П.В.Румянцев

РАБОТА С ФАЙЛАМИ В WIN 32 API

Изложены вопросы создания программных приложений для Windows. Рассмотрены основы работы с файлами в Win 32 API, структура исполняемого файла, его заголовки и разделы, экспорт и импорт функций, таблицы объектов, процессы и связанные с ними потоки.

В значительной степени материал книги развивает и дополняет другую книгу автора «Азбука программирования в Win 32 API», выдержавшую два издания.

Для программистов.

Оглавление

Здравствуй, мой читатель! (Вместо предисловия)

Уважаемый читатель! Некоторое время назад я написал книгу «Азбука программирования в Win32 API», посвящённую основам программирования для Windows'95 и Windows NT. В предисловии я написал, что я «предполагал дать быстрое и возможно более полное введение в программирование для Windows NT и Windows'95. При этом мне хотелось, чтобы материал, изложенный в ней, был полезен как начинающему программисту, так и специалисту, имеющему опыт в написании программ для Windows».

Издательство «Радио и связь» и я лично получили достаточно много откликов на эту книгу. В них читатели высказывали в целом хорошие впечатления от книги. Я очень благодарен читателям за их тёплые слова о моей работе. Эти слова – самая высокая награда автору за его труд. Но в то же время почти все, приславшие свои отклики, критиковали автора за то, что слишком много вопросов остались за пределами книги. Конечно, я оправдывался, что это, мол, только введение в программирование, а не полное описание, но вряд ли эти оправдания устраивали желающих в кратчайшие сроки узнать как можно больше о программировании для Windows.

Справедливости ради, я не задумывал ту книгу, которую Вы сейчас держите в руках, как продолжение предыдущей книги. Но, работая над книгой, я понял, что вольно или невольно я пишу продолжение «Азбуки...». Поэтому я не стал мудрствовать лукаво, как говорится, и для себя стал называть свою новую книгу, ту самую, которую Вы, уважаемый читатель, держите сейчас в руках, следующим образом – «Win32 API – продолжение азбуки».

Но, несмотря на то, что эта книга явилась продолжением предыдущей, ни в коем случае это не подразумевает, что её нельзя читать самостоятельно. Я постарался, чтобы и эта книга стала интересной как для начинающих, так и для опытных программистов.

В ходе работы над книгой я вспомнил о тех словах, которыми я закончил свою предыдущую книгу: «Теперь я должен расстаться со своим читателем и мне немного грустно. Я не знаю, что я должен сказать: «До *свидания*» или «*Прощайте*». *Надеюсь, что до свидания, мой читатель*». Получилось так, что мы расстались с Вами, уважаемый читатель, только на время. Поэтому я с радостью говорю: «Здравствуйте, мой читатель!»

И сразу же, не теряя времени, давайте начнём работу. Сначала я хотел поговорить о том, какие возможности по работе с файлами предлагает Win32 API. Но так как многие функции, обеспечивающие работу с файлами, требуют в качестве аргумента указатели на различные структуры, обеспечивающие безопасность того или иного объекта, давайте, уважаемый читатель, вкратце обсудим тему безопасности в Windows. Естественно, то, о чём мы будем говорить в этом разделе, касается прежде всего Windows NT.

Я хотел бы заметить, что вопросам безопасности сетей посвящено множество книг. Некоторые из них рассматривают вопросы физической безопасности сетей, то есть, например, ограничение физического доступа персонала в помещения с вычислительными машинами. Другие посвящены математическим основам шифрования информации в вычислительных машинах и надёжности средств шифрования. Третьи – непосредственно реализации системы безопасности. Я при рассказе о системе безопасности опираюсь на следующие посылки:

- понимая взаимодействие данных, легко понять и работу программы в целом;
- я пишу не подробное описание системы безопасности, а только очень краткое введение во взаимодействие данных, участвующих в обеспечении безопасности работы системы;
- в связи с деликатностью рассматриваемой темы что-то приходится недоговаривать и рассчитывать на то, что любознательный читатель, получив начальные сведения, сможет сам докопаться до более серьёзных вещей;
- я рассчитываю, что полученные знания читатель направит на укрепление и исследование систем безопасности, а не на «взломы» серверов.

Любое приложение, запущенное в системе, имеет свой собственный уровень привилегий, то есть ограничено определёнными рамками. Эти рамки, определяющие права приложений выполнять те или иные действия и осуществлять доступ к тем или иным данным, определяются системой безопасности операционной системы (извините за тавтологию).

Функции системы безопасности Win32 API позволяют разрешить или запретить приложению доступ к тем или иным ресурсам. Приложение может определить много различных уровней доступа для конечных пользователей и групп пользователей. Операционная система разрешает или запрещает доступ к ресурсам на основе сравнения характеристик уровня безопасности, присвоенных ресурсу, и уровня привилегий, которыми обладает приложение, запросившее доступ к ресурсу. Функции системы безопасности позволяют приложению запрашивать и манипулировать уровнями безопасности ресурса и привилегиями объекта.

Система безопасности оказывает минимальное воздействие на большинство функций Win32API, и большинство приложений, написанных для Windows, не требующих каких-либо манипуляций с объектами системы

безопасности, не требуют вставки какого-то специального кода. Тем не менее, разработчик может использовать некоторые возможности системы безопасности для того, чтобы ограничить доступ к некоторым системным ресурсам. Например, если приложение пытается изменить системное время, то система безопасности должна проверить, что у приложения есть права на выполнение этого действия. Приложение, обеспечивающее безопасность, может позволить пользовательской программе запросить атрибуты секретности файла, обеспечить соответствующую реакцию в том случае, если доступ к файлу по каким-то причинам запрещён, определять атрибуты файла или группы файлов так, чтобы только некоторые пользователи сети имели доступ к определённого рода информации.

Windows NT разработана с таким расчётом, чтобы обеспечить уровень безопасности C2, принятый Министерством Обороны Соединённых Штатов Америки. Некоторые наиболее важные требования, предъявляемые к уровню безопасности C2, приведены ниже:

- он должен обеспечивать контроль за доступом к ресурсам на уровне как отдельных пользователей, так и групп пользователей;
- память должна быть защищена, то есть её содержание не должно быть доступно для чтения после того, как процесс освободил память;
- в момент входа в систему пользователь должен однозначно идентифицировать себя. Система должна знать, кто осуществляет те или иные действия;
- системный администратор должен иметь возможности проверять все события, связанные с безопасностью. Доступ к данным, связанным с системой безопасности, должен иметь только администратор, имеющий соответствующие права;
- система должна предотвращать себя от внешнего воздействия или вмешательства в её работу.

Обеспечение безопасности объектов

Все именованные объекты в Windows и некоторые неименованные объекты могут быть защищены. Все атрибуты безопасности могут быть разделены по критерию принадлежности на общесистемные и принадлежащие объекту, а по критерию постоянства – на постоянные и временные. Атрибуты безопасности каждого объекта описываются дескриптором безопасности, в котором содержится информация о владельце объекта, а также списки контроля доступа (ACL – access control list), определяющие, какие пользователи и какие группы пользователей имеют доступ к объекту.

Дескриптор безопасности

Существуют пять категорий объектов, которые могут иметь дескриптор безопасности. Это файлы, объекты User'a (окна и т.д.), объекты ядра, объекты Registry, объекты, определённые пользователем.

Атрибуты секретности каждого защищённого объекта в системе описываются при помощи дескриптора безопасности. Структура дескриптора безопасности описана в файле winnt.h. Это описание я привожу ниже:

```
typedef struct _SECURITY_DESCRIPTOR {
  BYTE  Revision;
  BYTE  Sbz1;
  SECURITY_DESCRIPTOR_CONTROL Control;
  PSID Owner;
  PSID Group;
  PACL Sacl;
  PACL Dacl;
  } SECURITY_DESCRIPTOR, *PISECURITY_DESCRIPTOR;
```

Первое поле, Revision, определяет уровень просмотра дескриптора безопасности. Второе поле, Sbz1, не несёт никакой информационной нагрузки и предназначено всего-навсего для выравнивания первого поля на 16-битовую границу. Третье поле, Control, представляет собой набор флагов. Возможные значения флагов я привожу в таблице ниже:

Флаг	Значение
SE_OWNER_DEFAULTED	0x000
SE_GROUP_DEFAULTED	0x0002
SE_DACL_PRESENT	0x0004
SE_DACL_DEFAULTED	0x0008
SE_SACL_PRESENT	0x0010
SE_SACL_DEFAULTED	0x0020
SE_SELF_RELATIVE	0x8000

Поле четвёртое, Owner, является указателем на идентификатор безопасности (SID) пользователя, владеющего этим объектом. Если идентификатор безопасности записан в самоотносительной форме, то это поле содержит смещение соответствующего SID'а, а не указатель на него. Пятое поле, Group, является указателем на идентификатор безопасности (SID) первичной группы объекта. Если это поле содержит значение NULL, то, значит, у объекта нет первичной группы. Как и в случае предыдущего поля, это поле может содержать смещение соответствующего SID'а, а не указатель на него. Поле шестое, Sacl, содержит указатель или смещение системного ACL. Нужно заметить, что это поле содержит верное значение только в том случае, если в поле Control установлен флаг SE_SACL_PRESENT. Аналогично, седьмое поле, Dacl, содержит указатель или смещение частного ACL. Точно так же, поле содержит правильное значение только в том случае, если в поле Control установлен флаг SE_DACL_PRESENT.

О том, что представляет собой ACL, мы поговорим позже. А сейчас я хотел бы поговорить о том, при помощи каких функций мы можем получить доступ к дескриптору безопасности.

Функции доступа к дескрипторам безопасности

Первая функция из этой группы, позволяющая получить информацию об атрибутах безопасности файла и директории, называется GetFileSecurity(). Она описана в файле winbase.h. Её описание я привожу ниже:

```
WINADVAPI BOOL WINAPI GetFileSecurityA(LPCSTR lpFileName,
                        SECURITY_INFORMATION RequestedInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor,
                        DWORD nLength,
                        LPDWORD lpnLengthNeeded);
WINADVAPI BOOL WINAPI GetFileSecurityW(LPCWSTR lpFileName,
                        SECURITY_INFORMATION RequestedInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor,
                        DWORD nLength,
                        LPDWORD lpnLengthNeeded);
#ifdef UNICODE
#define GetFileSecurity  GetFileSecurityW
#else
#define GetFileSecurity  GetFileSecurityA
#endif // !UNICODE
```

Первый аргумент этой функции, lpFileName, является указателем на буфер, в котором записано имя файла или директории, информация об атрибутах безопасности которых запрашивается.

Второй аргумент, RequestedInformation, является набором флагов, определяющих, какая информация о безопасности запрашивается. Ниже я привожу возможные значения этих флагов:

Флаг	Значение	Назначение
OWNER_SECURITY_INFORMATION	0X00000001L	Запрашивается идентификатор владельца объекта
GROUP_SECURITY_INFORMATION	0X00000002L	Запрашивается идентификатор первичной группы объекта
DACL_SECURITY_INFORMATION	0X00000004L	Запрашивается информация о свободном ACL
SACL_SECURITY_INFORMATION	0X00000008L	Запрашивается информация о системном ACL

Третий аргумент, pSecurityDescriptor, является указателем на структуру типа SECURITY_DESCRIPTOR, в которую будет записана копия дескриптора безопасности объекта. В копию будет записана только та информация из дескриптора, которая запрашивалась вторым аргументом. Для того, чтобы получить подобную информацию, процесс должен иметь права просматривать информацию подобного характера.

Четвёртый аргумент, nLength, содержит размер буфера, на который указывает предыдущий аргумент.

И, наконец, пятый аргумент, lpnLengthNeeded, является указателем на двойное слово. В случае успешного выполнения функция записывает в это двойное слово нуль. В том случае, если буфер, определяемый третьим и четвёртым аргументами, слишком мал для того, чтобы вместить дескриптор безопасности, в это двойное слово записывается необходимый размер

буфера. Если это двойное слово не равно нулю, то в буфер ничего не копировалось.

Функцией, которая позволяет установить те или иные атрибуты безопасности файла или директории, является SetFileSecurity(). Описание этой функции, взятое из файла winbase.h, я привожу ниже:

```
WINADVAPI BOOL WINAPI SetFileSecurityA(LPCSTR lpFileName,
                        SECURITY_INFORMATION SecurityInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor);
WINADVAPI BOOL WINAPI SetFileSecurityW(LPCWSTR lpFileName,
                        SECURITY_INFORMATION SecurityInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor);
#ifdef UNICODE
#define SetFileSecurity  SetFileSecurityW
#else
#define SetFileSecurity  SetFileSecurityA
#endif // !UNICODE
```

Первый аргумент указывает на строку, содержащую имя файла. Второй аргумент является набором флагов, определяющих, какую часть поля дескриптора безопасности мы хотим получить. И третий аргумент указывает на структуру типа SECURITY_DESCRIPTOR, из которой будет взята та информация, которую мы хотим записать в дескриптор безопасности.

Теперь я хотел бы обратить внимание читателя на функции GetKernelObjectSecurity() и SetKernelObjectSecurity(), которые служат для получения информации о дескрипторах объектов ядра и установления атрибутов безопасности объектов ядра. Я приведу ниже их описания, взятые из файла winbase.h:

```
WINADVAPI BOOL WINAPI GetKernelObjectSecurity(HANDLE Handle,
                        SECURITY_INFORMATION RequestedInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor,
                        DWORD nLength,
                        LPDWORD lpnLengthNeeded);
WINADVAPI BOOL WINAPI SetKernelObjectSecurity(HANDLE Handle,
                        SECURITY_INFORMATION SecurityInformation,
                        PSECURITY_DESCRIPTOR SecurityDescriptor);
```

Внимательный читатель, бросив беглый взгляд на описания этих функций, сразу же заметит, что они практически ничем не отличаются от функций GetFileSecurity() и SetFileSecurity(), описанных несколько ранее. Единственное отличие состоит в том, что вместо указателя на строку, содержащую имя файла, во втором случае передаётся хэндл объекта ядра.

Функции GetUserObjectSecurity() и SetKernelObjectSecurity() также очень мало отличаются от функций GetKernelObjectSecurity() и SetKernelObjectSecurity(). Ниже я привожу описания этих функций, взятые из файла winuser.h:

```
WINUSERAPI BOOL WINAPI GetUserObjectSecurity(HANDLE hObj,
```

```
                                PSECURITY_INFORMATION pSIRequested,
                                PSECURITY_DESCRIPTOR pSID,
                                DWORD nLength,
                                LPDWORD lpnLengthNeeded);
WINUSERAPI BOOL WINAPI SetUserObjectSecurity(HANDLE hObj,
                                PSECURITY_INFORMATION pSIRequested,
                                PSECURITY_DESCRIPTOR pSID);
```

Отличие этих функций от предыдущих состоит в том, что в качестве второго аргумента функции передаётся не самоё структура типа SECURITY_INFORMATION, а *УКАЗАТЕЛЬ* на неё! Нужно ли после этого объяснять назначение остальных аргументов?

Ненамного отличаются от описанных выше и функции, предназначенные для получения и установки полей дескриптора безопасности ключей реестра. Описания этих функций, приведённые ниже, я взял из файла *winreg.h*:

```
WINADVAPI LONG APIENTRY RegGetKeySecurity(HKEY hKey,
                        SECURITY_INFORMATION SecurityInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor,
                        LPDWORD lpcbSecurityDescriptor);
WINADVAPI LONG APIENTRY RegSetKeySecurity(HKEY hKey,
                        SECURITY_INFORMATION SecurityInformation,
                        PSECURITY_DESCRIPTOR pSecurityDescriptor);
```

В этих функциях первым аргументом является хэндл ключа реестра, с информацией о котором мы хотим работать. О назначении остальных аргументов этих функция я предлагаю подумать читателю самостоятельно. Думаю, никаких трудностей эта задача не вызовет.

Разобравшись в целом с дескриптором безопасности, нам необходимо понять, что представляют собой

Списки управления доступом

Как мы отметили в предыдущей части, списки управления доступом (ACL), являющиеся частью дескриптора безопасности, определяют права доступа к объекту отдельных пользователей и групп пользователей.

Существуют два типа ACL – частный и системный списки.

Частный список администрируется владельцем объекта. Например, владелец объекта может определить, кто может и кто не может получить доступ к файлу.

Если в частном ACL объекта нет записей, это означает что права для доступа к объекту не предоставляются никому, то есть доступ к объекту неявно запрещён. Если же у объекта вообще нет частного ACL, то это означает, что доступ к объекту разрешён всем.

Кроме частного списка управления доступом, у объекта может быть и системный список, который администрируется системным администратором.

ACL состоит из двух частей – заголовка и непосредственно списка. Описание заголовка ACL находится в файле winnt.h, я привожу это описание ниже:

```
typedef struct _ACL {
   BYTE  AclRevision;
   BYTE  Sbz1;
   WORD  AclSize;
   WORD  AceCount;
   WORD  Sbz2;
} ACL;
typedef ACL *PACL;
```

Первое поле структуры, AclRevision, определяет уровень просмотра ACL. В настоящее время значение этого поля должно быть равно ACL_REVISION, которое в файле winnt.h определено так:

```
#define ACL_REVISION  (2)
```

Второе поле – это просто нулевой байт, который выравнивает поле AclRevision до 16-битовой границы. Третье поле, AclSize, хранит размер ACL. В данном случае под ACL понимается совокупный размер заголовка ACL и всех строк в списке. Четвёртое поле, AceCount; хранит число строк в списке управления доступом. Заметим следующее – раз это поле занимает всего одно слово, то, значит, в системе не может быть зарегистрировано более 65 535 пользователей и групп.. И, наконец, последнее поле, Sbz2, тоже является нулевым полем, предназначенным для выравнивания структуры ACL на 32-разрядную границу.

Каждая строка в списке управления доступом (ACL) называется входом (ACE – access control entry). Каждая вход состоит из заголовка входа, формат которого одинаков для всех входов, за которым следуют данные, зависящие от типа входа. Структуру заголовка ACE, описанную в файле winnt.h, я привожу ниже:

```
typedef struct _ACE_HEADER {
   BYTE  AceType;
   BYTE  AceFlags;
   WORD  AceSize;
} ACE_HEADER;
typedef ACE_HEADER *PACE_HEADER;
```

AceType, первое поле структуры, указывает тип входа. Список возможных типов я привожу в таблице ниже:

Тип	Значение	Назначение
ACCESS_ALLOWED_ACE_TYPE	0x0	Доступ разрешён
ACCESS_DENIED_ACE_TYPE	0x1	Доступ запрещён
SYSTEM_AUDIT_ACE_TYPE	0x2	Системная проверка
SYSTEM_ALARM_ACE_TYPE	0x3	В настоящее время не используется

То, какой тип доступа определён в поле типа входа, определяет формат информации, следующей за заголовком входа.

Типы входов ACL

Если в первом поле структуры типа ACE_HEADER указан тип входа «ACCESS_ALLOWED_ACE_TYPE», то фактически заголовок входа является частью структуры типа ACCESS_ALLOWED_ACE, описание которой, находящееся в файле winnt.h, я привожу ниже:

```
typedef struct _ACCESS_ALLOWED_ACE {
  ACE_HEADER Header;
  ACCESS_MASK Mask;
  DWORD SidStart;
} ACCESS_ALLOWED_ACE;
```

Эта структура определяет вход, который описывает права на доступ к объекту для объекта, идентификатор безопасности которого указан в поле SidStart. Особого разговора заслуживает второе поле этой структуры, Mask. Тип ACCESS_MASK описан в файле winnt.h следующим образом:

```
typedef DWORD ACCESS_MASK;
typedef ACCESS_MASK *PACCESS_MASK;
```

Однако, в том же файле я нашёл другое описание структуры типа ACCESS_MASK. К сожалению, оно всё было закомментировано, тем не менее, я приведу его также:

```
//    typedef struct _ACCESS_MASK {
//      WORD  SpecificRights;
//      BYTE  StandardRights;
//      BYTE  AccessSystemAcl : 1;
//      BYTE  Reserved : 3;
//      BYTE  GenericAll : 1;
//      BYTE  GenericExecute : 1;
//      BYTE  GenericWrite : 1;
//      BYTE  GenericRead : 1;
//    } ACCESS_MASK;
//    typedef ACCESS_MASK *PACCESS_MASK;
```

Это закомментированное описание структуры в точности совпадает с описанием, которое я нашёл в других источниках. Почему его закомментировали? Не понимаю. Очень удобное описание... Итак, биты с нулевого по пятнадцатый определяют права, специфические для объекта определённого типа. На них я останавливаться не буду. Биты с шестнадцатого по двадцать третий определяют стандартные для всех объектов права. Некоторые возможные значения флагов и их и назначения я приведу в таблице ниже:

Флаг	Значение	Назначение
SPECIFIC_RIGHTS_ALL	0x0000FFFFL	Все специфические права
DELETE	0x00010000L	*Право удалять объект*
READ_CONTROL	0x00020000L	Право чтения дескриптора безопасности (исключая системный ACL)
WRITE_DAC	0x00040000L	Право записи в частный ACL
WRITE_OWNER	0x00080000L	Право записи для владельца
STANDARD_RIGHTS_REQUIRED	0x000F0000L	Стандартные права
SYNCHRONIZE	0x00100000L	Синхронизация доступа
STANDARD_RIGHTS_ALL	0x001F0000L	Все стандартные права
ACCESS_SYSTEM_SECURITY	0x01000000L	Право доступа к системному ACL
MAXIMUM_ALLOWED	0x02000000L	
GENERIC_ALL	0x10000000L	
GENERIC_EXECUTE	0x20000000L	
GENERIC_WRITE	0x40000000L	
GENERIC_READ	0x80000000L	

Если в первом поле структуры типа ACE_HEADER указан тип входа ACCESS_DENIED_ACE_TYPE, то тогда заголовок входа является частью структуры ACCESS_DENIED_ACE, описание которой, взятое в файле winnt.h, я привожу ниже:

```
typedef struct _ACCESS_DENIED_ACE {
   ACE_HEADER Header;
   ACCESS_MASK Mask;
   DWORD SidStart;
} ACCESS_DENIED_ACE;
typedef ACCESS_DENIED_ACE *PACCESS_DENIED_ACE;
```

Всё отличие этой структуры от предыдущей состоит в предназначении. *В этой структуре во втором поле указываются флаги, определяющие* запрещение соответствующего вида доступа к объекту для пользователя, идентификатор которого равен SidStart.

Если тип ACE указан как SYSTEM_AUDIT_ACE_TYPE, то заголовок входа является частью структуры SYSTEM_AUDIT_ACE. Её описание из файла winnt.h выглядит так:

```
typedef struct _SYSTEM_AUDIT_ACE {
   ACE_HEADER Header;
   ACCESS_MASK Mask;
   DWORD SidStart;
} SYSTEM_AUDIT_ACE;
typedef SYSTEM_AUDIT_ACE *PSYSTEM_AUDIT_ACE;
```

В этой структуре определяются те события, которые вызывают необходимость оповещения о них системного администратора. Все попытки осуществления соответствующего доступа будут записаны в соответствующем лог – файле.

И, наконец, последний тип входа - SYSTEM_ALARM_ACE_TYPE. Он, к сожалению, зарезервирован для использования в будущем и в настоящее время не используется.

Обсудив типы входов, вернёмся к рассмотрению формата ACE_HEADER.

В следующей таблице я привожу список флагов, которые могут быть указаны в третьем поле структуры, AceFlags.

Флаг	Значение
OBJECT_INHERIT_ACE	0x1
CONTAINER_INHERIT_ACE	0x2
NO_PROPAGATE_INHERIT_ACE	0x4
INHERIT_ONLY_ACE	0x8
VALID_INHERIT_FLAGS	0xf
SUCCESSFUL_ACCESS_ACE_FLAG	0x40
FAILED_ACCESS_ACE_FLAG	0x80

Последнее поле структуры, AceSize, хранит размер входа.

Функции для работы с ACL

Если возникает необходимость создать новый ACL, то для этого можно воспользоваться функцией InitializeAcl(), которая в файле winbase.h описана следующим образом:

```
WINADVAPI BOOL WINAPI InitializeAcl (PACL pAcl,
                                     DWORD nAclLength,
                                     DWORD dwAclRevision);
```

При необходимости получить доступ к частному ACL мы можем использовать функцию, которая в файле winbase.h описана так:

```
WINADVAPI BOOL WINAPI GetSecurityDescriptorDacl
                          (PSECURITY_DESCRIPTOR pSecurityDescriptor,
                           LPBOOL lpbDaclPresent,
                           PACL *pDacl,
                           LPBOOL lpbDaclDefaulted);
```

Если же нам нужен доступ к системному ACL, то мы можем использовать функцию GetSecurityDescriptorSacl(), описание которой в файле winbase.h выглядит следующим образом:

```
WINADVAPI BOOL WINAPI GetSecurityDescriptorSacl
                          (PSECURITY_DESCRIPTOR pSecurityDescriptor,
                           LPBOOL lpbSaclPresent,
                           PACL *pSacl,
```

```
LPBOOL lpbSaclDefaulted);
```

Во время подключения к системе каждому пользователю присваивается признак доступа (access token), который идентифицирует пользователя и все группы, к которым принадлежит пользователь. Каждый процесс, запущенный этим пользователем, содержит копию этого признака доступа. Когда процесс пытается осуществить доступ к объекту, система сравнивает атрибуты безопасности, записанные в признаке доступа, с соответствующими строками в ACL, и на основе результатов этого анализа принимает решение о том, имеет ли пользователь право осуществлять доступ к объекту.

Идентификаторы безопасности

Помимо признака доступа, у каждого пользователя и каждой группы пользователей есть свой идентификатор безопасности (SID – security identifier). SID – это структура переменной длины, которая однозначно определяет пользователя или группу пользователей. SID'ы хранятся в базе данных о безопасности. SID'ы всегда уникальны. Если SID хотя бы один раз использовался, то возможность его повторного использования исключена.

SIDы идентифицируют сразу несколько элементов. Это можно заметить по структуре SID'а, которая описана в файле winnt.h:

```
typedef struct _SID {
  BYTE  Revision;
  BYTE SubAuthorityCount;
  SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
#ifdef MIDL_PASS
  [size_is(SubAuthorityCount)] DWORD SubAuthority[*];
#else // MIDL_PASS
  DWORD SubAuthority[ANYSIZE_ARRAY];
#endif // MIDL_PASS
} SID, *PISID;
```

Давайте рассмотрим каждый элемент этой структуры более подробно. Первое поле, Revision, определяет уровень просмотра SID'а. Точнее, этот уровень определяется только в младших четырёх разрядах байта, занимаемого полем, а старшие четыре разряда используются только для выравнивания поля на границу байта. Второе поле, SubAuthorityCount, указывает сколько значений поставторизации записано в поле SID'а. Идентификатор авторизации записан в третьем поле, IdentifierAuthority. Я обращу внимание читателя на то, что тип третьего поля, SID_IDENTIFIER_AUTHORITY, в файле winnt.h описан следующим образом:

```
typedef struct _SID_IDENTIFIER_AUTHORITY {
   BYTE  Value[6];
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;
```

Другими словами, это поле занимает всего–навсего шесть байтов. И, наконец, размер последнего поля, SubAuthority, определяется переменной ANYSIZE_ARRAY. В файле winnt.h эта переменная определяется следующим образом:

```
#define ANYSIZE_ARRAY 1
```

Очевидно, что массив, состоящий из одного элемента, создавать просто-напросто нецелесообразно, поэтому предположим, что это поле является просто-напросто первым элементом массива, в котором определяются уровни поставторизации.

Идентификатор авторизации, состоящий из двух частей, является наиболее важной частью SID'а. Он содержит сорокавосьмибитовый идентификатор компьютера, выдавшего SID, а также тридцатидвухбитовый относительный идентификатор (RID), однозначно определяющий пользователя или группу на этом компьютере. При соединении этих значений мы можем быть уверены, что полученное семидесятибитовое значение никогда не повторится, то есть SID всегда будет уникальным.

Стандартная краткая форма записи SID'а выглядит следующим образом:

S-*R-I-S-S*.....

В этой записи первая буква S указывает, что следующая за ней серия цифр представляет собой SID, в котором R – уровень просмотра, I – идентификатор авторизации, а последующие буквы S представляют собой уровни поставторизации. Например, запись

S-1-2758-43

определяет SID с уровнем просмотра 1, идентификатором авторизации 2758 и уровнем поставторизации 43.

Приложения никогда не манипулирует SID'ами напрямую. Для того, чтобы манипулировать SID'ами, приложение может использовать специальные функции.

Функции для работы с идентификаторами безопасности

Для того, чтобы инициализировать структуру SID, приложению необходимо вызвать функцию InitializeSid(), которая в файле winbase.h описана следующим образом:

```
WINADVAPI BOOL WINAPI InitializeSid
                    (PSID Sid,
                     PSID_IDENTIFIER_AUTHORITY pIdentifierAuthority,
                     BYTE nSubAuthorityCount);
```

Эта функция позволяет инициализировать идентификатор безопасности. Первый аргумент, Sid, должен быть указателем на инициализируемую структуру типа SID. Второй аргумент, pIdentifierAuthority, указывает на структуру типа SID_IDENTIFIER_AUTHORITY, которая определяет идентификатор авторизации. Некоторые возможные значения этого аргумента я привожу в таблице ниже:

Идентификатор	Значение
SECURITY_NULL_SID_AUTHORITY	{0,0,0,0,0,0}
SECURITY_WORLD_SID_AUTHORITY	{0,0,0,0,0,1}
SECURITY_LOCAL_SID_AUTHORITY	{0,0,0,0,0,2}
SECURITY_CREATOR_SID_AUTHORITY	{0,0,0,0,0,3}
SECURITY_NON_UNIQUE_AUTHORITY	{0,0,0,0,0,4}

И, наконец, последний аргумент, nSubAuthorityCount, указывает число записей о поставторизации.

Если работа функции завершена нормально, то возвращённое ею значение равно TRUE. Значение FALSE говорит о том, что при работе функции произошла какая-то ошибка.

Если мы хотим не только инициализировать структуру, но и возложить на систему заботу о выделении для неё памяти и т.д., мы можем воспользоваться другой функцией, AllocateAndInitializeSid(). Если мы обратимся к файлу winbase.h, то мы увидим там следующее описание этой функции:

```
WINADVAPI BOOL WINAPI AllocateAndInitializeSid
                    (PSID_IDENTIFIER_AUTHORITY pIdentifierAuthority,
                     BYTE nSubAuthorityCount,
                     DWORD nSubAuthority0,
                     DWORD nSubAuthority1,
                     DWORD nSubAuthority2,
                     DWORD nSubAuthority3,
                     DWORD nSubAuthority4,
                     DWORD nSubAuthority5,
                     DWORD nSubAuthority6,
                     DWORD nSubAuthority7,
                     PSID *pSid);
```

Первый аргумент этой функции, pIdentifierAuthority, является указателем на структуру типа SID_IDENTIFIER_AUTHORITY, в которой определяется идентификатор авторизации. Второй аргумент, nSubAuthorityCount, определяет число уровней поставторизации, другими словами, показывает, сколько последующих аргументов нужно принимать во внимание. Это значение может быть в пределах от одного до восьми включительно. Аргументы с третьего по десятый должны содержать упомянутые выше уровни поставторизации. Повторюсь, что принимать во внимание нужно ровно столько аргументов, сколько указано в nSubAuthorityCount. И последний аргумент, pSid, должен указывать на переменную, в которую будет записан указатель созданного SID'а.

При нормальном завершении функция возвращает TRUE. Если же произошла ошибка, то возвращённое функцией значение будет равно FALSE.

Если у нас есть имя пользователя, а мы хотим получить его SID, то нам необходимо воспользоваться функцией LookupAccountName(), которая в файле winbase.h описана так:

```
WINADVAPI BOOL WINAPI LookupAccountNameA
                                        (LPCSTR lpSystemName,
                                         LPCSTR lpAccountName,
                                         PSID Sid,
                                         LPDWORD cbSid,
                                         LPSTR ReferencedDomainName,
                                         LPDWORD cbReferencedDomainName,
                                         PSID_NAME_USE peUse);
WINADVAPI BOOL WINAPI LookupAccountNameW
                                        (LPCWSTR lpSystemName,
                                         LPCWSTR lpAccountName,
                                         PSID Sid,
                                         LPDWORD cbSid,
                                         LPWSTR ReferencedDomainName,
                                         LPDWORD cbReferencedDomainName,
                                         PSID_NAME_USE peUse);
#ifdef UNICODE
#define LookupAccountName  LookupAccountNameW
#else
#define LookupAccountName  LookupAccountNameA
#endif // !UNICODE
```

Давайте попробуем разобраться со всем многообразием этих аргументов. Аргумент первый, lpSystemName, указывает на строку, содержащую название компьютера, с которого пользователь осуществляет доступ. Если этот аргумент равен нулю, это означает, что имя пользователя должно быть найдено на локальном компьютере. Второй аргумент, lpAccountName, представляет собой указатель на строку, которая содержит имя пользователя. Третий аргумент, Sid, указывает на буфер, в который будет записан SID для пользователя, имя которого указано во втором аргументе. Следующий аргумент, cbSid, содержит размер буфера ,указанного в предыдущем параметре. Если буфер слишком мал для того, чтобы в нём был размещён требуемый SID, то функция запишет в двойное слово, на которое указывает аргумент, требуемый размер буфера. ReferencedDomainName, пятый аргумент, указывает на строку, в которую после выполнения функции будет записано имя домена, на котором будет найдено имя пользователя. Размер этой строки определяется шестым аргументом, cbReferencedDomainName. Если строка слишком мала, то после завершения работы функции в двойное слово, на которое указывает cbReferencedDomainName, будет записан требуемый размер строки. Последний аргумент этой функции, peUse, указывает на переменную типа SID_NAME_USE, которая после выполнения функции будет содержать тип SID'a. Тип SID_NAME_USE описан в файле winnt.h следующим образом:

```
typedef enum _SID_NAME_USE {
  SidTypeUser = 1,
  SidTypeGroup,
  SidTypeDomain,
  SidTypeAlias,
```

```
    SidTypeWellKnownGroup,
    SidTypeDeletedAccount,
    SidTypeInvalid,
    SidTypeUnknown
} SID_NAME_USE, *PSID_NAME_USE;
```

Я думаю, что значения, которые может принимать эта переменная, самоочевидны и не требуют каких-либо объяснений.

В случае успешного завершения функция возвращает значение TRUE. Если же функция вернула FALSE, то нам необходимо разобраться, что вызвало появление ошибки

Если мы не хотим полагаться на собственную интуицию и хотим быть уверены в том, что размер буфера, выделенного нами для SID'а всегда будет достаточным для размещения SID'а, мы можем воспользоваться функцией GetSidLengthRequired(), которая в файле winbase.h описана следующим образом:

```
WINADVAPI DWORD WINAPI GetSidLengthRequired
                                        (UCHAR nSubAuthorityCount);
```

Единственный аргумент этой функции – количество уровней поставторизации SID'а. Возвращаемое функцией значение всегда содержит количество байтов, необходимых для сохранения SID'а с данным количеством уровней поставторизации. Но как нам определить количество уровней поставторизации? А для этой цели у нас есть ещё одна функция, которой мы можем воспользоваться, - GetSidSubAuthorityCount(). Для того, чтобы найти её описание, мы обратимся к файлу winbase.h. Найденное в нём описание этой функции я привожу ниже:

```
WINADVAPI PUCHAR WINAPI GetSidSubAuthorityCount(PSID pSid);
```

В качестве единственного аргумента этой функции используется указатель на SID, а возвращаемое значение содержит указатель на поле SubAuthorityCount в структуре SID.

Если же нам необходимо получить имя пользователя, зная его SID, то для этой цели нам необходимо вызвать функцию LookupAccountSid(), описание которой, взятое из файла winbase.h, я привожу ниже:

```
WINADVAPI BOOL WINAPI LookupAccountSidA(LPCSTR lpSystemName,
                                        PSID Sid,
                                        LPSTR Name,
                                        LPDWORD cbName,
                                        LPSTR ReferencedDomainName,
                                        LPDWORD cbReferencedDomainName,
                                        PSID_NAME_USE peUse);
WINADVAPI BOOL WINAPI LookupAccountSidW(LPCWSTR lpSystemName,
                                        PSID Sid,
                                        LPWSTR Name,
                                        LPDWORD cbName,
                                        LPWSTR ReferencedDomainName,
```

```
                                    LPDWORD cbReferencedDomainName,
                                    PSID_NAME_USE peUse);
#ifdef UNICODE
#define LookupAccountSid  LookupAccountSidW
#else
#define LookupAccountSid  LookupAccountSidA
#endif // !UNICODE
```

Эта функция подобна функции LookupAccountName(), но, в отличие от последней, она ищет не SID, соответствующий данному имени пользователя, а, наоборот, имя пользователя, соответствующее данному SID'у. *Первый аргумент этой функции, lpSystemName, - это указатель на строку*, в которой записано имя компьютера. Если этот аргумент равен NULL, то имя пользователя ищется только на локальном компьютере. Второй аргумент, Sid, указывает на структуру типа SID, соответствующее которой имя пользователя и ищется. Третий аргумент, Name, является указателем на строку, в которую будет записано искомое имя пользователя. Размер этой строки в двойном слове, на которое указывает следующий, четвёртый аргумент, cbName. В строку, на которую указывает пятый аргумент, ReferencedDomainName, записывается имя того компьютера, на котором был найден SID, информацию о котором мы ищем. Шестой аргумент, cbReferencedDomainName, указывает на двойное слово, определяющее размер буфера для имени компьютера. И, наконец, тип SID'а указывается в поле, указателем на которое является последний аргумент, peUse.

Отработав с инициализированной нами структурой типа SID, мы должны, естественно, освободить память. Это делается при помощи функции FreeSid().Функция FreeSid() описана в файле winbase.h следующим образом:

```
WINADVAPI PVOID WINAPI FreeSid(PSID pSid);
```

Единственным аргументом этой функции является указатель на SID, который мы получили при помощи обращения к функции AllocateAndInitializeSid(). Освободив занимаемую структурой, мы можем считать, что работу с SID'ом мы закончили.

Привилегии

Привилегии используются для того, чтобы более строго управлять доступом к ресурсам системы. Администратор сети использует привилегии для того, чтобы определять, кто из пользователей имеет право манипулировать системными ресурсами. Приложения используют привилегии в тех случаях, когда им необходимо изменить системные ресурсы.

Привилегии представляют собой локальный уникальный идентификатор (LUID – locally unique identifier), который идентифицируется символьной строкой. Привилегии могут быть представлены в одном из следующих видов:

- обычное имя, так называемое глобальное имя программы, например, «SE_SYATEMTIME_NAME»;

- удобочитаемое имя, которое может при необходимости быть отображено пользователю, например, «Change the system time»;
- *локальное представление, которое изменяется от компьютера к компьютеру.*

Привилегии предполагают доступ к таким сервисам, которые редко бывают нужны обычным пользователям. Для обычных пользователей привилегии запрещены, поэтому для того, чтобы пользователь мог использовать привилегии, их необходимо разрешить.

Привилегии, которые определены в Windows NT, я привожу в таблице ниже.

Привилегия	Значение	Назначение
SE_CREATE_TOKEN_NAME	TEXT("SeCreateToken Privilege")	Создать первичный признак доступа
SE_ASSIGNPRIMARYTOKEN_ NAME	TEXT("SeAssignPrimaryToken Privilege")	Присвоить первичный признак доступа процессу
SE_LOCK_MEMORY_NAME	TEXT("SeLockMemory Privilege")	Зафиксировать физические страницы в памяти
SE_INCREASE_QUOTA_NAME	TEXT("SeIncreaseQuotaPrivilege")	Увеличить квоту, выделенную процессу
SE_UNSOLICITED_INPUT_ NAME	TEXT("SeUnsolicitedInputPrivilege")	Осуществлять ввод с терминального устройства
SE_MACHINE_ACCOUNT_ NAME	TEXT("SeMachineAccountPrivilege")	
SE_TCB_NAME	TEXT("SeTcbPrivilege")	*This privilege identifies its* holder as part of the trusted computer base. Some trusted protectedë subsystems are granted this privilege.
SE_SECURITY_NAME	TEXT("SeSecurityPrivilege")	Произвести некоторые действия, такие, например, как управление и просмотр проверочных сообщений. Эта привилегия идентифицирует держателя её как оператора системы безопасности.
SE_TAKE_OWNERSHIP_ NAME	TEXT("SeTakeOwnershipPrivilege")	Получить объект в собственность в том случае, если в частном ACL нет соответствующей записи.
SE_LOAD_DRIVER_NAME	TEXT("SeLoadDriverPrivilege")	Загрузить или выгрузить *системный драйвер*
SE_SYSTEM_PROFILE_NAME	TEXT("SeSystemProfilePrivilege")	Собирать информацию обо всей системе
SE_SYSTEMTIME_NAME	TEXT("SeSystemtimePrivilege")	Изменить системное время
SE_PROF_SINGLE_PROCESS_ NAME	TEXT("SeProfileSingleProcess Privilege")	Собирать информацию о конкретном процессе
SE_INC_BASE_PRIORITY_ NAME	TEXT("SeIncreaseBasePriority Privilege")	Изменить основной приоритет процесса
SE_CREATE_PAGEFILE_ NAME	TEXT("SeCreatePagefile Privilege")	Создать страничный файл
SE_CREATE_PERMANENT_ NAME	TEXT("SeCreatePermanent Privilege")	Создать постоянный объект
SE_BACKUP_NAME	TEXT("SeBackupPrivilege")	Выполнять операции возвратного сохранения
SE_RESTORE_NAME	TEXT("SeRestorePrivilege")	Выполнять операции восстановления информации после её сохранения
SE_SHUTDOWN_NAME	TEXT("SeShutdown Privilege")	Осуществлять выключение локального компьютера
SE_DEBUG_NAME	TEXT("SeDebugPrivilege")	Отладка процесса
SE_AUDIT_NAME	TEXT("SeAuditPrivilege")	Генерировать записи аудит-лога. Эта привилегия обычно устанавливается на серверах безопасности

SE_SYSTEM_ENVIRONMENT_ NAME	TEXT("SeSystemEnvironment Privilege")	Осуестивить запись в неизменяемую часть системной памяти (RAM), которая используется для хранения информации о конфигурации системы (environment)
SE_CHANGE_NOTIFY_NAME	TEXT("SeChangeNotifyPrivilege")	Получать оповещения об изменениях, произошедших в файловой системе. По умолчанию она разрешена для всех пользователей.
SE_REMOTE_SHUTDOWN_ NAME	TEXT("SeRemoteShutdown Privilege")	Осуществлять останов системы по сети

Для того, чтобы осуществлять доступ к привилегиям, пользователь может использовать несколько функций. Первая из них, PrivilegeCheck, в файле winbase.h описана так:

```
WINADVAPI BOOL WINAPI PrivilegeCheck (HANDLE ClientToken,
                                      PPRIVILEGE_SET RequiredPrivileges,
                                      LPBOOL pfResult);
```

Первый аргумент – это хэндл метки пользователя, привилегии которого проверяются. Второй – указатель на набор привилегий, информацию о которых мы хотим получить. И третий – указатель на флаг, который будет установлен в зависимости от того, всеми ли запрашиваемыми привилегиями обладает пользователь.

Остальные функции из этой группы не так важны и используются для того, чтобы получить какую-то информацию в удобном для пользователя виде. Информация о функциях LookupPrivilegeValue(), LookupPrivilegeDisplayName() и LookupPrivilegeName() может легко быть найдена в любом справочном руководстве.

И, наконец, вершиной того, о чём я говорил в этом разделе, является так называемый

Маркер доступа

В тот момент, когда пользователь регистрируется для того, чтобы войти в сеть, система проверяет его пароль и в соответствии с правами пользователя, определёнными в базе данных по безопасности, присваивает пользователю маркер доступа (access token). В этом маркере доступа указываются идентификаторы безопасности пользователя и групп, к которым он принадлежит, частный список управления доступом, привилегии пользователя и многое другое. Этот маркер доступа остаётся с пользователем до того времени, пока он не выйдет из системы. Таким образом, если пренебречь малой вероятностью наличия ошибок в системе безопасности, пользователь не может выйти за пределы отведённых ему рамок.

Выше я рассказал о структурах данных, используемых при работе системы безопасности Windows NT. Естественно, как я уже говорил, этих сведений недостаточно для того, чтобы начать писать полноценную программу, предназначенную для встраивания её в систему безопасности. Но,

я надеюсь, что читатель получил здесь информацию, достаточную для того, чтобы НАЧАТЬ ИЗУЧЕНИЕ системы безопасности Windows NT.

Перед тем, как начать рассказ об исполняемых файлах и об информации, которую мы можем извлечь из них, я вкратце расскажу о некоторых вещах, которые мы будем использовать в наших демонстрационных программах. Мы поговорим о взломщиках сообщений, а также о функциях, которые позволяют нам узнать о том, какие процессы, потоки и модули находятся в памяти системы, как распределены кучи. После этого мы перейдем к анализу исполняемого файла.

Перехватчики сообщений

Как уважаемый читатель, наверное, уже знает, Windows по сути своей является объектно-ориентированной системой. Тем не менее, для того, чтобы писать для нее программы, вовсе не нужно знать объектно-ориентированное программирование. Обычно оконная функция программы для Windows, написанной без применения объектно-ориентированной технологии программирования, выглядит примерно так :

```
switch(Message)
 {
   case WM_.NUMBER_ONE:
     <что-то делается>
     break;
   case WM_NUMBER_TWO:
     <что-то делается>
     break;
   default:
     return DefWindowProc(hWnd, Message, wParam, lParam);
}
```

Писать такую программу было в достаточной степени неприятно. Почему? Да потому что допустить в ней ошибку было проще простого. Представьте себе, что можно забыть указать какой-нибудь break или не определить действия по умолчанию. Результат, как говорится, непредсказуем.

Что же делать в этой ситуации? Изучать C++ и использовать при разработке программ объектно-ориентированную технологию? Но на это может уйти несколько месяцев, а время - очень дорогая штука. Продолжать использовать связки switch - case - break? А это, как говорится, себе дороже. В какие затраты это может вылиться при отладке, можно только представить.

Кроме этого, параметры сообщений Windows кодируются в каждом сообщении по-своему. Я думаю, что без справочника достаточно трудно вспомнить, где хранится, скажем в сообщении WM_COMMAND код нотификации, а где - хэндл окна. В старшем или младшем слове lParam? А, может быть, и не в lParam вовсе, а в wParam? И так далее, так далее, так далее...

