773169/.

21. Controls [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/aa511482/.

22. Keyboard Accelerators [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms645526/.

23. Keyboard and Mouse Input [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms632585/.

24. Security and Identity [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ee663293/.

25. Services [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms685141/.

26. Synchronization [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms686353/.

27. Windows and Messages [Электронный ресурс] // Microsoft Developer Network. – Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms632586/.

Учебное издание

Глухоедов Андрей Владимирович

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Лабораторный практикум

Подписано в печать 25.09.17 Формат 60x84/16 Усл. печ. л. 22,5. Уч.-изд. л. 24,2
Тираж 50 экз. Заказ Цена
Отпечатано в Белгородском государственном технологическом университете
им. В.Г. Шухова
308012, г. Белгород, ул. Костюкова, 46