Наверное, специалисты из фирмы Microsoft тоже были задавались подобными вопросами, ибо они решили эту задачу. Причем, как всегда, решили ее достаточно оригинально и интересно. Они предложили использовать взломщики сообщений (message crackers). При использовании взломщиков сообщений каждая часть оконной процедуры от case до break включительно будет выглядеть так, как показано ниже :

```
HANDLE_MSG(hWnd, WM_NUMBER_ONE, Cls_OnNumberOn);
```

А эквивалент фрагмента, приведенного выше, будет выглядеть следующим образом:

```
switch(Message)
 {
  HANDLE_MSG(hWnd, WM_NUMBER_ONE, Cls_OnNumberOne);
  HANDLE_MSG(hWnd, WM_NUMBER_TWO, Cls_OnNumberTwo);
  default:
   return DefWindowProc(hWnd, Message, wParam, lParam);
 }
```

Выглядит намного лучше первого варианта, не правда ли? На обработку каждого сообщения в оконной функции затрачивается ровно одна строка исходного кода (что происходит внутри этой строчки - это уже другой разговор). Текст оконной функции становится намного более читабельным, ибо фактически оконная функция превращается в совокупность обращений к макросам. Естественно, что и ошибку в таком случае допустить значительно сложнее. Итак, одна задача - упрощение кода оконной функции (или хотя бы часть этой задачи) решена. А теперь давайте попробуем разобраться, каким образом мы решаем вторую задачу - упрощение обработки параметров сообщений.

В поставках Borland C++ или Visual C++ есть файл под названием windowsx.h. Макро HANDLE_MSG описано в нем следующим образом:

```
#define HANDLE_MSG(hwnd, message, fn)   \
  case (message): return HANDLE_##message((hwnd), (wParam), (lParam), (fn))
```

Даже беглого взгляда на определение этого макро достаточно для того чтобы понять, что при каждом обращении к HANDLE_MSG вызывается другое макро, имя которого зависит от имени обрабатываемого сообщения. Например, для обработки WM_COMMAND вызывается макро HANDLE_WM_COMMAND, а для обработки WM_QUIT - макро HANDLE_WM_QUIT. Параметры этого макро понятны - хэндл окна, которому послано сообщение, параметры сообщения и адрес функции, которая *будет производить обработку этого сообщения. Об этой функции мы еще* поговорим.

Теперь обратимся непосредственно к вопросу взлома сообщений. Дело в том, что каждый из макросов HANDLE_##message вызываемой функции передает информацию таким образом, чтобы функции не пришлось заботиться об анализе wParam и lParam. Все разбиение wParam и lParam на отдельные компоненты, несущие смысловую нагрузку, производит макрос. Приведу два примера.

Допустим, что какой-то элемент управления послал нашему окну сообщение WM_COMMAND. При обработке сообщения WM_COMMAND параметры кодируются следующим образом:

- младшее слово wParam содержит идентификатор элемента, пославшего сообщение;
- lParam содержит хэндл окна, пославшего сообщение;
- старшее слов wParam содержит код нотификации, то есть определяет, какое именно действие пользователь произвел с элементом управления.

Посмотрим теперь, как реализовано макро HANDLE_WM_COMMAND в файле windowsx.h:

```
/*void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)*/
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
  ((fn)((hwnd), (int)(LOWORD(wParam)), (HWND)(lParam), (UINT)HIWORD(wParam)),
0L)
```

Видно, что макрос разложил сообщения на части, соответствующие приведенным ниже. Теперь очевидно, какие параметры должна иметь функция, которую мы должны написать для обработки сообщения WM_COMMAND. Кстати, для взломщиков сообщений, которые включены в windowsx.h, в качестве комментариев перед определением макроса приведены прототипы функций-обработчиков.

Второй пример - получение окном сообщения WM_DESTROY. В этом случае параметры сообщения нас вовсе не интересуют. Взломщик сообщения тоже игнорирует их:

```
/* void Cls_OnDestroy(HWND hwnd) */
#define HANDLE_WM_DESTROY(hwnd, wParam, lParam, fn) \
  ((fn)(hwnd), 0L)
```

Понятно, что таким образом можно обработать любое сообщение и при необходимости написать новый макрос.

Возникает вопрос: а как же быть в тех случаях, когда функция-обработчик должна передать сообщение на обработку по умолчанию или, скажем, вызвать другую функцию, которая требует передачи ей wParam и lParam, а не их частей? И здесь нет ничего сложного. В том же файле windowsx.h, помимо макросов HANDLE_##message существуют также макросы FORWARD_##message, которые восстанавливают исходный вид параметров, разобранных на части для функции-обработчика. Скажем, для WM_COMMAND соответствующий макрос имеет следующий вид:

```
#define FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, fn) \
  (void)(fn)((hwnd), WM_COMMAND, MAKEWPARAM((UINT)(id), (UINT)(codeNotify)),
(LPARAM)(HWND)(hwndCtl))
```

При необходимости передать сообщение на обработку по умолчанию, функция-обработчик должна вызвать макрос FORWARD_##message и указать адрес той функции, которую необходимо вызвать. Скажем, если

после обработки WM_COMMAND необходимо вызвать DefWindowProc(), то для этого в функцию-обработчик необходимо включить следующую строку:

```
FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, DefWindowProc);
```

Взломщики сообщений мы неоднократно будем использовать при написании демонстрационных программ, разбираемых в этой книге. Я очень надеюсь, что читатель, ознакомившись с работой этих макросов, станет их применять и при разработке собственных программ.

Основы работы с файлами в Win32 API

Одним из важнейших вопросов в любой операционной системе являются те возможности по работе с файлами, которые предлагает система. В настоящее время Windows использует несколько операционных систем. Первая – старая добрая FAT, которая используется со времени появления персональных компьютеров. Естественно, у этой операционной системы есть некоторые «возрастные» проблемы. Эти проблемы широко известны, поэтому я на них останавливаться не буду. То, что Windows'95 внесла в FAT некоторые изменения и позволила осуществлять поддержку длинных имён файлов, принципиально проблему не решило.

HPFS – high performance file system (высокопроизводительная файловая система) – была разработана для операционной системы OS/2 с целью устранения недостатков, присущих FAT, но и она не разрешила всех проблем, связанных с FAT. Тогда в Windows NT была предложена новая файловая система – NTFS. Для работы с компакт – дисками используется ещё одна файловая система – CDFS.

Но как бы то ни было, во всех операционных системах используются одни и те же методы доступа к файлам на диске и методы работы с файлами. Этим вопросам и посвящён этот раздел книги.

Получение информации о дисках, установленных в компьютере

Первая задача, которая обычно необходимо решить программисту, работающему с файловой системой, - это определение конфигурации системы, то есть определить, какие диски каких типов установлены на компьютере. Давайте попробуем выяснить, каким образом мы можем получить эту информацию.

Простейшей функцией, определяющей, какие логические диски присутствуют в системе, является функция GetLogicalDrives(), которая в файле winbase.h описана следующим образом:

```
WINBASEAPI DWORD WINAPI GetLogicalDrives(VOID);
```

Двойное слово, которое возвращает эта функция, фактически является логической шкалой, нулевой бит (самый младший) в которой соответствует диску A, первый бит – диску B, второй – диску C и так далее.

Эта функция достаточно проста и понятна, но, к сожалению, для того, чтобы извлечь из возращённого ею значения полезную информацию, приходится писать ещё одну функцию. Это, наверное, не совсем удобно, поэтому более удобна функция GetLogicalDriveStrings(), описание которой, находящееся в файле winbase.h я привожу ниже:

```
WINBASEAPI DWORD WINAPI GetLogicalDriveStringsA
                                    (DWORD nBufferLength, LPSTR lpBuffer);
WINBASEAPI DWORD WINAPI GetLogicalDriveStringsW
                                   (DWORD nBufferLength, LPWSTR lpBuffer);
#ifdef UNICODE
#define GetLogicalDriveStrings  GetLogicalDriveStringsW
#else
#define GetLogicalDriveStrings  GetLogicalDriveStringsA
#endif // !UNICODE
```

Первый аргумент этой функции – это длина буфера, который будет заполнен списком корневых директорий дисков, установленных на компьютере. Второй аргумент – указатель на этот буфер. При успешном завершении функция возвращает количество символов, скопированных в буфер. В том случае, если возвращаемое значение больше длины буфера, это является признаком того, что функции передан слишком маленький буфер, и необходимо увеличить его размер. Это наводит на мысль о том, что лучше сначала Если при выполнении функции произошла ошибка, то функция возвращает нулевое значение.

Но я ни слова не сказал о том, в каком виде будут представлены данные, записанные в буфер. Это просто несколько строк, завершённых нулями. В конце списка стоит ещё один нуль. Например, на моём компьютере список дисков выглядит следующим образом:

```
A:
C:
D:
E:
<null>
```

К сожалению, эта функция при работе в Windows'95 всегда возвращает нуль, то есть в Windows'95 она реализована только в виде заглушки.

Итак, мы уже получили имена дисков на нашем компьютере. Вполне вероятно, что у нас возникнет необходимость знать не только перечень дисков, но и типы дисков. Для этой цели существует функция GetDriveType(), описание которой, находящееся в файле winbase.h, приведено ниже:

```
WINBASEAPI UINT WINAPI GetDriveTypeA(LPCSTR lpRootPathName);
WINBASEAPI UINT WINAPI GetDriveTypeW(LPCWSTR lpRootPathName);
#ifdef UNICODE
#define GetDriveType  GetDriveTypeW
```

```
#else
#define GetDriveType  GetDriveTypeA
#endif // !UNICODE
```

В качестве единственного аргумента функции необходимо передать наименование корневой директории диска, тип которого мы хотим получить. Возвращаемое функцией значение определяет тип диска. Перечень возможных значений я привожу в таблице:

Идентификатор	Значение	Назначение
DRIVE_UNKNOWN	0	Тип устройства определить не удалось
DRIVE_NO_ROOT_DIR	1	Корневой директории не существует
DRIVE_REMOVABLE	2	Дисковод со сменным носителем – гибкий диск
DRIVE_FIXED	3	Дисковод с несменным носителем – жёсткий диск
DRIVE_REMOTE	4	Удалённый дисковод – сетевой диск
DRIVE_CDROM	5	Дисковод для компакт-дисков
DRIVE_RAMDISK	6	Эмулируемый в оперативной памяти диск – так называемый RAM-диск

Итак, мы уже знаем, как определить количество дисков в системе, как найти наименования корневых директорий этих дисков, как определить тип дисков. Вполне вероятно, что этой информации нам окажется недостаточно и что нам потребуется более подробная информация о дисках. Что ж, и на этот случай в Windows предусмотрена функция. Описание этой функции ,*которое расположено в файле winbase.h, я привожу ниже:*

```
WINBASEAPI BOOL WINAPI GetVolumeInformationA
                                    (LPCSTR lpRootPathName,
                                     LPSTR lpVolumeNameBuffer,
                                     DWORD nVolumeNameSize,
                                     LPDWORD lpVolumeSerialNumber,
                                     LPDWORD lpMaximumComponentLength,
                                     LPDWORD lpFileSystemFlags,
                                     LPSTR lpFileSystemNameBuffer,
                                     DWORD nFileSystemNameSize);
WINBASEAPI BOOL WINAPI GetVolumeInformationW
                                    ( LPCWSTR lpRootPathName,
                                     LPWSTR lpVolumeNameBuffer,
                                     DWORD nVolumeNameSize,
                                     LPDWORD lpVolumeSerialNumber,
                                     LPDWORD lpMaximumComponentLength,
                                     LPDWORD lpFileSystemFlags,
                                     LPWSTR lpFileSystemNameBuffer,
                                     DWORD nFileSystemNameSize);
#ifdef UNICODE
#define GetVolumeInformation  GetVolumeInformationW
#else
#define GetVolumeInformation  GetVolumeInformationA
```

```
#endif // !UNICODE
```

Первый аргумент этой функции, lpRootPathName, указывает на наименование корневой директории того диска, информацию о котором мы хотим получить. Если этот параметр равен NULL, используется корневая директория текущего диска.

Второй и третий аргументы функции работают в связке. Второй аргумент, lpVolumeNameBuffer, должен указывать на буфер, в который будет записано имя диска, а третий, nVolumeNameSize, определяет размер этого буфера в байтах.

В переменную, на которую указывает четвёртый аргумент, lpVolumeSerialNumber, функция записывает серийный номер диска. Если, однако, при вызове функции эта переменная будет равна NULL, то система определит, что информация о серийном номере вам не нужна и не станет возвращать её.

Очередной, пятый аргумент, lpMaximumComponentLength, указывает на двойное слово, в которое будет записана информация о том, какая максимальная дина имени файла вместе с путём, естественно, допускается в этой системе. Например, в FAT и NTFS это значение будет равно 255.

Шестой параметр, lpFileSystemFlags, указывает на двойное слово, в *которое будут записаны флаги, дающие дополнительную информацию о* файловой системе. Их перечень я приведу в таблице ниже.

Значение	Назначение
FS_CASE_IS_PRESERVED	Если этот флаг установлен, то при записи на диск сохраняется регистр букв в имени файла
FS_CASE_SENSITIVE	Если этот флаг установлен, то файловая система поддерживает поиск файлов с учётом регистра букв в именах
FS_UNICODE_STORED_ON_DISK	Если этот флаг установлен, то файловая система поддерживает хранение на диске имён файлов в Unicode.
FS_PERSISTENT_ACLS	Если этот флаг установлен, то файловая система способна оперировать со списками контроля доступа (ACL) (доступно только в NTFS).
FS_VOL_IS_COMPRESSED	Если этот флаг установлен, то том, информация о котором запрашивается, был подвергнут сжатию (например, DoubleSpace).
FS_FILE_COMPRESSION	Если этот флаг установлен, то файловая система поддерживает сжатие файлов.

Седьмой параметр, lpFileSystemNameBuffer, указывает на буфер, в который будет записано название файловой системы, такое как FAT, HPFS или NTFS. Если этот параметр при вызове функции равен NULL, то операционная система не будет возвращать имя файловой системы.

Последний, восьмой параметр, nFileSystemNameSize, определяет размер буфера, предназначенного для имени файловой системы. Если предыдущий аргумент равен NULL, то этот аргумент игнорируется.

Остался, как мне кажется, только один вопрос, на который мы пока не нашли ответ. Это вопрос о разбивке файла на сектора и кластеры. Для получения этой информации может быть использована функция GetDiskFreeSpace(), которая, кстати, что следует из названия функции, помимо информации о разбивке тома, возвращает и информацию о свободном пространстве на диске. Эта функция в файле winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI GetDiskFreeSpaceA(LPCSTR lpRootPathName,
                                         LPDWORD lpSectorsPerCluster,
                                         LPDWORD lpBytesPerSector,
                                         LPDWORD lpNumberOfFreeClusters,
                                         LPDWORD lpTotalNumberOfClusters);
WINBASEAPI BOOL WINAPI GetDiskFreeSpaceW(LPCWSTR lpRootPathName,
                                         LPDWORD lpSectorsPerCluster,
                                         LPDWORD lpBytesPerSector,
                                         LPDWORD lpNumberOfFreeClusters,
                                         LPDWORD lpTotalNumberOfClusters);
#ifdef UNICODE
#define GetDiskFreeSpace  GetDiskFreeSpaceW
#else
#define GetDiskFreeSpace  GetDiskFreeSpaceA
#endif // !UNICODE
```

Если внимательно посмотреть на аргументы этой функции, то сразу становится ясно, какие аргументы требуются для этой функции и какие данные она возвращает. Тем не менее, я опишу все аргументы.

Аргумент первый, lpRootPathName, указывает на наименование корневой директории того диска, информацию о котором мы хотим получить. Если этот параметр равено NULL, используется корневая директория текущего диска.

Второй аргумент, lpSectorsPerCluster, указывает на двойное слово, в которое система запишет количество секторов в одном кластере.

Аргумент третий, lpBytesPerSector, является указателем на двойное слово, в которое будет записано количество байтов в одном секторе.

lpNumberOfFreeClusters, четвёртый аргумент, содержит указатель на двойное слово, предназначенное для записи в него количества свободных кластеров на диске.

И, наконец, последний аргумент, lpTotalNumberOfClusters, указывает на двойное слово, в которое система запишет общее число кластеров на диске.

Если возвращённое функцией значение равно TRUE, то это означает, что функция завершила свою работу нормально. Если же функция вернула FALSE, то это является признаком того, что при выполнении функции возникла ошибка.

Кстати, имея теперь столь подробную информацию о томе, нам не составит никакого труда вычислить общий объём тома и объём свободного пространства на диске.

Ну, кажется, сейчас мы уже знаем, как нам найти любую информацию, которую мы только пожелаем узнать о диске. Если читателю интересно, как происходит работа с диском на более низком уровне, я рекомендую изучить работу функции DeviceIoControl(), которая предназначена для посылки команд и приёма информации непосредственно от драйвера диска. Она приведёт читателя к изучению команд устройства, кодов состояния и прочего, а потом... Как говорится, совершенству нет предела. ☺

И сейчас настало время для демонстрационной программы. Эта программа при помощи функций, которые мы изучили в этом разделе книги, выдаст нам информацию о тех дисках, которые установлены на компьютере. При запуске она определяет количество дисков, которые установлены на компьютере и включает ссылку на каждый диск в меню «File». После выбора какого-либо из элементов меню, соответствующих одному из дисков, на экране возникает окно сообщений с информацией о данном диске.

Для своей работы программа использует следующий файл ресурсов:

```
DisksMenu MENU
{
 POPUP "&File"
 {
  MENUITEM SEPARATOR
  MENUITEM "E&xit", 102
 }
}
```

А теперь непосредственно текст программы:

```
#include <windows.h>
#include <stdio.h>

#define IDM_EXIT 102

HINSTANCE hInst;

typedef struct _FLAGDISK
 {
   DWORD dwFlag;
   char* cDisk;
 } FLAGDISK;

   FLAGDISK FlagDisk[26] = { 0x1, "A:\\",
                             0x2, "B:\\",
                             0x4, "C:\\",
                             0x8, "D:\\",
                             0x10, "E:\\",
                             0x20, "F:\\",
                             0x40, "G:\\",
                             0x80, "H:\\",
```

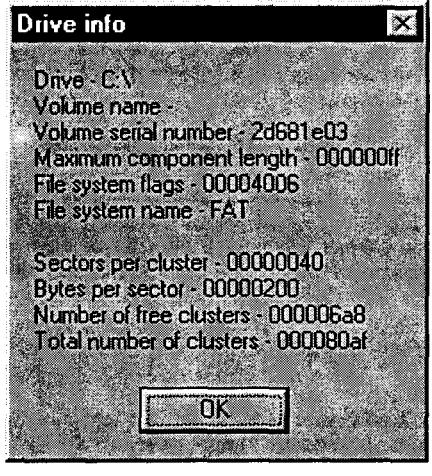
```
                    0x100, "I:\\",
                    0x200, "J:\\",
                    0x400, "K:\\",
                    0x800, "L:\\",
                    0x1000, "M:\\",
                    0x2000, "N:\\",
                    0x4000, "O:\\",
                    0x8000, "P:\\",
                    0x10000, "Q:\\",
                    0x20000, "R:\\",
                    0x40000, "S:\\",
                    0x80000, "T:\\",
                    0x100000, "U:\\",
                    0x200000, "V:\\",
                    0x400000, "W:\\",
                    0x800000, "X:\\",
                    0x1000000, "Y:\\",
                    0x2000000, "Z:\\" };

long WINAPI DisksWndProc( HWND, UINT, UINT, LONG );
void ViewDiskInfo(WORD);
int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow)
 {
   HWND hWnd;
   WNDCLASS WndClass;
   MSG Msg;

   hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = (WNDPROC) DisksWndProc;
   WndClass.cbClsExtra = 0;
   WndClass.cbWndExtra = 0;
   WndClass.hInstance =  hInstance ;
   WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
   WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
   WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
   WndClass.lpszMenuName = "DisksMenu";
   WndClass.lpszClassName = "Disks";

   if ( !RegisterClass(&WndClass) )
     {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
     }

hWnd = CreateWindow("Disks", "Viewer of disks",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
```

```
                CW_USEDEFAULT,
                NULL, NULL,
                hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

long WINAPI DisksWndProc ( HWND hWnd, UINT Message, UINT wParam,
                           LONG lParam )
 {

  DWORD dwDisks;
  int i, j;

  switch(Message)
   {
    case WM_CREATE:
     dwDisks = GetLogicalDrives();
// Теперь проверяем возвращённое функцией значение.
// В случае нахождения диска его букву добавляем в меню.
     j = 0;
     for(i = 0; i < 26; i++)
                                {
       if(dwDisks & FlagDisk[i].dwFlag)
         {
// Необходимо добавить все найденные диски в меню.
          InsertMenu(GetSubMenu(GetMenu(hWnd), 0), j,
                     MF_STRING | MF_BYPOSITION, IDM_EXIT + i + 1,
                     FlagDisk[i].cDisk);
          j++;
         }
      }
     return DefWindowProc(hWnd,Message,wParam, lParam);
    case WM_COMMAND:
     switch(LOWORD(wParam))
      {
       case IDM_EXIT + 1:
       case IDM_EXIT + 2:
```

```
        case IDM_EXIT + 3:
        case IDM_EXIT + 4:
        case IDM_EXIT + 5:
        case IDM_EXIT + 6:
        case IDM_EXIT + 7:
        case IDM_EXIT + 8:
        case IDM_EXIT + 9:
        case IDM_EXIT + 10:
        case IDM_EXIT + 11:
        case IDM_EXIT + 12:
        case IDM_EXIT + 13:
        case IDM_EXIT + 14:
        case IDM_EXIT + 15:
        case IDM_EXIT + 16:
        case IDM_EXIT + 17:
        case IDM_EXIT + 18:
        case IDM_EXIT + 19:
        case IDM_EXIT + 20:
        case IDM_EXIT + 21:
        case IDM_EXIT + 22:
        case IDM_EXIT + 23:
        case IDM_EXIT + 24:
        case IDM_EXIT + 25:
        case IDM_EXIT + 26:
          ViewDiskInfo(LOWORD(wParam) - IDM_EXIT - 1);
          return 0;
        case IDM_EXIT:
          SendMessage(hWnd, WM_DESTROY, 0, 0);
        default:
          return DefWindowProc(hWnd,Message,wParam, lParam);
      }
    case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
    default:
      return DefWindowProc(hWnd,Message,wParam, lParam);
  }
}

void ViewDiskInfo(WORD wId)
 {
  char cBuffer[0x400];

  char cVolumeNameBuffer[0x80];
  DWORD dwVolumeNameSize;
  DWORD dwVolumeSerialNumber;
  DWORD dwMaximumComponentLength;
  DWORD dwFileSystemFlags;
  char  cFileSystemNameBuffer[0x80];
  DWORD dwFileSystemNameSize;

  DWORD dwSectorsPerCluster;
  DWORD dwBytesPerSector;
```

```
  DWORD dwNumberOfFreeClusters;
  DWORD dwTotalNumberOfClusters;

 if(GetVolumeInformation(FlagDisk[wId].cDisk, cVolumeNameBuffer, 0x80,
                         &dwVolumeSerialNumber,
                         &dwMaximumComponentLength,
                        &dwFileSystemFlags, cFileSystemNameBuffer, 0x80))
   {
    GetDiskFreeSpace(FlagDisk[wId].cDisk, &dwSectorsPerCluster,
                     &dwBytesPerSector, &dwNumberOfFreeClusters,
                     &dwTotalNumberOfClusters);
    sprintf(cBuffer, "Drive - %03s\nVolume name - %s\n
          Volume serial number - %08x\n
          Maximum component length - %08x\nFile system flags - %08x\n
          File system name - %s\n\n
          Sectors per cluster - %08x\nBytes per sector - %08x\n
          Number of free clusters - %08x\nTotal number of clusters - %08x",
          FlagDisk[wId].cDisk, cVolumeNameBuffer, dwVolumeSerialNumber,
          dwMaximumComponentLength, dwFileSystemFlags,
          cFileSystemNameBuffer, dwSectorsPerCluster, dwBytesPerSector,
          dwNumberOfFreeClusters, dwTotalNumberOfClusters);
    MessageBox(NULL, cBuffer, "Drive info", MB_OK);
   }
  else
   MessageBox(NULL, "May be, there is not diskette or CD in this drive", "Error",
              MB_OK);
}
```

Окно сообщений, которое выдаёт информацию о диске C моего компьютера, имеет следующий вид:



Я надеюсь, читатель уже понял, что всё это не так уж и сложно. И теперь, разобравшись с файловой системой, получив информацию о дисках, установленных в компьютере, нам необходимо научиться манипули-

ровать файлами. Так как уважаемый читатель уже знает, как создавать файлы, то нам необходимо научиться создавать директории, копировать и перемещать файлы. Итак,

Работа с каталогами и манипулирование файлами

Определение и изменение текущей директории

Я надеюсь, уважаемый читатель уже имеет некоторый опыт программирования и знает, что каждый процесс запускается из так называемой текущей директории. Все операции с файлами также выполняются в текущей директории. Перво-наперво на

м необходимо научиться определять, какая директория является в настоящее время текущей. Для этого предназначена функция GetCurrentDirectory(). Её описание мы можем найти в файле winbase.h. Это описание я и привожу ниже:

```
WINBASEAPI DWORD WINAPI GetCurrentDirectoryA(DWORD nBufferLength,
                                             LPSTR lpBuffer);
WINBASEAPI DWORD WINAPI GetCurrentDirectoryW(DWORD nBufferLength,
                                             LPWSTR lpBuffer);
#ifdef UNICODE
#define GetCurrentDirectory  GetCurrentDirectoryW
#else
#define GetCurrentDirectory  GetCurrentDirectoryA
#endif // !UNICODE
```

После того, что мы уже изучили, я очень надеюсь, что читатель, бросив беглый взгляд на аргументы функции, сможет догадаться о назначении этих аргументов. Попробуйте, уважаемый читатель! Конечно же, второй аргумент, lpBuffer, – это указатель на буфер, в который будет записано наименование текущей директории. Первый аргумент, nBufferLength, - это размер этого буфера в байтах.

После успешного завершения функция возвращает число байтов, скопированных в буфер. Если произошла какая-то ошибка, то функция возвращает ноль. Но если возвращённое функцией значение больше того размера буфера, который был выделен, то это означает, что размер выделенного буфера недостаточен для того, чтобы в нём полностью поместилось наименование текущей директории. Какой можно найти выход из этой ситуации? Я здесь вижу два возможных выхода:

1. Вызвать функцию GetCurrentDirectory() с нулевым размером буфера, получить требуемое количество байтов, динамически выделить буфер требуемого размера (не забывайте о конечном нулевом байте, пожалуйста!), после чего повторно вызвать GetCurrentDirectory().

2. Перед вызовом функции GetCurrentDirectory() вызвать функцию GetVolumeInformation(), с её помощью определить значение максимальной длины компонента, после чего динамически выделить буфер такого размера, чтобы он заведомо вместил бы наименование текущей директории. Здесь у читателя может возникнуть вопрос – а зачем нам это делать, если максимальный размер пути нам известен – 256 символов? Вопрос справедлив. Я бы ответил на него так. Да, нам известен максимальный размер пути в той операционной системе, в которой мы сейчас работаем (я надеюсь, читатель работает сейчас в одной из трёх операционных систем – Windows'95, Windows'98 или Windows NT). Но что произойдёт, если, скажем, та операционная система, которая придёт на смену упомянутым выше, будет поддерживать максимальный размер пути, равный 512 байтам? Я думаю, что если у программиста нет каких-либо ограничений по скорости выполнения программы или по размеру выделенной памяти, то один вызов лишней функции, как говорится, погоды не сделает.

Итак, теперь мы знаем, как определять текущую директорию. Но как поступать в том случае, когда нам необходимо сменить текущую директорию? Для этого мы должны воспользоваться функцией SetCurrentDirectory(). Эта функция описана в заголовочном файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI SetCurrentDirectoryA(LPCSTR lpPathName);
WINBASEAPI BOOL WINAPI SetCurrentDirectoryW(LPCWSTR lpPathName);
#ifdef UNICODE
#define SetCurrentDirectory  SetCurrentDirectoryW
#else
#define SetCurrentDirectory  SetCurrentDirectoryA
#endif // !UNICODE
```

В качестве единственного аргумента этой функции используется указатель на строку, в которой записано наименование директории, которую необходимо сделать текущей. Если функция завершает свою работу нормально, то возвращаемое ею значение равно TRUE. После какой-либо ошибки функция возвращает FALSE.

Определение системной и основной директорий Windows

Иногда в программе необходимо узнать, какая директория является т.н. системной, то есть в каком каталоге хранятся основные файлы, обеспечивающие работоспособность системы. К таким файлам мы можем отнести основные динамические библиотеки, драйверы ,шрифты и прочее. Для того, чтобы получить наименование этой директории мы можем воспользоваться функцией GetSystemDirectory(). Описание её, приведённое ниже, можно найти в файле winbase.h:

```
WINBASEAPI UINT WINAPI GetSystemDirectoryA(LPSTR lpBuffer,
```

```
                                                    UINT uSize);
WINBASEAPI UINT WINAPI GetSystemDirectoryW(LPWSTR lpBuffer,
                                                    UINT uSize);
#ifdef UNICODE
#define GetSystemDirectory  GetSystemDirectoryW
#else
#define GetSystemDirectory  GetSystemDirectoryA
#endif // !UNICODE
```

Сколько раз мы уже встречались с такой связкой аргументов! Первый аргумент этой функции, lpBuffer, является указателем на буфер, в который будет записано наименование системной директории. А второй аргумент, uSize, это всего-навсего размер этого буфера. При успешном завершении функция заполняет буфер и возвращает число байтов, записанных в буфер. В случае ошибки система возвращает нуль. Я надеюсь, читатель может догадаться, что будет означать такая ситуация, при которой возвращённое функцией значение будет больше указанного при вызове функции размера буфера.

Аналогичным образом можно узнать и путь основной директории Windows. Это путь мы можем получить при вызове функции GetWindowsDirectory(). Она описана в заголовочном файле winbase.h следующим образом:

```
WINBASEAPI UINT WINAPI GetWindowsDirectoryA(LPSTR lpBuffer,
                                                    UINT uSize);
WINBASEAPI UINT WINAPI GetWindowsDirectoryW(LPWSTR lpBuffer,
                                                    UINT uSize);
#ifdef UNICODE
#define GetWindowsDirectory  GetWindowsDirectoryW
#else
#define GetWindowsDirectory  GetWindowsDirectoryA
#endif // !UNICODE
```

Назначение аргументов этой функции практически совпадает с аргументами функции GetSystemDirectory(), описанной чуть ранее, поэтому я не стану останавливаться на их описании.

А теперь – небольшая демонстрационная программа, которая выдаст нам данные о текущей директории, а также об основной директории Windows и системной директории Windows.

Программа использует следующий файл ресурсов:

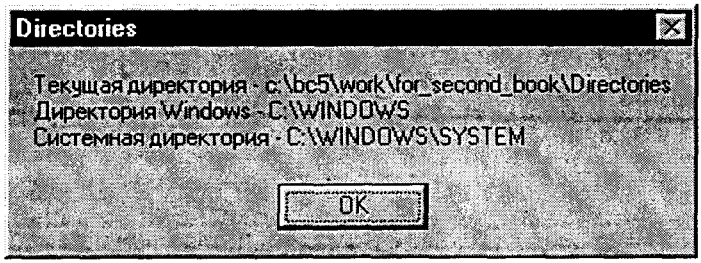
```
DirectoriesMenu MENU
{
 POPUP "&File"
 {
  MENUITEM "&Info", 101
  MENUITEM SEPARATOR
  MENUITEM "E&xit", 102
 }
}
```

А теперь основной файл программы:

```
#include <windows.h>
#include <stdio.h>

#define IDM_INFO 101
#define IDM_EXIT 102

HINSTANCE hInst;

long WINAPI DirectoriesWndProc(HWND, UINT, UINT, LONG);

int APIENTRY WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpszCmdParam, int nCmdShow )
 {
   HWND hWnd ;
   WNDCLASS WndClass ;
   MSG Msg;

   hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = (WNDPROC) DirectoriesWndProc;
   WndClass.cbClsExtra = 0;
   WndClass.cbWndExtra = 0;
   WndClass.hInstance =  hInstance ;
   WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
   WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
   WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
   WndClass.lpszMenuName = "DirectoriesMenu";
   WndClass.lpszClassName = "Directories";

   if ( !RegisterClass(&WndClass) )
     {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
     }

   hWnd = CreateWindow("Directories", "Directories demo",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT,
                       CW_USEDEFAULT,
                       CW_USEDEFAULT,
                       CW_USEDEFAULT,
                       NULL, NULL,
                       hInstance,NULL);
   if(!hWnd)
     {
       MessageBox(NULL,"Cannot create window","Error",MB_OK);
       return 0;
     }
```

```
/* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
   return Msg.wParam;
 }

long WINAPI DirectoriesWndProc ( HWND hWnd, UINT Message, UINT wParam,
     LONG lParam )
 {

   char cBuffer[0x400];
   char cBufferForCurrentDirectory[0x400];
   char cBufferForWindowsDirectory[0x400];
   char cBufferForSystemDirectory[0x400];

   switch(Message)
    {
     case WM_COMMAND:
      switch(LOWORD(wParam))
       {
        case IDM_INFO:
         GetCurrentDirectory(0x400, cBufferForCurrentDirectory);
         GetWindowsDirectory(cBufferForWindowsDirectory, 0x400);
         GetSystemDirectory(cBufferForSystemDirectory, 0x400);
         sprintf(cBuffer, "Текущая директория - %s\n
                           Директория Windows - %s\n
                           Системная директория - %s",
                           cBufferForCurrentDirectory,
                           cBufferForWindowsDirectory,
                           cBufferForSystemDirectory);
         MessageBox(hWnd, cBuffer, "Directories", MB_OK);
         return 1;
        case IDM_EXIT:
         SendMessage(hWnd, WM_DESTROY, 0, 0);
         default:
        return DefWindowProc(hWnd,Message,wParam, lParam);
       }
     case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
     default:
      return DefWindowProc(hWnd,Message,wParam, lParam);
    }
 }
```

Внешний вид окна с информацией о директориях, которое выдаёт программа, приведен ниже:



Создание и удаление каталогов

А как нам поступить в том случае, если нам необходимо завести собственный каталог? Нет ничего проще! Нам необходимо просто-напросто обратиться к функции CreateDirectory(), описанной в файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI CreateDirectoryA(LPCSTR lpPathName,
                                        LPSECURITY_ATTRIBUTES lpSecurityAttributes);
WINBASEAPI BOOL WINAPI CreateDirectoryW(LPCWSTR lpPathName,
                                        LPSECURITY_ATTRIBUTES lpSecurityAttributes);
#ifdef UNICODE
#define CreateDirectory  CreateDirectoryW
#else
#define CreateDirectory  CreateDirectoryA
#endif // !UNICODE
```

Первый аргумент этой функции – это указатель на буфер, в котором записано наименование той директории, которую мы пытаемся создать. Для того, чтобы назначить директории особые привилегии, мы можем соответствующим образом заполнить структуру типа SECURITY_ATTRIBUTES.

При нормальном завершении функция создаёт директорию и возвращает значение TRUE. В случае возникновения какой-либо ошибки функция возвращает FALSE.

Для того чтобы удалить директорию, нам необходимо воспользоваться функцией RemoveDirectory(). Эта функция в файле winbase.h описывается так:

```
WINBASEAPI BOOL WINAPI RemoveDirectoryA(LPCSTR lpPathName);
WINBASEAPI BOOL WINAPI RemoveDirectoryW(LPCWSTR lpPathName);
#ifdef UNICODE
#define RemoveDirectory  RemoveDirectoryW
#else
#define RemoveDirectory  RemoveDirectoryA
#endif // !UNICODE
```

Как и предыдущая функция, эта функция также возвращает TRUE при удачном завершении и FALSE в случае возникновения какой-либо ошибки. Я хотел бы обратить внимание на то, что во многих случаях ошибка не является ошибкой. Например, как воспринимать значение FALSE в том случае, если, скажем, я не имею прав на удаление той или иной директории? Для того, чтобы правильно реагировать на сообщения об ошибке (а чем является возвращённое значение FALSE, как не сообщением об ошибке?), необходимо почаще обращаться к функции GetLastError(). Эта функция, описанная в файле winbase.h следующим образом

```
WINBASEAPI DWORD WINAPI GetLastError(VOID);
```

Выдаст код встретившейся ошибки. Коды ошибок можно найти в заголовочном файле error.h. В этом великолепно комментированном файле приведены коды ошибок и их обозначения. Прочитав обозначение, можно даже без комментариев догадаться о том, какая ошибка встретилась при выполнении функции.

Копирование файлов

Если у программиста возникла необходимость скопировать тот или иной файл, то для выполнения этого действия проще всего воспользоваться функцией CopyFile(), описанной в файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI CopyFileA(LPCSTR lpExistingFileName,
                                 LPCSTR lpNewFileName,
                                 BOOL bFailIfExists);
WINBASEAPI BOOL WINAPI CopyFileW(LPCWSTR lpExistingFileName,
                                 LPCWSTR lpNewFileName,
                                 BOOL bFailIfExists);
#ifdef UNICODE
#define CopyFile  CopyFileW
#else
#define CopyFile  CopyFileA
#endif // !UNICODE
```

Первый аргумент этой функции, lpExistingFileName, указывает на буфер, в котором записано имя файла, подлежащего копированию. Второй аргумент, lpNewFileName, указывает на буфер, в котором хранится имя нового файла. Работа функции в какой-то степени зависит от третьего аргумента, bFailIfExists. Этот аргумент определяет, что должна делать функция в том случае, если файл с именем, указанным во втором аргументе, уже существует. Если третий аргумент функции равен TRUE и файл уже существует, то функция вернёт значение FALSE, которое фактически является сообщением об ошибке. Если третий аргумент равен FALSE и файл уже существует, то функция удалит старый файл и создаст новый. В обычной ситуации возвращённое функцией значение TRUE говорит о том, что работа функции завершена успешно, а значение FALSE говорит о том, что

при работе функции встретилась какая-то ошибка, код которой можно получить при помощи функции GetLastError().

Удаление файлов

Удаление файла выполняется при помощи вызова функции DeleteFile(). Найдя описание этой функции в файле winbase.h, я привожу его ниже:

```
WINBASEAPI BOOL WINAPI DeleteFileA(LPCSTR lpFileName);
WINBASEAPI BOOL WINAPI DeleteFileW(LPCWSTR lpFileName);
#ifdef UNICODE
#define DeleteFile  DeleteFileW
#else
#define DeleteFile  DeleteFileA
#endif // !UNICODE
```

Как читатель может догадаться, единственный аргумент этой функции, lpFileName, должен являться указателем на буфер, в котором хранится имя удаляемого файла. Как и у функций, описанных ранее, возвращаемое значение TRUE говорит о том, что работа функции завершена нормально, а значение FALSE является индикатором встретившейся при выполнении функции ошибки. Я бы хотел предупредить читателя об одной особенности работы этой функции под управлением опреационной системы Windows'95. Дело в том, что вызов функции DeleteFile() в этом случае может удалить и открытый файл, и файл отображённый в память, что может привести к безвозвратной потере данных. Чтобы предотвратить потерю данных, перед удалением файла необходимо его закрыть.

Перемещение файлов

Как ни странно, но многие мои студенты и ученики не сразу понимали, какая разница между копированием и перемещением файлов. Я в таком случае объяснял следующим образом: после операции копирования появляются две копии файла. Точнее, оригинал и копия. После перемещения файла на диске так и остаётся одна-единственная копия файла. Грубо говоря, сначала файл копируется, а потом оригинал уничтожается. Только потом я объяснял, что файл реально перемещается только тогда, когда он перемещается с одного диска на другой. Иначе просто изменяется ссылка на файл в таблице размещения файлов. Но, как бы то ни было, давайте разберёмся, как осуществляется перемещение файлов.

Для того, чтобы переместить файл, можно воспользоваться двумя функциями. Первая из них, MoveFile(), в заголовочном файле описана так:

```
WINBASEAPI BOOL WINAPI MoveFileA(LPCSTR lpExistingFileName,
                                 LPCSTR lpNewFileName);
WINBASEAPI BOOL WINAPI MoveFileW(LPCWSTR lpExistingFileName,
                                 LPCWSTR lpNewFileName);
#ifdef UNICODE
#define MoveFile  MoveFileW
#else
```

```
#define MoveFile  MoveFileA
#endif // !UNICODE
```

Что можно сказать об аргументах этой функции? Первый аргумент, lpExistingFileName, должен указывать на буфер, в котором записано имя перемещаемого файла. Второй аргумент, lpNewFileName, должен указывать на другой буфер, в котором записано новое имя файла, естественно, с путём к нему. В случае успешного выполнения функция возвращает TRUE. Если функция вернула нам FALSE, то необходимо разобраться, что вызвало ошибку при выполнении функции.

Но если функция MoveFile() перемещает файлы, то зачем нужна функция MoveFileEx()? Кстати, эта функция в файле заголовков winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI MoveFileExA(LPCSTR lpExistingFileName,
                                   LPCSTR lpNewFileName,
                                   DWORD dwFlags);
WINBASEAPI BOOL WINAPI MoveFileExW(LPCWSTR lpExistingFileName,
                                   LPCWSTR lpNewFileName,
                                   DWORD dwFlags);
#ifdef UNICODE
#define MoveFileEx  MoveFileExW
#else
#define MoveFileEx  MoveFileExA
#endif // !UNICODE
```

Бросив беглый взгляд на аргументы этой функции, мы можем заметить, что первые два аргумента функционально подобны аргументам функции MoveFile(), поэтому я на этих аргументах сейчас останавливаться не буду. Вопрос вызывает только третий аргумент, dwFlags. Возможно, это флаги, определяющие поведение функции в различных ситуациях? Естественно, так оно и есть. Список этих флагов, а также их значение и назначение я приведу в таблице ниже.

Флаг	Значение	Назначение
MOVEFILE_REPLACE_EXISTING	0x00000001	Если файл с именем, определяемым lpNewFileName, уже существует, то он замещается.
MOVEFILE_COPY_ALLOWED	0x00000002	Если файл перемещается на другой том, то для этого используются функции CopyFile() и DeleteFile().
MOVEFILE_DELAY_UNTIL_REBOOT	0x00000004	Файл реально не перемещается до перезагрузки системы. Обычно используется в установочных программах. Работает

		только в Windows NT.

И ещё один нюанс при использовании функции MoveFileEx(). А что произойдёт, если мы в качестве второго аргумента укажем NULL? То есть решим переместить файл в никуда? То и произойдёт, файл переместится в никуда, в небытие, то есть уничтожится, то есть произойдёт УДАЛЕНИЕ, а не перемещение файла.

Переименование файлов

А теперь я бы хотел, чтобы читатель задумался над таким вопросом – а чем отличается операция по переименованию файла от операции по перемещению файла? Могу ли я сказать, что в тот момент, когда я перемещаю файл, я переименовываю его? Конечно, да! Даже в том случае, если у файла осталось старое имя, у него изменяется путь! Другими словами, ОПЕРАЦИИ ПЕРЕИМЕНОВАНИЯ И ПЕРЕМЕЩЕНИЯ СУТЬ ОДНО И ТО ЖЕ! Именно поэтому в Win32 API нет функции типа RenameFile(). Для того, чтобы переименовать файл, необходимо просто-напросто вызвать функцию MoveFile() или MoveFileEx() и в качестве второго аргумента передать ей указатель на буфер с новым именем файла. Тем самым проблема будет решена.

Демонстрационная программа

А теперь наступила очередь демонстрационной программы. Вряд ли *эта программа покажется кому-то полезной, но я преследую только цели* демонстрации. В этой программе мы сделаем следующее:

1. Создадим на диске C две директории с именами «Demo_Directory_For_Book_1» и «Demo_Directory_For_Book_2».
2. В каждой из этих директорий создадим файлы с именами «File_1.dem».
3. Постараемся скопировать файл из первой директории во вторую и убедимся, что произошла ошибка, так как файл уже существует.
4. Переименуем файл в первой директории в «File_2.dem».
5. Скопируем файл из первой директории во вторую.
6. *Удалим файлы и директории.*

Итак, текст демонстрационной программы.

```
#include <windows.h>
#include <stdio.h>

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpszCmdParam, int nCmdShow )
 {
  char cBuffer[0x400];
  char cBufferForCurrentDirectory[0x400];
  HANDLE hFile;

// Сохраняем текущую директорию.
```

```
   GetCurrentDirectory(0x400, cBufferForCurrentDirectory);
   strcpy(cBuffer, "c:\\Demo_Directory_For_Book_1");
// Создаём первую директорию.
   if(CreateDirectory(cBuffer, NULL))
     MessageBox(NULL, "Директория c:\\Demo_Directory_For_Book_1
                 создана!", "Info", MB_OK);
    else
     {
      MessageBox(NULL, "Не могу создать директорию
                 c:\\Demo_Directory_For_Book_1", "Error", MB_OK);
      return 0;
     }
// Делаем только что созданную директорию текущей.
   SetCurrentDirectory(cBuffer);
// Создаём в текущей директории файл с именем "File_1.dem".
   if(INVALID_HANDLE_VALUE == (hFile =
                 CreateFile("File_1.dem", GENERIC_READ | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL, CREATE_NEW,
                       FILE_ATTRIBUTE_NORMAL, 0)) )
     {
      MessageBox(NULL, "Не могу создать файл в директории
                 c:\\Demo_Directory_For_Book_1", "Error", MB_OK);
      return 0;
     }
    else
// Закрываем файл, иначе будем получать сообщения
// об ошибке доступа к файлу.
     CloseHandle(hFile);
   strcpy(cBuffer, "c:\\Demo_Directory_For_Book_2");
// Создаём вторую директорию.
   if(CreateDirectory(cBuffer, NULL))
     MessageBox(NULL, "Директория c:\\Demo_Directory_For_Book_2
                 создана!", "Info", MB_OK);
    else
     {
      MessageBox(NULL, "Не могу создать директорию
                 c:\\Demo_Directory_For_Book_2", "Error", MB_OK);
      return 0;
     }
// Делаем только что созданную директорию текущей.
   SetCurrentDirectory(cBuffer);
// Создаём в текущей директории файл с именем "File_1.dem".
   if(INVALID_HANDLE_VALUE == (hFile =
                 CreateFile("File_1.dem", GENERIC_READ | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL, CREATE_NEW,
                       FILE_ATTRIBUTE_NORMAL, 0)) )
     {
      MessageBox(NULL, "Не могу создать файл в директории
                 c:\\Demo_Directory_For_Book_2", "Error", MB_OK);
      return 0;
     }
```

```
    else
// Закрываем файл, иначе будем получать сообщения об ошибке доступа к файлу.
      CloseHandle(hFile);
// Делаем попытку скопировать файл. Файл с таким именем уже существует,
// поэтому получаем сообщение об ошибке.
   if(!CopyFile("c:\\Demo_Directory_For_Book_1\\File_1.dem",
                "c:\\Demo_Directory_For_Book_1\\File_1.dem", TRUE))
      MessageBox(NULL, "Не могу скопировать файл!", "Error", MB_OK);
// Переименовываем файл.
   if(!MoveFile("c:\\Demo_Directory_For_Book_1\\File_1.dem",
                "c:\\Demo_Directory_For_Book_1\\File_2.dem"))
      {
       MessageBox(NULL, "Не могу переименовать файл", "Error", MB_OK);
       return 0;
      }
// Делаем вторую попытку скопировать файл.
// Файла с таким именем уже не существует,
// поэтому копирование происходит успешно.
   if(!CopyFile("c:\\Demo_Directory_For_Book_1\\File_2.dem",
                "c:\\Demo_Directory_For_Book_2\\File_2.dem", TRUE))
      {
       MessageBox(NULL, "Не могу скопировать файл!", "Error", MB_OK);
       return 0;
      }
     else
      MessageBox(NULL, "Файл скопирован успешно", "Info", MB_OK);
// Делаем попытку удалить директорию, в которой ещё есть файлы,
// поэтому получаем сообщение об ошибке.
   if(!RemoveDirectory("c:\\Demo_Directory_For_Book_1"))
      MessageBox(NULL, "Не могу удалить директорию.\n
                        Возможно, в ней ещё есть файлы", "Error", MB_OK);
// Удаляем файлы из второй директории.
   DeleteFile("c:\\Demo_Directory_For_Book_2\\File_1.dem");
   DeleteFile("c:\\Demo_Directory_For_Book_2\\File_2.dem");
// Удаляем файл из первой директории.
   DeleteFile("c:\\Demo_Directory_For_Book_1\\File_2.dem");
// Удаляем первую директорию.
   RemoveDirectory("c:\\Demo_Directory_For_Book_1");
// Делаем попытку удалить текущую директорию.
   if(!RemoveDirectory("c:\\Demo_Directory_For_Book_2"))
      MessageBox(NULL, "Не могу удалить директорию.\n
                        Возможно, она является текущей", "Error", MB_OK);
// Делаем текущей ту директорию, которая была текущей
// перед началом работы программы.
   SetCurrentDirectory(cBufferForCurrentDirectory);
// Удаляем вторую директорию.
   RemoveDirectory("c:\\Demo_Directory_For_Book_2");
   return 1;
 }
```

Я настоятельно рекомендую читателю как можно подробнее ознакомиться с работой данной программы, после чего немного "поиграть" с ней. Мне бы хотелось, чтобы читатель сделал бы следующие вещи:

1. Постарался бы попробовать поработать с незакрытыми файлами, особенно в Windows'95/98.
2. Попробовал бы удалить текущую директорию.

Я уверен, что подобные упражнения помогут лучше «почувствовать» операционную систему и способы и методы её работы с файлами.

Что ж, уважаемый читатель, мы изучили методы работы с файлами, начиная от создания файла и кончая удалением файла. Но ведь файлы создаются не только для того, чтобы их создать, скопировать, переместить или переименовать, и удалить! (В тот момент, когда я писал предыдущее предложение, я вспомнил знаменитое «...жители города рождаются лишь затем, чтобы побриться, остричься, освежить голову вежеталем и сразу же умереть» из «12-ти стульев» Ильфа и Петрова ☺). Действительно, файлы создаются для того, чтобы хранить в них какие-то данные, будь то текст программы или бухгалтерская база данных! А в таком случае нам необходимо не просто манипулировать файлами, но и производить запись данных в файл и, естественно, чтение данных из файла.

Кроме того, я хотел бы обратить внимание читателя на следующий факт. Диск является очень медленным устройством. Дело в том, что для того, чтобы осуществить доступ к данным, нам, в общем случае, необходимо включить двигатель, который обеспечивает вращение винчестера, дождаться пока он наберёт необходимую скорость вращения, позиционировать считывающие головки... Естественно, эти операции происходят намного медленнее, чем работает программа. Поэтому в следующих разделах мы постараемся узнать, не только то, как осуществлять доступ к данным файла, но и постараемся разобраться, как мы можем «распараллелить» работу программы, то есть, как сделать так, чтобы ввод и вывод данных на диск не замедлял работу программы.

Запись информации в файл и чтение информации из файла.

Перед тем, как начинать большой разговор о записи информации в файл и чтении информации из файла, я бы хотел поговорить о том, что такое

Открытие файла

Итак, нам необходимо открыть файл. Для этого нам необходимо воспользоваться функцией CreateFile(), которая в файле winbase.h описана следующим образом:

```
WINBASEAPI HANDLE WINAPI CreateFileA(LPCSTR lpFileName,
                DWORD dwDesiredAccess, DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
```

```
WINBASEAPI HANDLE WINAPI CreateFileW(LPCWSTR lpFileName,
                DWORD dwDesiredAccess,
                DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
#ifdef UNICODE
#define CreateFile  CreateFileW
#else
#define CreateFile  CreateFileA
#endif // !UNICODE
```

Рассмотрим параметры этой функции. Итак, первый параметр lpFileName представляет собой указатель на имя файла (естественно, вместе с путём), который программист хочет создать или открыть. Ничего особенного в этом параметре нет.

Второй параметр dwDesiredAccess определяет, какие действия мы можем произвести с содержимым файла, то есть определяет права на запись и на чтение данных. При определении прав мы можем указать следующие значения:

Флаг	Значение	Назначение
0		Содержимое файла нельзя считывать, в файл нельзя производить запись. Флаг используется только тогда, когда необходимо всего лишь получить информацию о файле.
GENERIC_READ	0x80000000L	Разрешается чтение из файла
GENERIC_WRITE	0x40000000L	Разрешена запись в файл
GENERIC_READ | GENERIC_WRITE	0xC000000L	Разрешено чтение из файла и запись в файл.

Третий параметр, dwShareMode, определяет права по совместному доступу разных процессов к файлу. В качестве этого параметра могут быть указаны следующие значения:

Флаг	Значение	Назначение
0		Исключительный доступ к файлу, сторонние пользователи открыть файл не могут
FILE_SHARE_READ	0x00000001	Разрешёно чтение содержимого файла сторонними пользователями
FILE_SHARE_WRITE	0x00000002	Разрешена запись в файл сторонними пользователями
FILE_SHARE_READ | FILE_SHARE_WRITE	0x00000003	Разрешены чтение из файла и запись в файл сторонними пользователями

Четвёртый параметр, lpSecurityAttributes, заслуживает особого разговора о нём. представляет собой указатель на структуру типа SECURITY_ATTRIBUTES, которая в файле winbase.h описана так:

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES,
  *LPSECURITY_ATTRIBUTES;
```

Если никакой особой защиты файлу не требуется, то в это поле можно занести NULL. Тем не менее, я вкратце расскажу о полях структуры.

Поле nLength должно содержать длину структуры типа SECURITY_ATTRIBUTES. Другими словами, в большинстве программ, которым приходится заполнять эту структуру, должен присутствовать оператор типа

```
lpSecurity_Attributes->nLength = sizeof(SECURITY_ATTRIBUTES);
```

Когда мне встречается структура, в одно из полей которых записывается длина этой структуры, я беру, как говорится, это на карандаш, ибо это – один из признаков того, что в будущих версиях системы эта структура претерпит некоторые изменения.

Второе поле содержит указатель на структуру типа SECURITY_DESCRIPTOR, которая в файле winnt.h описана следующим образом:

```
typedef struct _SECURITY_DESCRIPTOR {
  BYTE  Revision;
  BYTE  Sbz1;
  SECURITY_DESCRIPTOR_CONTROL Control;
  PSID Owner;
  PSID Group;
  PACL Sacl;
  PACL Dacl;
  } SECURITY_DESCRIPTOR, *PISECURITY_DESCRIPTOR;
```

Здесь я не хотел бы обсуждать все поля этой структуры. Мы их обсудим в главе, посвящённой безопасности при работе с операционной системой.

И, наконец, третье поле, bInheritHandle, говорит нам о том, должен ли возвращаемый хэндл наследоваться при создании нового процесса. Если это поле установлено в TRUE, то хэндл должен наследоваться.

Теперь опять вернёмся к параметрам функции CreateFile(). Мы рассмотрели четыре параметра. Пятый – dwCreationDisposition – определяет, какие действия необходимо произвести в тех случаях, когда файл уже существует и когда файл ещё не существует. Это поле может принимать одно из следующих значений:

Параметр	Значение	Назначение
CREATE_NEW	1	Создаётся новый файл. Если файл с указанным именем уже существует, функция завершает

		работу с ошибкой.
CREATE_ALWAYS	2	Создаётся новый файл. Если файл с указанным именем уже существует, то он переписывается.
OPEN_EXISTING	3	Открывается существующий файл. Если файла с указанным именем не существует, то функция завершает работу с ошибкой.
OPEN_ALWAYS	4	Если файл с указанным именем существует, то он открывается. Если файла с указанным именем не существует, то он создаётся.
TRUNCATE_EXISTING	5	Открывается существующий файл, причём сразу после открытия он усекается до нулевой длины. Если файл с указанным именем не существует, то функция завершает работу с ошибкой.

Очередной параметр, dwFlagsAndAttributes, указывает атрибуты и флаги для файла. В данном случае это поле является логической шкалой, то есть флаги можно комбинировать друг с другом. Возможные значения этого поля я привожу в таблице ниже:

Флаг или атрибут	Значение	Назначение
FILE_ATTRIBUTE_READONLY	0x00000001	Файл только для чтения. Приложения могут открывать этот файл, читать из него, но не могут производить в него запись.
FILE_ATTRIBUTE_HIDDEN	0x00000002	Скрытый файл, то есть в обычных условиях (если не указано обратное) в списке файлов не появляется.
FILE_ATTRIBUTE_SYSTEM	0x00000004	Файл является частью операционной системы или используется только операционной системой.
FILE_ATTRIBUTE_DIRECTORY	0x00000010	
FILE_ATTRIBUTE_ARCHIVE	0x00000020	Файл является архивным файлом. Используется для того, чтобы пометить файл либо как подлежащиц резервному копированию, либо как подлежащий удалению.
FILE_ATTRIBUTE_NORMAL	0x00000080	Этот атрибут не может использоваться в сочетании с другими атрибутами. Он используется только в тех случаях, когда у файла нет других атрибутов.

FILE_ATTRIBUTE_TEMPORARY	0x00000100	Создаётся временный файл, система пытается не записывать его на диск, а держать в памяти, что ускоряет доступ к файлу.
FILE_ATTRIBUTE_COMPRESSED	0x00000800	
FILE_FLAG_POSIX_SEMANTICS	0x01000000	При доступе к файлу используются правила POSIX (в частности, в этом случае могут употребляться файлы, имена которых различаются только регистром букв).
FILE_FLAG_BACKUP_SEMANTICS	0x02000000	Контролировать наличие у процесса прав доступа и, если таковые есть, открывать файл только для резервного копирования или восстановления.
FILE_FLAG_DELETE_ON_CLOSE	0x04000000	Файл после закрытия удаляется. Обычно этот флаг используется для временных файлов.
FILE_FLAG_SEQUENTIAL_SCAN	0x08000000	Режим кэширования файла оптимизируется для последовательного доступа к файлу.
FILE_FLAG_RANDOM_ACCESS	0x10000000	Режим кэширования файла оптимизируется для произвольного доступа к файлу.
FILE_FLAG_NO_BUFFERING	0x20000000	Система не буферизирует файловый ввод-вывод и не осуществляет опережающее чтение и кэширование данного файла.
FILE_FLAG_OVERLAPPED	0x40000000	Устанавливается асинхронный доступ к файлу (в Windows'95/98 не реализован).
FILE_FLAG_WRITE_THROUGH	0x80000000	Отключение промежуточного кэширования для снижения вероятности потери данных.

Я хотел бы обратить внимание читателя на то, что атрибуты файла используют младшие биты двойного слова, а флаги – старшие. Именно поэтому их и можно комбинировать, так как они никоим образом не могут повлиять друг на друга.

И последний аргумент функции CreateFile() – hTemplateFile – указывает либо хэндл уже открытого файла, либо равен NULL. В первом случае значение параметра dwFlagsAndAttributes игнорируется и функция использует флаги и атрибуты файла, на который указывает hTemplateFile. Для того, чтобы такая схема могла бы сработать, файл, на который указывает hTemplateFile, должен быть открыт заранее с флагом GENERIC_READ.

Если функция отработала нормально, то она возвращает хэндл открытого файла. А если неправильно... Почти все функции, которые возвращают хэндлы, в случае неудачного завершения возвращают значение NULL. Функция CreateFile() отличается от всех – в случае неуспешного завершения она возвращает значение INVALID_HANDLE_VALUE, которое в файле winbase.h определено следующим образом:

```
#define INVALID_HANDLE_VALUE (HANDLE)-1
```

то есть фактически равно 0xffffffff.

Открыв файл, нам в конце работы программы нужно обязательно закрыть его. Рассмотрим поэтому

Закрытие Файла

Для того, чтобы закрыть файл, нам необходимо вызвать функцию CloseHandle(), которая описана в файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI CloseHandle(HANDLE hObject);
```

Единственным аргументом этой функции является хэндл того файла, который мы должны закрыть. Всё легко и просто! Уважаемый читатель, я хотел бы, чтобы Вы сравнили две функции – CreateFile() и CloseHandle()! Как долго мы разбирали первую и как быстро мы изучили вторую!

Как я уже сказал выше, файловые ввод и вывод – это очень медленные операции. В зависимости от того, как нам необходимо осуществлять ввод-вывод, мы можем использовать либо синхронный, либо асинхронный режим чтения и записи файлов. При синхронном режиме мы выдаём команду на осуществление ввода–вывода и до тех пор пока операция ввода – вывода не закончится, программа не выполняет никаких действий. Естественно, это применимо только в тех случаях, когда мы имеем дело с очень небольшими блоками данных. Если блок данных достаточно велик, мы могли бы поступить по-другому. Мы могли бы дать системе указание прочитать или записать файл, а программа тем временем спокойно бы занималась другими делами. Естественно, нам пришлось бы в данном случае писать многопоточную программу и использовать средства синхронизации потоков, но, наверное, овчинка стоит выделки, а?

Сначала мы рассмотрим

Синхронный режим чтения и записи файлов

Я надеюсь, что читателю уже приходилось осуществлять операции ввода – вывода в DOS или Windows 3.x. Принципы синхронного ввода – вывода с приходом Windws'95/98 и Windows NT не изменились. Поэтому то, о чём я буду говорить в этом разделе, уже может быть известно читателю. Если это так на самом деле, уважаемый читатель может сразу перейти к изучению раздела об асинхронном вводе выводе.

Итак, как же выглядит процесс выборки данных при синхронном вводе-выводе?

1. Программист выделяет в памяти буфер определённого размера.
2. Программист открывает файл, к которому он намерен обращаться.
3. Программист устанавливает указатель файла на то место в файле ,где находятся интересующие его данные.
4. Данные из файла считываются в буфер.
5. В буфере производятся определённые действия.
6. Буфер записывается на то же место в файле или добавляется в конец файла.

Естественно, что в реальной ситуации те или иные шаги могут быть опущены. Но для того, чтобы наш рассказ был полным, нам необходимо рассмотреть все возможные шаги.

О выделении в памяти буфера я говорить не буду. Это предмет для отдельного разговора и я не хочу сейчас останавливаться на этом. Открытие файла мы рассмотрели чуть раньше, когда говорили об операциях с файлами. Значит, сейчас нам необходимо начать с вопроса о том, как осуществляется

Позиционирование указателя файла

Для того чтобы избежать какой-либо путаницы, я хотел бы сказать, что с каждым файлом связана некая внутренняя системная переменная, которая указывает на то место в файле, с которого будут начинаться операции ввода – вывода. Эта переменная, доступ к которой напрямую программист получить не может, называется указателем файла. При открытии файла этот указтель обычно устанавливается перед самым первым символом файла. Для того, чтобы манипулировать этой переменной, существуют специальные функции. Одной из них является функция SetFilePointer(), описание которой, взятое из файла winbase.h, я привожу ниже:

```
WINBASEAPI DWORD WINAPI SetFilePointer(HANDLE hFile,
                                       LONG lDistanceToMove,
                                       PLONG lpDistanceToMoveHigh,
                                       DWORD dwMoveMethod);
```

Первый аргумент этой функции, hFile, очевиден. Это хэндл того файла, с которым мы сейчас работаем и в котором перемещаем указатель. Второй аргумент, lDistanceToMove, является числом символов, на которое необходимо передвинуть указатель файла. Для установки нового значения указателя файла это число просто-напросто складывается со старым значением указателя файла, поэтому мы можем определять, в какую сторону будет смещён указатель. Если аргумент lDistanceToMove больше нуля, то указатель файла перемещается вперёд, то есть к концу файла. Отрицательное значение говорит о том, что указатель файла перемещается назад, то есть к началу файла. Я хотел бы, чтобы читатель обратил внимание на следующий факт. Тип этого аргумента – LONG, то есть используя только этот аргумент, мы не можем переместить указатель файла больше,

чем на 2**32 байта. А что нам делать, если мы работаем с очень большим файлом? В этом случае нам на помощь приходит третий аргумент, через который мы можем передать указатель на буфер, содержащий старшие тридцать два разряда требуемого значения. Подводя итог сказанному выше, мы приходим к следующему выводу. При определении нового значения указателя файла используется следующая формула:

НЗУ = СЗУ + lDistanceToMove + *lpDistanceToMoveHigh,

Где: НЗУ – это Новое Значение Указателя;
СЗУ – это Старое Значение Указателя.

Я думаю, что читатель достаточно проницателен и у него возник вопрос – а зачем нам передавать УКАЗАТЕЛЬ на старшие тридцать два разряда? Почему мы не можем передать это значение напрямую? Ответ на этот вопрос очень прост. Я, заглядывая несколько вперёд, хотел бы, чтобы читатель обратил внимание на тип возвращаемого функцией значения – двойное слово. В этом двойном слове функция возвращает новое значение указателя файла. В том случае, когда тридцати двух разрядов для этого мало, старшие тридцать два разряда записываются по адресу lpDistanceToMoveHigh.

Но вернёмся к аргументам нашей функции. Последний аргумент, dwMoveMethod, определяет точку отсчёта для перемещения указателя файла. Возможные значения этого аргумента я приведу в таблице.

Метод	Значение	Назначение
FILE_BEGIN	0	Старое значение указателя игнорируется и считается равным нулю, то есть СЗУ в данном случае равно нулю.
FILE_CURRENT	1	Для вычислений используется текущее значение указателя.
FILE_END	2	Старое значение указателя принимается равным числу байтов в файле, то есть фактически указатель устанавливается в конец файла

Осталось только довести до конца разговор о том, какое значение и в каких случаях возвращает эта функция. О том, что в случае успешного завершения функция возвращает новое значение указателя, я уже говорил. Но возможны ещё два случая.

Во-первых, функция может вернуть значение 0xffffffff, при этом слово, на которое указывает третий аргумент, может быть нулевым. Это будет означать, что при работе функции произошла ошибка.

Во-вторых, функция может вернуть значение 0xffffffff, при этом слово, на которое указывает третий аргумент, может быть не равно нулю. В этом случае для того, чтобы определить, произошла ли ошибка, необходимо вызвать функцию GetLastError(). Если эта функция вернёт значение, отличное от NO_ERROR, это будет являться сообщением об ошибке.

Установка конца файла

Обычно при закрытии файла система сама устанавливает конец файла и программисту не приходится об этом заботиться. Но иногда возникает необходимость увеличить или уменьшить размер файла. Для этих целей в Windows существует функция SetEndOfFile(), которая в файле winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI SetEndOfFile(HANDLE hFile);
```

Единственным аргументом этого файла является хэндл файла, с которым мы работаем. Конец файла устанавливается в том месте, на котором находится указатель файла. Я хотел бы напомнить читателю о том, что после того, как будет установлен конец файла, необходимо закрыть файл. Другими словами, если мы хотим изменить длину файла и сделать её равной, скажем, 65535, нам необходимо выполнить следующие действия:

```
HFILE hFile = CreateFile(...);
SetFilePointer(hFile, 65535, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);
```

Теперь нам необходимо остановиться на той функции, которая позволит нам произвести

Чтение данных из файла

в буфер. Эта функция называется ReadFile() и в файле winbase.h она описана следующим образом:

```
WINBASEAPI BOOL WINAPI ReadFile(HANDLE hFile, LPVOID lpBuffer,
                                DWORD nNumberOfBytesToRead,
                                LPDWORD lpNumberOfBytesRead,
                                LPOVERLAPPED lpOverlapped);
```

Перед тем, как начать описывать аргументы этой функции, я хотел бы сказать, что для того, чтобы функция могла бы выполниться успешно, файл, из которого мы собираемся производить чтение, должен быть открыт с флагом GENERIC_READ.

Первый аргумент этой функции, hFile, является хэндлом того файла, из которого мы производим чтение данных. Второй аргумент, lpBuffer, указывает на буфер, в который мы будем производить чтение данных. Третий аргумент, nNumberOfBytesToRead, определяет число байтов, которые мы хотим прочесть из файла. В буфер, на который указывает lpNumberOfBytesRead, будет записано число реально прочитанных байтов. Последний аргумент, lpOverlapped, является указателем на структуру типа OVERLAPPED, но так как этот аргумент используется только для асинхронного ввода-вывода, мы его рассмотрим тогда ,когда будем изучать асинхронный ввод-вывод. При синхронном вводе нам необходимо передать функции через этот аргумент значение NULL.

Я думаю, читатель уже заметил, что аргументы функции не определяют, из какого места файла нам необходимо производить чтение информации. Дело в том, что чтение производится с того места, на которое указывает указатель файла. Причём, к примеру, если файл только что открыт и мы производим чтение одного килобайта данных из начала файла, то после операции чтения указатель файла будет установлен на 1025-й байт. Если мы не сместим указатель, то после второй операции чтения он будет установлен на 2049-й байт, и так далее. Я надеюсь, читатель понял принцип перемещения указателя файла во время исполнения операций чтения.

В том случае, если функция завершена нормально, она возвращает значение TRUE.

Для того, чтобы осуществить

Запись данных в файл

предназначена функция WriteFile(). Её описание может быть найдено в файле заголовков winbase.h. Я привожу его ниже:

```
WINBASEAPI BOOL WINAPI WriteFile(HANDLE hFile, LPCVOID lpBuffer,
                                 DWORD nNumberOfBytesToWrite,
                                 LPDWORD lpNumberOfBytesWritten,
                                 LPOVERLAPPED lpOverlapped);
```

Я хочу обратить внимание уважаемого читателя на то, что для того ,чтобы мы были бы в состоянии произвести запись данных в файл, файл должен быть открыт с флагом GENERIC_WRITE. А теперь поговорим об аргументах этой функции.

Значение первого аргумента, hFile, очевидно. Это хэндл файла, в который мы собираемся произвести запись. Второй аргумент, lpBuffer, - это указатель на буфер, данные из которого мы хотим записать в файл. Число байтов, которые мы хотим записать в файл, должно быть указано в третьем аргументе, nNumberOfBytesToWrite. После того, как функция выполнит свою работу, в буфер, на который указывает четвёртый аргумент, lpNumberOfBytesWritten, будет записано число байтов, записанных в файл. Последний аргумент, lpOverlapped, который указывает на структуру типа OVERLAPPED, которая используется только в случае асинхронного ввода – вывода, поэтому сейчас я на ней останавливаться не буду.

Как и в случае чтения файла, запись данных начинается с того места, на которое указывает указатель файла. После каждой операции чтения указатель файла смещается на величину, равную тому количеству байтов, которые были записаны в файл.

Если функция отработала нормально, то она возвращает значение TRUE. Значение FALSE возвращается в том случае, если при работе функции произошла ошибка.

Блокировка и разблокировка файла

Но здесь я хотел бы остановиться на одной проблеме, связанной с возможностью одновременного доступа к файлу нескольких процессов.

Представим себе такую ситуацию. С базой данных большого предприятия работает одновременно несколько человек. Если они имеют право только читать данные из файла, никаких проблем не возникает. Но, естественно, если двум или более сотрудникам потребуется одновременно произвести запись в одно и то же место файла? Целостность данных будет нарушена. Может быть, открывать файл только с флагом FILE_SHARE_READ? Но тогда никто, кроме открывшего файл, не сможет произвести запись в файл. Простейший выход, который можно найти – это блокировка файлов. Другими словами, если нам необходим файл для монопольного использования в течение какого-то времени, мы можем заблокировать файл. Если флаги FILE_SHARE_* влияют на весь файл в целом, то блокировать можно только какую-то часть файла. Перед тем, как производить запись в файл, нам необходимо убедиться в том, что никто, кроме нас, не сможет в то же время произвести запись в то же место файла. В этом нам поможет функция LockFile(). Описание этой функции, взятое из файла заголовков winbase.h, я привожу ниже:

```
WINBASEAPI BOOL WINAPI LockFile(HANDLE hFile,
                                DWORD dwFileOffsetLow,
                                DWORD dwFileOffsetHigh,
                                DWORD nNumberOfBytesToLockLow,
                                DWORD nNumberOfBytesToLockHigh);
```

На первом аргументе, hFile, останавливаться не стоит – это хэндл блокируемого файла. Второй и третий аргументы, dwFileOffsetLow и dwFileOffsetHigh соответственно, содержат младшие тридцать два разряда и старшие тридцать два разряда смещения той части файла которую необходимо блокировать. Четвёртый и пятый аргументы, nNumberOfBytesToLockLow и nNumberOfBytesToLockHigh, содержат младшую и старшую части числа, определяющего размер блокируемой области в байтах.

При успешном завершении работы функция возвращает TRUE. Если функция вернула FALSE, нам необходимо разобраться, в чём причина ошибки.

Если мы заблокировали какой-то участок файла ,то никто, кроме нас, не сможет осуществить обращение к этой области файла, то есть опасности разрушения данных из-за одновременного доступа к файлу нет.

А как нам поступить в том случае, если мы собираемся добавлять данные в конец файла? ТОЧНО ТАК ЖЕ! Нам необходимо блокировать участок файла ЗА КОНЦОМ ФАЙЛА, после чего спокойно производить туда запись.

При помощи функции LockFileEx() мы можем более гибко управлять блокировкой файла. Эта функция в файле winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI LockFileEx(HANDLE hFile,
                                  DWORD dwFlags,
                                  DWORD dwReserved,
```

```
                                    DWORD nNumberOfBytesToLockLow,
                                    DWORD nNumberOfBytesToLockHigh,
                                    LPOVERLAPPED lpOverlapped);
```

Что принципиально нового предлагает нам эта функция по сравнению с функцией LockFile()? Обратим внимание на второй аргумент, dwFlags. Их возможные значения и назначение я привожу в таблице ниже:

Флаг	Значение	Назначение
LOCKFILE_FAIL_IMMEDIATELY	0x00000001	Если заблокировать область не удаётся, функция НЕМЕДЛЕННО возвращает управление. Если флаг не установлен, функция ждёт, пока область будет разблокирована, чтобы получить к ней доступ.
LOCKFILE_EXCLUSIVE_LOCK	0x00000002	Если флаг установлен, то другие пользователи не могут ни читать, ни записывать в файл. В противном случае другим пользователям разрешено чтение из файла.

Со вторым флагом, LOCKFILE_EXCLUSIVE_LOCK, кажется, всё ясно. Несколько слов я скажу о первом флаге, LOCKFILE_FAIL_IMMEDIATELY. Дело в том, что если нам нужно заблокировать файл, а он уже заблокирован другим пользователем, то функция LockFile() сразу же вернёт нам управление. Нам придётся постоянно вызывать функцию LockFile() до тех пор, пока мы не сможем заблокировать требуемую область файла. Задачу ожидания освобождения области можно возложить на функцию LockFileEx(), не устанавливая флага LOCKFILE_FAIL_IMMEDIATELY.

Третий аргумент зарезервирован для дальнейшего использования и должен быть установлен в нуль. Четвёртый и пятый аргументы определяют размер блокируемой области точно таким же образом, как это делается в функции LockFile(). И возникает вопрос – а как же нам указать смещение в файле, начиная с которого мы хотим блокировать область? Для этого существует шестой аргумент, lpOverlapped. Он указывает на структуру типа OVERLAPPED, которая в файле winbase.h описана так:

```
typedef struct _OVERLAPPED {
   DWORD  Internal;
   DWORD  InternalHigh;
   DWORD  Offset;
   DWORD  OffsetHigh;
   HANDLE  hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

В целом об этой структуре мы поговорим позже, а сейчас я скажу, что функция LockFileEx() использует только поля Offset и OffsetHigh. В них до вызова функции должны быть записаны соответственно младшие и старшие тридцать два разряда смещения области, которую необходимо блокировать.

Так же, как и функция LockFile(), функция LockFileEx() возвращает значение TRUE в случае успешного завершения и FALSE, если встретилась какая-то ошибка.

Раз мы получили временный эксклюзивный доступ к области файла, то, естественно, после окончания работы с этой областью нам необходимо её разблокировать. Это делается при помощи функции UnlockFile(). Эта функция описана в файле winbase.h, я привожу это описание ниже:

```
WINBASEAPI BOOL WINAPI UnlockFile(HANDLE hFile,
                                  DWORD dwFileOffsetLow,
                                  DWORD dwFileOffsetHigh,
                                  DWORD nNumberOfBytesToUnlockLow,
                                  DWORD nNumberOfBytesToUnlockHigh);
```

Назначение аргументов этой функции совпадает с назначением аргументов функции LockFile(), поэтому я не стану останавливаться на них. Единственное, на что я хотел бы обратить внимание уважаемого читателя, это то, что разблокировать нужно ИМЕННО ТУ область файла, которая была ранее заблокирована.

В недрах Win32 API существует и функция UnlockFileEx(). Она описана в файле winbase.h, я привожу её описание ниже:

```
WINBASEAPI BOOL WINAPI UnlockFileEx(HANDLE hFile,
                                    DWORD dwReserved,
                                    DWORD nNumberOfBytesToUnlockLow,
                                    DWORD nNumberOfBytesToUnlockHigh,
                                    LPOVERLAPPED lpOverlapped);
```

Аргументы этой функции подобны аргументам функции LockFileEx(), поэтому, надеюсь, у читателя не возникнет вопросов о назначении этих аргументов.

Честно говоря, я не нашёл в ней никаких отличий от функции UnlockFile(), кроме размещения аргументов. Может, конечно, в будущем фирма Microsoft добавит этой функции новые возможности...

Что ж, уважаемый читатель, мы почти полностью рассмотрели возможности синхронного ввода и вывода в Windows. Я надеюсь, что теперь у вас не возникнет ни малейшего затруднения в тех случаях, когда потребуется произвести запись в файл или чтение данных из файла.

Но я должен предупредить читателя об одной не очень приятной возможности. Дело в том, что не все функции, имеющиеся в Win32 API, реализованы в Windows'95/98. Многие функции реализованы как заглушки. Поэтому я настоятельно рекомендую уважаемому читателю следующее. Если вы работаете в Windows'95/98 и если при вызове функции она возвращает вам FALSE, хотя, кажется, всё в порядке, попробуйте вызвать

функцию GetLastError(). Если возвращённый код ошибки равен 120 (ERROR_CALL_NOT_IMPLEMENTED), то считайте, что вам не повезло, эта функция как раз и реализована как заглушка. Придётся изменять код программы и искать те функции, которые работают в Windows'95/98.

Демонстрационная программа

```
#include <windows.h>

int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow )
 {
   char cBuffer[0x400];
   char cDirectoryBuffer[0x400];
   char cTextBuffer[] = {"Синьора Спагетти, рождённая в Пизе,\n\
                         Любила висеть босиком на карнизе."};
   HANDLE hFile;
   DWORD dwBytes;
   int i;

// Определяем текущую директорию и запоминаем её
// в массиве cDirectoryBuffer.
   GetCurrentDirectory(0x400, cDirectoryBuffer);
// Создаём директорию, в которой будем производить все манипуляции.
   CreateDirectory("c:\\Demo_Directory_For_Book", NULL);
// Делаем вновь созданную директорию текущей.
   SetCurrentDirectory("c:\\Demo_Directory_For_Book");
// Создаём файл, в который мы сбросим данные.
   if(INVALID_HANDLE_VALUE == (hFile = CreateFile("TestFile.tst",
                                        GENERIC_WRITE, FILE_SHARE_READ,
                                        NULL, CREATE_NEW,
                                        FILE_ATTRIBUTE_NORMAL, NULL)) )
     {
       MessageBox(NULL, "Can't create file", "Error", MB_OK);
       return 0;
     }
// Записываем данные в файл.
   WriteFile(hFile, cTextBuffer, sizeof(cTextBuffer), &dwBytes, NULL);
// Закрываем файл.
   CloseHandle(hFile);
// Очищаем буфер для чистоты эксперимента.
   for(i = 0; i < sizeof(cTextBuffer); *(cTextBuffer + i) = 0, i++);
// Выдаём содержимое буфера на отображения.
   MessageBox(NULL, cTextBuffer, "Clear buffer", MB_OK);
// Открываем файл для чтения.
   if(INVALID_HANDLE_VALUE == (hFile = CreateFile("TestFile.tst",
                                        GENERIC_READ, FILE_SHARE_READ,
                                        NULL, OPEN_EXISTING,
                                        FILE_ATTRIBUTE_NORMAL, NULL)) )
     {
       MessageBox(NULL, "Can't open file", "Error", MB_OK);
       return 0;
     }
```

```
// Читаем из файла.
   ReadFile(hFile, cTextBuffer, sizeof(cTextBuffer), &dwBytes, NULL);
// Выдаём содержимое буфера на отображение.
   MessageBox(NULL, cTextBuffer, "Fulled buffer", MB_OK);
// Закрываем файл.
   CloseHandle(hFile);
// Удаляем созданный файл.
   DeleteFile("TestFile.tst");
// Делаем текущей ту же директорию, которая была текущей
// до начала работы программы.
   SetCurrentDirectory(cDirectoryBuffer);
// Удаляем созданную директорию.
   RemoveDirectory("c:\\Demo_Directory_For_Book");
   return 1;
 }
```

Асинхронный режим чтения и записи файлов.

Перед тем, как начать описание асинхронного режима чтения и записи файлов, я хочу обратить внимание читателя на то, что асинхронный ввод – вывод в Windows'95/98 не работает, если, конечно, речь не идёт о работе с последовательным портом. Поэтому если уважаемый читатель не предполагает работать с последовательным портом и пишет программу, предназначенную для работы только в Windows'95/98, он может пропустить эту главу без какого-либо ущерба для понимания содержания дальнейших разделов.

Я уже говорил о том, что такое асинхронный ввод – вывод и не хочу повторяться. Поэтому давайте, уважаемый читатель, возьмём, как говорится, быка за рога. Для того, чтобы осуществить асинхронный доступ к файлу, мы должны открыть файл при помощи функции CreateFile() и указать в аргументе dwFlagsAndAttributes флаг FILE_FLAG_OVERLAPPED, который укажет, что файл будет открыт именно для асинхронного доступа к данным.

Кроме этого, в данном случае, в отличие от операций, производимых в синхронном режиме, мы должны использовать структуру типа OVERLAPPED. И сейчас, кажется, настало время рассказать об этой структуре поподробнее. Я уже говорил, что эта структура описана в файле winbase.h следующим образом:

```
typedef struct _OVERLAPPED {
   DWORD  Internal;
   DWORD  InternalHigh;
   DWORD  Offset;
   DWORD  OffsetHigh;
   HANDLE hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

Давайте теперь подробнее поговорим о полях этой структуры. Первое поле, Internal, зарезервировано для внутреннего использования системой. В некоторых случаях оно показывает состояние системы. Второе по-

ле, InternalHigh, также зарезервировано для использования системой, оно показывает число прочитанных или записанных в файл байтов.

Третье и четвёртое поля работают вместе. В третье поле, Offset, записываются младшие тридцать два разряда смещения в файле, начиная с которого необходимо производить чтение или запись в файл. Четвёртое поле, OffsetHigh, содержит старшие тридцать два разряда этого слова

И, наконец, нужно особо остановиться на пятом поле, h Event. Здесь я постараюсь объяснить его назначение поподробнее. Я надеюсь, читатель уже имеет определённый опыт написания многопоточных программ, то есть мне не нужно объяснять, что такое многопоточность и как происходит синхронизация потоков. Если читатель всё же хочет освежить свои знания о многопоточных программах, я могу порекомендовать ему обратиться к моей книге «Азбука программирования для Win32 API" (2-е изд., М., «Радио и связь», 1999).

Вполне очевидно, что при асинхронных операциях ввода – вывода программа, начавшая операцию чтения или записи, должна определить факт окончания записи и свои дальнейшие действия производить в зависимости от результата операции. Другими словами, ПРОГРАММА ДОЛЖНА ИМЕТЬ ВОЗМОЖНОСТЬ СИНХРОНИЗАЦИИ СВОЕЙ РАБОТЫ С РАБОТОЙ ОПЕРАЦИИ ВВОДА _ ВЫВОДА. Так вот, hEvent – это хэндл синхронизирующего объекта «событие», который до начала операции ввода – вывода должен быть создан при помощи обраения к функции CreateEvent(). Проще говоря, hEvent – это тот самый объект, который даст нам знать об окончании операции ввода – вывода. Как только операция ввода – вывода закончится, система устанавливает событие, хэндлом которого является hEvent. Как только он стал ненулевым, система переводит его в незанятое состояние, а затем тут же делает незанятым и хэндл файла. Я надеюсь, принцип работы понятен даже после этого краткого описания. Я бы хотел обратить внимание читателя на один момент, который вытекает из сказанного выше. При асинхронной операции мы можем дожидаться двух событий. Событие первое – это окончание операции ввода – вывода. Индикатором наступления этого события является перевод в незанятое состояние объекта «событие», хэндл которого равен hEvent. Событие второе – освобождение хэндла файла. В зависимости от того, какие цели преследует программа, она может реагировать на каждое из этих событий.

В случае использования асинхронного ввода – вывода можно запрограммировать выполнение нескольких операций ввода – вывода. Для этого нам придётся создать объект типа «событие», получить хэндл на это событие, после чего вызвать функцию ReadFile() или WriteFile(). После того, как нам потребуется синхронизировать дальнейшую работу программы с результатами ввода – вывода, нам необходимо обратиться к функции WaitForSingleObject(), передав ей в описатель созданного нами события. В этом случае мы легко и просто можем выполнить сразу несколько операций ввода – вывода.

Для того, чтобы ещё больше функций возложить на операционную систему и освободить программу от заботы о результатах ввода – вдля синхронизации нашей программы мы можем использовать функцию GetO-

verlappedResult(), которая в заголовочном файле winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI GetOverlappedResult(HANDLE hFile,
                                           LPOVERLAPPED lpOverlapped,
                                           LPDWORD lpNumberOfBytesTransferred,
                                           BOOL bWait);
```

Аргументы этой функции достаточно просты. Первый аргумент, hFile, - это хэндл файла, к которому мы обращаемся. Второй аргумент, lpOverlapped, - это указатель на структуру типа OVERLAPPED, на эту же структуру должен указывать и соответствующий аргумент при вызове функций ReadFile() или WriteFile(). Третий аргумент, lpNumberOfBytesTransferred, указывает на двойное слово, в которое будет записано число прочитанных или записанных байтов.

И, наконец, последний аргумент, bWait, который нас и интересует более всего. Этот аргумент определяет, должна ли эта функция дожидаться результатов операции ввода – вывода. Если мы присвоим этому аргументу значение TRUE, то функция будет дожидаться результата выполнения операции ввода – вывода, то есть самостоятельно вызовет WaitForSingleObject() и передаст ей хэндл события. Если же этот аргумент равен FALSE, то функция не будет дожидаться результатов операции ввода – вывода, вернёт значение FALSE и результатом выполнения функции GetLastError() в этом случае будет значение ERROR_IO_INCOMPLETE. Здесь я хотел бы отметить один факт. Этот код ошибки описан в заголовочном файле winerror.h. Для того, чтобы обратить внимание уважаемого читателя на этот файл, я приведу описания кода ошибки так, как оно дано в этом файле:

```
//
// MessageId: ERROR_IO_INCOMPLETE
//
// MessageText:
//
//  Overlapped I/O event is not in a signalled state.
//
#define ERROR_IO_INCOMPLETE              996L
```

Скажет ли кто-нибудь после этого, что заголовочные файлы не нужно изучать? ☺

Чтение информации из файла в асинхронном режиме

Две функции, ReadFile() и WriteFile(), мы рассмотрели тогда, когда изучали синхронный режим ввода – вывода. Но мы не рассматривали функции ReadFileEx() и WriteFileEx(), которые предназначены специально для асинхронного ввода – вывода.

Первая из этих функций, ReadFileEx(), описана в заголовочном файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI ReadFileEx(HANDLE hFile,
        LPVOID lpBuffer,
        DWORD nNumberOfBytesToRead,
        LPOVERLAPPED lpOverlapped,
        LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

Первые четыре аргумента этой функции нам уже известны. Это хэндл файла, с которым мы работаем, указатель на буфер, в который мы будем читать данные, число байтов, которые необходимо считать и указатель на структуру типа OVERLAPPED соответственно. Но пятый аргумент нам неизвестен. Этот аргумент является указателем на так называемую функцию завершения. Прототип этой функции мы можем найти опять-таки в файле winbase.h:

```
typedef VOID (WINAPI *LPOVERLAPPED_COMPLETION_ROUTINE)
                                        (DWORD dwErrorCode,
                                         DWORD dwNumberOfBytesTransfered,
                                         LPOVERLAPPED lpOverlapped);
```

Для чего нужна эта функция? Она используется для того, чтобы произвести какие-либо действия после завершения операции ввода – вывода. Здесь реализован очень тонкий и сложный механизм.

Представим себе, что мы запускаем несколько операций ввода вывода. Система ставит наши запросы в очередь, периодически эту очередь просматривает и выполняет требуемые действия. Когда операции ввода-вывода заканчиваются, система создаёт список завершившихся событий и ставит его в соответствие тому потоку, который запустил операции ввода – вывода, но не вызывает функцию завершения. Для того, чтобы функция завершения была бы вызвана, нам необходимо «разбудить» наш поток при помощи одной из «тревожных» функций - SleepEx(), WaitForSingleObject() или WaitForMultipleObjects(). Каким образом происходит «пробуждение» потока? Система просматривает список завершённых операций ввода-вывода, находит завершённую операцию, при помои «тревожной» функции «пробуждает» поток, и дальнейшая работа потока продолжается с функции завершения. После отработки функции завершения завершённая операция ввода – вывода удаляется из списка и система опять ждёт завершения какой-либо операции ввода – вывода. После того, как все операции ввода – вывода завершатся, продолжается нормальная работа программы. Честно говоря, мне кажется, что здесь всё продумано удивительно точно!

Запись информации в файл в асинхронном режиме

Для того, чтобы завершить наш разговор, посвящённый асинхронному режиму ввода – вывода, я приведу описание функции WriteFileEx(), которое может быть найдено в файле winbase.h:

```
WINBASEAPI BOOL WINAPI WriteFileEx(HANDLE hFile,
```

```
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPOVERLAPPED lpOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

Я надеюсь, что после нашего обсуждения функции ReadFileEx(), аргументы функции WriteFileEx() полностью ясны для читателя, поэтому останавливаться на них надобности нет. И, наконец, демонстрационная программа, в которой мы будем осуществлять копирование файла с винчестера на дискету.

Уважаемый читатель! Мы уже рассмотрели вопросы получения информации о дисках, установленных в системе, научились манипулировать файлами, рассмотрели вопросы записи информации в файл и чтения информации из файла. Теперь нам, наверное, надо было бы рассмотреть вопрос о том, какую же информацию непосредственно о файлах мы можем получить. Например, нам могут потребоваться данные об атрибутах файла, его размере, времени создания... Да мало ли какая информация может нам потребоваться! Поэтому следующая тема нашего разговора –

Характеристики файлов

Наверное, самыми основными характеристиками файла являются его так называемые атрибуты, которые присваиваются файлу при его создании. Но каким образом мы можем узнать атрибуты того файла, который был создан не нами? Для этого в Win32 API включена функция GetFileAttributes(). Эта функция описана в файле заголовков winbase.h следующим образом:

```
WINBASEAPI DWORD WINAPI GetFileAttributesA(LPCSTR lpFileName);
WINBASEAPI DWORD WINAPI GetFileAttributesW(LPCWSTR lpFileName);
#ifdef UNICODE
#define GetFileAttributes  GetFileAttributesW
#else
#define GetFileAttributes  GetFileAttributesA
#endif // !UNICODE
```

Единственным аргументом этой функции является имя файла, атрибуты которого мы можем получить. Возвращает функция двойное слово, биты которого соответствуют атрибутам файла. Об атрибутах файла мы уже говорили при рассмотрении функции CreateFile(). Нужно ли рассматривать их здесь ещё раз?

Иногда возникает необходимость изменить какие-либо атрибуты файла. Для этой цели существует другая функция, SetFileAttributes(). Она также описана в заголовочном файле winbase.h:

```
WINBASEAPI BOOL WINAPI SetFileAttributesA(LPCSTR lpFileName,
                                          DWORD dwFileAttributes);
WINBASEAPI BOOL WINAPI SetFileAttributesW(LPCWSTR lpFileName,
                                          DWORD dwFileAttributes);
```

```
#ifdef UNICODE
#define SetFileAttributes  SetFileAttributesW
#else
#define SetFileAttributes  SetFileAttributesA
#endif // !UNICODE
```

Первый аргумент этой функции – имя файла, атрибуты которого мы хотим установить. Второй аргумент – это двойное слово, показывающее, какие атрибуты файла должны быть установлены. В случае успешного завершения функция возвращает значение TRUE. Возврат значение FALSE говорит о том, что по каким-то причинам установить атрибуты файла не удалось. Для получения причины ошибки в этом случае необходимо обратиться к функции GetLastError().

Если мы хотим узнать размер файла, то мы можем обратиться к функции GetFileSize(), описанной в файле winbase.h следующим образом:

```
WINBASEAPI DWORD WINAPI GetFileSize(HANDLE hFile,
                                    LPDWORD lpFileSizeHigh);
```

Первый аргумент этой функции, hFile, содержит хэндл файла, размер которого мы хотим получить. Отсюда мы можем сделать очень важный вывод, что при помощи этой функции мы можем получить ТОЛЬКО РАЗМЕР ОТКРЫТОГО ФАЙЛА, то есть для получения размера файла нам необходимо сначала его открыть только после этого передавать его хэндл функции. Возвращаемое функцией значение содержит младшие тридцать два разряда размера файла. Если нам нужны старшие тридцать два разряда размера файла, то функция поместит их в двойное слово, указатель на которое передан функции в качесве второго аргумента. Если старшие тридцать два разряда размера файла нам не нужны, то в качестве второго аргумента мы можем указать NULL.

Начиная с Windows'95, с файлом связаны три метки времени. Первая метка – это время создания файла, вторая – время последнего обращения к файлу, и третья – это время последней записи в файл. Для того, чтобы получить значения этих меток, нам необходимо обратиться к функции GetFileTime(), описание которой, взятое из файла winbase.h, я привожу ниже:

```
WINBASEAPI BOOL WINAPI GetFileTime(HANDLE hFile,
                                   LPFILETIME lpCreationTime,
                                   LPFILETIME lpLastAccessTime,
                                   LPFILETIME lpLastWriteTime);
```

Я думаю, читатель уже обратил внимание на то, что в качестве первого аргумента функции должен быть передан хэндл файла, то есть и временные метки мф можем получить только для открытого файла. Второй, третий и четвёртый аргументы представляют собой указатели на структуры типа FILETIME. По этим адресам будут записаны соответственно время создания файла, время последнего обращения к файлу и время последней

записи в файл. Если нам не нужна какая-то из этих меток, то в качестве указателя на соответствующую структуру нужно передать NULL.

Структура FILETIME описана в заголовочном файле winbase.h следующим образом:

```
typedef struct _FILETIME {
   DWORD dwLowDateTime;
   DWORD dwHighDateTime;
} FILETIME, *PFILETIME, *LPFILETIME;
```

В первое поле этой структуры, dwLowDateTime, функция запишет младшие тридцать два разряда метки времени, а во второе, dwHighDateTime, - старшие тридцать два разряда.

У читателя может возникнуть вопрос – а в каком формате представляется время, если для его представления необходимо шестьдесят четыре разряда? Дело в том, что в фирме Microsoft время создания файлов измеряют, начиная от нуля часов 1 января 1601 года, при этом выражают его в наносекундах. Мне, честно говоря, от такого выражения времени проку не очень много. ☺ Мне бы выразить время по старинке, в часах, минутах... Есть функция и для этого. Например, функция FileTimeToDosDateTime(), которая описана в файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI FileTimeToDosDateTime
                                    (CONST FILETIME *lpFileTime,
                                     LPWORD lpFatDate,
                                     LPWORD lpFatTime);
```

Первым аргументом этой функции, lpFileTime, является указатель на структуру типа FILETIME, содержащую время «в формате Microsoft», которое мы хотим преобразовать в формат времени, принятый в FAT. Второй и третий аргументы, lpFatDate и lpFatTime соответственно, являются указателями на два двойных слова, в которые будут записаны дата и время создания файла в формате, принятом в FAT. Как и обычно, функция возвращает значение TRUE при успешном завершении функции, значение FALSE говорит нам о том, что при работе функции встречена ошибка.

Обратное преобразование можно осуществить при помощи другой функции, DosDateToFile Time(), описанной в файле winbase.h так:

```
WINBASEAPI BOOL WINAPI DosDateTimeToFileTime
                                    (WORD wFatDate,
                                     WORD wFatTime,
                                     LPFILETIME lpFileTime);
```

Взглянув на аргументы этой функции, читатель сразу поймёт их назначение, поэтому я не стану останавливаться на этом.

Что еще можно сделать с временными метками файла? Ну, прежде всего, конечно, сравнить два файла и определить, который их них создан ранее. Для этой цели нам необходимо обратиться к функции CompareFi-

leTime(). Описание этой функции, приведённое ниже, можно найти в файле winbase.h:

```
WINBASEAPI LONG WINAPI CompareFileTime
                                    (CONST FILETIME *lpFileTime1,
                                     CONST FILETIME *lpFileTime2);
```

Аргументами этой функции являются указатели на две структуры типа FILETIME, в которых записаны сравниваемые времена. Значения, которые могут быть возвращены этой функцией, я привожу в таблице ниже:

Значение	Назначение
-1	Время, на которое указывает первый аргумент, более позднее
0	Оба времени равны
1	Время, на которое указывает первый аргумент, более раннее

Используя эту функцию, мы можем определить, производилась ли запись во время последнего обращения к файлу. Я надеюсь, что читатель без труда догадается, как это сделать.

Вполне вероятно, что нам захочется получить время в удобочитаемом виде. При этом может оказаться полезной функция FileTimeToSystemTime(), которая описана в файле winbase.h так:

```
WINBASEAPI BOOL WINAPI FileTimeToSystemTime
                                    (CONST FILETIME *lpFileTime,
                                     LPSYSTEMTIME lpSystemTime);
```

Что представляет собой первый аргумент, lpFileTime, понятно без объяснений. Это указатель на структуру типа FILETIME, в которой записано время в «формате Microsoft». А вот второй аргумент... В нём-то всё и дело! Второй аргумент, lpSystemTime, - это указатель на структуру типа SYSTEMTIME. Формат этой структуры описан в файле winbase.h следующим образом:

```
typedef struct _SYSTEMTIME {
   WORD wYear;
   WORD wMonth;
   WORD wDayOfWeek;
   WORD wDay;
   WORD wHour;
   WORD wMinute;
   WORD wSecond;
   WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;
```

Взглянув на поля этой структуры, так и хочется воскликнуть: «Так вот же время в удобочитаемом формате!» Да, уважаемый читатель, так оно и есть. В этой структуре всё записывается в таком виде, в каком мы легко сможем прочесть эту информацию.

Как всегда, если функция вернула значение TRUE, это означает, что функция отработала нормально. Ошибка при работе функции заставляет функцию вернуть значение FALSE.

Обратная функция по отношению к FileTimeToSystemTime(), которая называется SystemTimeToFileTime(), описана в файле winbase.h. Ниже приведено описание этой функции, взятое из этого файла:

```
WINBASEAPI BOOL WINAPI SystemTimeToFileTime
                                    (CONST SYSTEMTIME *lpSystemTime,
                                     LPFILETIME lpFileTime);
```

Я думаю, что читатель без труда догадался о назначении аргументов этой функции и о том, какие значения и в каких случаях функция возвращает.

Ну, и наконец, вполне вероятно, что может возникнуть необходимость изменить метки времени, связанные с файлом. Для этого существует функция SetFileTime(). В файле winbase.h мы без труда найдём описание этой функции. Это описание я привожу ниже:

```
WINBASEAPI BOOL WINAPI SetFileTime
                                    (HANDLE hFile,
                                     CONST FILETIME *lpCreationTime,
                                     CONST FILETIME *lpLastAccessTime,
                                     CONST FILETIME *lpLastWriteTime);
```

Первый вывод, который мы можем сделать – устанавливать метки времени можно только у открытого файла. Откуда это следует? Дело в том, что первый аргумент этой функции должен быть хэндлом файла, с которым мы работаем. Остальные три аргумента – это указатели на структуры типа FILETIME, в которых записаны времена создания файла, последнего доступа к нему и последней записи в файл. Если мы не хотим менять какую-то из меток времени, то нужно вместо соответствующего указателя передать значение NULL.

Возвращаемое функцией значение TRUE является свидетельством того, что функция отработала нормально. Если же возвращено значение FALSE, то, уважаемый читатель, придётся определять, в чём же причина ошибки.

Но квинтэссенцией этого рассказа про характеристики файлов, как мне кажется, может служить функция GetFileInformationByHandle(). Эта функция собирает ВСЮ информацию о файле. Описание этой функции, приведённое ниже, я взял из файла winbase.h:

```
WINBASEAPI BOOL WINAPI GetFileInformationByHandle
                    (HANDLE hFile,
                     LPBY_HANDLE_FILE_INFORMATION lpFileInformation);
```

Первый аргумент понятен – хэндл файла, информацию о котором мы хотим получить. Но какого рода информация может быть получена? Ответ на этот вопрос даёт второй аргумент, который является указателем

на структуру типа BY_HANDLE_FILE_INFORMATION, которая заполняется этой функцией. Структура описана в файле winbase.h. Это описание выглядит следующим образом:

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
   DWORD dwFileAttributes;
   FILETIME ftCreationTime;
   FILETIME ftLastAccessTime;
   FILETIME ftLastWriteTime;
   DWORD dwVolumeSerialNumber;
   DWORD nFileSizeHigh;
   DWORD nFileSizeLow;
   DWORD nNumberOfLinks;
   DWORD nFileIndexHigh;
   DWORD nFileIndexLow;
} BY_HANDLE_FILE_INFORMATION,
  *PBY_HANDLE_FILE_INFORMATION,
  *LPBY_HANDLE_FILE_INFORMATION;
```

Я не буду объяснять каждое поле подробно, я просто приведу краткое описание полей в том порядке, в каком они следуют в структуре:

- атрибуты файла;
- время создания файла;
- время последнего доступа к файлу;
- время последней записи в файл;
- серийный номер тома, на котором находится файл;
- старшие тридцать два разряда размера файла;
- младшие тридцать два разряда размера файла;
- число ссылок на файл;
- старшие тридцать два разряда идентификатора файла;
- младшие тридцать два разряда идентификатора файла.

Когда я нашёл эту функцию, я был просто поражён! Одна эта функция заменяет несколько других! Но, кажется, я немного отвлёкся. Последние два поля вызывают вопрос – о каком идентификаторе файла идёт речь? Дело в том, что при открытии файла система присваивает файлу уникальный идентификатор. Значение именно этого идентификатора и записывается в последние два поля структуры типа BY_HANDLE_FILE_INFORMATION.

А теперь – небольшая демонстрационная программа, в которой мы определим некоторые характеристики файлов, используя те функции, которые мы только что изучили. Я хотел бы, чтобы читатель дополнил эту программу теми функциями, которые мы не использовали в демонстрационной программе.

Программа использует следующий файл ресурсов:

```
CharacteristicsMenu MENU
{
 POPUP "&File"
```

```
 {
  MENUITEM "&Open", 101
  MENUITEM SEPARATOR
  MENUITEM "E&xit", 102
 }
}
```

Текст демонстрационной программы я привожу ниже:

```
#include <windows.h>
#include <stdio.h>

#define IDM_OPEN 101
#define IDM_EXIT 102

long WINAPI CharacteristicsWndProc( HWND, UINT, UINT, LONG );

HINSTANCE hInst;

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpszCmdParam, int nCmdShow )
 {
   HWND hWnd ;
   WNDCLASS WndClass ;
   MSG Msg;

   hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = (WNDPROC) CharacteristicsWndProc;
   WndClass.cbClsExtra = 0;
   WndClass.cbWndExtra = 0;
   WndClass.hInstance =  hInstance ;
   WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
   WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
   WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
   WndClass.lpszMenuName = "CharacteristicsMenu";
   WndClass.lpszClassName = "Characteristics";

   if ( !RegisterClass(&WndClass) )
      {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
      }

   hWnd = CreateWindow("Characteristics", "Characteristics of file",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT,
                       CW_USEDEFAULT,
                       CW_USEDEFAULT,
                       CW_USEDEFAULT,
                       NULL, NULL,
```

```
                        hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

long WINAPI CharacteristicsWndProc(HWND hWnd, UINT Message,
                                   UINT wParam, LONG lParam )
 {
  static char lpstrFile[MAX_PATH] = "";
  HANDLE hFile;
  char cBuffer[0x400];

  switch(Message)
   {
    case WM_COMMAND:
     switch(LOWORD(wParam))
      {
       case IDM_OPEN:
        OPENFILENAME OpenFileName;
        OpenFileName.lStructSize = sizeof(OPENFILENAME);
        OpenFileName.hwndOwner = hWnd;
        OpenFileName.hInstance = hInst;
        OpenFileName.lpstrFilter = "Applications (*.exe)\0*.exe\0
                                    Application Extension (*.dll)\0*.dll\0\0";
        OpenFileName.lpstrCustomFilter = NULL;
        OpenFileName.nFilterIndex = 0;
        OpenFileName.lpstrFile = lpstrFile;
        OpenFileName.nMaxFile = MAX_PATH;
        OpenFileName.lpstrFileTitle = NULL;
        OpenFileName.lpstrInitialDir = NULL;
        OpenFileName.lpstrTitle = "Open PE-files for viewing of section table...";
        OpenFileName.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST |
                             OFN_HIDEREADONLY | OFN_LONGNAMES |
                             OFN_PATHMUSTEXIST;
        OpenFileName.lpstrDefExt = NULL;
        if(!GetOpenFileName(&OpenFileName))
          MessageBox(hWnd, "Cannot open file", "Error of file opening",
                     MB_OK);
```

```
else
 {
  if(INVALID_HANDLE_VALUE ==
    (hFile = CreateFile(OpenFileName.lpstrFile,
                        GENERIC_READ | GENERIC_WRITE,
                        0, NULL, OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL, NULL)) )
    {
     MessageBox(hWnd, "Cannot open file", "Error of file opening",
                MB_OK);
     return 0;
    }
   else
    {
     strcpy(cBuffer, "");
     DWORD dwFileAttributes =
                              GetFileAttributes(OpenFileName.lpstrFile);
     if(dwFileAttributes & FILE_ATTRIBUTE_READONLY)
        strcat(cBuffer, "FILE_ATTRIBUTE_READONLY");
     if(dwFileAttributes & FILE_ATTRIBUTE_HIDDEN)
        strcat(cBuffer, " | FILE_ATTRIBUTE_HIDDEN");
     if(dwFileAttributes & FILE_ATTRIBUTE_SYSTEM)
        strcat(cBuffer, " | FILE_ATTRIBUTE_SYSTEM");
     if(dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
        strcat(cBuffer, " | FILE_ATTRIBUTE_DIRECTORY");
     if(dwFileAttributes & FILE_ATTRIBUTE_ARCHIVE)
        strcat(cBuffer, " | FILE_ATTRIBUTE_ARCHIVE");
     if(dwFileAttributes & FILE_ATTRIBUTE_NORMAL)
        strcat(cBuffer, " | FILE_ATTRIBUTE_NORMAL");
     if(dwFileAttributes & FILE_ATTRIBUTE_TEMPORARY)
        strcat(cBuffer, " | FILE_ATTRIBUTE_TEMPORARY");
     if(dwFileAttributes & FILE_ATTRIBUTE_COMPRESSED)
        strcat(cBuffer, " | FILE_ATTRIBUTE_COMPRESSED");
     if(dwFileAttributes & FILE_FLAG_POSIX_SEMANTICS)
        strcat(cBuffer, " | FILE_FLAG_POSIX_SEMANTICS");
     if(dwFileAttributes & FILE_FLAG_BACKUP_SEMANTICS)
        strcat(cBuffer, " | FILE_FLAG_BACKUP_SEMANTICS");
     if(dwFileAttributes & FILE_FLAG_DELETE_ON_CLOSE)
        strcat(cBuffer, " | FILE_FLAG_DELETE_ON_CLOSE");
     if(dwFileAttributes & FILE_FLAG_SEQUENTIAL_SCAN)
        strcat(cBuffer, " | FILE_FLAG_SEQUENTAL_SCAN");
     if(dwFileAttributes & FILE_FLAG_RANDOM_ACCESS)
        strcat(cBuffer, " | FILE_FLAG_RANDOM_ACCESS");
     if(dwFileAttributes & FILE_FLAG_NO_BUFFERING)
        strcat(cBuffer, " | FILE_FLAG_NO_BUFFERING");
     if(dwFileAttributes & FILE_FLAG_OVERLAPPED)
        strcat(cBuffer, " | FILE_FLAG_OVERLAPPED");
     if(dwFileAttributes & FILE_FLAG_WRITE_THROUGH)
        strcat(cBuffer, " | FILE_FLAG_WRITE_THROUGH");
     MessageBox(hWnd, cBuffer, "Attributes of file", MB_OK);
     sprintf(cBuffer, "File Size = %08d", GetFileSize(hFile, NULL));
     MessageBox(hWnd, cBuffer, "FileSize", MB_OK);
    }
```

```
            }
            return 0;
          case IDM_EXIT:
            SendMessage(hWnd, WM_DESTROY, 0, 0);
            default:
          return DefWindowProc(hWnd,Message,wParam, lParam);
        }
      case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
      default:
        return DefWindowProc(hWnd,Message,wParam, lParam);
    }
}
```

Кажется, нам осталось рассмотреть всего лишь две небольших темы о файлах. Одна из них – это

Поиск файлов

Сейчас поиск файлов стал задачей простой настолько, что фирма Microsoft даже разработала целый набор диалоговых окон, которые позволяют пользователям самостоятельно находить свои файлы на дисках. Но иногда перед программистами встают задачи относительно нестандартные, в которых возможностей поиска, представляемых упомянутыми выше диалоговыми окнами, совершенно недостаточно. Вот в этих случаях и приходится использовать те возможности, которые заложены в Win32 API.

Допусим, программе необходимо получить полный путь файла. Для этого она может вызвать функцию GetFullPathName(). Описание этой функции, взятое из заголовочного файла winbase.h, я привожу ниже:

```
WINBASEAPI DWORD WINAPI GetFullPathNameA(LPCSTR lpFileName,
                                         DWORD nBufferLength,
                                         LPSTR lpBuffer,
                                         LPSTR *lpFilePart);
WINBASEAPI DWORD WINAPI GetFullPathNameW(LPCWSTR lpFileName,
                                         DWORD nBufferLength,
                                         LPWSTR lpBuffer,
                                         LPWSTR *lpFilePart);
#ifdef UNICODE
#define GetFullPathName  GetFullPathNameW
#else
#define GetFullPathName  GetFullPathNameA
#endif // !UNICODE
```

Первым аргументом этой функции, lpFileName, является указатель на имя файла, полное имя которого мы хотим получить. Второй аргумент, nBufferLength, определяет длину буфера, в который системы будет записывать полныйы путь файла. Третий аргумент, lpBuffer, является указателем на буфер, в который система занесёт полный путь файла. Длина именно этого буфера и указывается в качестве второго аргумента. И, наконец, последний аргумент, lpFilePart, - в него функция запишет указатель на

тот символ в полном пути, с которго начинается собственно имя файла. Я не совсем понимаю, зачем нужен этот параметр, ведь, во-первых, мы уже знаем имя файла, а, во-вторых, если у нас есть полный путь файла, мы легко можем найти имя файла, воспользовавшись функцией strrchr(). Но, как бы то ни было, этот аргумент присутствует и функция данные в него записывает.

Функция возвращает количество байтов, записанных в буфер. Если возвращённое значение больше размера буфера, это является свидетельством того, что размер буфера недостаточен для размещения в нём полного имени файла. Если же при работе функции встретилась ошибка, то свидетельством этого будет возврвщённое функцией нулевое значение.

Фактически эта функция никаких файлов не ищет. Она простонапросто пользуется данными о текущем каталоге и текущем диске процесса. Если необходимо действительно НАЙТИ файл на диске, то лучше воспользоваться другой функцией, SearchPath(). Приведённое ниже описание этой функции я выбрал из заголовочного файла winbase.h:

```
WINBASEAPI DWORD WINAPI SearchPathA(LPCSTR lpPath,
                                    LPCSTR lpFileName,
                                    LPCSTR lpExtension,
                                    DWORD nBufferLength,
                                    LPSTR lpBuffer,
                                    LPSTR *lpFilePart);
WINBASEAPI DWORD WINAPI SearchPathW(LPCWSTR lpPath,
                                    LPCWSTR lpFileName,
                                    LPCWSTR lpExtension,
                                    DWORD nBufferLength,
                                    LPWSTR lpBuffer,
                                    LPWSTR *lpFilePart);
#ifdef UNICODE
#define SearchPath  SearchPathW
#else
#define SearchPath  SearchPathA
#endif // !UNICODE
```

В качестве первого аргумента этой функции, lpPath, используется указатель на строку, определяющую пути к тем директориям, в которых будет осуществляться поиск файла. Если этот параметр равен NULL, то тогда поиск осуществляется в следующих директориях:

1. Директория, из которой запущена программа;
2. Текущая директория;
3. Системная директория;
4. Основная директория Windows;
5. Директории, перечисленные в переменной окружения PATH.

Указатель на имя файла, путь к которому мы ищем, является вторым аргументом функции.

Второй и третий аргументы работают в связке. Если указатель, являющийся вторым аргументом, указывает на строку, содержащую имя

файла с расширением, то в этом случае третий аргумент должен быть равен NULL. Если же второй аргумент указывает на строку, содержащую имя файла бкз расширения, то в этом случае третий аргумент должен указывать на строку, содержащую расширение файла, при этом расширение должно начинаться с точки.

Пятый аргумент, lpBuffer, указывает на буфер, в который будет занесён полный путь файла. Четвёртый аргумент, nBufferLength, определяет длину этого буфера.

И, наконец, последний, шестой аргумент, lpFilePart, после выполнения функции будет указывать на символ в буфере, с которого начинается непосредственно имя файла.

Если функция отработала без ошибок, то она возвращает число байтов, записанных в буфер. Если возвращённое значение превышает размер буфера, то в этом случае размер буфера недостаточен для размещения в нём полного пути файла. И если функция вернула нулевое значение, нам необходимо разобраться, по какой причине при работе функции произошла ошибка.

Но у функции SearchPath() есть одно очень серьёзное ограничение. Она ает файлы только в ограниченном числе директорий. А как быть в том случае, если нам необходимо просмотреть весь диск? В этом случае нам на помощь приходит функция FindFirstFile(). Описание этой функции находится в файле заголовков winbase.h, я привожу его ниже:

```
WINBASEAPI HANDLE WINAPI FindFirstFileA(LPCSTR lpFileName,
                                        LPWIN32_FIND_DATAA lpFindFileData);
WINBASEAPI HANDLE WINAPI FindFirstFileW(LPCWSTR lpFileName,
                                        LPWIN32_FIND_DATAW lpFindFileData);
#ifdef UNICODE
#define FindFirstFile  FindFirstFileW
#else
#define FindFirstFile  FindFirstFileA
#endif // !UNICODE
```

Назначение первого аргумента, lpFileName, понятно. Это указатель на строку, содержащую имя файла. Точнее, наверное, будет сказать, что первый аргумент содержит указатель на строку, содержащую не имя файла, а спецификации файла, потому что эта одна из немногих функций, которая в качестве имени файла допускает символы подстановки («*» и «?»). Второй аргумент, lpFindFileData, является указателем на структуру типа WIN32_FILE_DATA, в которую будет записана информация о найденном файле. Это достаточно длинная структура также описана в файле winbase.h. Ниже я привожу её описание:

```
typedef struct _WIN32_FIND_DATAA {
   DWORD dwFileAttributes;
   FILETIME ftCreationTime;
   FILETIME ftLastAccessTime;
   FILETIME ftLastWriteTime;
   DWORD nFileSizeHigh;
```

```
    DWORD nFileSizeLow;
    DWORD dwReserved0;
    DWORD dwReserved1;
    CHAR  cFileName[ MAX_PATH ];
    CHAR  cAlternateFileName[ 14 ];
} WIN32_FIND_DATAA, *PWIN32_FIND_DATAA, *LPWIN32_FIND_DATAA;
typedef struct _WIN32_FIND_DATAW {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD dwReserved0;
    DWORD dwReserved1;
    WCHAR  cFileName[ MAX_PATH ];
    WCHAR  cAlternateFileName[ 14 ];
}WIN32_FIND_DATAW, *PWIN32_FIND_DATAW, *LPWIN32_FIND_DATAW
#ifdef UNICODE
typedef WIN32_FIND_DATAW WIN32_FIND_DATA;
typedef PWIN32_FIND_DATAW PWIN32_FIND_DATA;
typedef LPWIN32_FIND_DATAW LPWIN32_FIND_DATA;
#else
typedef WIN32_FIND_DATAA WIN32_FIND_DATA;
typedef PWIN32_FIND_DATAA PWIN32_FIND_DATA;
typedef LPWIN32_FIND_DATAA LPWIN32_FIND_DATA;
#endif // UNICODE
```

Теперь нам необходимо узнать, какого рода информация о файле записывается в поля этой структуры. Итак, в первое поле этой структуры, dwFileAttributes, записываются атрибуты файла. Во второе, третье и четвёртое поля записываются соответственно времена создания файла, последнего доступа к файлу и последней записи к файлу в «формате Microsoft». Следующие два поля, nFileSizeHigh и nFileSizeLow,содержат соответственно тридцать два старших разряда и тридцать два младших разряда размера файла. Ещё два поля, dwReserved0 и dwReserved1, зарезервированы фирмой Microsoft для использования в будущем, и поэтому на них можно не обращать внимания. В массив cFileName функция записывает полный путь файла. И, наконец, в массив cAlternateFileName записывается так называемое альтернативное имя файла, то есть имя файла в формате 8.3. Кстати, для того, чтобы получить имя файла в формате 8.3, можно воспользоваться функцией GetShrtPathName(), описанная в файле winbase.h следующим образом:

```
WINBASEAPI DWORD WINAPIGetShortPathNameA(LPCSTR lpszLongPath,
                                         LPSTR  lpszShortPath,
                                         DWORD  cchBuffer);
WINBASEAPI DWORD WINAPI GetShortPathNameW(LPCWSTR lpszLongPath,
                                          LPWSTR  lpszShortPath,
                                          DWORD   cchBuffer);
#ifdef UNICODE
```

```
#define GetShortPathName  GetShortPathNameW
#else
#define GetShortPathName  GetShortPathNameA
#endif // !UNICODE
```

Я предлагаю читателю самостоятельно разобраться с назначением аргументов этой функции. Очень надеюсь, что эта задача не вызовет никаких трудностей.

Если функция FindFirstFile не нашла требуемые файлы, то возвращаемое ею значение равно INVALID_HANDLE_VALUE. Если же работа функции завершена успешно, то функция возвращает некий хэндл, который может быть использован в тех случаях, когда необходимо продолжить поиск файлов, соответствующих заданной спецификации. Для продолжения поиска файлов может быть использована функция FindNextFile(). В файле winbase.h эта функция описана следующим образом:

```
WINBASEAPI BOOL WINAPI FindNextFileA(HANDLE hFindFile,
                                     LPWIN32_FIND_DATAA lpFindFileData);
WINBASEAPI BOOL WINAPI FindNextFileW(HANDLE hFindFile,
                                     LPWIN32_FIND_DATAW lpFindFileData);
#ifdef UNICODE
#define FindNextFile  FindNextFileW
#else
#define FindNextFile  FindNextFileA
#endif // !UNICODE
```

Первый аргумент, hFindFile, - это тот самый хэндл, который был возвращён при предыдущем вызове функции FindFirstFile(). Второй аргумент, lpFindFileData, является указателем на структуру типа WIN32_FIND_DATA, о которой мы говорили чуть выше. Если функция нашла файл, соответствующий указанным условиям, то она возвращает значение TRUE. В этом случае после обработки полученных результатов вызов функции мржет быть повторён для поиска очередного файла. Окончить поиск можно тогда, когда функция вернёт значение FALSE, говорящее о том,что файлов, соответствующих указанным спецификациям, на диске больше нет. Для того, чтобы указать системе, что поиск закончен и необходимо удалить информацию, связанную с поиском, в том числе и поисковый хэндл, необходимо обратиться к функции FindClose(), которая описана в файле winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI FindClose(HANDLE hFindFile);
```

Единственный аргумент этой функции, hFindFile, представляет собой поисковый хэндл, который был возвращён при вызове функции FindFirstFile(). Это один из немногих случаев, когда для закрытия объекта не применяется функция CloseHandle().

Выше я сказал, что если нам нужно просмотреть весь диск, то мы можем воспользоваться функциями FindFirstFile() и FindNextFile(). Но дело-то в том, что эти функции просматривают только одну директорию! Для

того, чтобы просмотреть весь диск, нам необходимо написать рекурсивную функцию, перебирающую все директории на диске.

И, наконец, последняя тема из раздела о файлах –

Уведомления об изменениях в файловой системе

Представим себе такую ситуацию – стандартных диалоговых окон не существует. Мы сами создали диалоговое окно, в котором выдаётся список файлов. Щёлкнув на одном из файлов, мы хотим открыть этот файл. Но в период времени между созданием нашего диалогового окна со списком файлов и щелчком мыши на одном из файлов этот файл был удалён другим процессом. Пытаясь открыть файл, который отображён у нас в списке, мы получаем сообщение об ошибке. Естественно, такая ситуация для нас нежелательна. Нам бы хотелось, чтобы при каждом изменении в файловой системе система посылала бы какие-то сообщения, которые получали бы все работающие приложения. Эти приложения при необходимости могли бы реагировать на эти сообщения и при необходимости корректировать список файлов, который был выдан в диалоговом окне.

Работа над возможностью посылки подобных сообщений была начата в ранних версиях Windows, но реализовано полностью было только в Windows'95 и Windows NT.

Принцип работы с подобными уведомлениями об изменениях в файловой системе следующий. Сначала приложение должно уведомит систему о том, что оно заинтересовано в получении подобных (их ещё называют нотификационными) сообщений. При этом оповещении поток создаёт некий объект, который может быть использован для синхронизации. При наступлении события, вызвавшего изменение состояния объекта (удаление файла, копирование файла, перемещёние файла и т.д.), поток реагирует на событие, после чего снова оповещает систему о том, что он вновь заинтересован в получении подобных уведомлений. И так должно продолжаться до тех пор, пока поток не примет решение о том, что получать уведомляющие сообщения ему больше не нужно и не оповестит об этом систему. Теперь рассмотрим то, о чём я только что рассказал, более подробно.

Для того, чтобы оповестить систему о том, что приложение хотело бы получать уведомления об изменениях в файловой системе, приложение должно обратиться к функции FindFirstChangeNotification(). Эта функция описана в заголовочном файле winbase.h следующим образом:

```
WINBASEAPI HANDLE WINAPI FindFirstChangeNotificationA
                                        (LPCSTR lpPathName,
                                         BOOL bWatchSubtree,
                                         DWORD dwNotifyFilter);
WINBASEAPI HANDLE WINAPIFindFirstChangeNotificationW
                                        (LPCWSTR lpPathName,
                                         BOOL bWatchSubtree,
                                         DWORD dwNotifyFilter);
#ifdef UNICODE
```

```
#define FindFirstChangeNotification  FindFirstChangeNotificationW
#else
#define FindFirstChangeNotification  FindFirstChangeNotificationA
#endif // !UNICODE
```

Первый аргумент, передаваемый этой функции, lpPathName, является указателем на строку, содержащую путь к директории. Приложение указывает, что хотело бы получать сообщения об изменениях, произошедших с файлами как минимум указанной директории. О поддиректориях, находящихся в данной директории, разговор, как говорится, особый. Если мы укажем не корневой каталог диска, а одну из поддиректорий, то приложение не будет получать сообщения об изменениях, произошедших в директориях, стоящих по иерархии выше указанной. Для того, чтобы получать сообщения о файлах на нескольких дисках, нам придётся несколько раз вызвать функцию FindFirstChangeNotification и указать разные начальные директории.

Я чуть раньше сказал, что о том, будет ли приложение получать уведомления об изменениях в поддиректориях, находящихся в указанной директории, у нас будет отдельный разговор. Вот второй аргумент, bWatchSubtree, и определяет, нужно ли получать уведомления о нижестоящих директориях. Если он равен TRUE, это означает, что приложение хочет получать сообщения об изменениях в нижестоящих директориях. Если этот аргумент равен FALSE, то это означает, что приложение будет получать сообщения об изменениях только в указанной директории.

Третий аргумент указывает, уведомления о каких событиях желательно получать приложения. Возможные значения этого аргумента я привожу в таблице ниже.

Флаг	Значение	Назначение
FILE_NOTIFY_CHANGE_FILE_NAME	0x00000001	Любые изменения (переименование, создание удаление), произведённые с именами файлов, вызывают выдачу оповещения
FILE_NOTIFY_CHANGE_DIR_NAME	0x00000002	Любые изменения (переименование, создание удаление), произведённые с именами директорий, вызывают выдачу оповещения
FILE_NOTIFY_CHANGE_ATTRIBUTES	0x00000004	Любые изменения атрибутов вызывают выдачу оповещений
FILE_NOTIFY_CHANGE_SIZE	0x00000008	Любые изменения размеров файлов вызывают выдачу оповещений
FILE_NOTIFY_CHANGE_LAST_WRITE	0x00000010	Любые изменения времени последней записи в файл вызывают выда-

		чу оповещений
FILE_NOTIFY_CHANGE_SECURITY	0x00000100	Любые изменения дескриптора защиты вызывают выдачу оповещений

Если функция закончила свою работу с ошибкой, то возвращённое ею значение равно INVALID_HANDLE_VALUE. В случае нормального завершения возвращаемое функцией значение является хэндлом объекта, который в дальнейшем может быть использован при вызове таких синхронизирующих функций как WaitForSingleObject() и WaitForMultipleObject().

Этот объект, хэндл которого мы получили, реагирует на изменение в системе и изменяет своё состояние каждый раз при изменении в файловой системе. Если мы обработали изменение и хотим и дальше получать нотификационные уведомления, мы должны вызвать функцию FindNextChangeNotification(). Описание этой функции, взятое из файла winbase.h, я привожу ниже:

```
WINBASEAPI BOOL WINAPI FindNextChangeNotification
                                    (HANDLE hChangeHandle);
```

Единственным аргументом этой функции является тот хэндл объекта, который был нам возвращён при вызове функции FindFirstChangeNotification().

Вызывать функцию FindNextChangeNotification() необходимо до тех пор, пока у нас есть необходимость получать нотификационные сообщения. Как только надобность в них отпала, мы должны оповестить систему, что больше получать нотификационные сообщения не хотим. Это делается при помощи обращения к функции FindCloseChangeNotification(), которая в файле winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI FindCloseChangeNotification
                                    (HANDLE hChangeHandle);
```

Аргументом и этой функции является тот хэндл объекта, который был нам возвращён при вызове функции FindFirstChangeNotification().

Я прошу читателя обратить внимание на то, что и в этом случае объект закрывается не при помощи вызова функции CloseHandle(), а при помощи другой функции.

Уважаемый читатель! Мы закончили рассмотрение функций, предназначенных для работы с файлами. Я очень надеюсь на то, что при помощи этой книги Вы получили достаточное представление о том, как осуществляется работа с файлами в Windows'95, Windows'98 и Windows NT. Я надеюсь, что сейчас Вы готовы к тому, чтобы приступить к программированию операций чтения информации из файла и записи информации в файл.

Файлы, отображаемые в память

Выше я описал стандартный способ работы с файлами. Он подразумевает открытие файла, чтение части файла в буфер, создаваемый в оперативной памяти и запись содержимого буфера обратно в файл. Способ достаточно громоздкий и неудобный. Однако с приходом Windows'95 ситуация несколько изменилась. Одним из новшеств, предлагаемых этой операционной системой, были т.н. файлы, проецируемые в память. При использовании проецируемых в память файлов мы можем, практически не занимая оперативную память, обращаться к файлу таким образом, словно файл полностью загружен в ОЗУ. Для анализа файлов, в частности, для такого анализа, который мы будем производить в этой книге, более удобного средства, наверное, и не придумаешь, так как мы можем обойтись без операций файлового ввода – вывода и буферизации.

Для того чтобы отобразить файл в память, нам необходимо выполнить следующие операции:

1. Создать или открыть файл, который мы хотим спроецировать в память.
2. Создать объект ядра системы «проецируемый файл». Тем самым мы сможем сообщить системе размер файла и способ доступа к нему.
3. Указать системе, что необходимо сделать – загрузить файл в память целиком или загрузить только часть файла.

После того, как мы завершили работу с файлом, спроецированным в память, нам необходимо выполнить следующие действия:

1. Отменить проецирование файла в память, то есть закрыть объект «проецируемый файл».
2. Закрыть проецируемый файл.

О том, как это сделать, мы поговорим более подробно. О том, как открыть файл я останавливаться не буду, ибо этой теме я посвятил достаточно много времени раньше. Теперь, после того, как мы научились открывать файлы, нам необходимо обсудить

Создание объекта «проецируемый файл»

Этот объект создаётся при помощи функции CreateFileMapping(), которая в файле winbase.h описана следующим образом:

```
WINBASEAPI HANDLE WINAPI CreateFileMappingA(HANDLE hFile,
                LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                DWORD flProtect,
                DWORD dwMaximumSizeHigh,
                DWORD dwMaximumSizeLow,
                LPCSTR lpName);
WINBASEAPI HANDLE WINAPI CreateFileMappingW(HANDLE hFile,
                LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                DWORD flProtect,
                DWORD dwMaximumSizeHigh,
                DWORD dwMaximumSizeLow,
                LPCWSTR lpName);
#ifdef UNICODE
#define CreateFileMapping  CreateFileMappingW
```

```
#else
#define CreateFileMapping  CreateFileMappingA
#endif // !UNICODE
```

Теперь нам опять необходимо разобраться во множестве аргументов этой функции. О том, что представляет собой первый аргумент, hFile, легко догадаться. Конечно же, это хэндл файла, который мы хотим спроецировать в память, то есть тот хэндл, который был возвращён нам при вызове функции CreateFile().

Второй аргумент, указатель на структуру типа SECURITY_ATTRIBUTES, мы обсудили при рассмотрении функции CreateFile(), поэтому сейчас повторяться я не стану. Скажу только, что обычно, как и в предыдущем случае, ему присваивается значение NULL.

А третий аргумент, flProtect, заставляет остановиться на нём чуть подробнее. Этот аргумент определяет, какой атрибут защиты будет присвоен странице физической памяти, на которую проецируется файл. Обычно используются следующие атрибуты:

Атрибут	Значение	Назначение
PAGE_READONLY	0x02	После проецирования из –файла можер считывать данные. При вызове функции CreateFile() необходимо было использовать флаг GENERIC_READ
PAGE_READWRITE	0x04	После проецирования можно считывать данные из файла и записывать данные в файл. При вызове функции CreateFile() необходимо было использовать флаги GENERIC_READ | GENERIC_WRITE.
PAGE_WRITECOPY	0x08	После проецирования можно считывать данные из файла и записывать данные в файл. Запись приводит к созданию копии страницы. При вызове функции CreateFile() необходимо было использовать флаги GENERIC_READ или GENERIC_READ | GENERIC_WRITE.

Кроме этого, именно через этот аргумент передаются и атрибуты защиты разделов проецируемой памяти. Я приведу список возможных атрибутов защиты разделов в таблице.

Атрибут	Значение	Назначение
SEC_IMAGE	0x1000000	Используется тогда, когда проецируемый файл является PE-файлом. При проецировании система просматривает содержимое файла для того, чтобы определить, какие атрибуты присвоить тому или иному разделу файла.
SEC_NOCACHE	0x10000000	Данные из файла, спроецированного в память, кэшировать не нужно.

Но вернёмся, как говорится, к нашим баранам, то есть к аргументам функции CreateFileMapping(). Четвёртый и пятый аргументы этой функции, dwMaximumSizeHigh и dwMaximumSizeLow соответственно, определяют максимальный размер проецируемого файла в байтах. Так как Win32 может работать с файлами, размер которых выражается 64-битными числами, то в параметре dwMaximumSizeHigh необходимо указать старшие 32 бита, а в параметре dwMaximumSizeLow необходимо указать младшие 32 бита значения. Но если нам необходимо спроецировать файл в память таким, каким он есть, то есть чтобы размер спроецированного файла совпадал с размером файла на диске, нам необходимо передать в этих параметрах нули.

И, наконец, последний аргумент, lpName. В нём указывается имя объекта «проецируемый файл». Это необходимо для того, чтобы к спроецированному файлу могли бы получить доступ и другие процессы. Но, однако, в подавляющем большинстве случаев этого не нужно, поэтому обычно через данный аргумент передают значение NULL.

При удачном завершении этой функции система возвращает хэндл объекта «спроецированный файл». Если же создание объекта завершилось с какой-то ошибкой, то функция возвращает NULL.

Проецирование данных файла на адресное пространство процесса

Мы создали объект «спроецированный в память файл». Теперь нам необходимо, чтобы операционная система, зарезервировав определённое адресное пространство под данные файла, передала это пространство как физическую память. Это делается при помощи функции MapViewOfFile(). Эта функция в файле winbase.h описана следующим образом:

```
WINBASEAPI LPVOID WINAPI MapViewOfFile(HANDLE hFileMappingObject,
                                       DWORD dwDesiredAccess,
                                       DWORD dwFileOffsetHigh,
                                       DWORD dwFileOffsetLow,
                                       DWORD dwNumberOfBytesToMap);
```

Придётся опять разбирать все аргументы этой функции. Понятно, что первый аргумент, hFileMappingObject, представляет собой хэндл отображённого в память на предыдущем этапе файла. Второй аргумент... Да, опять эти бесконечные режимы доступа к данным! Честно говоря, я не совсем понимаю, зачем нужно было делать эти бесконечные проверки режимов доступа. Единственное, что я могу предположить, это то, что разработчики фирмы Microsoft хотели дать программисту максимальную гибкость при работе с операционной системой. Что ж, опять приведу все допустимые значения аргумента.

Значение	Назначение

FILE_MAP_WRITE	Данные файла доступны как по чтению, так и по записи. Объект «проецируемый файл» должен быть создан с атрибутом PAGE_READWRITE
FILE_MAP_READ	Данные файла доступны только для чтения. Объект «проецируемый файл» должен быть создан с атрибутом PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY.
FILE_MAP_ALL_ACCESS	То же, что и FILE_MAP_WRITE
FILE_MAP_COPY	Данные файла доступны только для чтения. При записи создаётся закрытая копия страницы. Объект «проецируемый файл» должен быть создан с атрибутом PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY.

Два следующих параметра служат для того, чтобы указать системе, какой байт файла необходимо считать первым. Через аргумент dwFileOffsetHigh передаются старшие тридцать два разряда этого смещения, а через аргумент dwFileOffsetLow – младшие тридцать два разряда.

А после того, как мы указали системе, с какого места начинать отображение файла, что нам осталось ещё указать? Правильно, только размер отображаемой части файла. Этот размер передаётся через последний аргумент, dwNumberOfBytesToMap. Если, однако, мы сделаем этот аргумент равным нулю, то система будет считать, что нам необходимо отобразить ВЕСЬ файл.

Функция MapViewOfFile() в случае удачного завершения возвращает указатель на начало области отображения файла. Если же функция завершает свою работу с ошибкой, то она возвращает NULL. Получив указатель на область отображения, мы можем забыть о буферировании и работать с файлом так, словно это вовсе не файл, а область памяти.

Что ж, уважаемый читатель, мы научились отображать требующиеся нам файлы в память, получать указатели на начало области отображения и работать с файлом без буферирования, обращаясь к содержимому файла напрямую. Но одно из правил хорошего тона программирования гласит, что после окончания работы программы система должна быть возвращена в то же самое состояние, в котором она находилась до запуска программы. Если мы хотим следовать этому правилу, то у нас должен возникнуть вопрос – как сделать так, чтобы отображение файла было отменено, как закрыть объект «проецируемый файл» и как закрыть основной файл, над которым мы производили все эти манипуляции. На все эти вопросы я постараюсь ответить в следующих разделах.

Отмена отображения файла на адресное пространство процесса

Функцией, выполняющей обратные по отношению к MapViewOfFile() действия, является UnmapViewOfFile(). Эта функция в файле winbase.h описана следующим образом:

```
WINBASEAPI BOOL WINAPI UnmapViewOfFile(LPCVOID lpBaseAddress);
```

Даже беглого взгляда на эту функцию достаточно для того, чтобы понять, что необходимо передать этой функции в качестве аргумента. Естественно, указатель на начало области отображения файла. Другими словами, этот аргумент должен совпадать с тем значением, которое было возвращено нам при отображении файла функцией MapViewOfFile().

Закрытие объекта «проецируемый файл»

После того, как мы отменили отображение файла на адресное пространство процесса, нам необходимо закрыть объект «проецируемый файл». Это делается при помощи функции CloseHandle(). В файле winbase.h эта функция описана следующим образом:

```
WINBASEAPI BOOL WINAPI CloseHandle(HANDLE hObject);
```

Я думаю, читатель и в этом случае догадался о том, какое значение должно быть передано этой функции в качестве аргумента. Конечно же, это тот самый хэндл объекта «проецируемый файл», который был возвращён нам после вызова функции CreateFileMapping(). Надеюсь, что никаких объяснений в данном случае не требуется. В случае успешного завершения функция возвращает TRUE. Если во время выполнения функции произошла какая-то ошибка, то функция возвращает FALSE.

Закрытие файла производится стандартно, при помощи функции CloseHandle(), которой в качестве аргумента передаётся хэндл закрываемого файла.

Насколько долго мне пришлось рассказывать об отображении файла настолько быстро мы справились с отменой отображения и закрытием объектов! Единственное, что осталось сделать, для того, чтобы закончить разговор об отображаемых в память файлах, это написать демонстрационную программу. В той программе, которую я написал и текст которой приведён ниже, можно открыть файл с расширением «.exe» и узнать, является ли этот файл PE-файлом. О том, что такое PE-файл, читатель узнает в самом начале изучения формата исполняемого файла Win32. А пока я скажу, что исполняемые файлы, разработанные для Win32, имеют сигнатуру «PE», поэтому они и называются PE-файлами. Естественно, что не все exe-файлы являются PE-файлами, поэтому демонстрационная программа проецирует файл в память, проверяет сигнатуру и выводит сообщение о том, является ли файл PE_файлом.

Демонстрационная программа использует файл ресурсов, который я привожу ниже.

```
MainMenu MENU
{
 POPUP "&File"
 {
  MENUITEM "&Open", 101
  MENUITEM SEPARATOR
  MENUITEM "E&xit", 102
```

```
}
}
```

А теперь я привожу и основной файл программы:

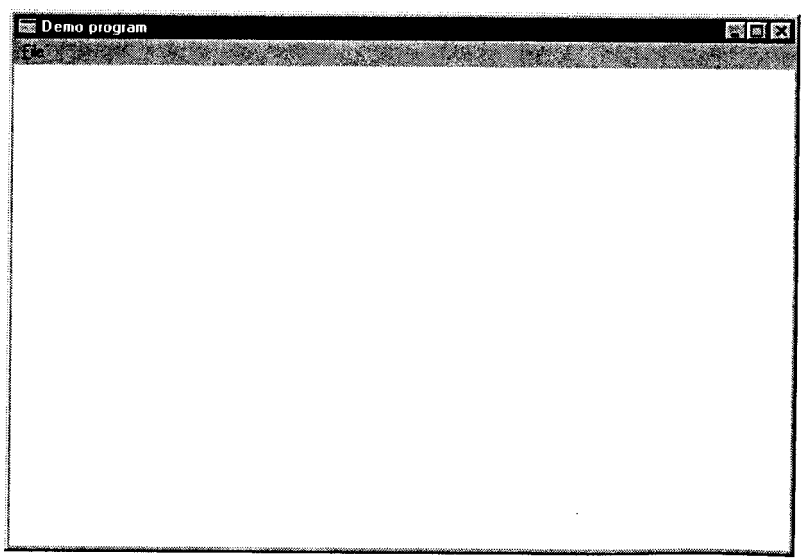
```
#include <windows.h>

#define IDM_OPEN 101
#define IDM_EXIT 102

HINSTANCE hInst;

long WINAPI MappedFilesWndProc( HWND, UINT, UINT, LONG );
LPVOID MapFile(LPTSTR);
BOOL TestFile(PIMAGE_DOS_HEADER);

int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow)
 {
   HWND hWnd ;
   WNDCLASS WndClass ;
   MSG Msg;

   hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = (WNDPROC) MappedFilesWndProc;
   WndClass.cbClsExtra = 0;
   WndClass.cbWndExtra = 0;
   WndClass.hInstance =  hInstance ;
   WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
   WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
   WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
   WndClass.lpszMenuName = "MainMenu";
   WndClass.lpszClassName = "MappedFiles";

   if ( !RegisterClass(&WndClass) )
     {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
     }

   hWnd = CreateWindow("MappedFiles", "Demo program",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                NULL, NULL,
                hInstance,NULL);
   if(!hWnd)
     {
       MessageBox(NULL,"Cannot create window","Error",MB_OK);
```

```
      return 0;
    }

/* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
   return Msg.wParam;
 }

long WINAPI MappedFilesWndProc(HWND hWnd, UINT Message,
                                    UINT wParam, LONG lParam )
 {

   static char lpstrFile[MAX_PATH] = "";
   PIMAGE_DOS_HEADER pIDH;

   switch(Message)
    {
     case WM_COMMAND:
      switch(LOWORD(wParam))
       {
        case IDM_OPEN:
         OPENFILENAME OpenFileName;
         OpenFileName.lStructSize = sizeof(OPENFILENAME);
         OpenFileName.hwndOwner = hWnd;
         OpenFileName.hInstance = hInst;
         OpenFileName.lpstrFilter="Applications(*.exe)\0*.exe\0
                                   Application Extension (*.dll)\0*.dll\0\0";
         OpenFileName.lpstrCustomFilter = NULL;
         OpenFileName.nFilterIndex = 0;
         OpenFileName.lpstrFile = lpstrFile;
         OpenFileName.nMaxFile = MAX_PATH;
         OpenFileName.lpstrFileTitle = NULL;
         OpenFileName.lpstrInitialDir = NULL;
         OpenFileName.lpstrTitle = "Open PE-files for viewing of section table...";
         OpenFileName.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST |
                              OFN_HIDEREADONLY | OFN_LONGNAMES |
                              OFN_PATHMUSTEXIST;
         OpenFileName.lpstrDefExt = NULL;
         if(!GetOpenFileName(&OpenFileName))
           MessageBox(hWnd, "Невозможно открыть файл",
                      "Ошибка открытия файла", MB_OK);
          else
           {
            if((pIDH=(PIMAGE_DOS_HEADER)
               MapFile(OpenFileName.lpstrFile)) == 0)
```

```
                MessageBox(hWnd, "Невозможно спроецировать файл в память",
                                "Ошибка проецирования файла", MB_OK);
              else
               {
                if(!TestFile(pIDH))
                  MessageBox(hWnd, "Файл не является PE-файлом",
                                "Ошибка", MB_OK);
                 else
                  MessageBox(hWnd, "Файл является PE-файлом",
                                "Всё отлично!", MB_OK);
                UnmapViewOfFile((LPVOID) pIDH);
               }
            }
           return 0;
          case IDM_EXIT:
           SendMessage(hWnd, WM_DESTROY, 0, 0);
           default:
          return DefWindowProc(hWnd,Message,wParam, lParam);
         }
      case WM_DESTROY:
       PostQuitMessage(0);
       return 0;
      default:
       return DefWindowProc(hWnd,Message,wParam, lParam);
    }
}

LPVOID MapFile(LPTSTR lpstrFile)
 {
  HANDLE hFile, hMappedFile;
  LPVOID lpMappedFile;

  hFile = CreateFile(lpstrFile, GENERIC_READ,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                     NULL);
  if(INVALID_HANDLE_VALUE == hFile)
     return 0;
    else
     {
      hMappedFile = CreateFileMapping(hFile, NULL, PAGE_READONLY,
                                      0, 0, NULL);
      CloseHandle(hFile);
      if(!hMappedFile)
         return 0;
       else
        {
         lpMappedFile = MapViewOfFile(hMappedFile, FILE_MAP_READ,
                                      0, 0, 0);
         CloseHandle(hMappedFile);
         if(!lpMappedFile)
            return 0;
          else
```

```
            return lpMappedFile;
        }
    }
}

BOOL TestFile(PIMAGE_DOS_HEADER pIDH)
{
  if(pIDH->e_magic != IMAGE_DOS_SIGNATURE)
     return FALSE;
   else
     if(pIDH->e_lfarlc >= 0x40)
       {
        PIMAGE_NT_HEADERS pINTH = (PIMAGE_NT_HEADERS)
                                          ((PBYTE) pIDH + pIDH->e_lfanew);
        if(pINTH->Signature != IMAGE_NT_SIGNATURE)
          return FALSE;
         else
          return TRUE;
       }
      else
       return FALSE;
}
```

А теперь – внешний вид окна, создаваемого программой:



Я хотел бы обратить внимание читателя на некоторые особенности этой программы. Если внимательно всмотреться в этот текст, то можно увидеть, что я закрываю файл в тот момент, когда создаётся объект «проецируемый файл». Фактически хэндл открытого файла В ЭТОЙ ПРОГРАММЕ нам больше не нужен. Точно так же я закрываю объект «про-

ецируемый файл» в тот момент, когда проецирую содержимое файла на адресное пространство процесса. Причина точно та же – В ЭТОЙ ПРОГРАММЕ нам хэндл этого объекта больше не пригодится. Я думаю, незачем держать в памяти объекты, которые нам больше в работе программы не пригодятся.

Внутренности исполняемого файла Win32

Программисты – народ особый. Во-первых, в силу профессии, которая сродни искусству (иногда даже говорят «искусство программирования»). Во-вторых, в отличие, скажем, от художника или музыканта, чьё произведение зависит только от наличия таланта, программист постоянно испытывает недостаток информации. В частности, книг и материалов по системному программированию недостаточно. Возможность общения через Интернет всей проблемы не решает. Если программист пишет оригинальную программу, то, естественно, и вопросы у него будут достаточно интересные и вряд ли он получит точный и исчерпывающий ответ на интересующий его вопрос. Таким образом, для того, чтобы получить необходимую ему информацию, программист должен осуществлять САМОСТОЯТЕЛЬНОЕ ИССЛЕДОВАНИЕ операционной системы, в которой он работает, а также интересующих его программ. Очень часто тех, кто осуществляет такие исследования, называют хакерами.

Кроме этого, немаловажен тот факт, что практически единственным способом освоить хороший стиль программирования и научиться тем или иным секретам остаётся опять-таки САМОСТОЯТЕЛЬНОЕ ИССЛЕДОВАНИЕ программ и операционных систем. Другими словами, эти исследования позволяют программисту повысить свою квалификацию.

Ну, и, в конце концов, именно САМОСТОЯТЕЛЬНОЕ ИССЛЕДОВАНИЕ – это замечательный вид интеллектуального спорта и способ удовлетворить извечный вопрос «А как это сделано?».

Известны два типа исследования программ – статический и динамический. Динамический способ – это запуск программы под отладчиком и анализ её работы шаг за шагом. Статический способ предусматривает создание такого представления программы, которое было бы наиболее удобно для её анализа. К статическим способам можно отнести получение шестнадцатеричного дампа программы или обычный просмотр программы в текстовом режиме.

Конечно же, существуют и комбинированные способы исследования программ. Можно привести пример, когда исследуемая программа запускается из-под какого-либо монитора, производящего вывод в файл какой-либо информации о работе исследуемой программы, например, об обращении к реестру или о вызовах импортируемых функций. После окончания работы программы, запущенной из-под подобного монитора, анализируется информация, выведенная в файл. Во многих случаях этот анализ позволяет получить требующуюся информацию с наименьшими усилиями.

Но я сознательно умолчал о наиболее известном и, наверное, наиболее широко применяющемся статическом способе анализа программ – дизассемблировании исполняемого файла, то есть создании такого представления программы, в котором, как минимум, все машинные команды заменены мнемоническими обозначениями кодов операций и операнды представлены в удобном для человека виде. Естественно, что программа в таком виде (собственно говоря, это уже не программа, а просто дизас-

семблированный модуль) намного легче воспринимается человеком, производящим анализ программы. Дизассемблированный модуль, получаемый в результате работы дизассемблера со средними возможностями, состоит как минимум из двух частей. Первая часть – это непосредственно дизассемблированный исполняемый код. Анализ этой части программы позволяем определить, каким образом изменяются данные, используемые программой. Задачей исследователя в данном случае является восстановить ход мысли программиста, написавшего дизассемблированную программу. Это задача творческая, успех её реализации зависит, прежде всего, от квалификации человека, производящего исследование дизассемблированного модуля. Формализовать этот процесс, а тем более, автоматизировать его, практически невозможно. Вторая часть содержит данные, используемые программой и операционной системой. Показать, как из данных извлечь максимум информации о программе, а также каким образом представить данные в наиболее удобной для анализа форме, и является задачей этой книги.

В некотором смысле это книга о том, с чего нужно начать процесс «взлома» программы. В данном случае я рассчитываю, что читать эту книгу будут не те, кому результаты «взлома» интересны количеством доставленных ближним неприятностей. Меня вовсе не прельщает слава печально известной книги «Пишем вирус и антивирус». Я рассчитываю, что эту книгу будут читать люди, которые задаются вопросом «Как это сделано и как это работает?», а не «Как напакостить?». Попутно отмечу, что, так как исполняемый файл не может исполняться сам по себе, а исполняется в конкретной операционной системе, то ещё одной задачей этой книги является дать читателю более полное представление о работе упомянутых выше операционных систем, в частности о порядке загрузки файла в память и взаимодействии частей модуля.

В этой книге я намерен рассмотреть формат исполняемого файла, применяемого в операционных системах Win32s (умершей практически не родившейся), Windows'95, Windows'98 и Windows NT. Эта книга написана автором, который произвёл самостоятельное исследование этого формата. В начале своей работы у меня не было никаких других дополнительных источников информации, кроме операционной системы Windows'95, установленной на моём домашнем компьютере, самих исполняемых файлов и компилятора Borland C++ 5.01. Как не странно, на первых порах этого оказалось больше чем достаточно. Я рассчитываю что читатель, который будет читать эту книгу, будет читать её не на ходу и не будет воспринимать её как лёгкое чтение на ночь. Я надеюсь, что читатель будет во время чтения сидеть за компьютером и вместе со мной пройдет по всем закоулкам исполняемых файлов Win32. Я очень надеюсь на то, что, прочитав эту книгу, читатель станет более глубоко понимать не только то, что запрятано в недрах исполняемых файлов Win32, но и работу операционных систем семейства Win32.

По моему убеждению, все технические книги, в особенности в части вычислительной техники, могут быть разделены на две большие категории. Первая категория - это "самоучители». Читая «самоучитель», про-

граммист «въезжает» (*извините за профессиональный сленг*) в новую для себя область и получает ПЕРВОНАЧАЛЬНЫЕ знания о предмете, что позволяет ему практически НЕМЕДЛЕННО начать использовать полученные знания. В частности, моя первая книга, «Азбука программирования в Win32 API», принадлежала именно к категории «самоучителей». При написании «самоучителей» необходимо придерживаться нескольких правил. Первое – книга должна быть написана живым языком и быть достаточно интересной для читателя, чтобы у последнего не возникло желания бросить её на полпути. Второе правило – после прочтения каждого раздела у читателя должны появиться новые знания, которые он может начать использовать сразу же. Вторая категория книг – это книги, позволяющие уже достаточно подготовленному читателю расширить или углубить уже имеющиеся знания. Они в большей степени предназначены для профессионалов и *используются в большинстве своём как справочники по определённым*, иногда достаточно узким вопросам. Эта книга относится ко второй категории. По глубокому убеждению автора, изложенный в ней материал необходим каждому серьёзному программисту для Windows и позволит более глубоко понять процессы взаимодействия операционной системы и исполняемого файла, а также находить необходимую информацию в недрах исполняемых файлов (PE-файлов) Windows.

Одним из осязаемых результатов этой книги будет разработанная программа, при помощи которой читатель сможет разобраться в хитросплетениях исполняемого файла. Более того, результатом каждого раздела будет небольшая, но законченная и, самое главное, полезная функция этой программы. Я хотел бы в этом месте остановиться на одном моменте. Когда я писал демонстрационные программы, я, естественно, не заботился об их дизайне. Это не коммерческие программы. Моей целью было просто показать читателю способы, при помощи которых можно найти ту или иную информацию в исполняемом файле. Поэтому пусть читатель не судит меня строго. Есть ещё одна тонкость. Иногда демонстрационные программы можно было бы сделать более короткими, например, поместить строковые константы в таблицы ресурсов и обращаться к ним в цикле. Но, как и всегда, выигрывая в одном, мы проигрываем в другом. Выигрывая в объёме программы, я бы заставил читателя делать двойную работу. Сначала читателю пришлось бы разобраться в структуре программы и понять, как она устроена, и только потом обращать внимание непосредственно на семантику программы. Я решил, пожертвовав краткостью программы, сэкономит время и усилия читателя.

Я бы хотел остановиться ещё на одном моменте. Почему-то все описания формата исполняемого файла, которые мне пришлось изучить при написании этой книги, не доводили описание до конца. Возможно, потому, что, например, формат ресурсов не считается форматом файла. Но как же так – ведь ресурсы же являются составной частью исполняемого файла? Поэтому мне пришлось проделать определённую работу для того, чтобы самостоятельно докопаться до понимания формата некоторых частей исполняемого файла. В других изданиях я не смог найти интересую-

щей меня информации. В этом смысле содержимое книги является уникальным.

При написании книги я предполагал, что читателем этой книги будет человек, уже имеющий опыт написания программ для Windows. Тех же, которые такого опыта не имеют, я отсылаю к своей книге «Азбука программирования в Win32 API», второе издание которой вышло в 1999 году в издательстве «Радио и связь».

Общая структура файла

Перед тем, как начать описание общей структуры файла, я бы хотел сделать одно замечание. Когда я писал эту книгу, зачастую у меня не было никакой документации, за исключением заголовочных файлов, поставляемых в комплекте Borland C++ 5.01. Я хотел бы поблагодарить тех, кто разрабатывал эти файлы, то есть создавал форматы, описывал переменные и структуры, давая им осмысленные имена. Очень часто той информации, которую я находил в заголовочных файлах, мне хватало для того, чтобы прийти к определённым выводам. И отправной точкой для этого были ОСМЫСЛЕННЫЕ ИМЕНА СТРУКТУР, ПОЛЕЙ И ПЕРЕМЕННЫХ. Уж если посторонний человек, не работник фирмы–разработчика, смог разобраться в хитросплетениях всех этих полей и переменных, то, значит, программисты-разработчики использовали хороший стиль программирования. Спасибо вам, незнакомые мои коллеги!

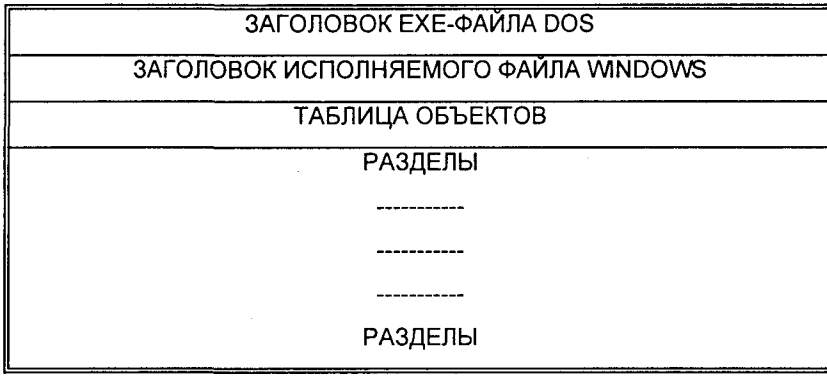
Ну, а теперь можно перейти и непосредственно к разговору об исполняемых файлах.

В операционных системах, входящих в семейство Win32, существует несколько типов исполняемых файлов. Типы файлов различаются по расширению файла, а также по сигнатуре (определённой последовательности байтов, находящихся по определённым смещениям в теле исполняемого файла). Все сигнатуры исполняемых файлов перечислены в заголовочном файле Windows SDK winnt.h. Ниже я привожу список этих сигнатур в таком виде, в каком они приведены в winnt.h:

Сигнатура	Значение	Описание
IMAGE_DOS_SIGNATURE	0x5A4D	MZ
IMAGE_OS2_SIGNATURE	0x454E	NE
IMAGE_OS2_SIGNATURE_LE	0x454C	LE
IMAGE_VXD_SIGNATURE	0x454C	LE
IMAGE_NT_SIGNATURE	0x00004550	PE00

В этой книге мы будем рассматривать только формат PE-файла, поэтому, не останавливаясь на других форматах, поговорим о структуре именно этого формата.

Как и обычно, после сигнатуры файла следует заголовок исполняемого файла. Причём, первым идёт заголовок исполняемого файла DOS. Почему DOS? Да только лишь потому, что любой исполняемый файл Windows является в то же время и исполняемым файлом DOS. Парадокс? Нет, и об этом мы поговорим чуть позже. А пока я приведу общую схему формата исполняемого файла Windows.



Как видно, ничего сложного в структуре исполняемого файла Windows нет. Заголовок DOS, заголовок Windows, таблица объектов, которая практически является оглавлением разделов, и непосредственно разделы. Всё! Тем не менее, этот формат заслуживает того, чтобы поговорить о нём подробнее.

Заголовки исполняемого файла

Я уже говорил, что, как и любой исполняемый файл DOS, исполняемый файл Windows начинается с заголовка.

Заголовок DOS

Заголовок исполняемого файла DOS уже неоднократно описан в литературе, поэтому я не буду утомлять читателя очередным его описанием, а всего лишь приведу описание этого заголовка, взятое из файла winnt.h:

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    WORD   e_magic;                 // Magic number
    WORD   e_cblp;                  // Bytes on last page of file
    WORD   e_cp;                    // Pages in file
    WORD   e_crlc;                  // Relocations
    WORD   e_cparhdr;               // Size of header in paragraphs
    WORD   e_minalloc;              // Minimum extra paragraphs needed
    WORD   e_maxalloc;              // Maximum extra paragraphs needed
    WORD   e_ss;                    // Initial (relative) SS value
    WORD   e_sp;                    // Initial SP value
    WORD   e_csum;                  // Checksum
```

```
  WORD   e_ip;                    // Initial IP value
  WORD   e_cs;                    // Initial (relative) CS value
  WORD   e_lfarlc;                // File address of relocation table
  WORD   e_ovno;                  // Overlay number
  WORD   e_res[4];                // Reserved words
  WORD   e_oemid;                 // OEM identifier (for e_oeminfo)
  WORD   e_oeminfo;               // OEM information; e_oemid specific
  WORD   e_res2[10];              // Reserved words
  LONG   e_lfanew;                // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Нас формат этого заголовка интересует постольку, поскольку именно с DOS-заголовка начинаются исполняемые файлы Windows. Если читатель помнит, то при запуске Windows-программ из DOS в большинстве случаев выдается сообщение о том, что программу нужно запускать из Windows. Это сообщение выдается DOS-частью программы. Вместо программы выдачи сообщения при линковании программы мы можем подставить любую другую программу, но в данный момент нам это безразлично. Нам сейчас важно, что кроме этого, если слово, которое находится по смещению 0x18 в заголовке DOS-файла, больше или равно 0x40, то по смещению 0x3c находится смещение (прошу прощения у читателя за тавтологию), по которому мы можем найти сигнатуру какого-то другого исполняемого файла (OS/2, Windows, VxD). Другими словами, исполняемый файл Windows сам по себе существовать не может. Вот и разрешение того парадокса, о котором я упоминал. Запомним этот вывод:

Если слово, которое находится по смещению 0x18 в заголовке DOS-файла, больше или равно 0x40, то по смещению 0x3c находится смещение (прошу прощения у читателя за тавтологию), по которому мы можем найти сигнатуру другого исполняемого файла. Если сигнатура исполняемого файла представляет собой «PE00», то файл является исполняемым файлом Windows.

Интересно, что формат исполняемого файла практически полностью документирован в файле winnt.h. Этот файл мы будем упоминать в этой книге наиболее часто.

Итак, будем считать, что мы определили способ нахождения и определения смещения начала исполняемого файла Windows. Естественно, что он, как и любой исполняемый файл, также начинается с заголовка.

Заголовок исполняемого файла Windows

Давайте посмотрим, как этот заголовок описан в заголовочном файле winnt.h:

```
typedef struct _IMAGE_NT_HEADERS {
   DWORD Signature;
   IMAGE_FILE_HEADER FileHeader;
   IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Внимательно присмотревшись к этому описанию, можно сделать вывод о том, что заголовок исполняемого файла Windows состоит из трёх частей. Часть первая – это сигнатура исполняемого файла. Вторая часть – это заголовок файла (впредь я буду его называть «обязательным»). И, наконец, третья часть – это ещё один заголовок, «необязательный». Две последние части и хранят в себе всю необходимую для работы информацию. Перед тем, как мы начнем рассмотрение, мне хотелось бы условиться с читателем о некоторых терминах. Под словом «файл» мы будем понимать исследуемый файл на диске, а под словом «образ (image)» - ту часть файла, которая загружается в память при вызове файла для исполнения.

О сигнатуре исполняемого файла мы уже говорили. Здесь я повторюсь, что мы будем рассматривать только тот случай, когда сигнатура исполняемого файла представляет собой «PE00», то есть когда исполняемый файл является исполняемым файлом Windows. Давайте рассмотрим формат обязательной и необязательной частей, а также назначение каждого поля заголовка.

Обязательная часть заголовка PE-файла

В файле winnt.h обязательный заголовок описан следующим образом:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Рассмотрим назначение каждого поля в отдельности.

Поле Machine

Из описания этого поля видно, что оно занимает всего одно слово. Как читатель помнит, при компиляции проекта мы можем указать компилятору, для какого процессора необходимо генерировать код. Информация о том, для какого процессора скомпилирован исполняемый файл, хранится в поле «Machine». Ниже приведены коды процессоров так, как они описаны в файле winnt.h:

Обозначение	Значение	Описание
IMAGE_FILE_MACHINE_UNKNOWN	0	Неизвестно, для какого процессора скомпилирован файл
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386

	0x14d	Intel 486
	0x14e	Intel Pentium
	0x160	MIPS big-indian
IMAGE_FILE_MACHINE_R3000	0x162	MIPS Mark I little-indian (R2000, R3000)
	0x163	MIPS Mark II (R6000)
IMAGE_FILE_MACHINE_R4000	0x166	MIPS Mark III little-indian (R4000)
IMAGE_FILE_MACHINE_R10000	0x168	MIPS little-indian
IMAGE_FILE_MACHINE_ALPHA	0x184	Alpha_AXP
IMAGE_FILE_MACHINE_POWERPC	0x1F0	IBM PowerPC Little-indian
MOTOROLA 68000	0x268	
PA RISC	0x290	

Я рекомендую читателю попробовать откомпилировать какую-нибудь программу для процессоров Pentium II, Pentium Pro и Pentium III. Тогда, возможно, к этой табличке можно будет добавить еще несколько строчек.

Поле NumberOfSections

Следующее поле - NumberOfSections. Оно содержит в себе число разделов исполняемого файла или, что то же самое, число входов в таблице разделов. Т.к. это поле занимает одно слово, то максимальное число разделов в исполняемом файле – 65 536. К этим разделам относятся разделы экспортируемых и импортируемых функций, раздел ресурсов, непосредственно кода и данных программы и другие. Уважаемый читатель, я хотел бы, чтобы вы обратили особое внимание на это поле. Нам придется еще использовать его при дальнейшем рассмотрении формата исполняемого файла.

Поле TimeDateStamp

Метка времени создания файла хранится в поле TimeDateStamp, которое занимает одно двойное слово. Особого интереса это поле не представляет. В этом поле указывается число секунд, которое прошло от 16 часов 31 декабря 1969 года до момента создания файла. Интересно только, почему в качестве точки отсчёта выбран именно этот момент, а не, скажем, 00 часов 01 января 1970 года? Интересно, чем руководствовались разработчики фирмы Microsoft при выборе столь странной точки отсчёта?

Поле PointerToSymbolTable

Названия этого поля практически говорит само за себя. В том случае, когда в откомпилированном модуле сохраняется отладочная информация, то в этом поле хранится смещение таблицы символов.

Поле NumberOfSymbols

Название следующего поля – NumberOfSymbols -, также говорит сами за себя. Оно содержит количество символов в таблице символов отладочной информации. Об этих указателях мы еще поговорим при изучении таблицы символов.

Поле SizeOfOptionalHeader

Поле очередное - SizeOfOptionalHeader - хранит, как видно из его названия, размер необязательного заголовка файла. До него мы тоже дойдем, но уже чуть раньше, чем до таблицы символов. Лично мне не совсем понятно назначение этого поля. Оно постоянно и равно 224. Это подтверждает и следующий макрос, также находящийся в файле winnt.h:

```
#define IMAGE_SIZEOF_NT_OPTIONAL_HEADER     224
```

Наверное, в более ранних версиях Win32 размер необязательного файла не был фиксированным и это поле оставлено для совместимости с системой тех программ, которые были написаны с учётом этого факта. Косвенным подтверждением этого предположения является тот факт, что в описании TIS (Tool Interface Standart)-комитета за 1993 год это поле описывается как «число байтов в заголовке NT, которое следует за полем характеристик».

Поле Characteristics

И, наконец, последнее поле, Characteristics. Значения, которые может принимать это поле, я привожу в таблице ниже.

Характеристика	Значение	Описание
IMAGE_FILE_RELOCS_STRIPPED	0x0001	Информация о таблице перемещений извлечена из файла
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	Файл является исполняемым
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	Номера строк извлечены из файла
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	Локальные символы извлечены из файла
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Байты машинного слова зарезервированы

IMAGE_FILE_32BIT_MACHINE	0x0100	Файл скомпилирован для 32-битного компьютера
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Отладочная информация извлечена из файла
IMAGE_FILE_SYSTEM	0x1000	Системный файл
IMAGE_FILE_DLL	0x2000	Файл является DLL
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Байты машинного слова зарезервированы

Что можно сказать об этом поле? Из приведённых выше значений можно сделать вывод, что оно используется как логическая шкала, каждый бит которой имеет своё назначение. Допускаются и комбинации приведённых выше флагов. Например, в одной моей программе значение этого поля равно 0x818e. Но обращает на себя внимание тот факт, что некоторые биты этого поля недокументированы. Например, например, не определены поля, соответствующие значениям 0x0010, 0x0020, 0x0040, 0x0400, 0x0800, 0x4000.

Уважаемый читатель, обратите особое внимание на флаг IMAGE_FILE_EXECUTABLE_IMAGE. Если этот флаг не установлен, то это означает, что при линковании файла была встречена ошибка, но исполняемый exe-файл все же был создан. В этом случае exe-файл фактически исполняемым не является. Таким образом, если какой-то exe-файл при запуске зависает или начинает нести ерунду, есть небольшая вероятность того, что у файла не установлен этот флаг.

Представляет также определённый интерес и флаг IMAGE_FILE_DLL. Если этот флаг установлен, то исполняемый файл является библиотекой динамической компоновки (DLL), а не просто исполняемым файлом.

Рассмотрев структуру и назначение полей «обязательной» части заголовка исполняемого файла, мы можем перейти к рассмотрению «необязательной» части заголовка. Для того, чтобы получить смещение «необязательной» части нам необходимо просто-напросто посчитать размер «обязательной» части и прибавить его к смещению «обязательной» части. Конечно, мы можем самостоятельно посчитать этот размер, благо мы знаем размер каждого из полей. Но фирма Microsoft, вероятно понимая, что эту операцию придётся производить достаточно часто, в файле winnt.h определила макрос IMAGE_SIZEOF_FILE_HEADER, который определяет размер «обязательной» части. Ниже я привожу этот макрос так, как он определён в winnt.h:

```
#define IMAGE_SIZEOF_FILE_HEADER        20
```

На этом мы заканчиваем рассмотрение обязательной части заголовка и впереди у нас –

Мне думается, что в ранних версиях Win32 этой части заголовка могло не быть вообще. Однако, разобравшись со значением всех этих полей, утверждаю, что в настоящее время эта часть заголовка является ОБЯЗАТЕЛЬНОЙ!

Теперь необходимо рассмотреть формат этой части заголовка. Она описана следующим образом:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    //
    // NT additional fields.
    //
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Reserved1;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
                DataDirectory [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

Как мы видим, и этот «необязательный заголовок» в свою очередь разбит на две части, стандартные поля и дополнительные поля для Windows NT (а также для Windows'95 и Windows'98). Давайте пройдёмся по всем этим полям.

Поле Magic

Итак, первое поле, Magic. В настоящее время оно не несёт какой-либо информационной нагрузки и всегда равно 0x10b. В файле winnt.h существует следующий макрос, подтверждающий этот вывод:

```
#define IMAGE_NT_OPTIONAL_HDR_MAGIC      0x10b
```

Поле MajorLinkerVersion

В этом поле записана старшая часть номера версии линкера, который использовался при создании программы.

Поле MinorLinkerVersion

В этом поле записана младшая часть номера версии линкера, который использовался при создании программы.

Например, в моих программах в этих полях записано 0x2 и 0x19 соответственно. Другими словами, я использую версию линкера 2.25.

Поле SizeOfCode

Поле SizeOfCode хранит суммарный размер всех разделов, содержащих код программы, округлённый к верхней границе. Так как в подавляющем большинстве случаев в программе есть всего один раздел с кодом, то обычно значение этого поля равно размеру раздела с кодом программы.

Поле SizeOfInitializedData

Судя по названию этого поля, оно должно хранить суммарный размер всех инициализированных данных программы.

Поле SizeOfUninitializedData

А это поле также судя по названию, содержит размер неинициализированных данных соответственно, то есть тех данных, для которых загрузчик выделяет место в памяти при загрузке программы, но которые не занимают никакого места в файле на диске.

А наличие следующих трех полей заставляет задуматься о том, правильно ли необязательный заголовок называть необязательным. Может ли существовать исполняемый файл, если у него отсутствует

Поле AddressOfEntryPoint

Это поле, что видно по его названию, содержит адрес, а точнее, относительный виртуальный адрес (RVA) точки входа в программу. Но раз адрес относительный, то должна быть и база, которая является точкой отсчёта. Эту базу хранит в себе поле ImageBase.

Поле BaseOfCode

Содержит относительный виртуальный адрес, с которого обычно начинаются программные секции файла, а

Поле BaseOfData

Содержит относительный виртуальный адрес, с которого начинаются секции данных. Обычно программные секции располагаются сразу после заголовков, до секций данных, а секци данных идут последними в памяти.

Нужно ли говорить о том, что базой программных секций и данных является значение в поле ImageBase? Но почему-то фирма Microsoft посчитала, что на этом стандартные поля должны закончиться, и относительные виртуальные адреса могут существовать без базы. ☺ Как бы то ни было, давайте примем этот факт к сведению и заметим, что в заголовочном файле winnt.h определён макрос IMAGE_SIZEOF_STD_OPTIONAL_HEADER. Ниже я привожу его описание в том виде, в каком он встречается в этом файле:

```
#define IMAGE_SIZEOF_STD_OPTIONAL_HEADER    28
```

Поле ImageBase

Когда линкер создаёт исполняемый файл, то он предполагает, что файл будет отображён в определённое место в памяти. Адрес этого места, так называемый базовый адрес, и хранится в поле ImageBase.

Естественно, что для более быстрого доступа каждый раздел программы начинается с адреса, кратного какой-то определённой величине. Эту величину определяет

Поле SectionAlignment.

Обычно в программах, созданных при помощи линкера фирмы Microsoft, эта величина равна 0x1000, а в программах, скомпонованных линкером фирмы Borland – 0x10000.

Данные, входящие в состав каждой секции, внутри секции также начинаются с адреса, кратного какой-то определённой величине. Эту величину содержит

Поле FileAlignment

По умолчанию значение этого поля равно 0x200. Причина, по которой выбрано именно это значение, достаточно очевидна – это размер сектора на диске.

Поле MajorOperatingSystemVersion

и

Поле MinorOperatingSystemVersion

Содержат соответственно старшую и младшую части номера самой старой версии операционной системы, которая позволяет запускать данный исполняемый файл. В подавляющем большинстве случаев это поле имеет значение 1.0. Но в таком случае остаются неясными значения, которые хранятся в

Поле MajorSubsystemVersion

и

Поле Minor SubsystemVersion

Эти два поля содержат самую старую версию ПОДсистемы, позволяюей использовать этот исполняемый файл. Но в таком случае для меня совершенно непонятно, что такое «система» и сто такое «подсистема». Как бы то ни было, в большинстве случаев в этих двух полях записано 4.0, что, по всей вероятности, соответствует Windows'95 (Windows'95 – это и есть Windows 4.0).

Но сейчас мне хотелось бы остановиться на следующем. Для программиста иногда хотелось бы иметь возможность каким-то образом помечать версии своих файлов. Для того, чтобы хранить старшую часть номера версии файла, существует

Поле MajorImageVersion

хранит младшую часть номера версии исполняемого файла, а

Поле MinorImageVersion

хранит младшую часть номера версии исполняемого файла.

Поле Reserved1

Как следует из его названия, пока не используется. В тех файлах, которые я просматривал, его значение всегда равно нулю.

Поле SizeOfImage

Хранит, что следует из его названия, общий размер загруженного модуля в памяти, начиная от базового адреса загрузки (ImageBase) до адреса конца последней секции, выровненного до соответствующего значе-

ния. Другими словами, значение этого поля должно быть кратно значению SectionAlignment.

В

Поле SizeOfHeaders

записан не суммарный размер всех заголовков файла, а общий размер всех заголовков файла (DOS, PE – FileHeader и OptionalHeader) и таблицы разделов.

Поле CheckSum

В этом поле содержится контрольная сумма исполняемого файла.

Достаточно интересным является

Поле Subsystem

В нём указывается тип подсистемы, которую данный исполняемый файл использует в качестве интерфейса с пользователем. По полю Subsystem возможно определить характер исполняемого модуля. Некоторые возможные значения поля Subsystem я приведу в таблице.

Подсистема	Значение	Описание
IMAGE_SUBSYSTEM_UNKNOWN	0	Подсистема неизвестна
IMAGE_SUBSYSTEM_NATIVE	1	Подсистема не требуется (например, для драйвера устройства)
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	Модуль откомпилирован для графического пользовательского интерфейса Windows
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	Модуль откомпилирован для символьного пользовательского интерфейса пользователя Windows (консоль)
IMAGE_SUBSYSTEM_OS2_CUI	5	Модуль откомпилирован для символьного пользовательского интерфейса OS/2
IMAGE_SUBSYSTEM_POSIX_CUI	7	Модуль откомпилирован для символьного пользовательского интерфейса POSIX

Я надеюсь, что после этой таблицы уже не требуется никаких объяснений, что представляет собой поле Subsystem.

Поле DLLCharacteristics

Это поле показывает, при каких обстоятельствах должна вызываться функция инициализации DLL. Поле может принимать следующие значения:

Характеристика	Значение	Описание
DLL_PROCESS_ATTACH	1	Вызов, когда DLL впервые загружается в адресное пространство процесса
DLL_THREAD_ATTACH	2	

Поле SizeOfStackReserve
и
Поле SizeOfStackCommit

работают в связке. Первое из них определяет размер необходимого стека, то есть размер резервируемого блока памяти. Однако используемым является только та часть стека, которая равна SizeOfStackCommit. Следующая страница стека называется «дежурной» страницей (guarded page). Когда используемый стек достигает «дежурной страницы», последняя становится рабочей, а следующая становится «дежурной». Так продолжается до тех пор, пока не будет исчерпан весь зарезервированный для стека размер памяти. Аналогичным образом работают поля SizeOfHeapReserve и SizeOfHeapCommit, только в данном случае речь идет не о стеке, а о куче.

Поле LoaderFlags

В настоящее время не употребляется.

Поле NumberOfRvaAndSizes

Мне, честно говоря, не совсем понятно назначение этого поля. Я думаю, что оно является отголоском более ранних форматов заголовка файла. Дело в том, что это поле должно определять число элементов в массиве структур типа IMAGE_DATA_DIRECTORY, который начинается вслед за этим полем. Но, с другой стороны, в заголовочном файле winnt.h есть макрос, который определяет число этих элементов:

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES   16
```

и именно этот макрос определяет число элементов массива в описании заголовка. Что же получается? С одной стороны, число элементов массива определяется в заголовке, а с другой стороны, он является постоянным значением? Как бы то ни было, отметим, что поле NumberOfRvaAndSizes содержит число элементов массива, упомянутого выше.

А что же это за массив, о котором я говорил выше? Дело в том, что этот массив определяет расположение данных в исполняемом файле. Элементами этого массива являются структуры типа IMAGE_DATA_DIRECTORY, описание которого я привожу:

```
typedef struct _IMAGE_DATA_DIRECTORY {
   DWORD  VirtualAddress;
   DWORD  Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Как читатель, вероятно, уже догадался, каждая из структур содержит виртуальный адрес того или иного раздела, а также размер этого раздела. Массив является позиционным, то есть информация о конкретном разделе (директории, секции, как их часто называет фирма Microsoft) *должна находиться в строго определённом элементе массива. Сведения о* том, какому порядковому элементу массива соответствует какой раздел, приведены в таблице ниже.

Макрос	Значение	Описание
IMAGE_DIRECTORY_ENTRY_EXPORT	0	Директория экспорта
IMAGE_DIRECTORY_ENTRY_IMPORT	1	Директория импорта
IMAGE_DIRECTORY_ENTRY_RESOURCE	2	Директория ресурсов
IMAGE_DIRECTORY_ENTRY_EXCEPTION	3	Директория исключений
IMAGE_DIRECTORY_ENTRY_SECURITY	4	Директория безопасности
IMAGE_DIRECTORY_ENTRY_BASERELOC	5	Таблица перемещений
IMAGE_DIRECTORY_ENTRY_DEBUG	6	Директория с отладочной информацией
IMAGE_DIRECTORY_ENTRY_COPYRIGHT	7	Строка описания
IMAGE_DIRECTORY_ENTRY_GLOBALPTR	8	
IMAGE_DIRECTORY_ENTRY_TLS	9	Директория локальной памяти потока
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	10	
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	11	
IMAGE_DIRECTORY_ENTRY_IAT	12	Таблица адресов импорта

Как мы видим, в настоящее время из шестнадцати возможных элементов массива используются только двенадцать.

Итак, я надеюсь, читатель в целом уже имеет представление о том, какого рода информация содержится в заголовке исполняемого файла, а также в некоторой степени о том, как развивался формат заголовка и какие несуразности присутствуют в нём и поныне. Что ж, наверное, в таких больших системах, каковыми являются Windows'95 иWindows NT, без ошибок не обойтись.

А теперь настало время написать программу, которая будет анализировать заголовок PE-файла и выдавать результаты в файл (подобие TDUMP из Borland C++) с именем PEHEADER.TXT, который будет располагаться в текущей директории. Наша программа должна предусматривать следующее:

- Возможность выбора имени файла;
- Проверку, является ли файл исполняемым (в данном случае мы будем проверять расширение файла и его сигнатуру, список расширений файлов будет ограничен .exe и .dll);
- Проверку, является ли файл PE-файлом.

В том случае, если файл не является исполняемым или не является PE-файлом, об этом должно быть выдано сообщение и программа должна быть готова к вводу нового имени файла.

В этой программе для доступа к содержимому файла я использую проецирование файла в память. Эта возможность стала доступной только в системах, поддерживающих Win32 API, то есть в Windows'95 и Windows NT. Для того, чтобы не нарушать логику повествования, рассказ о файлах, отображаемых в память, я вынес в приложение 1 к этой книге.

Это, наверное, будет самый длинный листинг в книге. Это объясняется тем, что структуры заголовков – самые объёмные структуры во всём исполняемом файле. Я постарался сократить программу, но, к сожалению, я не могу сократить объём заголовков PE-файла. Так что я заранее прошу читателя набраться терпения и изучить этот листинг. В программе нет никаких технических сложностей. Главная «сложность» – это перечислить все поля заголовков и при этом ни разу не ошибиться. ☺

Ниже приведён файл описания ресурсов к программе:

```
PEHeader_menu MENU
{
 POPUP "&File"
 {
  MENUITEM "&Open", IDM_OPEN
  MENUITEM SEPARATOR
  MENUITEM "E&xit", IDM_EXIT
 }
}
```

Теперь я приведу заголовочный файл программы:

```
#define IDM_OPEN 101
#define IDM_EXIT 102
#define nListWndId 10001
```

А теперь я прошу уважаемого читателя набраться терпения. Теперь – основной файл программы. Я настоятельно рекомендую изучить его, ибо в нём я активно использую все структуры из winnt.h, которые описывают формат заголовка исполняемого файла Windows (PE-файла):

```
#include <windows.h>
#include <stdio.h>
#include "peheader.h"

long WINAPI PEHeaderWndProc ( HWND, UINT, UINT, LONG );
HINSTANCE hInst;

int APIENTRY WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpszCmdParam, int nCmdShow )

// «Стандартное заклинание» понятно без комментариев
// и объяснений не требует.
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = (WNDPROC) PEHeaderWndProc;
  WndClass.cbClsExtra = 0;
  WndClass.cbWndExtra = 0;
  WndClass.hInstance =  hInstance ;
  WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
  WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
  WndClass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
  WndClass.lpszMenuName = "PEHeader_menu";
  WndClass.lpszClassName = "PEHeaderViewer";

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow("PEHeaderViewer", "PE-file's header viewer",
              WS_OVERLAPPEDWINDOW,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              CW_USEDEFAULT,
              NULL, NULL,
              hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
```

```
        return 0;
      }

/* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
   return Msg.wParam;
 }

long WINAPI PEHeaderWndProc ( HWND hWnd, UINT Message, UINT wParam,
                              LONG lParam )
 {

   static char lpstrFile[MAX_PATH] = "";
   char cBuffer[0x80];
   DWORD dwFileSize;
   HANDLE hFile, hMappedFile;
   LPVOID lpMappedFile;
   static HWND hListWnd;
   RECT Rect;

   PIMAGE_DOS_HEADER pIDH;
   int nOffset = 0;

   switch(Message)
    {
     case WM_CREATE:
// При создании окна необходимо предусмотреть создание поверх основного
// окна списка, совпадающего с основным окном и изменяющего свой размер
// вместе с размером основного окна.
       GetClientRect(hWnd, &Rect);
       hListWnd = CreateWindow("LISTBOX", NULL, WS_CHILD | WS_VISIBLE|
                               WS_VSCROLL | LBS_HASSTRINGS |
                               LBS_NOINTEGRALHEIGHT, 0, 0,
                               Rect.right, Rect.bottom, hWnd,
                               (HMENU) nListWndId, hInst, NULL);
// Если окно списка создать невозможно, то выдаём сообщение об этом и
// прекращаем работу программы.
       if(!hListWnd)
         {
          MessageBox(hWnd, "Cannot create Listbox window", "Error", MB_OK);
          PostQuitMessage(0);
         }
       return 1;
     case WM_COMMAND:
      switch(LOWORD(wParam))
```

```
        {
         case IDM_OPEN:
// Если нам необходимо открыть файл, выбираем файл при помощи
// стандартного диалога для открытия файлов.
          OPENFILENAME OpenFileName;
          OpenFileName.lStructSize = sizeof(OPENFILENAME);
          OpenFileName.hwndOwner = hWnd;
          OpenFileName.hInstance = hInst;
          OpenFileName.lpstrFilter = "Applications (*.exe)\0*.exe\0Application
                                                    Extension (*.dll)\0*.dll\0\0";
          OpenFileName.lpstrCustomFilter = NULL;
          OpenFileName.nFilterIndex = 0;
          OpenFileName.lpstrFile = lpstrFile;
          OpenFileName.nMaxFile = MAX_PATH;
          OpenFileName.lpstrFileTitle = NULL;
          OpenFileName.lpstrInitialDir = NULL;
          OpenFileName.lpstrTitle = "Open PE-files for viewing of header...";
          OpenFileName.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST |
                               OFN_HIDEREADONLY | OFN_LONGNAMES |
                               OFN_PATHMUSTEXIST;
          OpenFileName.lpstrDefExt = NULL;
          if(GetOpenFileName(&OpenFileName))
            {
             strcpy(lpstrFile, OpenFileName.lpstrFile);
// Открываем файл, выбранный в окне открытия файла.
             hFile = CreateFile(lpstrFile, GENERIC_READ, FILE_SHARE_READ|
                                FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL, NULL);
// Если файл открыт с ошибкой, то выдаём сообщение об ошибке.
             if(INVALID_HANDLE_VALUE == hFile)
               {
                sprintf(cBuffer, "Cannot open file %s", lpstrFile);
                MessageBox(hWnd, cBuffer, "Error", MB_OK);
               }
              else
               {
// Если файл открыт нормально, то тогда отображаем его в память.
// Об отображении файлов в память можно прочесть в приложении 1 к этой книге.
                dwFileSize = GetFileSize(hFile, NULL);
// Во-первых, создаём объект "проецируемый в память файл".
                hMappedFile = CreateFileMapping(hFile, NULL,
                                                PAGE_READONLY, 0, 0, NULL);
// После создания объекта "проецируемый в память файл" хэндл открытого
// файла нам больше не нужен и его можно закрыть.
                CloseHandle(hFile);
// Если объект "проецируемый в память файл" не создан или создан с ошибкой,
// то выдаём сообщение об ошибке и прекращаем выполнение программы.
                if(!hMappedFile)
                  {
                   sprintf(cBuffer, "Cannot create map of file %s", lpstrFile);
                   MessageBox(hWnd, cBuffer, "Error", MB_OK);
                  }
                 else
```

```
                    {
// Проецируем данные файла на адресное пространство процесса.
                    lpMappedFile = MapViewOfFile(hMappedFile,
                                                 FILE_MAP_READ, 0, 0, 0);
// После этого хэндл объекта "проецируемый в память файл" нам больше
// не нужен и его можно закрыть.
                    CloseHandle(hMappedFile);
// Если файл спроецирован с ошибкой, то выдаём сообщение об этом и
// прекращаем работу программы.
                    if(!lpMappedFile)
                      {
                       sprintf(cBuffer, "Cannot create mapped view of file %s",
                               lpstrFile);
                       MessageBox(hWnd, cBuffer, "Error", MB_OK);
                      }
                     else
                      {
// Проверяем, является ли файл исполняемым PE-файлом.
// 1. Проверяем, присутствует ли в файле сигнатура исполняемого файла DOS.
                       pIDH = (PIMAGE_DOS_HEADER) lpMappedFile;
                       if(pIDH->e_magic != IMAGE_DOS_SIGNATURE)
                         MessageBox(hWnd, "File isn't executable!", "Wrong file
                                    type!", MB_OK);
                        else
                         {
// Раскрываем заголовок DOS.
                          sprintf(cBuffer, "         0x%08x –
                                IMAGE_DOS_HEADER", nOffset);
                          SendMessage(hListWnd, LB_ADDSTRING,
                                      0, (LPARAM) cBuffer);
                          sprintf(cBuffer, "0x%08x - e_magic - %04x",
                                offsetof(IMAGE_DOS_HEADER, e_magic),
                                pIDH->e_magic);
                          SendMessage(hListWnd, LB_ADDSTRING,
                                      0, (LPARAM) cBuffer);
                          sprintf(cBuffer, "0x%08x - e_cblp - %04x",
                                offsetof(IMAGE_DOS_HEADER, e_cblp),
                                pIDH->e_cblp);
                          SendMessage(hListWnd, LB_ADDSTRING,
                                      0, (LPARAM) cBuffer);
                          sprintf(cBuffer, "0x%08x - e_cp - %04x",
                                offsetof(IMAGE_DOS_HEADER, e_cp),
                                pIDH->e_cp);
                          SendMessage(hListWnd, LB_ADDSTRING,
                                      0, (LPARAM) cBuffer);
                          sprintf(cBuffer, "0x%08x - e_crlc - %04x",
                                offsetof(IMAGE_DOS_HEADER, e_crlc),
                                pIDH->e_crlc);
                          SendMessage(hListWnd, LB_ADDSTRING,
                                      0, (LPARAM) cBuffer);
                          sprintf(cBuffer, "0x%08x - e_cparhdr - %04x",
                                offsetof(IMAGE_DOS_HEADER, e_cparhdr),
                                pIDH->e_cparhdr);
```

```
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_minalloc - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_minalloc),
                        pIDH->e_minalloc);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_maxalloc - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_maxalloc),
                        pIDH->e_maxalloc);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_ss - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_ss),
                        pIDH->e_ss);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_sp - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_sp),
                        pIDH->e_sp);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_csum - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_csum),
                        pIDH->e_csum);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_ip - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_ip),
                        pIDH->e_ip);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_cs - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_cs),
                        pIDH->e_cs);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_lfarlc - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_lfarlc),
                        pIDH->e_lfarlc);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_ovno - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_ovno),
                        pIDH->e_ovno);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_res[4]",
                        offsetof(IMAGE_DOS_HEADER, e_res));
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_oeminfo - %04x",
                        offsetof(IMAGE_DOS_HEADER, e_oeminfo),
```

```
                    pIDH->e_oeminfo);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_res2[10]",
                    offsetof(IMAGE_DOS_HEADER, e_res2));
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - e_lfanew - %08x",
                    offsetof(IMAGE_DOS_HEADER, e_lfanew),
                    pIDH->e_lfanew);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
// Проверяем, является ли файл исполняемым файлом Windows.
                if(pIDH->e_lfarlc >= 0x40)
                  {
                   PIMAGE_NT_HEADERS pINTH =
                       (PIMAGE_NT_HEADERS)((PBYTE) pIDH +
                       pIDH->e_lfanew);
                   if(pINTH->Signature != IMAGE_NT_SIGNATURE)
                     {
                      MessageBox(hListWnd, "This file there isn't PE-file",
                              "Wrong file type", MB_OK);
                      return 0;
                     }
                    else
                     {
                      nOffset += pIDH->e_lfanew;
                      sprintf(cBuffer, "       0x%08x –
                          IMAGE_NT_HEADERS", nOffset);
                      SendMessage(hListWnd, LB_ADDSTRING,
                              0, (LPARAM) cBuffer);
                      sprintf(cBuffer, "0x%08x - Signature - %08x",
                          nOffset +
                          offsetof(IMAGE_NT_HEADERS, Signature),
                              pINTH->Signature);
                      SendMessage(hListWnd, LB_ADDSTRING,
                              0, (LPARAM) cBuffer);
                      nOffset += sizeof(DWORD);
                      sprintf(cBuffer, "              0x%08x –
                          IMAGE_FILE_HEADER FileHeader", nOffset);
                      SendMessage(hListWnd, LB_ADDSTRING,
                              0, (LPARAM) cBuffer);
                      sprintf(cBuffer, "0x%08x - Machine - %04x",
                          nOffset +
                          offsetof(IMAGE_FILE_HEADER, Machine),
                              pINTH->FileHeader.Machine);
                      SendMessage(hListWnd, LB_ADDSTRING,
                              0, (LPARAM) cBuffer);
                      switch(pINTH->FileHeader.Machine)
                       {
                        case 0:
                         SendMessage(hListWnd, LB_ADDSTRING,
                                 0, (LPARAM) "
```

```
    IMAGE_FILE_MACHINE_UNKNOWN");
   break;
  case 0x14c:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_I386");
   break;
  case 0x14d:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_I486");
   break;
  case 0x14e:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_IPentium");
   break;
  case 0x160:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_R3000
    MIPS big-endian");
   break;
  case 0x162:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_R3000
    MIPS little-endian");
   break;
  case 0x163:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_R6000
    MIPS Mark II");
   break;
  case 0x166:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_R4000
    MIPS little-endian");
   break;
  case 0x168:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_R10000
    MIPS little-endian");
   break;
  case 0x184:
   SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) "
    IMAGE_FILE_MACHINE_ALPHA");
   break;
  case 0x1f0:
```

```
        SendMessage(hListWnd, LB_ADDSTRING,
                    0, (LPARAM) "
          IMAGE_FILE_MACHINE_POWERPC");
        break;
      case 0x268:
        SendMessage(hListWnd, LB_ADDSTRING,
                    0, (LPARAM) "
          MOTOROLA 68000");
        break;
      case 0x290:
        SendMessage(hListWnd, LB_ADDSTRING,
                    0, (LPARAM) "
          PA RISC");
        break;
    }
    sprintf(cBuffer, "0x%08x - NumberOfSections –
          %04x",
          nOffset +
          offsetof(IMAGE_FILE_HEADER,
                NumberOfSections),
          pINTH->FileHeader.NumberOfSections);
    SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
    sprintf(cBuffer, "0x%08x - TimeDateStamp –
          %08x",
          nOffset +
          offsetof(IMAGE_FILE_HEADER,
                TimeDateStamp),
          pINTH->FileHeader.TimeDateStamp);
    SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
    sprintf(cBuffer, "0x%08x –
          PointerToSymbolTable - %08x",
          nOffset +
          offsetof(IMAGE_FILE_HEADER,
                PointerToSymbolTable),
          pINTH->FileHeader.PointerToSymbolTable);
    SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
    sprintf(cBuffer, "0x%08x –
          NumberOfSymbols - %08x",
          nOffset +
          offsetof(IMAGE_FILE_HEADER,
                NumberOfSymbols),
          pINTH->FileHeader.NumberOfSymbols);
    SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
    sprintf(cBuffer, "0x%08x –
          SizeOfOptionalHeader - %04x",
          nOffset +
          offsetof(IMAGE_FILE_HEADER,
                SizeOfOptionalHeader),
          pINTH->FileHeader.SizeOfOptionalHeader);
```

```
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                sprintf(cBuffer, "0x%08x - Characteristics –
                    %04x",
                    nOffset +
                    offsetof(IMAGE_FILE_HEADER,
                        Characteristics),
                     pINTH->FileHeader.Characteristics);
                SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
                WORD i = 0x0001;
                while(i)
                 {
                  if(pINTH->FileHeader.Characteristics & i)
                  switch(i)
                   {
                    case 0x0001:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
                (LPARAM) "   IMAGE_FILE_RELOCS_STRIPPED");
                     break;
                    case 0x0002:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
              (LPARAM) "   IMAGE_FILE_EXECUTABLE_IMAGE");
                     break;
                    case 0x0004:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
            (LPARAM) "   IMAGE_FILE_LINE_NUMS_STRIPPED");
                     break;
                    case 0x0008:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
          (LPARAM) "   IMAGE_FILE_LOCAL_SYMS_STRIPPED");
                     break;
                    case 0x0080:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
           (LPARAM) "   IMAGE_FILE_BYTES_REVERSED_LO");
                     break;
                    case 0x0100:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
                 (LPARAM) "   IMAGE_FILE_32BIT_MACHINE");
                     break;
                    case 0x0200:
                     SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
              (LPARAM) "   IMAGE_FILE_DEBUG_STRIPPED");
                     break;
                    case 0x1000:
                     SendMessage(hListWnd,
```

```
                                LB_ADDSTRING, 0,
                        (LPARAM) "    IMAGE_FILE_SYSTEM");
                    break;
                  case 0x2000:
                    SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
                            (LPARAM) "    IMAGE_FILE_DLL");
                    break;
                  case 0x8000:
                    SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
          (LPARAM) "    IMAGE_FILE_BYTES_REVERSED_HI");
                    break;
                  default:
                    SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
(LPARAM) "    IMAGE_FILE_UNKNOWN_CHARACTERISTICS");
                    break;
                  }
                 i <<= 1;
                }
              nOffset += IMAGE_SIZEOF_FILE_HEADER;
              sprintf(cBuffer, "        0x%08x –
      IMAGE_OPTIONAL_HEADER OptionalHeader", nOffset);
              SendMessage(hListWnd, LB_ADDSTRING,
                          0, (LPARAM) cBuffer);
              sprintf(cBuffer, "0x%08x - Magic - %04x",
                    nOffset +
                    offsetof(IMAGE_OPTIONAL_HEADER,
                    Magic),
                              pINTH->OptionalHeader.Magic);
              SendMessage(hListWnd, LB_ADDSTRING,
                          0, (LPARAM) cBuffer);
              sprintf(cBuffer, "0x%08x - MajorLinkerVersion -
                    %02x",
                     nOffset +
                    offsetof(IMAGE_OPTIONAL_HEADER,
                          MajorLinkerVersion),
                 pINTH->OptionalHeader.MajorLinkerVersion);
              SendMessage(hListWnd, LB_ADDSTRING,
                          0, (LPARAM) cBuffer);
              sprintf(cBuffer, "0x%08x - MinorLinkerVersion -
                    %02x",
                    nOffset +
                     offsetof(IMAGE_OPTIONAL_HEADER,
                          MinorLinkerVersion),
                 pINTH->OptionalHeader.MinorLinkerVersion);
              SendMessage(hListWnd, LB_ADDSTRING,
                          0, (LPARAM) cBuffer);
              sprintf(cBuffer, "0x%08x - SizeOfCode –
                    %08x",
                    nOffset +
                    offsetof(IMAGE_OPTIONAL_HEADER,
```

```
                SizeOfCode),
                pINTH->OptionalHeader.SizeOfCode);
SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
sprintf(cBuffer, "0x%08x - SizeOfInitializedData –
        %08x",
        nOffset +
        offsetof(IMAGE_OPTIONAL_HEADER,
                SizeOfInitializedData),
      pINTH->OptionalHeader.SizeOfInitializedData);
SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
sprintf(cBuffer, "0x%08x - SizeOfUninitializedData –
        %08x",
        nOffset +
        offsetof(IMAGE_OPTIONAL_HEADER,
                SizeOfUninitializedData),
   pINTH->OptionalHeader.SizeOfUninitializedData);
SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
sprintf(cBuffer, "0x%08x - AddressOfEntryPoint –
        %08x",
        nOffset +
        offsetof(IMAGE_OPTIONAL_HEADER,
                AddressOfEntryPoint),
      pINTH->OptionalHeader.AddressOfEntryPoint);
SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
sprintf(cBuffer, "0x%08x - BaseOfCode –
        %08x",
        nOffset +
        offsetof(IMAGE_OPTIONAL_HEADER,
        BaseOfCode),
                pINTH->OptionalHeader.BaseOfCode);
SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
sprintf(cBuffer, "0x%08x - BaseOfData –
        %08x",
        nOffset +
        offsetof(IMAGE_OPTIONAL_HEADER,
        BaseOfData),
                pINTH->OptionalHeader.BaseOfData);
sprintf(cBuffer, "0x%08x - ImageBase –
        %08x",
        nOffset +
        offsetof(IMAGE_OPTIONAL_HEADER,
                ImageBase),
                pINTH->OptionalHeader.ImageBase);
SendMessage(hListWnd, LB_ADDSTRING,
                0, (LPARAM) cBuffer);
sprintf(cBuffer, "0x%08x - SectionAlignment –
        %08x",
        nOffset +
```

```
                offsetof(IMAGE_OPTIONAL_HEADER,
                        SectionAlignment),
                  pINTH->OptionalHeader.SectionAlignment);
        sprintf(cBuffer, "0x%08x - FileAlignment –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        FileAlignment),
                    pINTH->OptionalHeader.FileAlignment);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x –
                MajorOperatingSystemVersion -- %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        MajorOperatingSystemVersion),
    pINTH->OptionalHeader.MajorOperatingSystemVersion);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x –
                MinorOperatingSystemVersion -- %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        MinorOperatingSystemVersion),
     pINTH->OptionalHeader.MinorOperatingSystemVersion);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - MajorImageVersion –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        MajorImageVersion),
                 pINTH->OptionalHeader.MajorImageVersion);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - MinorImageVersion –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        MinorImageVersion),
                pINTH->OptionalHeader.MinorImageVersion);
        SendMessage(hListWnd, LB_ADDSTRING,
                         0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - MajorSubsystemVersion –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        MajorSubsystemVersion),
          pINTH->OptionalHeader.MajorSubsystemVersion);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - MinorSubsystemVersion –
                %08x",
```

```
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        MinorSubsystemVersion),
          pINTH->OptionalHeader.MinorSubsystemVersion);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - DWORD Reserved1 –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        Reserved1),
                         pINTH->OptionalHeader.Reserved1);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - SizeOfImage –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                        SizeOfImage),
                       pINTH->OptionalHeader.SizeOfImage);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - SizeOfHeaders –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                         SizeOfHeaders),
                    pINTH->OptionalHeader.SizeOfHeaders);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        sprintf(cBuffer, "0x%08x - CheckSum –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                         CheckSum),
                         pINTH->OptionalHeader.CheckSum);
        SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
        for(int i = 0; i < 8; i++)
         {
          if(pINTH->OptionalHeader.Subsystem == i)
            switch(i)
             {
              case 1:
               SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
            (LPARAM) "   IMAGE_SUBSYSTEM_NATIVE");
               break;
              case 2:
               SendMessage(hListWnd,
                                LB_ADDSTRING, 0,
(LPARAM) "   IMAGE_SUBSYSTEM_WINDOWS_GUI");
               break;
```

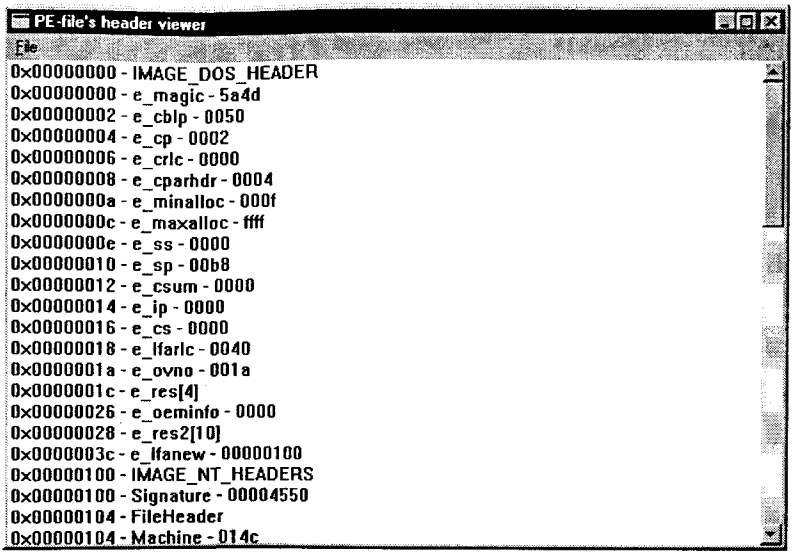
```
                case 3:
                  SendMessage(hListWnd,
                              LB_ADDSTRING, 0,
(LPARAM) "   IMAGE_SUBSYSTEM_WINDOWS_CUI");
                  break;
                case 5:
                  SendMessage(hListWnd,
                              LB_ADDSTRING, 0,
    (LPARAM) "   IMAGE_SUBSYSTEM_OS2_CUI");
                  break;
                case 7:
                  SendMessage(hListWnd,
                              LB_ADDSTRING, 0,
  (LPARAM) "   IMAGE_SUBSYSTEM_POSIX_CUI");
                  break;
                default:
                  SendMessage(hListWnd,
                              LB_ADDSTRING, 0,
  (LPARAM) "   IMAGE_SUBSYSTEM_UNKNOWN");
                  break;
              }
            }
            sprintf(cBuffer, "0x%08x - DllCharacteristics –
                %04x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                         DllCharacteristics),
                pINTH->OptionalHeader.DllCharacteristics);
            SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
            sprintf(cBuffer, "0x%08x - SizeOfStackReserve –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                         SizeOfStackReserve),
                pINTH->OptionalHeader.SizeOfStackReserve);
            SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
            sprintf(cBuffer, "0x%08x - SizeOfStackCommit –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                         SizeOfStackCommit),
                pINTH->OptionalHeader.SizeOfStackCommit);
            SendMessage(hListWnd, LB_ADDSTRING,
                        0, (LPARAM) cBuffer);
            sprintf(cBuffer, "0x%08x - SizeOfHeapReserve –
                %08x",
                nOffset +
                offsetof(IMAGE_OPTIONAL_HEADER,
                         SizeOfHeapReserve),
                pINTH->OptionalHeader.SizeOfHeapReserve);
            SendMessage(hListWnd, LB_ADDSTRING,
```

```
                                        0, (LPARAM) cBuffer);
                                sprintf(cBuffer, "0x%08x - SizeOfHeapCommit –
                                        %08x",
                                        nOffset +
                                        offsetof(IMAGE_OPTIONAL_HEADER,
                                                SizeOfHeapCommit),
                                         pINTH->OptionalHeader.SizeOfHeapCommit);
                                SendMessage(hListWnd, LB_ADDSTRING,
                                                0,  (LPARAM) cBuffer);
                                sprintf(cBuffer, "0x%08x - LoaderFlags –
                                        %08x",
                                        nOffset +
                                        offsetof(IMAGE_OPTIONAL_HEADER,
                                                 LoaderFlags),
                                                pINTH->OptionalHeader.LoaderFlags);
                                SendMessage(hListWnd, LB_ADDSTRING,
                                                0, (LPARAM) cBuffer);
                                sprintf(cBuffer, "0x%08x - LoaderFlags –
                                        %08x",
                                         nOffset +
                                         offsetof(IMAGE_OPTIONAL_HEADER,
                                                 LoaderFlags),
                                                pINTH->OptionalHeader.LoaderFlags);
                                SendMessage(hListWnd, LB_ADDSTRING,
                                                0, (LPARAM) cBuffer);
                                sprintf(cBuffer, "0x%08x – NumberOfRvaAndSizes –
                                        %08x",
                                         nOffset +
                                         offsetof(IMAGE_OPTIONAL_HEADER,
                                                 NumberOfRvaAndSizes),
                                   pINTH->OptionalHeader.NumberOfRvaAndSizes);
                                SendMessage(hListWnd, LB_ADDSTRING,
                                                0, (LPARAM) cBuffer);
                                sprintf(cBuffer, "0x%08x –
            DataDirectory[IMAGE_NUMBEROF_DIRECTORY__ENTRIES]", nOffset);
                                SendMessage(hListWnd, LB_ADDSTRING,
                                                0, (LPARAM) cBuffer);
                            }
                        }
                    }
                }
            }
        }
      return 0;
    case IDM_EXIT:
    SendMessage(hWnd, WM_DESTROY, 0, 0);
    default:
      return DefWindowProc(hWnd,Message,wParam, lParam);
  }
case WM_DESTROY:
  PostQuitMessage(0);
  return 0;
```

```
    default:
      return DefWindowProc(hWnd,Message,wParam, lParam);
  }
}
```

В качестве упражнения я оставляю читателю возможность раскрыть характеристики последнего поля «необязательного» заголовка - DataDirectory[IMAGE_NUMBEROF_DIRECTORY__ENTRIES].

Внешний вид окна, создаваемого этой программой, я привожу ниже:



Уважаемый читатель! Я просто устал от этого заголовка! Автор – он ведь тоже живой человек и ему тоже могут наскучить описания бесконечных полей! Я очень надеюсь, что нам больше не встретятся столь длинные структуры и мне не придётся тратить на одну тему столько времени. Я попробовал представить, как бы чувствовали себя студенты после лекции только лишь о заголовках PE-файла! ☺ Давайте пока немного прервёмся, автор выпьет чашечку кофе, после чего мы продолжим наш разговор о формате исполняемого файла и поговорим о той структуре, которая называется

Таблица объектов (object table)

Итак, давайте подведём некоторые итоги того, о чём говорилось в предыдущей главе. Как мне кажется, главный итог состоит в том, что мы, помимо назначения каждого поля заголовка исполняемого файла, узнали о том, что исполняемый файл состоит из разделов, причём в каждом из разделов хранится информация строго определённого назначения. Для того

чтобы описать расположение этих разделов, используется так называемая таблица объектов.

По моему мнению, это название - таблица объектов - не совсем верно. Но если уж так такое наименование применяется в фирме Microsoft, то я буду придерживаться их терминологии.

Под объектом (разделом) в данном случае понимается совокупность данных определенного назначения, например, об экспортируемых или импортируемых функциях, ресурсах, элементах перемещений (relocations) и так далее, которые компактно размещены в исполняемом файле. Соответственно таблица объектов - это данные, описывающие размещение объектов (разделов) в исполняемом файле и в памяти после загрузки файла.

Другими словами, таблица объектов – это просто оглавление разделов. Таблица объектов размещается сразу же после заголовка PE-файла. В очередной раз перед нами встаёт вопрос, – а как нам узнать смещение начала таблицы объектов? В общем задача достаточно тривиальна – взять смещение заголовка PE-файла и прибавить к нему размер заголовка. Но неужели же придётся вручную считать размер всех заголовков PE-файла или использовать длинные вычисления? Я думаю, что читатель уже догадался, что ответ будет отрицательным. И опять на помощь нам приходит макрос. Я приведу его описание в том виде, в котором он появляется в файле winnt.h:

```
#define IMAGE_FIRST_SECTION(ntheader) ((PIMAGE_SECTION_HEADER)
((DWORD)ntheader +FIELD_OFFSET(IMAGE_NT_HEADERS, OptionalHeader) +
((PIMAGE_NT_HEADERS)(ntheader))->FileHeader.SizeOfOptionalHeader))
```

Другими словами, для того, чтобы получить смещение таблицы разделов, нам необходимо знать всего лишь смещение заголовка PE-файла. При помощи макроса, упомянутого выше, можно легко вычислить смещение таблицы разделов.

Теперь я попрошу читателя порассуждать вместе со мной. Логично предположить, что описание каждого раздела должно быть похожим на описание других разделов. Если бы это было не так, то тогда программисту при создании собственного раздела пришлось бы придумывать и формат его описания. Но ведь сейчас этого делать не приходится! Значит, наше предположение о том, что описания разделов имеют один и тот же формат, соответствует действительности. А если формат один и тот же, но, наверное, вся таблица объектов представляет собой массив! Теперь остаётся два вопроса – что представляет собой каждый элемент этого массива и где найти число элементов этого массива.

Ответ на второй вопрос (о числе элементов массива) находится очень просто. Надеюсь, читатель помнит поле NumberOfSections, которое находится в «обязательном» заголовке файла? Так вот, это поле и определяет число элементов в массиве описания разделов. «А ларчик просто открывался», как говорится в одной из басен И.А. Крылова. ☺

Ответ на второй вопрос (о формате элементов массива описания разделов) несколько более сложен, но и он не представляет особой трудности. Как всегда, нашим помощником явится заголовочный файл winnt.h.

Порывшись в нём, мы без труда найдём описание структуры IMAGE_SECTION_HEADER (заголовок образа раздела). Это описание фактически является и описанием элемента массива, то есть описывает раздел исполняемого файла. Ниже я привожу описание структуры IMAGE_SECTION_HEADER так, как оно приведено в заголовочном файле winnt.h:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE   Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
       DWORD  PhysicalAddress;
       DWORD  VirtualSize;
  } Misc;
  DWORD  VirtualAddress;
  DWORD  SizeOfRawData;
  DWORD  PointerToRawData;
  DWORD  PointerToRelocations;
  DWORD  PointerToLinenumbers;
  WORD   NumberOfRelocations;
  WORD   NumberOfLinenumbers;
  DWORD  Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Размер этого заголовка определен в виде макроса IMAGE_SIZEOF_SECTION_HEADER и принимает значение 40. Насколько я могу судить, наличие подобных макросов говорит о том, что все значения могут в недалеком будущем поменяться. В случае использования подобных макросов программисту не придется заново переписывать свои приложения.

Давайте посмотрим, что мы можем выжать из этого заголовка. Прежде всего,

Массив Name

Этот массив определяется первым. Этот массив является символьным и содержит наименование раздела программы. Когда я постарался выяснить, какой же длины этот массив, то в файле winnt.h я нашёл макрос IMAGE_SIZEOF_SHORT_NAME, который определён следующим образом:

```
#define IMAGE_SIZEOF_SHORT_NAME          8
```

Другими словами, длина массива Name постоянна и равна восьми байтам. Что из этого следует? Ничего, кроме того, что имя раздела исполняемого файла не может быть более восьми символов. И в этом месте исследователя исполняемого файла подстерегает одна ловушка.

Если число символов в имени раздела меньше восьми, то оно дополняется справа нулями. В этом случае имя раздела можно считать ASCIIZ-строкой (не строкой в Unicode!). НО! Я прошу читателя обратить внимание на следующее. Практически в любой публикации по формату исполняемого файла (такие публикации можно без труда найти в Интернете) говорится, что имя раздела заканчивается нулевым кодом.

НЕВЕРНО! Если число символов в имени равно восьми, то тогда никаких нулей НЕ ДОБАВЛЯЕТСЯ и попытка проанализировать или вывести на отображение строку, указывающую на имя раздела, с большой долей вероятности приведет к ошибке. Я НАСТОЯТЕЛЬНО рекомендую принять к сведению это утверждение!

Далее идёт

Объединение Misc

Поля этого объединения имеют различное назначение в зависимости от того, находятся ли они в объектном или же в исполняемом файле. В этой книге мы не рассматриваем объектные файлы, поэтому я остановлюсь только на значении этого поля в исполняемом файле. В исполняемом файле это поле содержит (точнее, должно содержать) размер раздела (НЕОКРУГЛЁННЫЙ!). Но, судя по всему, в файлах, скомпилированных при помощи компилятора Borland, это поле содержит округлённое значение, при этом оно округляется до значения, записанного в поле SectionAlignment «необязательного» заголовка. Это и есть одно из тех исключений, которые только подтверждают правила. Перефразируя известную фразу, мы можем сказать, что «формат файла – не догма, а руководство к действию».

Это поле работает в определённом смысле в связке с полем SizeOfRawData, о котором мы поговорим ниже.

Поле VirtualAddress

Это поле имеет очень важное значение, ибо содержит смещение, по которому загрузчик должен отобразить раздел. Это смещение вычисляется относительно базового адреса загрузки, то есть того значения, которое указано в поле ImageBase «необязательного» заголовка.

Выше я уже упоминал о том, что мы должны рассмотреть

Поле SizeOfRawData

В файлах, созданных при помощи средств Microsoft, это поле содержит размер раздела, округлённый до значения, записанного в поле FileAlignment «необязательного» заголовка. Возникает один только вопрос —как в компиляторах фирмы Borland определить точный размер раздела? Я хотел бы переадресовать этот вопрос представителям этой фирмы.

Поле PointerToRawData

Значение этого поля также по-разному описывается в разных документах. Чего только о нём не пишут! Я беру на себя смелость заявить следующее – это поле содержит смещение начала раздела относительно начала файла. Только и всего. Не более. И из этого заявления есть одно КРАЙНЕ ВАЖНОЕ СЛЕДСТВИЕ: при анализе содержимого файла должна появиться поправка, которая будет учитывать разность между значением

поля VirtualAddress и полем PointerToRawData. При написании программ просмотра различных частей исполняемого файла нам придётся воспользоваться этим свойством PE-файлов.

Следующее на очереди -

Поле PointerToRelocations

В exe-файлах не несёт какой-либо информационной нагрузки и устанавливается в нуль. Информация о необходимых перемещениях записывается в раздел перемещений. Думаю, это ещё одно поле, напоминающее нам о том, что заголовок исполняемого файла перерабатывался неоднократно и что некоторые поля в нём оставлены только для совместимости с ранее написанными программами. Из этого следует, что

Поле NumberOfRelocations

в исполняемых файлах также не используется. В тех файлах, которые я просматривал, поля PointerToRelocations и NumberOfRelocations имеют нулевые значения.

Поле PointerToLinenumbers

Ещё одно пример того, что многие поля уже не используются или могут не использоваться. Судя по названию этого поля, в нём должен находиться указатель на таблицу строк. Таблица строк используется для отладки и ставит номера строк исходной программы в соответствие тем смещениям, по которым в исполняемом файле находится код, сформированный для этих строк. В настоящее время отладочная информация приписывается компилятором в конец исполняемого файла.

После сказанного выше становится вполне понятно, что и

Поле NumberofLinenumbers

Практически не используется.

Поле Characteristics

Это поле определяет атрибуты раздела и используется как логическая шкала. Все возможные атрибуты приведены в заголовочном файле winnt.h. Я привожу эти описания вместе с теми описаниями, которые фактически удалены из файла, то есть сделаны комментариями. Возможно, и информация в комментариях кому-нибудь окажется полезной.

Характеристика	Значение	Назначение
IMAGE_SCN_TYPE_REG	0x00000000	Зарезервировано (комментарий)
IMAGE_SCN_TYPE_DSECT	0x00000001	Зарезервировано (комментарий)
IMAGE_SCN_TYPE_NOLOAD	0x00000002	Зарезервировано (комментарий)
IMAGE_SCN_TYPE_GROUP	0x00000004	Зарезервировано (комментарий)
IMAGE_SCN_TYPE_NO_PAD	0x00000008	Зарезервировано (комментарий)
IMAGE_SCN_TYPE_COPY	0x00000010	Зарезервировано (комментарий)
IMAGE_SCN_CNT_CODE	0x00000020	Раздел содержит программный код

IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Раздел содержит инициализированные данные
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Раздел содержит неинициализированные данные
IMAGE_SCN_LNK_OTHER	0x00000100	Зарезервировано
IMAGE_SCN_LNK_INFO	0x00000200	Раздел содержит комментарии или какой-нибудь другой тип информации
IMAGE_SCN_TYPE_OVER	0x00000400	Зарезервировано
IMAGE_SCN_LNK_REMOVE	0x00000800	Содержимое раздела не должно быть помещено в исполняемый файл.
IMAGE_SCN_LNK_COMDAT	0x00001000	Раздел содержит «comdat» (может быть, общие данные?)
	0x00002000	Зарезервировано
IMAGE_SCN_MEM_PROTECTED	0x00004000	Устарело (комментарий)
IMAGE_SCN_MEM_FARDATA	0x00008000	
IMAGE_SCN_MEM_SYSHEAP	0x00010000	Устарело (комментарий)
IMAGE_SCN_MEM_PURGEABLE	0x00020000	
IMAGE_SCN_MEM_16BIT	0x00020000	
IMAGE_SCN_MEM_LOCKED	0x00040000	
IMAGE_SCN_MEM_PRELOAD	0x00080000	
IMAGE_SCN_ALIGN_1BYTES	0x00100000	
IMAGE_SCN_ALIGN_2BYTES	0x00200000	
IMAGE_SCN_ALIGN_4BYTES	0x00300000	
IMAGE_SCN_ALIGN_8BYTES	0x00400000	
IMAGE_SCN_ALIGN_16BYTES	0x00500000	Выравнивание по умолчанию, если не указано другого.
IMAGE_SCN_ALIGN_32BYTES	0x00600000	
IMAGE_SCN_ALIGN_64BYTES	0x00700000	
Не используется	0x00800000	
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	Секция содержит дополнительные данные о таблице перемещений
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	Раздел не должен кэшироваться
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	Раздел не должен разбиваться на страницы
IMAGE_SCN_MEM_SHARED	0x10000000	Раздел является совместно используемым
IMAGE_SCN_MEM_EXECUTE	0x20000000	Раздел является исполняемым, то есть помечен «для исполнения»
IMAGE_SCN_MEM_READ	0x40000000	Раздел после загрузки помечен «для чтения».
IMAGE_SCN_MEM_WRITE	0x80000000	Раздел после загрузки в память помечен «для записи». Если этот флаг не установлен, то загрузчик должен установить в памяти для страниц раздела атрибуты «только для чтения» или «только для исполнения»

А теперь настало время для демонстрационной программы. Я очень рассчитываю на то, что читатель отнесётся с пониманием к некоторым длиннотам программы. Перебрать все возможные значения параметров и вместо них вывести их словесные обозначения.... Это не сложно, это просто длинно и достаточно надоедливо. Что ж, не только красота, но и наука требует жертв.

Ниже я привожу текст заголовочного файла, используемого демонстрационной программой:

```
#define IDM_OPEN 101
#define IDM_EXIT 102

#define nListWndId 10001
```

Программа использует следующие ресурсы:

```
#define IDM_EXIT   102
#define IDM_OPEN 101

ObjectTable_menu MENU
{
 POPUP "&File"
 {
  MENUITEM "&Open", IDM_OPEN
  MENUITEM SEPARATOR
  MENUITEM "E&xit", IDM_EXIT
 }
}
```

И, наконец, непосредственно текст программы:

```
#include <windows.h>
#include <stdio.h>
#include "objtable.h"

long WINAPI ObjTableWndProc( HWND, UINT, UINT, LONG );
LPVOID MapFile(LPTSTR);
BOOL TestFile(PIMAGE_DOS_HEADER);
VOID FillList(HWND, WORD, PIMAGE_SECTION_HEADER, int);

HINSTANCE hInst;

int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow )
 {
   HWND hWnd ;
   WNDCLASS WndClass ;
   MSG Msg;

   hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = (WNDPROC) ObjTableWndProc;
   WndClass.cbClsExtra = 0;
   WndClass.cbWndExtra = 0;
   WndClass.hInstance =  hInstance ;
   WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
   WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
   WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
```

```
  WndClass.lpszMenuName = "ObjectTable_menu";
  WndClass.lpszClassName = "ObjTableViewer";

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow("ObjTableViewer", "Object table's viewer",
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

long WINAPI ObjTableWndProc(HWND hWnd, UINT Message, UINT wParam,
                            LONG lParam )
 {
  static char lpstrFile[MAX_PATH] = "";
  PIMAGE_DOS_HEADER pIDH;
  RECT Rect;
  static HWND hListWnd;

  switch(Message)
   {
    case WM_CREATE:
     GetClientRect(hWnd, &Rect);
     hListWnd = CreateWindow("LISTBOX", NULL,
                             WS_CHILD | WS_VISIBLE | WS_VSCROLL |
                             LBS_HASSTRINGS | LBS_NOINTEGRALHEIGHT,
                             0, 0, Rect.right, Rect.bottom, hWnd,
                             (HMENU) nListWndId, hInst, NULL);
```

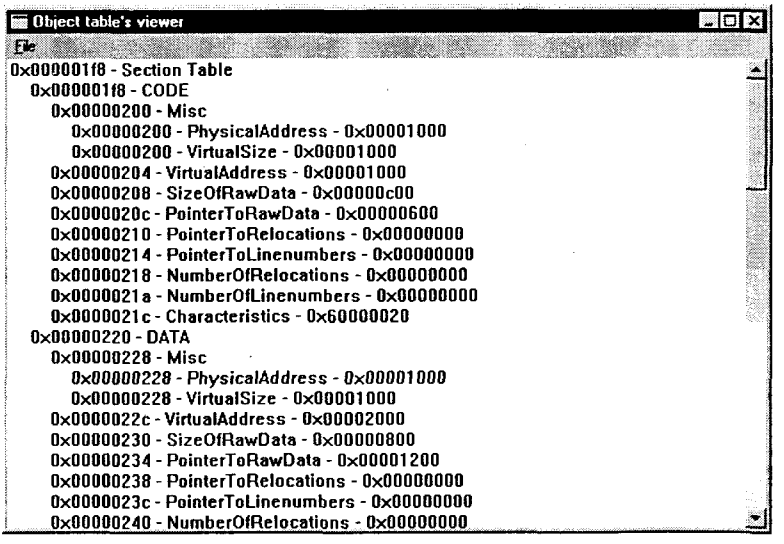
```
 if(!hListWnd)
    {
     MessageBox(hWnd, "Невозможно создать окно списка", "Ошибка",
                MB_OK);
     PostQuitMessage(0);
    }
 return 1;
case WM_COMMAND:
 switch(LOWORD(wParam))
  {
   case IDM_OPEN:
    OPENFILENAME OpenFileName;
    OpenFileName.lStructSize = sizeof(OPENFILENAME);
    OpenFileName.hwndOwner = hWnd;
    OpenFileName.hInstance = hInst;
    OpenFileName.lpstrFilter = "Applications (*.exe)\0*.exe\0
                                Application Extension (*.dll)\0*.dll\0\0";
    OpenFileName.lpstrCustomFilter = NULL;
    OpenFileName.nFilterIndex = 0;
    OpenFileName.lpstrFile = lpstrFile;
    OpenFileName.nMaxFile = MAX_PATH;
    OpenFileName.lpstrFileTitle = NULL;
    OpenFileName.lpstrInitialDir = NULL;
    OpenFileName.lpstrTitle = "Open PE-files for viewing of section table...";
    OpenFileName.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST |
                         OFN_HIDEREADONLY | OFN_LONGNAMES |
                         OFN_PATHMUSTEXIST;
    OpenFileName.lpstrDefExt = NULL;
    if(!GetOpenFileName(&OpenFileName))
      MessageBox(hWnd, "Невозможно открыть файл",
                 "Ошибка открытия файла", MB_OK);
     else
      {
       if( (pIDH = (PIMAGE_DOS_HEADER)
            MapFile(OpenFileName.lpstrFile)) == 0)
         MessageBox(hWnd, "Невозможно спроецировать файл в память",
                    "Ошибка проецирования файла", MB_OK);
        else
         {
          if(!TestFile(pIDH))
            MessageBox(hWnd, "Файл не является PE-файлом",
                       "Ошибка", MB_OK);
           else
            {
             PIMAGE_NT_HEADERS pINTH =
                        (PIMAGE_NT_HEADERS) ((PBYTE) pIDH +
                        pIDH->e_lfanew);
             PIMAGE_SECTION_HEADER pISH =
                        IMAGE_FIRST_SECTION(pINTH);
             int nOffset = (int) pISH - (DWORD) pIDH;
             FillList(hListWnd, pINTH->FileHeader.NumberOfSections,
                      pISH, nOffset);
            }
```

```
                UnmapViewOfFile((LPVOID) pIDH);
              }
            }
           return 0;
          case IDM_EXIT:
           SendMessage(hWnd, WM_DESTROY, 0, 0);
           default:
          return DefWindowProc(hWnd,Message,wParam, lParam);
        }
      case WM_DESTROY:
       PostQuitMessage(0);
       return 0;
      default:
       return DefWindowProc(hWnd,Message,wParam, lParam);
    }
}

LPVOID MapFile(LPTSTR lpstrFile)
 {
  HANDLE hFile, hMappedFile;
  LPVOID lpMappedFile;

  hFile = CreateFile(lpstrFile, GENERIC_READ,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                     NULL);
  if(INVALID_HANDLE_VALUE == hFile)
     return 0;
    else
     {
      hMappedFile = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0,
                                      NULL);
      CloseHandle(hFile);
      if(!hMappedFile)
         return 0;
        else
         {
          lpMappedFile = MapViewOfFile(hMappedFile, FILE_MAP_READ,
                                       0, 0, 0);
          CloseHandle(hMappedFile);
          if(!lpMappedFile)
             return 0;
            else
             return lpMappedFile;
         }
     }
}

BOOL TestFile(PIMAGE_DOS_HEADER pIDH)
 {
  if(pIDH->e_magic != IMAGE_DOS_SIGNATURE)
     return FALSE;
    else
```

```
    if(pIDH->e_lfarlc >= 0x40)
      {
       PIMAGE_NT_HEADERS pINTH = (PIMAGE_NT_HEADERS)
                                        ((PBYTE) pIDH + pIDH->e_lfanew);
       if(pINTH->Signature != IMAGE_NT_SIGNATURE)
         return FALSE;
        else
         return TRUE;
      }
     else
      return FALSE;
}

VOID FillList(HWND hListWnd, WORD wNumberOfSections,
            PIMAGE_SECTION_HEADER pISH, int nOffset)
{
  char cBuffer[0x80];

  sprintf(cBuffer, "0x%08x - Section Table", nOffset);
  SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
  for(int i = 0; i < wNumberOfSections; i++, pISH++,
                                nOffset += IMAGE_SIZEOF_SECTION_HEADER)
   {
    sprintf(cBuffer, "   0x%08x - %.8s", nOffset, pISH->Name);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "      0x%08x - Misc",
          nOffset + offsetof(IMAGE_SECTION_HEADER, Misc));
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "         0x%08x - PhysicalAddress - 0x%08x",          nOffset +
          offsetof(IMAGE_SECTION_HEADER, Misc.PhysicalAddress),
          pISH->Misc.PhysicalAddress);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "         0x%08x - VirtualSize - 0x%08x", nOffset +
          offsetof(IMAGE_SECTION_HEADER, Misc.VirtualSize),
          pISH->Misc.VirtualSize);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "      0x%08x - VirtualAddress - 0x%08x", nOffset +
          offsetof(IMAGE_SECTION_HEADER, VirtualAddress),
          pISH->VirtualAddress);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "      0x%08x - SizeOfRawData - 0x%08x", nOffset +
          offsetof(IMAGE_SECTION_HEADER, SizeOfRawData),
          pISH->SizeOfRawData);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "      0x%08x - PointerToRawData - 0x%08x", nOffset +
          offsetof(IMAGE_SECTION_HEADER, PointerToRawData),
          pISH->PointerToRawData);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "      0x%08x - PointerToRelocations - 0x%08x", nOffset +
          offsetof(IMAGE_SECTION_HEADER, PointerToRelocations),
          pISH->PointerToRelocations);
    SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    sprintf(cBuffer, "      0x%08x - PointerToLinenumbers - 0x%08x", nOffset +
```

```
            offsetof(IMAGE_SECTION_HEADER, PointerToLinenumbers),
            pISH->PointerToLinenumbers);
        SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
        sprintf(cBuffer, "      0x%08x - NumberOfRelocations - 0x%08x", nOffset +
            offsetof(IMAGE_SECTION_HEADER, NumberOfRelocations),
            pISH->NumberOfRelocations);
        SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
        sprintf(cBuffer, "      0x%08x - NumberOfLinenumbers - 0x%08x", nOffset +
            offsetof(IMAGE_SECTION_HEADER, NumberOfLinenumbers),
            pISH->NumberOfLinenumbers);
        SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
        sprintf(cBuffer, "      0x%08x - Characteristics - 0x%08x", nOffset +
            offsetof(IMAGE_SECTION_HEADER, Characteristics),
            pISH->Characteristics);
        SendMessage(hListWnd, LB_ADDSTRING, 0, (LPARAM) cBuffer);
    }
}
```

И, как всегда, окно, созданное при помощи приведённой выше программы:



Надеюсь, при помощи этой программы читатель без труда сможет просмотреть таблицу секций любого PE-файла. Я очень надеюсь, что все скучные теоретизирования по поводу тех или иных полей уже позади. Длиннейшие описания форматов структур и полей этих структур, которые можно будет встретить, скажем, при описании раздела ресурсов, будут несколько более живыми. Их можно будет увидеть, почти пощупать руками... Но до этого необходимо ещё дойти. А пока нам необходимо узнать, какие секции мы можем встретить в исполняемых файлах, а также вкратце назначение каждой секции.

Разделы в исполняемом файле

Итак, мы выяснили, что исполняемый файл состоит из сигнатуры, набора заголовков, таблицы разделов и непосредственно разделов. Что мы можем сказать о разделах вообще? Наверное, не так уж много. Во-первых, то, что по умолчанию подавляющее большинство разделов имеют наименование, начинающееся с точки. Примерами наименований разделов могут служить .idata, .edata, .rsrc и другие. Во вторых, большинство наименований разделов осмысленны. Например, раздел .idata содержит данные об импортируемых функциях (ImportDATA), раздел .rsrc содержит данные о ресурсах, хранящихся где-то в глубине исполняемого файла (ReSouRCes), в разделе .edata хранятся данные об экспортируемых функциях (ExportDATA). Естественно, что программист может создавать собственные разделы и давать разделам произвольные имена. Например, в компиляторах фирмы Borland (нынешняя Inprise) некоторым разделам присваиваются имена CODE и DATA. Ну, и в третьих, в большинстве случаев для того, чтобы определить наименования разделов файла, не требуется какой-либо специальной программы просмотра. Если мы попробуем просмотреть интересующий нас файл при помощи обычного wpview из Norton Commander'a, то мы увидим наименования разделов буквально на первой странице. Не требуется особых навыков для того, чтобы найти среди набора символов сочетания «.reloc», «.debug», «.text» и прочие. Но это так, для сведения. Я рекомендую читателю попробовать просмотреть таким образом какие-нибудь исполняемые PE-файлы. И как результат? ☺

Теперь, когда первоначальное понятие о том, что такое раздел, мы приобрели, необходимо остановиться на том, какие разделы встречаются наиболее часто и какую информацию они содержат.

Наверное, наиболее важной из секций является

Секция программного кода

Эта секция содержит код, который генерирует компилятор. В компиляторах фирмы Microsoft эта секция называется .text, а в компиляторах фирмы Borland (нынешняя Inprise) эта секция обычно называется CODE. С этой секцией неразрывно связаны –секции импорта и экспорта функций.

Мне думается, что без специального инструментария какой-либо полезной информации из этой секции извлечь нам не удастся. Разве что мы посмотрим на содержимое файла при помощи обычного wpview из Norton Commander'a. Что мы можем увидеть? Какие-нибудь литералы, не более того. Понимания того, как работает программа, нам это не даст. В качестве специального инструментария мы можем использовать всевозможные дизассемблеры, отладчики, мониторы и так далее. Естественно, это очень интересная и обширная тема, но здесь мы её обсуждать не будем.

Секция инициализированных данных

В случае использования компиляторов фирмы Micrsoft по умолчанию все инициализированные во время компиляции данные попадают в секцию .edata. Если же используются компиляторы фирмы Borland (нынешняя Inprise), то эта секция называется DATA. Кроме глобальных и статических переменных, в этой секции содержатся также литералы.

Но, если есть отдельная секция для инициализированных данных, то, наверное, логично предположить, что существует и секция для неинициализированных данных. Действительно, такая секция существует и в случае использования компилятора фирмы Microsoft такой секцией является

Секция .bss

Но так как никаких данных в этом случае хранить не нужно, то обычно её размер в исполняемом файле равен нулю. Точнее, как таковой этой секции в исполняемом файле нет. Есть только упоминание о ней в таблице секций. Компиляторы фирмы Borland (нынешняя Inprise) секцию .bss не создают.

Секция .idata

В этой секции находятся таблицы, обеспечивающие импорт функций из другиз библиотек динамической компоновки. С её форматом мы познакомимся поближе тогда, когда будем изучать механизм импорта функций.

Секция .edata

Секция .edata содержит данные о тех функциях, которые экспортируются данным модулем. О том, что такое экспорт функций и о механизме экспорта мы поговорим позже.

Одной из самых больших и сложных секций в исполняемом файле является

Секция .rsrc

Эта секция хранит в себе все ресурсы, которые присутствуют в исполняемом файле. Я думаю, что на описание формата всех ресурсов у нас уйдёт очень много места и времени. Если уважаемый читатель интересуется форматом ресурсов, то он может обратиться к главе «Ресурсы в исполняемом файле».

Секция .reloc

Эта секция содержит таблицу базовых поправок и ничего более. Она используется только тогда, когда загрузчик не смог загрузить файл по

адресу, начиная с которого планировал произвести загрузку редактор связей. Формат её крайне прост.

Секция .tls

Если мы предполагаем использовать локальную память потока, то эти данные не попадают ни в секцию инициализированных данных, ни в секцию неинициализированных данных. Вместо этого они попадают в специальную секцию .tls (Thread Local Storage).

Естественно, что я не смог описать назначение всех секций, которые могут быть встречены читателем в исполняемых файлах. Во-первых, это просто невозможно, так как секции очень часто определяются программистом, и предусмотреть, какие данные будет хранить та или иная секция, просто невозможно. Но, во-вторых, в том случае, если программист обладает хорошим стилем программирования или стремится к нему, то секции приобретают несколько осмысленные имена. Например, в одном из файлов я встретил секцию, которая называлась "PRINT". Логично предположить, что в этой секции находятся данные, каким-то образом связанные с печатью. Конечно, это может быть и отвлекающим манёвром, но, надеюсь, такие отвлекающие манёвры встречаются достаточно редко.

Экспорт функций и механизм экспорта

Выше мы говорили о том, что такое импорт и предположили, что к тому времени, когда запускается наша программа, уже готовы какие-то библиотеки динамической компоновки, функции из которых мы импортируем. По аналогии с внешнеторговыми операциями (☺) в таком случае мы можем сказать, что эти библиотеки ЭКСПОРТИРУЮТ функции. Естественно, что для того, чтобы функции были экспортируемыми, они должны быть каким-то образом описаны и к ним можно было бы легко осуществить доступ. Всё описание расположения экспортируемых функций обычно находится в разделе, который называется «.edata».

Каждая экспортируемая функция имеет имя и порядковый номер внутри модуля. Обращение к ним может осуществляться и по имени, и по порядковому номеру, то есть, например, я могу вызвать экспортируемую функцию MyFunction, а могу и обратиться к функции, порядковый номер которой равен, скажем, пяти. Экспортируемыми могут, кстати, быть не только функции, но и другие данные, хранящиеся в исполняемом файле.

Но для того, чтобы не внести путаницы, давайте будем говорить об экспортируемых функциях (наверное, функции экспортируются наиболее часто), но подразумевать при этом все возможные экспортируемые данные. Из раздела экспорта мы попытаемся извлечь информацию о том, какие функции, с какими порядковыми номерами экспортируются исследуемым модулем. Кроме этого, мы постараемся выяснить, где находятся экспортируемые функции. Для этого, естественно, нам необходимо знать формат данных об экспорте, то есть формат раздела экспорта.

Итак, как всегда, первым делом мы должны определить смещение таблицы экспорта в исполняемом файле. Как мы уже видели, это не настолько тривиальная задача, как, например, нахождение смещения таблицы разделов. Что же нам необходимо сделать для этого? Алгоритм нахождени смещения раздела экспорта в имполняемом файле может быть представлен следующими шагами:

1. Входим в массив IMAGE_DATA_DIRECTORY, которым заканчиваетсмя «необязательный заголовок» файла и выбираем в нём строку с индексом IMAGE_DIRECTORY_ENTRY_EXPORT, то есть нулевую строку.
2. Из этой строки выбираем значение VirtualAddress, то есть смещение раздела экспорта в загруженном в память ОБРАЗЕ исполняемого файла. В дальнейшем все смещения в разделе будут приводиться относительно этого значения.
3. Выше я упомянул, что при анализе исполняемого файла должна появиться поправка, которая будет учитывать разность между значением поля VirtualAddress и полем PointerToRawData из описания раздела в таблице разделов. С этой целью нам необходимо перебрать таблицу разделов до тех пор, пока мы не найдём в ней раздел с именем «.edata». Выбрав из описания этого раздела поля VirtualAdress и PointerToRawData, вычисляем искомую поправку.
4. Смещение раздела экспорта находим как разность между выбранным на втором шаге алгоритма виртуальным адресом и поправкой. Это смещение отсчитывается от начала исполняемого файла.

После того, как уважаемый читатель прочёл этот алгоритм, у него, возможно, появился один вопрос, – зачем городить весь этот огород, если из элемента таблицы разделов, соответствующего разделу экспорта, можно просто-напросто выбрать значение поля PointerToRawData?

Как говорится, гладко было на бумаге... Когда я писал большую программу для просмотра исполняемых файлов, я сначала так и определял смещение требующегося мне раздела относительно начала файла как значение поля PointerToRawData. Когда я начал отладку программы, то я проверял её работоспособность на каждом из более чем двух тысяч .exe- и .dll-файлов, которые находились у меня на винчестере. Несколько раз оказывалось, что выбранное программой значение PointerToRawData не соответствует фактическому расположению его в файле. Пришлось изобретать новый способ определения смещения данных раздела. Тот способ, который изложен выше, наверное, несколько неизящен, зато он у меня работал ВО ВСЕХ СЛУЧАЯХ. Именно поэтому я решил предложить уважаемому читателю свой способ.

Итак, считаем, что раздел экспорта мы можем найти без труда. Теперь, естественно, необходимо рассмотреть формат и назначение той информации, которая хранится в этом разделе.

Структура раздела экспорта приведена ниже.

Оглавление
Таблица адресов
Таблица указателей на имена
Таблица порядковых номеров функций
Таблица экспортируемых имен

Давйте рассмотрим каждую из этих частей более подробно.

Оглавление раздела экспорта

Раздел экспорта начинается с оглавления. Формат оглавления документирован в заголовочном файле winnt.h:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
   DWORD   Characteristics;
   DWORD   TimeDateStamp;
   WORD    MajorVersion;
   WORD    MinorVersion;
   DWORD   Name;
   DWORD   Base;
   DWORD   NumberOfFunctions;
   DWORD   NumberOfNames;
   PDWORD  *AddressOfFunctions;
   PDWORD  *AddressOfNames;
   PWORD   *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Итак, давайте рассмотрим каждое поле отдельно.

Первое поле - Characteristics - в настоящее время не используется и должно быть заполнено нулями. Второе поле - TimeDateStamp - определяет время, в которое был создан файл. Третье и четвертое поля - старшая и младшая часть версии файла. Представляют ли эти поля интерес?

Поле Name используется для того, чтобы определить смещение, по которому находится строка, содержащая имя файла библиотеки. Это поле представляет интерес только после загрузки файла в оперативную память, когда мы исследуем содержимое оперативной памяти, поэтому при исследовании формата файла (а не образа) мы его опустим.

Поле Base определяет начальный порядковый номер для экспортируемых функций. Обычно значение этого поля равно 1.

Поле NumberOfFunctions определяет число экспортируемых адресов функций.

Поле NumberOfNames, в свою очередь, определяет число экспортируемых имен и, соответственно, порядковых номеров функций. Дело в том, что иногда при вызове разных функций обращение происходит к одному и тому же адресу. Ярким примером этого является библиотека kernel32.dll, в которой обращение к функции со смещением 13d4h происходит при вызо-

ве восьми функций. Поэтому значение поля NumberOfFunctions всегда *меньше или равно значению поля NumberOfNames*.

Как явствует из предыдущих рассуждений, где-то «в недрах» исполняемого файла должны храниться таблицы, содержащие адреса экспортируемых функций, их имена и порядковые номера. Следующие три поля определяют смещения этих таблиц (таблицы адресов, таблицы имён имен и таблицы порядковых номеров) в загруженном образе относительно начала образа файла. Лично мне не совсем понятно, почему разработчики вместо трех таблиц не создали одну, элементами которой были бы структуры, содержащие адрес, имя и порядковый номер экспортируемых функций. Для того, чтобы читатель мог более наглядно представить себе взаимоотношения между таблицами адресов функций, адресов имен функций и порядковых номеров экспортируемых функций, ниже приведен рисунок, иллюстрирующий это взаимодействие.

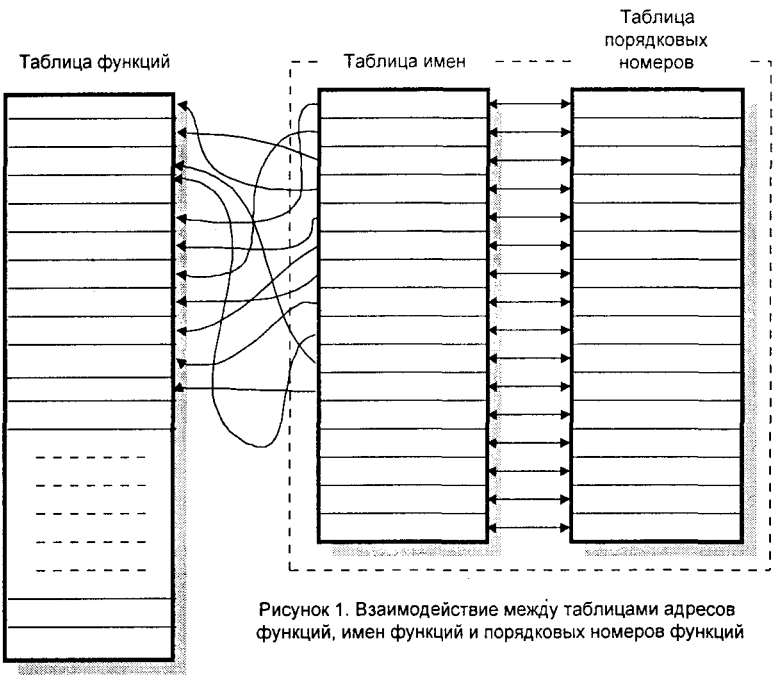


Рисунок 1. Взаимодействие между таблицами адресов функций, имен функций и порядковых номеров функций

Как следует из общей структуры раздела экспорта, за оглавлением раздела следует

Таблица адресов

Таблица адресов содержит смещения экспортируемых функций относительно начала образа файла. Каждый элемент этой таблицы занимает 4 байта (одно двойное слово) и содержит смещение точки входа одной

функции. Количество элементов в этой таблице равно числу экспортируемых функций (NumberOfFunctions), которое определяется в оглавлении раздела. При этом нужно помнить о том, что иногда не все элементы этой таблицы задействованы. Некоторые элементы могут вместо смещений экспортируемых функций содержать нули. Это является признаком того, что данный элемент таблицы не используется. Порядковый номер (ordinal) экспортируемой функции вычисляется как сумма индекса данной строки таблицы (индекс отсчитывается от нуля) и поля Base в заголовке раздела экспорта.

За таблицей адресов следует

Таблица указателей на имена

Эта таблица содержит указатели на ASCII-строки, заканчивающиеся нулем, содержащие имена экспортируемых функций. Как и в предыдущем случае, каждый элемент этой таблицы занимает 4 байта (одно двойное слово) и содержит указатель на имя экспортируемой функции. Количество элементов в этой таблице равно числу экспортируемых имен (NumberOfNames), определенное в заголовке раздела.

Таблица порядковых номеров функций

Таблица порядковых номеров содержит индексы строк таблицы экспортируемых функций. Эта таблица отличается от предыдущих таблиц тем, что ее поля занимают не четыре, а два байта (другими словами, в файле может быть не более 65 536 экспортируемых функций). Количество элементов и в этой таблице равно числу экспортируемых имен (NumberOfNames).

Две последние таблицы работают в связке, то есть каждая строка таблицы адресов имен функций соответствует строке таблицы порядковых имен.

Таблица экспортируемых имен

Таблица экспортируемых имен определенного размера не имеет. Каждый элемент в этой таблице представляет собою ASCII-строку, завершающуюся нулем. Нужно ли повторять, что число элементов в этой таблице также равно числу экспортируемых программой имен?

Теперь рассмотрим каким образом осуществляется

Обращение к экспортируемой функции

Обращение к экспортируемой функции (другими словами, импорт функции) может происходить как по имени, так и по порядковому номеру. Итак, случай первый.

Импорт функции по номеру

При обращении к функции по ее порядковому номеру система производит следующие действия:

1. Вычисляет индекс адреса функции в таблице адресов функций (этот индекс равен разности между порядковым номером функции и значением поля Base в заголовке раздела);

2. В таблице адресов функций выбирает элемент, индекс которого равен вычисленному на предыдущем шаге и осуществляет передачу управления соответствующей функции.

Как видно, все происходит с завидной простотой. Я хотел бы обратить внимание читателя, что при этом таблицы адресов имен и порядковых номеров функций даже не используются.

Импорт функции по имени

заставляет систему выполнить несколько больший объем работы. При импорте по имени происходит следующее:

Осуществляется поиск строки, соответствующей имени вызываемой функции, в таблице адресов имен функций. Отмечу, что в таблице адресов имен строки расположены в соответствии с упорядоченными по алфавиту именами функций. Естественно, при таком условии время, необходимое на поиск нужной строки, резко уменьшается.

Выбирается элемент таблицы порядковых номеров с индексом, соответствующим найденному на предыдущем шаге. Значением выбранного элемента является индекс в таблице адресов функций.

Выбирается элемент в таблице адресов функций, индекс которого равен найденному на предыдущем шаге.

После всего, сказанного выше, мы знаем почти все о разделе экспорта в выполняемом файле. Естественно, что неплохо было бы написать небольшую демонстрационную программу, иллюстрирующую сказанное выше. Я написал таковую программу. При её помощи можно открыть файл и просмотреть содержимое секции экспорта. При этом показываются смещения экспортируемых функций, их порядковые номера и названия. Кстати, я хотел бы заметить, что о названиях мы ещё поговорим чуть позже. В определённых случаях из них можно выжать много интересного...

Сначала я приведу заголовочный файл демонстрационной программы.

```
#define IDM_OPEN 101
#define IDM_EXIT 102

MainMenu MENU
{
 POPUP "&File"
 {
  MENUITEM "&Open", IDM_OPEN
  MENUITEM SEPARATOR
  MENUITEM "E&xit", IDM_EXIT
 }
```

```
}
```

А сейчас я приведу основной файл программы:

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

#define IDM_OPEN 101
#define IDM_EXIT 102

typedef struct
 {
  PDWORD pdwPointerInAOF;
  PDWORD pdwPointerInAON;
  WORD wNO;
 } EXPORTDATA, *PEXPORTDATA;

LRESULT CALLBACK DemanglingWndProc ( HWND, UINT, UINT, LONG );
BOOL TestFile(PIMAGE_DOS_HEADER);
LPVOID MapFile(LPTSTR);
void FillList(HWND, WORD, PIMAGE_SECTION_HEADER, int);

HINSTANCE hInst;
HWND hExportWnd;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "Demangling";

  hInst = hInstance;

/* Registering our window class */
/* Fill WNDCLASS structure */

  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = DemanglingWndProc;
  WndClass.cbClsExtra = 0;
  WndClass.cbWndExtra = 0;
  WndClass.hInstance = hInstance ;
  WndClass.hIcon = LoadIcon (NULL,IDI_APPLICATION);
  WndClass.hCursor = LoadCursor (NULL, IDC_ARROW);
  WndClass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
  WndClass.lpszMenuName = "MainMenu";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
```

```
      MessageBox(NULL,"Cannot register class","Error",MB_OK);
      return 0;
    }

  hWnd = CreateWindow(szClassName, "Demangling",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
      MessageBox(NULL,"Cannot create window","Error",MB_OK);
      return 0;
    }

  InitCommonControls();

  /* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
   }
  return Msg.wParam;

 }

LRESULT CALLBACK DemanglingWndProc (HWND hWnd, UINT Message,
                                    UINT wParam, LONG lParam )
 {

  HDC hDC;
  PAINTSTRUCT PaintStruct;
  RECT Rect;
  static HWND hListWnd;
  char* cBuf[3] = {"Functions", "Ordinals", "Names"};
  LV_COLUMN LV_Column;
  int i = 0;
  static char lpstrFile[MAX_PATH] = "";
  PIMAGE_DOS_HEADER pIDH;

  hExportWnd = hWnd;
  switch(Message)
   {
     case WM_CREATE:
       GetClientRect(hWnd, &Rect);
       hListWnd = CreateWindow(WC_LISTVIEW, "",
                               LVS_REPORT | WS_CHILD,
                               0, 0, Rect.right, Rect.bottom,
```

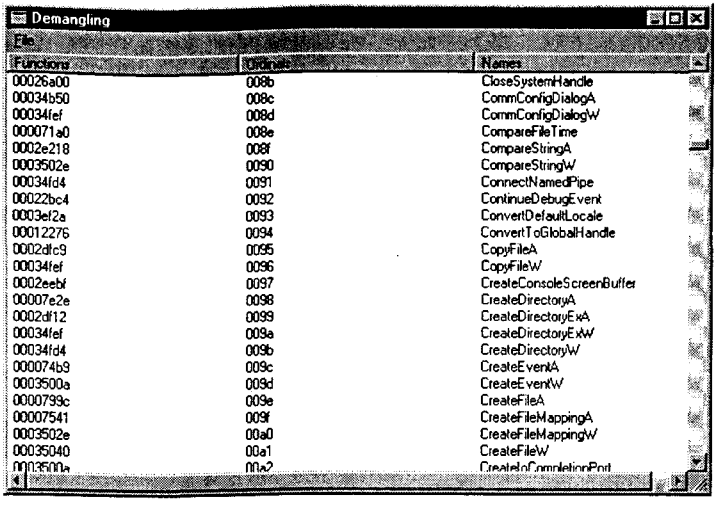
```
                                hWnd, NULL, hInst, NULL);
  ShowWindow(hListWnd, SW_SHOW);
  LV_Column.mask = LVCF_FMT | LVCF_SUBITEM | LVCF_TEXT |
                   LVCF_WIDTH;
  LV_Column.fmt = LVCFMT_LEFT;
  LV_Column.cx = Rect.right / 3;
  for( ; i < 3; i++)
   {
    LV_Column.iSubItem = i;
    LV_Column.pszText = cBuf[i];
    ListView_InsertColumn(hListWnd, i, &LV_Column);
   }
  return 0;
 case WM_COMMAND:
  switch(LOWORD(wParam))
   {
    case IDM_OPEN:
     OPENFILENAME OpenFileName;
     OpenFileName.lStructSize = sizeof(OPENFILENAME);
     OpenFileName.hwndOwner = hWnd;
     OpenFileName.hInstance = hInst;
     OpenFileName.lpstrFilter = "Applications (*.exe)\0*.exe\0
                                 Application Extension (*.dll)\0*.dll\0\0";
     OpenFileName.lpstrCustomFilter = NULL;
     OpenFileName.nFilterIndex = 0;
     OpenFileName.lpstrFile = lpstrFile;
     OpenFileName.nMaxFile = MAX_PATH;
     OpenFileName.lpstrFileTitle = NULL;
     OpenFileName.lpstrInitialDir = NULL;
     OpenFileName.lpstrTitle = "Open PE-files for viewing of section table...";
     OpenFileName.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST |
                          OFN_HIDEREADONLY | OFN_LONGNAMES |
                          OFN_PATHMUSTEXIST;
     OpenFileName.lpstrDefExt = NULL;
     if(!GetOpenFileName(&OpenFileName))
       MessageBox(hWnd, "Невозможно открыть файл",
                  "Ошибка открытия файла", MB_OK);
      else
       {
        if( (pIDH = (PIMAGE_DOS_HEADER)
          MapFile(OpenFileName.lpstrFile)) == 0)
          MessageBox(hWnd, "Невозможно спроецировать файл в память",
                     "Ошибка проецирования файла", MB_OK);
         else
          {
           if(!TestFile(pIDH))
             MessageBox(hWnd, "Файл не является PE-файлом",
                        "Ошибка", MB_OK);
            else
             {
              PIMAGE_NT_HEADERS pINTH =
                        (PIMAGE_NT_HEADERS) ((PBYTE) pIDH +
                        pIDH->e_lfanew);
```

```
                    PIMAGE_SECTION_HEADER pISH =
                                    IMAGE_FIRST_SECTION(pINTH);
                    int nOffset = (int) pISH - (DWORD) pIDH;
                    FillList(hListWnd, pINTH->FileHeader.NumberOfSections,
                            pISH, nOffset);
                  }
                UnmapViewOfFile((LPVOID) pIDH);
              }
            }
           return 0;
          case IDM_EXIT:
           PostQuitMessage(0);
           return 0;
        }
      case WM_DESTROY:
       PostQuitMessage(0);
       return 0;
      default:
       return DefWindowProc(hWnd,Message,wParam, lParam);
    }
 }

BOOL TestFile(PIMAGE_DOS_HEADER pIDH)
 {
  if(pIDH->e_magic != IMAGE_DOS_SIGNATURE)
     return FALSE;
    else
     if(pIDH->e_lfarlc >= 0x40)
       {
        PIMAGE_NT_HEADERS pINTH = (PIMAGE_NT_HEADERS)
                                        ((PBYTE) pIDH + pIDH->e_lfanew);
        if(pINTH->Signature != IMAGE_NT_SIGNATURE)
           return FALSE;
         else
           return TRUE;
       }
    else
      return FALSE;
 }

LPVOID MapFile(LPTSTR lpstrFile)
 {
  HANDLE hFile, hMappedFile;
  LPVOID lpMappedFile;

  hFile = CreateFile(lpstrFile, GENERIC_READ,
                     FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
                     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  if(INVALID_HANDLE_VALUE == hFile)
     return 0;
    else
     {
      hMappedFile = CreateFileMapping(hFile, NULL, PAGE_READONLY,
```

```
                                        0, 0, NULL);
    CloseHandle(hFile);
    if(!hMappedFile)
      return 0;
     else
      {
       lpMappedFile = MapViewOfFile(hMappedFile, FILE_MAP_READ,
                                        0, 0, 0);
       CloseHandle(hMappedFile);
       if(!lpMappedFile)
         return 0;
        else
         return lpMappedFile;
      }
   }
}

void FillList(HWND hListWnd, WORD wNumberOfSections,
          PIMAGE_SECTION_HEADER pISH, int nOffset)
{

  DWORD dwRva, dwDifference;
  PIMAGE_EXPORT_DIRECTORY dwExportOffset;
  int i, j;
  char cBuffer[0x400];
  PDWORD pdwPointerInAOF, pdwPointerInAON, pdwPointerInAON_Save;
  PWORD pwPointerInAONO, pwPointerInAONO_Save;
  LV_ITEM LV_Item;

  PIMAGE_DOS_HEADER pIDH = (PIMAGE_DOS_HEADER)
                                    ( (int) pISH - nOffset);
  PIMAGE_NT_HEADERS pINTH = (PIMAGE_NT_HEADERS)
                                    ((PBYTE) pIDH + pIDH->e_lfanew);
  dwRva = pINTH->OptionalHeader.DataDirectory[0].VirtualAddress;
  for(i = 0; i < wNumberOfSections; i++, pISH++)
   {
    if(!strcmp(pISH->Name, ".edata"))
      {
       i = wNumberOfSections;
       dwDifference = pISH->VirtualAddress - pISH->PointerToRawData;
      }
   }
  PIMAGE_EXPORT_DIRECTORY pIED = (PIMAGE_EXPORT_DIRECTORY)
                                    ( (PBYTE)pIDH + dwRva -dwDifference);
  pdwPointerInAOF = (PDWORD) ( (PBYTE) pIDH +
                        (DWORD) pIED->AddressOfFunctions - dwDifference);
  pdwPointerInAON = pdwPointerInAON_Save = (PDWORD)
          ( (PBYTE) pIDH + (DWORD) pIED->AddressOfNames -dwDifference);
  pwPointerInAONO = pwPointerInAONO_Save = (PWORD)
    ( (PBYTE) pIDH + (DWORD) pIED->AddressOfNameOrdinals -dwDifference);
  LV_Item.state = LV_Item.stateMask = 0;
  for( i = 0; i < pIED->NumberOfFunctions; i++ )
   {
```

```
      LV_Item.mask = LVIF_TEXT | LVIF_PARAM;
      PEXPORTDATA pExportData = new EXPORTDATA;
      j = 0;
// Теперь нам необходимо перебрать все таблицы экспорта.
// 1. Выбираем элемент из таблицы адресов функций.
      pExportData->pdwPointerInAOF = pdwPointerInAOF;
// 2. Проверяем, существует ли соответствующий элемент в таблице
// порядковых номеров.
      while( (*pwPointerInAONO != i) && (j++ < pIED->NumberOfNames) )
       {
        pdwPointerInAON++;
        pwPointerInAONO++;
       }
// Если нам удалось найти соответствующий элемент,
// то заполняем структуру pExportData
      if(j <= pIED->NumberOfNames)
        {
         pExportData->pdwPointerInAON = pdwPointerInAON;
         pExportData->wNO = i +pIED->Base;
        }
       else
        {
         pExportData->pdwPointerInAON = 0;
         pExportData->wNO = i +pIED->Base;
        }
      LV_Item.iItem = i;
      LV_Item.iSubItem = 0;
      sprintf(cBuffer, "%08x", *pdwPointerInAOF);
      LV_Item.pszText = cBuffer;
      LV_Item.lParam = (LPARAM) pExportData;
      ListView_InsertItem(hListWnd, &LV_Item);
      LV_Item.mask = LVIF_TEXT;
      LV_Item.iSubItem = 1;
      sprintf(cBuffer, "%04x", i + pIED->Base);
      LV_Item.pszText = cBuffer;
      ListView_SetItem(hListWnd, &LV_Item);
      if(j <= pIED->NumberOfNames)
        {
         LV_Item.iSubItem = 2;
         sprintf(cBuffer, "%s", (PBYTE) pIDH + (DWORD) *pdwPointerInAON –
                 dwDifference);
         LV_Item.pszText = cBuffer;
         ListView_SetItem(hListWnd, &LV_Item);
        }
      pdwPointerInAOF++;
      pdwPointerInAON = pdwPointerInAON_Save;
      pwPointerInAONO = pwPointerInAONO_Save;
    }
   SetWindowLong(hExportWnd, GWL_USERDATA, (LONG) hListWnd);
 }
```

Выглядит окно, созданное при помощи этой демонстрационной программы, следующим образом (в данном случае я просматриваю kernel32.dll):



Наверное, кое-какую для себя помощь из списка экспортируемых модулем функций мы уже можем извлечь. Скажем, найти недокументированные Microsoft функции или, например, определить, какие функции могут использоваться для того, чтобы произвести те или иные действия. Но, по моему мнению, намного больше информации мы можем извлечь в других случаях. Давайте попробуем посмотреть список экспорта не в kernel32.dll, а, например, в c:\windows\system\msvcrt20.dll. Здесь имена функций будут выглядеть совершенно по-другому. Итак, ...

Я надеюсь, что ни один программист, если он находится в здравом уме и твердой памяти (☺), не станет давать функциям в своей программе подобные имена. Вывод - в формирование подобных имен вмешался компилятор. О том, почему имена экспортируемых функций выглядят столь странно и о том, какую информацию мы сможем отсюда извлечь, мы поговорим в следующей главе. Естественно, что вопросы, связанные с искажением имен функций КОМПИЛЯТОРОМ, а НЕ СИСТЕМОЙ, нельзя считать относящимися к вопросам программирования в Win32 API, но без знания этих деталей невозможно произвести более глубокий анализ исполняемого файла.

Искажение имён в C++ (вариант фирмы Borland)

При рассмотрении вопроса об экспортируемых функциях я упомянул о том, что при компиляции в некоторых случаях имена, данные функциям программистом, изменяются компилятором и в объектном, а, следовательно, и в исполняемом файле получается что-то, напоминающее исходные имена функций очень отдаленно.

Представим себе, что существует какая-то функция, скажем, MyFunc(), которая в качестве аргументов должна получать два указателя, например, int и char*. Допустим, что программист в заголовочном файле написал неверный прототип этой функции и вызвал её с аргументами другого типа. В обычном C результат, как говорится, непредсказуем. Но язык C++ недаром пользуется славой языка, старающегося уберечь программиста ото всех возможных ошибок. Подобные ситуации в нём предусмотрены. Каким образом разработчикам компилятора языка C++ удалось решить эту проблему, мы и обсудим в этом разделе.

По моему глубокому убеждению, проблема разрешена достаточно оригинально. Компилятор ПРОИЗВОДИТ ИЗМЕНЕНИЕ имен функций таким образом, что каждое измененное имя функции содержит в себе массу информации, в том числе и информацию о типах требуемых функцией аргументов. Таким образом, вызов одной функции с аргументами другой в C++ просто невозможен - сразу будет выдано сообщение об ошибке.

Другими словами, в этом случае резко повышается надёжность программ. В данном случае мы не будем касаться вопроса надежности программы, нам необходимо просто извлечь максимум информации об исполняемом файле.

К сожалению, Microsoft C++ и Borland C++ используют разные алгоритмы изменения. Я опишу Borland'овский алгоритм (дело в том, что автор использует компилятор фирмы Borland), но надеюсь, что те, которые разберутся в одном алгоритме, без труда разберутся и в другом.

«Виртуальный механизм» в C++ реализован при помощи таблиц, которые обычно называют виртуальными таблицами или VMT (Virtual Method Table – таблица виртуальных методов). Различные установки компилятора определяют, либо виртуальная таблица заканчивается в Default Data

Segment (сегменте данных по умолчанию), или в Far Segment (дальнем сегменте).

Borland C++ использует четыре основные формы изменения имен. Поговорим о них более подробно.

Формы изменения имен

Я прошу уважаемого читателя обратить внимание на следующее: то, что написано в этом разделе, необходимо, прежде всего, не для программирования. Это необходимо прежде всего для исследования программ. Я достаточно часто посещаю сайты, посвящённые различным вопросам программирования и исследования программ, и достаточно часто в форумах вижу вопросы о том, как узнать, какие аргументы использует та или иная функция. Что здесь ответить? Во-первых, как я уже указал выше, если для написания программ использовался компилятор языка C, то узнать о аргументах той или иной функции можно только при анализе дизассемблированного файла. Но, если программа написана на языке C++, то информация об арументах функции хранится в самом изменённом имени функции и «выудить» её оттуда вполне по силам программисту или аналитику средней квалификации. Но, естественно, для того, чтобы сделать это, необходимо обладать сведениями о том, каким образом производится изменение имён компилятором языка C++.

Основные правила искажения имён в языке C++ (компилятор фирмы Borland)

Кодирование наименования функции (метода), принадлежащей классу

Имена классов и функций кодируются достаточно просто. Перед первым символом имени класса добавляется символ «@», перед первым символом наименования функции также добавляется символ "@». Перед списком аргументов добавляется символ «$». Например, если нам необходимо закодировать данные о функции с именем «FunctionName», использующей аргументы «args» и принадлежащей к классу «ClassName», то результатом этого кодирования будет что-то типа

```
@ClassName@FunctionName$args
```

За именем класса может следовать одна цифра, биты которой используются как логическая шкала и могут комбинироваться. Ниже я приведу значения, которые может принимать эта цифра:

Значение	Назначение
0x01	Класс использует дальнюю виртуальную таблицу
0x02	
0x04	

Но непосредственно эти значения (то есть 1, 2 и 4) в закодированной информации о функции не используются. В изменённом имени цифра кодируется как ASCII-представление битовой шкалы, уменьшенное на единицу. Например, в том случае, если используется дальняя виртуальная таблица, имя класса «MyClass» будет закодировано как «@MyClass@0».

Из сказанного выше делаем первый вывод – **если мы в закодированном имени можем обнаружить ПО МЕНЬШЕЙ МЕРЕ ДВА СИМВОЛА «@» И ОДИН СИМВОЛ «$», то мы можем быть уверены в том, что закодирована информация о функции, принадлежащей классу.**

Я прошу уважаемого читателя обратить внимание на то, что я использовал выражение «по меньшей мере». Чуть позже я объясню, почему символов «@» может быть сколь угодно много (в разумных пределах, конечно).

Кодирование имён членов класса.

В данном случае имя класса кодируется точно таким же образом, как и в первом случае, то есть перед первым символом имени класса добавляется символ «@». Перед первым символом имени члена класса также добавляется символ «@". Другими словами, если в моей программе встречается класс «MyClass», которому принадлежит член «MyMember», то в откомпилированной программе запись, соответствующая этому члену класса, будет выглядеть следующим образом:

```
@MyClass@MyMember
```

Очередной вывод, который мы можем сделать в данном случае, будет таким – **если в закодированном имени мы можем обнаружить ТОЛЬКО СИМВОЛЫ «@», НО НЕ СИМВОЛЫ «$», то в данном случае мы имеем дело с закодированным именем члена класса.**

Кодирование имени функции, не принадлежащей ни к какому классу.

Практически данный случай ничем не отличается от случая первого. Перед первым символом имени функции добавляется символ «@», а перед списком аргументов добавляется символ «$». Ниже я привёл пример кодирования имени той же фунции «FunctionName», имеющей те же аргументы «args», но не принадлежащей ни к одному классу.

```
@FunctionName$args
```

Сам собой напрашивается вывод - **если мы в закодированном имени функции может найти ОДИН И ТОЛЬКО ОДИН СИМВОЛ «@» И ОДИН И ТОЛЬКО ОДИН СИМВОЛ «$», то мы имеем дело с функцией, не принадлежащей ни к одному из классов.**

Кодирование обращения в виртуальной таблице класса

Если мы видим запись типа «@ClassName@», то мы можем сказать, что в данном случае речь идёт о каком-то обращении к виртуальным таблицам класса «ClassName».

Очередной вывод, который мы можем сделать, рассмотрев упомянутый выше формат записи, будет следующим – **если в закодированном имени функции присутствуют ТОЛЬКО СИМВОЛЫ «@», ПРИЧЁМ ЗАКОДИРОВАННОЕ ИМЯ И ЗАКАНЧИВАЕТСЯ СИМВОЛОМ "«@», то мы имеем дело с обращением к виртуальной таблице класса, имя которого заключено в символы «@».**

Кодирование имён классов – контейнеров и вложенных классов

Ни для кого не секрет, что в качестве членов классов могут использоваться другие классы. В данном случае тот класс, который использует в качестве члена другой класс, называется классом – контейнером. Тот класс, который используется как член другого класса, называется вложенным классом. Для того, чтобы закодировать имена функций и членов вложенного класса, используется уже хорошо знакомый нам алгоритм – перед первым символом каждого класса добавляется символ «@», имя вложенного класса добавляется к имени класса-контейнера. Приведу пример – если у меня есть класс – контейнер «Outer», которому принадлежит вложенный класс «Inner», то все записи о функциях, членах класса и т.д. будут начинаться с

«@Outer@Inner@...»

Теперь, я надеюсь, понятно, почему в выводе ,который мы сделали для первого случая (кодирование наименования функции (метода), принадлежащей классу), я использовал выражение «по меньшей мере». Символов «@» может быть сколько угодно.

Давайте попробуем сделать очередной вывод – **если в закодированном имени НАХОДИТСЯ ТРИ И БОЛЕЕ СИМВОЛОВ «@», ПРИЧЁМ ЗАКОДИРОВАННОЕ ИМЯ СИМВОЛОМ «@» НЕ ЗАКАНЧИВАЕТСЯ, то мы имеем дело с данными классов – контейнеров и вложенных классов.**

Что ж, с кодированием классов, методов, полей и прочего мы разобрались. Теперь нам предстоит рассмотреть кодирование шаблонов классов, которое стоит несколько особняком.

Кодирование шаблонов классов

При кодировании шаблонов классов кодируются имя шаблона класса и аргументы. Если у нас есть шаблон класса с именем «Template», ар-

гументами которого являются «arg1», «arg2», ... , «argn», то после кодирования мы получим следующий результат:

%Template$arg1$arg2 ... $argn%

Какой вывод мы можем сделать из сказанного выше? **Если закодированное имя НАЧИНАЕТСЯ И ЗАКАНЧИВАЕТСЯ СИМВОЛОМ «%», то мы имеем дело с шаблоном класса.**

Что ж, уважаемый читатель, я очень надеюсь, что мы разобрались с основами кодирования наименований классов, функций и т.д., которое применяется в компиляторе языка C++ фирмы Borland (нынешняя Inprise). Я специально употребил слово «основа». Дело в том, что изложенным выше правила кодирования далеко не исчерпываются. Теперь нам необходимо рассмотреть

Правила кодирования наименований функций и их аргументов

Выше мы рассмотрели основные правила кодирования (искажения) имён классов, функций и т.д. Но самое сложное у нас ещё впереди. Дело в том, что помимо основных правил существует ещё множество правил, определяющих правила кодирования в зависимости от того, что та или иная функция собою представляет – конструктор, деструктор, перегруженный оператор и т.д.

Кодирование имён обычных функций

При кодировании имён обычных функций, то есть тех функций, у которых нет особого предназначения, производится напрямую. Об этом мы уже говорили выше. Например, если у нас есть функция MyFunction(arg), то после компиляции её имя превратится в @MyFunction$arg. Если у нас есть та же функция, но принадлежащая классу MyClass, то после компиляции её имя будет будет выглядеть как @MyClass@MyFunction$arg.

Кодирование имён функций, имеющих особое назначение

Имена функций, имеющих особое предназначение, кодируются несколько отличным от обычного способом. Имена конструкторов, деструкторов и перегруженных операторов кодируются при помощи последовательности «$b<символ>». Символы, которые применяются в этих случаях, я привожу в таблице ниже.

Например, перегруженный оператор operator+(int) будет перекодирован в $badd$qi, а конструктор plot::plot() после преобразования будет выглялеть как @plot$bctr$qv. Деструктор того же класса plot::~plot() примет вид @plot$bdtr$qv.

Буква «q» после второго символа «$» означает, что после неё следуют данные об аргументах.

Символ	Значение
Ctr	Конструктор
Dtr	Деструктор
Add	+
Adr	&
And	&
Arow	->
Arwm	->*
Asg	=
Call	()
Cmp	~
Coma	,
Dec	--
Dele	Delete
Div	/
Eql	==
Geq	>=
Qtr	>
Inc	++
Ind	*
Land	&&
Lor	||
Leq	<=
Lsh	<<
Lss	<
Mod	%
Mul	*
Neq	!=
New	New
Not	!
Or	|
Rand	&=
Rdiv	/=
Rlsh	<<=
Rmin	-=
Rmod	%=
Rmul	*=
Ror	|=
Rplu	+=
Rrsh	>>=
Rsh	>>
Rxor	^=
Sub	-
Subs	[]
Xor	^
Nwa	New[]
Dla	Delete[]

Кодирование операций преобразования типов.

*Операции преобразования типов кодирубтся при помощи последо*вательности символов «$o», после которой следует буква, определяющая *тип значения, возвраемого операцией преобразования.* Примерами vогут служить следующие преобразования. MyFunction::operator int() будет преобразован в @MyFunctin@oiqv, а MyFunction::operator char*() примет вид @MyFunction@$opzc$qv.

Кодирование аргументов

Количество и сочетания аргументов и их типов делает кодирование аргументов наиболее сложной частью искажения имён.

*Список аргументов для функций всегда начинается с последова*тельности символов «$q». Буквы, соответствующие типам квалификатов типов, приведены в следующей таблице:

Символ	Значение
Up	Huge
Ur	_seg
U	Unsigned
Z	Signed
X	Const
W	Volatile

В кодированной форме за буквой, соответствующей квалификатору, следуют буквы, определяющие тип аргумента. Типы аргументов и соответствующие им буквы приведены в таблице:

Значение	Тип аргумента
V	Void
C	Char
S	Short
I	Int
L	Long
F	Float
D	Double
G	Long double

Более сложные типы аргументов кодируются в соответствии со следующей таблицей:

Символ	Значение
Цифра	Перечисление или имя класса
P	Near *
R	Near &
M	Far &
N	Far *
A	Array
M	Указатель на член класса (за ним обязаны быть указаны класс и основной тип).

Появление одной или нескольких цифр означает, что после них следует наименование перечисления или класса, а значение числа определяет длину имени перечисления или класса.

Например, функция MyClass::MyFunction(MyClass2) будет представлена в виде @MyClass@MuFunction$qr8MyClass2.

MyClass::MyFunction(OnceMoreClass) примет вид

«MyClass@MyFunction$qr13OnceMoreClass.

Символы «x» и «w» могут появляться после символов «p», «r», «m» и «n» для того чтобы обозначить квалификаторы «Const» и «volatile» соответственно. Символ «q», появляющийся после этих квалификаторов, означает функцию. За символом «q» следуют аргументы этой функции, вплоть до символа «$», после которого в закодированном виде указан тип возвращаемого функцией значения. Ниже я привожу примеры применения этих правил:

Если у меня есть функция MyClass::MyFunction(const char near*), то после преобразования её имя примет вид @MyClass@MyFunction$qpxzc.

В свою очередь, наименование функции MyFunction(const int) будет преобразовано в @MyFunction$qxi.

Вместо функции MyClass::MyFunction(int (near*) (int, int)) возникнет @MyClass@MyFunction$qpqii$i.

Появление массивов фиксируется буквой «a», за которой следует десятичное число, обозначаюее размерность массива, а после символа «$» указывается тип массива. Например, если у меня есть функция MyFunction(int (*x) [20]), то после преобразования возникнет следующее: @MyFunction$qpa20$i.

Закодированные аргументы появляются в том же порядке, в котором они появляются при вызове функции. Символом «t», за которым следует символ ASCII, кодируются повторяющиеся аргументы неосновных типов. Следующий за символом «t» символ определяет номер повторяющегося аргумента. Например, если мы раскодируем запись @MyClass-@MyFunction$qdddiiiiiiipzctata, то получим MyClass::MyFunction(duble, double, double, int, int, int, long, long, long, char near*, char near*, char near*).

Ну, кажется, я рассказал всё то, что мне известно об искажении имён компилятором C++ фирмы «Borland». Я надеюсь, что уважаемый читатель теперь обладает достаточной совокупностью знаний для того, чтобы самостоятельно декодировать искажённые имена и извлекать информацию, необходимую для того ,чтобы использовать функции из библиотек динамической компоновки в тех случаях, когда какая-либо информация об аргументах функций отсутствует. Кстати, практически любая программа, входящая в состав системы программирования фирмы «Borland», содержит искажённые имена. Имея информацию о том, какие функции с какими аргументами вызываются, мы может произвести анализ программы намного более серьёзно, чем обычно.

Импорт функций и механизм импорта

Выше я упомянул, что секция, в которой содержится программный код, неразрывным образом связана с секциями экспорта и импорта функций. Здесь я хотел бы остановиться на том, что такое импорт и каким образом исполняемый файл может экспортировать функции, содержащиеся в нём.

Я надеюсь, что уважаемый читатель имеет опыт программирования для DOS. Давайте попробуем вспомнить, как проходил процесс программирования для DOS. Программист писал программу на одном из языков программирования. При этом для решения тривиальных задач достаточно было использовать функции так называемой библиотеки исполнения компилятора языка программирования. Но что приходилось делать, если задача выходила за рамки тривиальной? Например, если требовалось написать программу, использующую окна? Писать оконный интерфейс от начала до конца? Конечно же, нет. Для решения нетривиальной задачи, но ранее уже кем-то решённой, программист использовал так называемые библиотеки функций. Программа компилировалась, после чего полученный объектный файл линковался с файлом библиотеки. Код библиотечных функций добавлялся к коду программы, то есть объём исполняемого файла увеличивался. Если программа интенсивно использовала библиотеки функций, то код исполняемого файла распухал прямо на глазах.

Выше приведён пример так называемого статического линкования, применяемого в DOS. При статическом линковании код функции из библиотеки становился частью исполняемого файла. Иное дело – программирование для Windows. В общем случае при программировании для DOS можно было обойтись только функциями из библиотеки исполнения и не обращаться к другим библиотекам. Но в Windows это невозможно принципиально. Если бы разработчики Windows пошли бы по пути реализации в этой системе только статического линкования, то исполняемые файлы для Windows поражали бы всех своим громадным размером. Учитывая это, было принято решение наряду с возможностью статического линкования добавить в систему возможность так называемого динамического линкования. Что это значит? При динамическом линковании в теле исполняемого файла код библиотечных функций отсутствует, но при обращении к библиотечной функции управление определённым образом передаётся непосредственно в код библиотеки, то есть некоторым образом происходит импорт кода функции в процесс выполнения программы. Честно говоря, термин «импорт» мне кажется более применимым именно к статическому линкованию, но раз уж «исторически» сложилось так, что этот термин применяется в основном тогда, когда речь идёт о динамическом линковании... Я не стану спорить.

На механизме импорта функций, то есть передачи управления непосредственно в тело библиотеки, я бы хотел остановиться особо.

Вполне очевидно, что та библиотека, в которой находятся необходимые для работы программы функции (в Windows такие библиотеки называются DLL – dynamic linking library – библиотеки динамического линкования), должна быть загружена в память процесса, порождённого вызы-

вающей программой. Эта загрузка может быть осуществлена двумя способами. В зависимости от того, как осуществляется загрузка DLL, различают два типа линкования.

Тип первый – так называемое неявное линкование или линкование при загрузке. Что это значит? При линковании программы мы ЯВНО указываем, что программа использует функции из такой-то DLL, а при запуске программы мы никоим образом не указываем, что какая-то библиотека используется нашей программой, то есть НЕЯВНО даём операционной системе знать о том, что мы работаем с какой-то библиотекой. В результате при запуске нашей программы операционная система проверяет, загружена ли требуемая DLL в память. Если библиотека загружена, то всё в порядке и дальше, как говорится, дело техники. Если же библиотека не загружена, то операционная система пытается найти файл библиотеки на диске и загрузить его. Поиск осуществляется следующим образом:

1. Поиск библиотеки производится в той директории, в которой располагается исполняемый файл, породивший процесс.
2. Поиск библиотеки производится в текущей директории.
3. Поиск производится в системной директории (в случае Windows'95 это обычно директория C:\Windows\System).
4. Поиск библиотеки производится в директории, в которой расположена операционная система Windows (в случае Windows'95 это обычно директория C:\Windows).
5. Поиск производится в директориях, перечисленных в переменной окружения PATH.

В том случае, если DLL не найдена, выполнение программы прекращается и выдаётся сообщение об ошибке.

Иначе обстоит дело при так называемом явном линковании. В этом случае при линковании программа не знает о том, что она импортирует функции из определённой библиотеки. В этом случае происходит следующее:

1. Программа ЯВНО вызывает функцию LoadLibrary() для того, чтобы загрузить в память требуемую DLL.
2. После загрузки требуемой DLL в память программа вызывает функцию GetProcAddress(), которая определяет начальный адрес требуемой функции.
3. Программа передаёт управление по полученному адресу функции. После отработки функции управление передаётся обратно в программу.
4. После того, как программа поработала с библиотекой и библиотека больше не нужна, программа вызывает функцию FreeLibrary(), которая производит удаление DLL из памяти.

И теперь неплохо было бы выяснить, каким образом осуществляется импорт функции в программе.

Вся информация об импортируемых программой функциях, естественно, хранится в секции .idata. Фактически эта секция является последовательностью структур типа IMAGE_IMPORT_DESCRIPTOR. Количество

этих структур нигде не определяется, однако последняя структура в последовательности имеет нулевые поля. Каждая структура этого типа соответствует одной DLL, функции которой импортируются программой. Формат этой структуры можно найти в заголовочном файле winnt.h, я привожу его ниже:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
   union {
      DWORD   Characteristics;                    // 0 for terminating null import descriptor
      PIMAGE_THUNK_DATA OriginalFirstThunk;       // RVA to original unbound IAT
#if defined(__cplusplus) || defined(_ANONYMOUS_UNION)
   };
#else
   }u;
#endif
   DWORD   TimeDateStamp;      // 0 if not bound,
                               // -1 if bound, and real date\time
                               // in IMAGE_DIRECTORY_ENTRY_BOUNT_IMPORT
                               // O.W. date/time stamp of DLL bound to
   DWORD   ForwarderChain;     // -1 if no forwarders
   DWORD   Name;
   PIMAGE_THUNK_DATA FirstThunk;// RVA to IAT
} IMAGE_IMPORT_DESCRIPTOR, *PIMAGE_IMPORT_DESCRIPTOR;
```

Первое поле этой структуры представляет собой объединение. Если эта структура является последней в последовательности, то значение этого поля равно нулю. Во всех остальных случаях это поле содержит смещение объединения типа IMAGE_THUNK_DATA. К нему мы ещё вернёмся, а пока продолжим разговор о структуре IMAGE_IMPORT_DESCRIPTOR. Второе поле, TimeDateStamp, как следует из его названия, должно содержать время создания данного файла. Тем не менее, обычно значение этого поля равно нулю. В поле Name хранится имя той библиотеки, которой соответствует данная структура типа IMAGE_IMPORT_DESCRIPTOR. И последнее поле содержит ещё один указатель на поле типа IMAGE_THUNK_DATA.

Я думаю, читатель уже догадался, что наиболее информативными в структуре IMAGE_THUNK_DATA являются имя DLL и поля, содержащие смещения на IMAGE_THUNK_DATA. Зачем нужно имя библиотеки, понятно. Но зачем нужны два параллельных массива структур типа IMAGE_THUNK_DATA? Кстати, я должен сказать, что каждый элемент типа IMAGE_THUNK_DATA соответствует одной экспортируемой из библиотеки функции.

Формат объединения типа IMAGE_THUNK_DATA описан в заголовочном файле winnt.h. Ниже я привожу формат этого описания:

```
typedef struct _IMAGE_THUNK_DATA {
   union {
      PBYTE  ForwarderString;
      PDWORD Function;
      DWORD Ordinal;
```

```
    PIMAGE_IMPORT_BY_NAME AddressOfData;
  } u1;
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

Для того, чтобы понять назначение полей, я напомню, что функция может импортироваться как по порядковому номеру, так и по имени. Если функция импортируется по порядковому номеру, то мы можем интерпретировать значение IMAGE_THUNK_DATA как её порядковый номер (поле Ordinal). Заметим, что в этом случае старший бит этого двойного слова должен быть установлен в единицу. Например, если поле Ordinal равно 0x80000001, то это значит, что экспортируется первая функция из DLL.

Для того, чтобы определить, каким образом экспортируется функция, можно воспользоваться макросами, определёнными в файле winnt.h. Ниже я привожу эти макросы.

```
#define IMAGE_ORDINAL_FLAG 0x80000000
#define IMAGE_SNAP_BY_ORDINAL(Ordinal) ((Ordinal & IMAGE_ORDINAL_FLAG) != 0)
```

Я уверен, что объяснений эти макросы не требуют, их назначение очевидно.

Совсем по-другому обстоит дело в том случае, если функция эксплуатируется по имени, а не по порядковому номеру. В этом случае IMAGE_THUNK_DATA интерпретируется как структура типа IMAGE_IMPORT_BY_NAME, описанная в файле winnt.h следующим образом:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
  WORD   Hint;
  BYTE   Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Назначение полей этой структуры понятно из их наименований. Первое поле – так называемая подсказка. С этого номера загрузчик начинает поиск соответствующей экспортируемой функции в DLL. Второе поле – это строка ASCII, завершающаяся нулём, содержащая наименование импортируемой функции.

Отличие между двумя параллельными массивами состоит в том, что первый массив (на него указывает поле OriginalFirstThunk) при работе программы не изменяется. Со вторым массивом всё обстоит несколько иначе. Во-первых, загрузчик определяет имя библиотеки, которой принадлежит наша функция (помните, поле Name bp IMAGE_IMPORT_DESCRIPTOR), и, во-вторых, по какому адресу в памяти находится интересующая нас функция. После в массив, адрес которого хранится в поле FirstThunk, загрузчик записывает адрес этой функции. Таким образом, мы можем интерпретировать значение второго IMAGE_THUNK_DATA как указатель на функцию (поле Function). Я думаю, это оправданный подход. От версии к версии адреса функций меняются, а порядковые номера могут оставаться без изменений. Постоянные данные (первый массив) не меняются, меняются только переменные (второй массив). Другими словами, для того, чтобы произвести статический анализ файла, то есть файла, знаходящегося на

диске, достаточно массива ,на который указывает OriginalFirstThunk. При этом мы можем получить порядковый номер или имя импортируемой функции. Если же нам необходимо произвести анализ исполняемого модуля, уже загруженного в память, то имеет смысл проанализировать и тот массив, на который указывает FirstThunk. В этом случае мы получим также адреса, по которым импортируемые нашим модулем функции расположены в памяти. Теперь, после того, как мы разобрали взаимотношения этих структур и массивов, мы можем понять, каким образом происходит импорт функций в Win32.

Для того, чтобы полностью разобраться в хитросплетениях этих данных, естественно, необходим пример. В качестве этого примера я приведу фрагмент одной из моих программ. Этот фрагмент делит окно программы на две части, в одной части появляется список импортируемых библиотек. При «клике» мышкой на имени библиотеки во втором окне появляется список импортируемых программой функций с подсказками.

```
LRESULT CALLBACK ImportWndProc(HWND hImportWnd, UINT Message,
                               UINT wParam, LONG lParam)
{
  HWND hTopListWnd, hBottomListWnd;
  static RECT Rect, rcDividerRect;
  LV_COLUMN LV_Column;
  char cBuffer[MAX_PATH];
  int i = 0, nIcon;
  static HIMAGELIST hImageList = 0;
  HICON hIcon;
  LV_ITEM LV_Item;
  PIMPORTDATA pImportData;
  static double dDividerProportion;
  PAINTSTRUCT PaintStruct;
  HDC hDC;
  static PIMAGE_IMPORT_DESCRIPTOR pIID;
  LPNM_LISTVIEW lpNM;
  PIMAGE_THUNK_DATA pITD;
  PIMAGE_IMPORT_BY_NAME pIIBN;

  switch(Message)
   {
    case WM_CREATE:
      pIID = dwImportOffset;
      if(!hImageList)
        hImageList = ImageList_Create(CX_SMALLICON, CY_SMALLICON,
                                      ILC_MASK, 16, 5);
      if(!hImageList)
        MessageBox(hImportWnd, "Cannot create image list for Tree View",
                   "Error", MB_OK);
       else
        {
         GetClientRect(hImportWnd, &Rect);
         hTopListWnd = CreateWindow(WC_LISTVIEW, "",
                                    LVS_REPORT |
                                    LVS_SORTASCENDING | WS_CHILD,
```

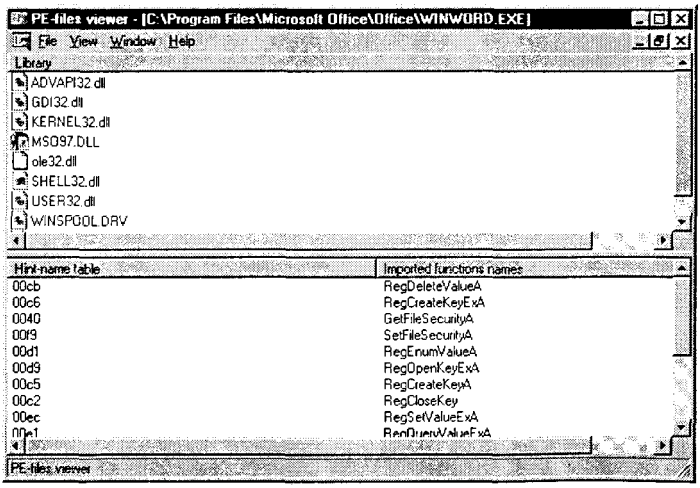
```
                                    CW_USEDEFAULT, CW_USEDEFAULT,
                                    CW_USEDEFAULT, CW_USEDEFAULT,
                                    hImportWnd, NULL, hInst, NULL);
ListView_SetImageList(hTopListWnd, hImageList, LVSIL_SMALL);
ShowWindow(hTopListWnd, SW_SHOW);
hBottomListWnd = CreateWindow(WC_LISTVIEW, "",
                                    LVS_REPORT | WS_CHILD,
                                    CW_USEDEFAULT, CW_USEDEFAULT,
                                    CW_USEDEFAULT, CW_USEDEFAULT,
                                    hImportWnd, NULL, hInst, NULL);
ShowWindow(hBottomListWnd, SW_SHOW);
SetWindowLong(hImportWnd, GWL_USERDATA,
                MAKELONG((WORD) hTopListWnd,
                (WORD) hBottomListWnd));
dDividerProportion = 0.5;
rcDividerRect.left = 0;
rcDividerRect.right = Rect.right;
rcDividerRect.top = (Rect.bottom - Rect.top) * dDividerProportion;
rcDividerRect.bottom = rcDividerRect.top + DIVIDER_HEIGHT;
LV_Column.mask = LVCF_FMT | LVCF_SUBITEM | LVCF_TEXT | LVCF_WIDTH;
LV_Column.fmt = LVCFMT_LEFT;
LV_Column.cx = rcDividerRect.right;
LV_Column.pszText = cBuffer;
LV_Column.iSubItem = 0;
LoadString(hInst, ID_HEADER_DLL, cBuffer, 0x80);
ListView_InsertColumn(hTopListWnd, LV_Column.iSubItem, &LV_Column);
LV_Column.cx = rcDividerRect.right / 2;
for( ; LV_Column.iSubItem < 2; )
 {
  LoadString(hInst, ID_HEADER_HINT_NAMES + LV_Column.iSubItem,
            cBuffer, 0x80);
  ListView_InsertColumn(hBottomListWnd, LV_Column.iSubItem++,
                        &LV_Column);
 }
if(dwPointerToRawData)
  dwNameDifference = dwImportOffset->Name - dwPointerToRawData;
 else
  dwNameDifference = dwImportDifference;
do
 {
  if(!(hIcon = ExtractIcon(hInst, (PBYTE) lpMappedFile +
                                    pIID->Name - dwNameDifference, 0)))
    {
     GetWindowsDirectory(cBuffer, MAX_PATH);
     strcat(cBuffer, "\\system\\shell32.dll");
     hIcon = ExtractIcon(hInst, cBuffer, 61);
    }
  nIcon = ImageList_AddIcon(hImageList, hIcon);
  LV_Item.iItem = i++;
  LV_Item.iSubItem = 0;
  LV_Item.mask = LVIF_TEXT | LVIF_IMAGE;
  LV_Item.pszText = (PBYTE) lpMappedFile +
                                    pIID->Name - dwNameDifference;
```

```
            LV_Item.iImage = nIcon;
            ListView_InsertItem(hTopListWnd, &LV_Item);
            pIID++;
          }
        while(pIID->Characteristics || pIID->FirstThunk);
      }
    ListView_SetItemState( (HWND)
                LOWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                0, LVIS_SELECTED, LVIS_SELECTED);
    return 0;
  case WM_PAINT:
    hDC = BeginPaint(hImportWnd, &PaintStruct);
    DrawEdge(hDC, &rcDividerRect, EDGE_RAISED,
              BF_TOP | BF_MIDDLE | BF_BOTTOM);
    EndPaint(hImportWnd, &PaintStruct);
    return 0;
  case WM_SIZE:
    if(wParam == SIZE_MAXIMIZED)
      bMaxFlag = TRUE;
    if(wParam == SIZE_RESTORED)
      bMaxFlag = FALSE;
    GetClientRect(hImportWnd, &Rect);
    rcDividerRect.top = HIWORD(lParam) * dDividerProportion - DIVIDER_HEIGHT / 2;
    rcDividerRect.right = LOWORD(lParam);
    rcDividerRect.bottom = rcDividerRect.top + DIVIDER_HEIGHT;
    MoveWindow((HWND)
                LOWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                0, 0, LOWORD(lParam), rcDividerRect.top, TRUE);
    MoveWindow((HWND)
                HIWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                0, rcDividerRect.bottom, LOWORD(lParam),
                HIWORD(lParam) - rcDividerRect.bottom, TRUE);
    return DefMDIChildProc(hImportWnd, Message, wParam, lParam);
  case WM_NOTIFY:
    lpNM = (LPNM_LISTVIEW) lParam;
    if(LOWORD(GetWindowLong(hImportWnd, GWL_USERDATA)) ==
      (WORD) lpNM->hdr.hwndFrom)
      {
        switch(lpNM->hdr.code)
        {
          case LVN_ITEMCHANGING:
            ListView_DeleteAllItems((HWND)
                HIWORD(GetWindowLong(hImportWnd, GWL_USERDATA)));
            return 0;
          case LVN_ITEMCHANGED:
            if(lpNM->uNewState & LVIS_SELECTED)
              {
                ListView_GetItemText( (HWND)
                  LOWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                  lpNM->iItem, 0, cBuffer, MAX_PATH);
                pIID = dwImportOffset;
                while(strcmp(cBuffer, (PBYTE) lpMappedFile + pIID->Name -
                            dwNameDifference) != 0)
```

```
    pIID++;
  if(pIID->Characteristics)
    pITD = (PIMAGE_THUNK_DATA)
           ( (PBYTE) lpMappedFile +
           pIID->Characteristics - dwImportDifference);
   else
    pITD = (PIMAGE_THUNK_DATA)
           ( (PBYTE) lpMappedFile +
           (DWORD) pIID->FirstThunk - dwImportDifference);
  i = 0;
  while(pITD->u1.Ordinal)
   {
    pImportData = new IMPORTDATA;
    LV_Item.mask = LVIF_TEXT | LVIF_PARAM;
    LV_Item.iItem = i;
    LV_Item.iSubItem = 0;
    LV_Item.state = LV_Item.stateMask = 0;
    if(pITD->u1.Ordinal & IMAGE_ORDINAL_FLAG)
      {
       pImportData->wOrdinal = IMAGE_ORDINAL(pITD->u1.Ordinal);
       pImportData->cName = "";
       sprintf(cBuffer, "%04x", IMAGE_ORDINAL(pITD->u1.Ordinal));
       LV_Item.pszText = cBuffer;
       LV_Item.lParam = (LPARAM) pImportData;
       ListView_InsertItem( (HWND)
           HIWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                                        &LV_Item);
      }
     else
      {
       pIIBN = (PIMAGE_IMPORT_BY_NAME)
               ( (PBYTE) lpMappedFile +
               (DWORD) pITD->u1.AddressOfData -
                dwNameDifference);
       pImportData->wOrdinal = pIIBN->Hint;
       pImportData->cName =  pIIBN->Name;
       sprintf(cBuffer, "%04x", pIIBN->Hint);
       LV_Item.pszText = cBuffer;
       LV_Item.lParam = (LPARAM) pImportData;
       ListView_InsertItem( (HWND)
           HIWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                                        &LV_Item);
       LV_Item.mask = LVIF_TEXT;
       LV_Item.iSubItem = 1;
       LV_Item.pszText = pIIBN->Name;
       ListView_SetItem( (HWND)
           HIWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                                        &LV_Item);
      }
    i++;
    pITD++;
   }
}
```

```
            return 0;
          default:
            break;
        }
      }
    else
      {
        switch(lpNM->hdr.code)
          {
            case LVN_COLUMNCLICK:
              ListView_SortItems((HWND)
                        HIWORD(GetWindowLong(hImportWnd, GWL_USERDATA)),
                                Compare2, (LPARAM) lpNM->iSubItem);
              break;
            default:
              break;
          }
      }
    return 0;
  case WM_DESTROY:
    nOpenImportWindows--;
    if(!nOpenImportWindows)
      ImageList_Destroy(hImageList);
    nOpenWindows--;
    if(!nOpenWindows)
      SendMessage(hWnd, WM_COMMAND,
                        MAKEWPARAM(IDM_CLOSE_ALL, 0), (LPARAM) 0);
    return 0;
  default:
    return DefMDIChildProc(hImportWnd, Message, wParam, lParam);
  }
}
```

Окно, создаваемое этой программой, приведено ниже.



Итак, мы рассмотрели разделы экспорта и импорта, то есть те разделы, которые имеют отношение непосредственно к программированию. Но ведь в исполняемом файле хранится масса данных, которые используются программами в основном с целью создания интерфейса окна. Это так называемые ресурсы.

Ресурсы в исполняемом файле

Очередным разделом исполняемого файла является раздел ресурсов. Из этого раздела мы также постараемся извлечь всю возможную информацию о ресурсах, имеющихся в программе. Мы постараемся раскопать всё, что можно, выяснить формат записи о каждом типе ресурса, выяснить всё, что нам необходимо для того ,чтобы найти информацию о ресурсах в исполняемом файле, при необходимости визуализировать ресурсы или декомпилировать их, то есть получить о них полное представление, а также научиться манипулировать ресурсами. Как всегда, первый вопрос, который может возникнуть, - каким образом мы можем найти смещение раздела ресурсов в исполняемом файле? Тот алгоритм, который я изложил при описании раздела экспорта, он в достаточной степени универсален. Для нахождения смещения раздела ресурсов необходимо:

1. Из массива IMAGE_DATA_DIRECTORY выбираем запись с индексом IMAGE_DIRECTORY_ENTRY_RESOURCE, то есть вторую запись (напомню, что отсчёт записей ведётся с нуля).
2. Из этой записи выбираю поле VirtualAddress.
3. Перебираю таблицу разделов до тех пор, пока не найду запись, соответствующую разделу с именем «.rsrc».
4. Определяю поправку как разность между значениями полей VirtualAddress и PointerToRawData.
5. Вычисляю смещение раздела ресурсов в исполняемом файле относительно начала файла как разность между полем VirtualAddress, выбранном на втором шаге, и поправкой на четвёртом шаге.

Теперь, когда смещение раздела ресурсов мы вычислили, нам необходимо разобраться с его структурой и назначением каждого из его полей. Я думаю, уважаемый читатель уже догадался, что в начале раздела ресурсов, как и в начале других разделов, находится

Оглавление раздела ресурсов

Формат этого оглавления мы можем найти в файле winnt.h:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
   DWORD   Characteristics;
   DWORD   TimeDateStamp;
   WORD    MajorVersion;
   WORD    MinorVersion;
   WORD    NumberOfNamedEntries;
   WORD    NumberOfIdEntries;
// IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[];
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

Я не случайно оставил в тексте строку комментария, которая присутствует в файле winnt.h. Определенный смысл в этом есть. Я надеюсь, что читатель, прочитав раздел о ресурсах, согласится со мной.

Поле Characteristics в настоящее время не используется и должно быть заполнено нулями. Следующие три поля содержат соответственно время создания файла, а также старшую и младшую части версии файла. Если, конечно, эти поля не используются с какими-то особыми целями, например, для защиты информации, для системного анализа никакого интереса эти поля не представляют.

А вот о следующих двух полях следует сказать особо. Дело в том, что ресурсы хранятся как бы двумя частями. Первая часть - это ресурсы, у которых есть имена. Имена - это строки, они отсортированы по алфавиту в порядке возрастания, прописные и строчные буквы при этом не различаются. Вторая часть - это ресурсы, идентифицирующиеся по номеру. Они расположены сразу за массивом имен. Эти идентификаторы также отсортированы в возрастающем порядке. Такое построение позволяет осуществлять быстрый поиск как по имени, так и по номеру ресурса. Необходимо заметить, что каждый ресурс может идентифицироваться либо по имени, либо по номеру, но не по имени и номеру одновременно. Это связано с синтаксисом .RC- и .RES-файлов, а также со структурой таблицы ресурсов.

Но вернемся к теме. Поля NumberOfNamedEntries и NumberOfIdEntries соответственно указывают число ресурсов, идентифицирующихся по имени и число ресурсов, идентифицирующихся по номеру. Таким образом, зная содержимое только этих двух полей, мы получаем самое первое представление о количестве ресурсов, хранящихся в исследуемом файле – общее число ресурсов равно сумме ресурсов, идентифицирующихся по имени и ресурсов, идентифицирующихся по номеру. Я хотел бы, чтобы читатель запомнил формат оглавления, ибо к нему мы еще не раз вернемся.

Строки таблицы ресурсов

Сразу за оглавлением таблицы ресурсов следуют непосредственно строки таблицы ресурсов. Число этих строк равно Формат строки ресурсов опять таки описан в файле winnt.h и приведён ниже:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
   union {
      struct {
         DWORD NameOffset:31;
         DWORD NameIsString:1;
#if defined(_ANONYMOUS_STRUCT)
            };
#else
            }s;
#endif
      DWORD   Name;
      WORD    Id;
#if defined(_ANONYMOUS_UNION) || !defined(__BORLANDC__)
         };
```

```
#else
        }u;
#endif
   union {
      DWORD   OffsetToData;
      struct {
         DWORD   OffsetToDirectory:31;
         DWORD   DataIsDirectory:1;
#if defined(_ANONYMOUS_STRUCT)
            };
#else
               }s;
#endif
#if defined(_ANONYMOUS_UNION) || !defined(__BORLANDC__)
        };
#else
        }u2;
#endif
}IMAGE_RESOURCE_DIRECTORY_ENTRY,
 *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Давайте попробуем разобраться в этом подробнее. Из описания видно, что каждая строка таблицы ресурсов состоит из двух частей, двух объединений. Каждая часть имеет своё назначение. Мы рассмотрим их последовательно.

Первая часть определяет, имеет ли ресурс имя или он идентифицируется по номеру. Определяется это по старшему биту первого двойного слова. Если старший бит первого двойного слова установлен в единицу, это значит, что у ресурса, определяемого данной строкой, есть имя, а не идентификатор. В этом случае оставшиеся 31 бит определяют смещение относительно начала таблицы ресурсов структуры (!), хранящей имя ресурса. Если старший бит строки равен 0, то в младшем слове этого двойного слова хранится 16-битный идентификатор ресурса.

Для того чтобы определить, каким образом идентифицируется ресурс, можно воспользоваться макросом

IMAGE_RESOURCE_NAME_IS_STRING

котрый в файле winnt.h описан следующим образом:

```
#define IMAGE_RESOURCE_NAME_IS_STRING        0x80000000
```

Структура, хранящая имя ресурса, описана следующим образом:

```
typedef struct _IMAGE_RESOURCE_DIR_STRING_U {
   WORD    Length;
   WCHAR   NameString[ 1 ];
}IMAGE_RESOURCE_DIR_STRING_U,
 *PIMAGE_RESOURCE_DIR_STRING_U;
```

Первый элемент структуры – поле Length – содержит длину наименования ресурса. Второе поле – NameString[1] – это первый символ наименования ресурса, причём наименование ресурса хранится в коде Unicode.

Из этого следует, что для обычной обработки строки необходимо либо преобразовать ее из Unicode в ANSI, либо работать в Unicode. Кроме этого, нужно не забывать добавлять нулевой код к концу строки. Возникает один закономерный вопрос - почему не используются обычные стандартные строки, завершающиеся нулевым кодом? Я могу только предположить, что это связано с тем, что в таком случае легче и быстрее вычисляется смещение байта, следующего за последним байтом строки, ибо сразу за строкой с именем ресурса хранятся данные, составляющие ресурс.

Перед тем, как рассказывать о втором двойном слове, я бы хотел несколько слов сказать об иерархии ресурсов. Дело в том, что все ресурсы одного типа (курсоры, иконки, диалоговые окна и так далее) объединяются в группы, каждая из которых начинается с оглавления точно такого же формата, что и основное оглавление таблицы ресурсов. Эти группы я называю первым уровнем иерархии ресурсов. Внутри группы ресурсы, имеющие одинаковые имена или одинаковые идентификаторы, но предназначенные для разноязычных пользователей (к примеру, диалоговое окно с русскими и английскими строками), объединяются в подгруппы. Я называю это вторым уровнем иерархии ресурсов. Подгруппы, естественно, также имеют оглавление, причём формат этого оглавления точно такой же, что и у основного оглавления таблицы ресурсов. И третий уровень иерархии ресурсов – ресурсы разделяются по языку. Всего уровней может быть тридцать два, но в настоящее время используются только три (тип, идентификатор или имя, язык). Если, наконец, мы доходим до элемента, а не группы, то мы должны получить указатель на данные, из которых необходимо формировать ресурс.

А теперь вернёмся ко второй части описания ресурса. Старший бит второго двойного слова строки таблицы ресурсов и определяет, ссылается ли текущая строка таблицы ресурсов на оглавления более низкого уровня. Бит установлен в единицу - текущая строка содержит оглавления. В этом случае оставшиеся 31 бит содержат смещение подоглавления относительно начала раздела ресурсов. Бит не установлен - оставшиеся биты содержат смещение данных, определяющих ресурс. Как и в предыдущем случае, для анализа старшего бита можно воспользоваться макросом IMAGE_RESOURCE_DATA_IS_DIRECTORY, который и в данном случае принимает значение 0x80000000.

Теперь обратимся к вопросу о том, каие идентификаторы ресурсов мы используем при анализе оглавлений на всез трёх уровнях иерархии. Идентификатор ресурса, используемый на первом уровне, фактически является номером типа ресурса. Ниже в таблице я привожу используемые в Windows номера типов ресурсов. Но для того, чтобы читатель смог понять, каким образом определяются RT_GROUP_CURSOR и RT_GROUP_ICON, я должен сказать, что макрос DIFFERENCE описан в файле winuser.h следуюим образом:

```
#define DIFFERENCE      11
```

А теперь непосредственно таблица.

Идентификатор	Значение
RT_CURSOR	MAKEINTRESOURCE(1)
RT_BITMAP	MAKEINTRESOURCE(2)
RT_ICON	MAKEINTRESOURCE(3)
RT_MENU	MAKEINTRESOURCE(4)
RT_DIALOG	MAKEINTRESOURCE(5)
RT_STRING	MAKEINTRESOURCE(6)
RT_FONTDIR	MAKEINTRESOURCE(7)
RT_FONT	MAKEINTRESOURCE(8)
RT_ACCELERATOR	MAKEINTRESOURCE(9)
RT_RCDATA	MAKEINTRESOURCE(10)
RT_MESSAGETABLE	MAKEINTRESOURCE(11)
RT_GROUP_CURSOR	MAKEINTRESOURCE((DWORD) RT_CURSOR + DIFFERENCE)
RT_GROUP_ICON	MAKEINTRESOURCE((DWORD) RT_ICON + DIFFERENCE)
RT_VERSION	MAKEINTRESOURCE(16)
RT_DLGINCLUDE	MAKEINTRESOURCE(17)
RT_PLUGPLAY	MAKEINTRESOURCE(19)
RT_VXD	MAKEINTRESOURCE(20)

Например, пятерка говорит о том, что описываемый ресурс является окном диалога. Идентификатор, который встречается на втором уровне, является действительно идентификатором ресурса. И, наконец, последний идентификатор определяет, для какого языка применяется этот ресурс.

Формат данных, описывающих ресурсы

Когда мы доходим до описания конкретного ресурса, а не оглавления, нам приходится иметь дело уже со структурами другого типа. Структура, определяющая размещение данных, описывается следующим образом:

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
   DWORD  OffsetToData;
   DWORD  Size;
   DWORD  CodePage;
   DWORD  Reserved;
} IMAGE_RESOURCE_DATA_ENTRY,
  *PIMAGE_RESOURCE_DATA_ENTRY;
```

Поле OffsetToData определяет смещение данных о ресурсе относительно начала таблицы ресурсов в образе загруженного файла. Честно говоря, ни в одном из доступных мне источников я не нашел даже упоми-

нания о том, как определить смещение данных не в образе, а в файле. Наверное, можно было бы посчитать число описываемых ресурсов, умножить это число на размер IMAGE_RESOURCE_DATA_ENTRY и получить смещение данных. Но меня такой способ не устраивал. Я нашел свой способ определения смещения относительно начала таблицы ресурсов. Проанализировав текст предлагаемой ниже демонстрационной программы, пользователь поймет, в чем состоит этот способ.

Наименование следующего поля - Size - говорит само за себя - оно определяет число байтов, занимаемых описываемым ресурсом, CodePage определяет номер кодовой страницы, которая должна использоваться при декодировании данных. Четвертое поле является зарезервированным (чем-то его заполнит Microsoft? ☺).

Что ж, мы рассмотрели все те структуры, которые описывают ресурсы, а также взаимосвязи между ними. Теперь для того, чтобы всё расставить на свои места, я предлагаю читателю то, о чём мы здесь говорили, записать в виде последовательности шагов:

1. Находим общее число ресурсов в файле как сумму полей NumberOfNamedEntries и NumberOfIdEntries из структуры IMAGE_RESOURCE_DIRECTORY.
2. Анализируем каждый ресурс, используя для этого структуру типа IMAGE_RESOURCE_DIRECTORY_ENTRY. Проверяем, с чем мы работаем на данном этапе - с группой ресурсов или отдельным ресурсом.
3. Если мы анализируем группу ресурсов, то запоминаем на каком уровне иерархии ресурсов мы находимся. Если мы находимся на самом верхнем уровне, то идентификаторы ресурсов являются ТИПАМИ РЕСУРСОВ (я привёл их в таблице выше). Если мы находимся на втором уровне, то идентификатор ресурсов является действительно идентификатором ресурса. Если мы находимся на третьем уровне, то идентификатор ресурсов является идентификатором языка, который используется данным ресурсом. После этого рекурсивно можно повторять анализ до тех пор, пока мы не дойдём до отдельного ресурса.
4. Если мы дошли до отдельного ресурса, то его анализ будет зависеть от того типа, к которому он принадлежит.

С именами и директориями все ясно. Сложности начинаются в тот момент, когда мы приступаем к анализу данных, связанных с элементом. Здесь придется анализировать свои структуры для каждого типа ресурсов, а стандартных типов ресурсов, как читатель мог заметить несколько раньше, на настоящий момент существует около двадцати. Я постараюсь в этой книге рассмотреть наиболее важные типы ресурсов. Наверное, начать можно с одного из наиболее часто использующихся типов ресурсов - диалоговых окон. Но перед этим - небольшая демонстрационная программа, которая позволяет определить, какие ресурсы присутствуют в исследуемом файле.

При запуске программы ей необходимо в качестве единственного параметра передать имя исследуемого файла. При работе программы ей необходим файл заголовков, который приводится ниже.

```
#define IDC_Listbox 101
#define IDC_SYSTREEVIEW1            101

BOOL CALLBACK ResourcesProc(HWND, UINT, WPARAM, LPARAM);
void Cls_OnClose(HWND);
BOOL Cls_OnInitDialog(HWND, HWND, LPARAM);
void AnalyzeResourceData(void*, HTREEITEM, int);
HTREEITEM InsertItemInTree(HWND, HTREEITEM, HTREEITEM, UINT,
                           LPSTR, int, int, LPARAM);
```

Кроме этого, программе необходим файл ресурсов:

```
#include "Resources.h"
Resources DIALOG 0, 0, 264, 187
STYLE DS_3DLOOK | WS_POPUP | WS_VISIBLE | WS_CAPTION |
      WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX
CAPTION "Resources in File"
FONT 8, "MS Sans Serif"
{
 CONTROL "Tree", IDC_SYSTREEVIEW1, "SysTreeView32", 39 | WS_CHILD |
            WS_VISIBLE | WS_BORDER, 4, 4, 256, 178
}
```

И, самое главное, текст программы:

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <stdio.h>
#include "Resources.h"

HANDLE hFile = 0;
HWND hTree;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR pszCmdLine, int nCmdShow)
 {
  if(!lstrcmp(pszCmdLine, ""))
    {
     MessageBox(NULL, "USAGE: Resources <file name>", "User's Error",
                MB_OK);
    }
   else
    {
     if( (hFile = CreateFile(pszCmdLine, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
        OPEN_EXISTING, 0, 0)) != INVALID_HANDLE_VALUE)
       {
        InitCommonControls();
        DialogBox(hInstance, "Resources", NULL, ResourcesProc);
       }
```

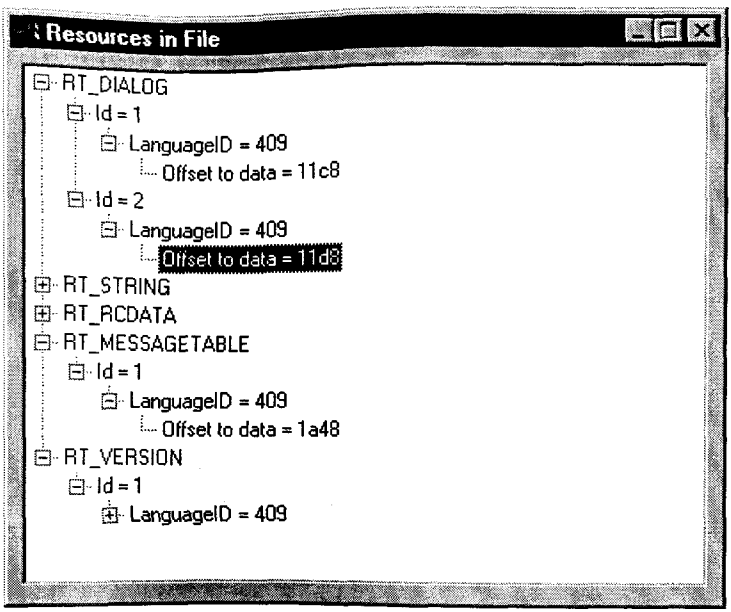
```
        else
          MessageBox(NULL, "Cannot open file", "File's name error", MB_OK);
      }
    if(hFile)
      CloseHandle(hFile);
    return 0;
  }

BOOL CALLBACK ResourcesProc(HWND hDlg, UINT Message,
                                    WPARAM wParam, LPARAM lParam)
  {
    switch(Message)
     {
      HANDLE_MSG(hDlg, WM_CLOSE, Cls_OnClose);
      HANDLE_MSG(hDlg, WM_INITDIALOG, Cls_OnInitDialog);
     }
    return FALSE;
  }

void Cls_OnClose(HWND hDlg)
  {
    EndDialog(hDlg, 0);
  }

BOOL Cls_OnInitDialog(HWND hDlg, HWND hWndFocus, LPARAM lParam)
  {
    char cBuffer[0x100];
    DWORD dwNOBR, dwCurrentPosition;
    WORD wNumberOfSections;
    int i;
    IMAGE_SECTION_HEADER I_S_H;
    IMAGE_EXPORT_DIRECTORY I_E_D;
    PDWORD pdwPointerInEAT, pdwPointerInENT;
    PWORD pwPointerInEOT;
    PCHAR pcPointer;

    hTree = GetDlgItem(hDlg, IDC_SYSTREEVIEW1);
    ReadFile(hFile, cBuffer, 0x40, &dwNOBR, NULL);
            if( cBuffer[0x18] >= 0x40 )
      {
       SetFilePointer(hFile, *((LPDWORD)(cBuffer + 0x3c)), NULL,
                      FILE_BEGIN);
       ReadFile(hFile, cBuffer, 0x18, &dwNOBR, NULL);
       if(lstrcmp((LPSTR) cBuffer, "PE0"))
          {
           wNumberOfSections = MAKEWORD(cBuffer[6], cBuffer[7]);
           SetFilePointer(hFile, MAKEWORD(cBuffer[0x14], cBuffer[0x15]),
                          NULL, FILE_CURRENT);
           for(i = 0; i < wNumberOfSections; i++)
            {
             ReadFile(hFile, &I_S_H, IMAGE_SIZEOF_SECTION_HEADER,
                      &dwNOBR, NULL);
             dwCurrentPosition = SetFilePointer(hFile, 0, NULL, FILE_CURRENT);
```

```
            SetFilePointer(hFile, I_S_H.PointerToRawData, NULL,
                           FILE_BEGIN);
            if(!strcmp(I_S_H.Name, ".rsrc"))
              {
               void* pPointer = GlobalAlloc(GMEM_FIXED,
                                            I_S_H.SizeOfRawData);
               ReadFile(hFile, pPointer, I_S_H.SizeOfRawData, &dwNOBR,
                        NULL);
               AnalyzeResourceData(pPointer, TVI_ROOT, 0);
              }
            SetFilePointer(hFile, dwCurrentPosition, NULL, FILE_BEGIN);
           }
         }
      }
   return TRUE;
 }

void AnalyzeResourceData(void* pPointer, HTREEITEM hParent, int nOffset)
 {
   int j;
   PIMAGE_RESOURCE_DIRECTORY pIRD;
   PIMAGE_RESOURCE_DIRECTORY_ENTRY pIRDE, pIRDE_Temp;
   HTREEITEM hNewParentItem;
   WORD wNumberOfEntries;
   static WORD wOffset;
   static short int nLevel;
   char cBuffer[256];
   LPSTR pszResourceTypes[] = {"RT_CURSOR", "RT_BITMAP", "RT_ICON",
                               "RT_MENU", "RT_DIALOG",
                               "RT_STRING", "RT_FONTDIR", "RT_FONT",
                               "RT_ACCELERATOR", "RT_RCDATA",
                               "RT_MESSAGETABLE",
                               "RT_GROUP_CURSOR", "",
                               "RT_GROUP_ICON", "", "RT_VERSION",
                               "RT_DLGINCLUDE", "", "RT_PLUGPLAY",
                               "RT_VXD"};

   pIRD = (PBYTE) pPointer + nOffset;
   pIRDE_Temp = (PBYTE) pIRD + sizeof(IMAGE_RESOURCE_DIRECTORY);
   wNumberOfEntries = pIRD->NumberOfNamedEntries +
                      pIRD->NumberOfIdEntries;
   for( j = 0 ; j <  wNumberOfEntries; j++, pIRDE_Temp++)
    {
     if(pIRDE_Temp->u.s.NameIsString)
       {
        PIMAGE_RESOURCE_DIR_STRING_U pIRDSU = (PBYTE) pPointer
                                              + pIRDE_Temp->u.Id;
        cBuffer[WideCharToMultiByte(CP_ACP, 0, &(pIRDSU->NameString),
                                    pIRDSU->Length, cBuffer, 256, NULL, NULL)] = 0;
        hNewParentItem = InsertItemInTree(hTree, hParent, TVI_LAST,
                                          TVIF_TEXT, cBuffer, 0, 0, 0);
       }
      else
```

```
        if(nOffset == 0)
          if(pIRDE_Temp->u.Id <= 20)
            {
             wOffset = pIRDE_Temp->u.Id;
             hNewParentItem = InsertItemInTree(hTree, hParent, TVI_LAST,
                                    TVIF_TEXT,
                                    pszResourceTypes[pIRDE_Temp->u.Id - 1],
                                    0, 0, 0);
            }
           else
            ;
         else
          {
           switch(nLevel)
            {
             case 0:
              sprintf(cBuffer, "%d", pIRDE_Temp->u.Id);
              break;
             case 1:
              sprintf(cBuffer, "Id = %d", pIRDE_Temp->u.Id);
              break;
             case 2:
              sprintf(cBuffer, "LanguageID = %x", pIRDE_Temp->u.Id);
              break;
            }
           hNewParentItem = InsertItemInTree(hListsWnd[1], hParent,
                                    TVI_LAST, TVIF_TEXT, cBuffer, 0, 0, 0);
          }
      if(pIRDE_Temp->u2.s.DataIsDirectory)
        {
         nLevel++;
         AnalyzeResourceData(pPointer, hNewParentItem,
                                pIRDE_Temp->u2.s.OffsetToDirectory);
         nLevel--;
        }
       else
        {
         sprintf(cBuffer, "Offset to data = %x", pIRDE_Temp->u2.OffsetToData);
         InsertItemInTree(hTree, hNewParentItem, TVI_LAST, TVIF_TEXT,
                        cBuffer, 0, 0, 0);
        }
    }
}

HTREEITEM InsertItemInTree(HWND hWnd, HTREEITEM hParent,
                           HTREEITEM hInsertAfter, UINT mask,
                           LPSTR pszText, int iImage, int iSelectedImage,
                           LPARAM lParam)
{
  TV_INSERTSTRUCT InsertStruct;

  InsertStruct.hParent = hParent;
  InsertStruct.hInsertAfter = hInsertAfter;
```

```
  InsertStruct.item.mask = mask;
  InsertStruct.item.pszText = pszText;
  InsertStruct.item.iImage = iImage;
  InsertStruct.item.iSelectedImage = iSelectedImage;
  InsertStruct.item.lParam = lParam;
  return TreeView_InsertItem(hWnd, &InsertStruct);
}
```

А теперь я приведу внешний вид создаваемого программой окна. В качестве параметра ей передано имя «c:\windows\system\kernel32.dll»:



Ресурс типа «диалоговое окно»

Описание диалогового окна состоит из двух частей. Первая часть описывает непосредственно диалоговое окно как таковое, вторая часть описывает элементы управления, которые входят в состав диалогового окна. В совокупности все эти описания составляют так называемый шаблон (template) диалогового окна. Для того чтобы посмотреть, что находится внутри этого диалогового окна, нам ничего, кроме этого шаблона, не нужно. Но ведь наша цель - не просто посмотреть, а извлечь максимум информации из исследуемого файла! Поэтому сейчас -

Описание диалогового окна

В том случае, если тип ресурса - диалоговое окно, то за структурой типа IMAGE_RESOURCE_DATA_ENTRY, о которой мы говорили чуть

раньше, следует заголовок диалогового окна. Этот заголовок описан в файле winuser.h:

```
typedef struct {
   DWORD style;
   DWORD dwExtendedStyle;
   WORD cdit;
   short x;
   short y;
   short cx;
   short cy;
} DLGTEMPLATE;
```

Первые два поля содержат стили и дополнительные стили диалогового окна. Возможные значения стилей я привожу в таблице ниже.

Стиль	Значение	Назначение
DS_ABSALIGN	0x01L	Положение диалогового окна исчисляется в экранных координатах
DS_SYSMODAL	0x02L	Создается системное модальное диалоговое окно
DS_3DLOOK	0x0004L	Создается диалоговое окно, имеющее зрительную иллюзию трехмерности
DS_FIXEDSYS	0x0008L	Вместо SYSTEM_FONT используется SYSTEM_FIXED_FONT
DS_NOFAILCREATE	0x0010L	Диалоговое окно создается, несмотря на то, что при его создании произошли ошибки
DS_LOCALEDIT	0x20L	В 32-битных приложениях не используется
DS_SETFONT	0x40L	Определяется шрифт, который будет применяться в диалоговом окне
DS_MODALFRAME	0x80L	Создается модальное диалоговое окно
DS_NOIDLEMSG	0x100L	
DS_SETFOREGROUND	0x200L	Поместить диалоговое окно на передний план
DS_CONTROL	0x400L	
DS_CENTER	0x800L	Диалоговое окно помещается в центр рабочей области
DS_CENTERMOUSE	0x1000L	
DS_CONTEXTHELP	0x2000L	

Естественно, что помимо стилей, указанных в таблице, диалоговое окно может использовать и стили, обозначения которых начинаются с «WS_», то есть стили, применимые ко всем окнам.

Третье поле - число элементов управления в диалоговом окне, четвертое и пятое поля - координаты левого верхнего угла окна, шестое и седьмое поля - ширина и высота окна соответственно. Ни одно из этих полей, надеюсь, не вызывает никаких вопросов.

За заголовком следуют еще несколько полей, которые полностью описывают диалоговое окно (но не элементы управления в диалоговом окне!). Причем обратите внимание, уважаемый читатель, несмотря на то, что размер каждого из этих полей переменный, общая длина этих полей выровнена по границе двойного слова. Первыми расположены два поля типа [Name or Ordinal]. Этот тип используется при описании ресурсов в тех случаях, когда за полем этого типа может следовать или строка Unicode

(скажем, имя ресурса), или число (например, идентификатор ресурса). Поле этого типа имеет переменный размер. Если первое слово этого поля равно 0xffff (символ, отсутствующий в Unicode), то следующее слово содержит идентификатор ресурса. В противном случае далее следует строка Unicode с именем ресурса. Из этого правила есть единственное исключение - если должна быть записана пустая строка Unicode, то первое и единственное слово равно 0.

Но вернемся к формату описания диалогового окна. Первое поле типа [Name or Ordinal] содержит имя или идентификатор меню. Второе поле этого типа содержит имя или идентификатор класса диалогового окна. Третье поле, которое следует за первыми двумя, хранит в себе строку Unicode, содержащую заголовок диалогового окна.

Еще два поля присутствуют в описании диалогового окна только в том случае, если в описании ресурса в файле типа .RC присутствует описание шрифта, используемого диалоговым окном. Другими словами, если значение выражения (style & DS_SETFONT) равно 1, то поля присутствуют в описании диалогового окна. Поле wPointSize типа WORD содержит в себе размер шрифта, а поле szFontName[] типа WCHAR содержит в себе наименование шрифта.

Все! С описанием диалогового окна как такового покончено! За ним следует

Описание элемента управления в составе диалогового окна

Сразу за описанием диалогового окна расположены описания элементов диалогового окна. Все они имеют одинаковый формат, что чрезвычайно облегчает их анализ.

Структура, определяющая формат описания элементов диалога, определена в файле winuser.h и приведена ниже:

```
typedef struct {
   DWORD style;
   DWORD dwExtendedStyle;
   short x;
   short y;
   short cx;
   short cy;
   WORD id;
} DLGITEMTEMPLATE;
```

Первые два поля функционально подобны первым двум полям структуры типа DLGTEMPLATE и тоже определяют стили и дополнительные стили элемента управления. Следующие четыре поля определяют положение элемента управления внутри родительского окна. И, наконец, последний элемент определяет идентификатор элемента управления. Таким образом, вроде бы имен у элементов управления быть не может. Но, как говорится, так ли это? А как мы определим, какого типа этот элемент? Кнопка это, список или, скажем, статическая надпись? И откуда мы узнаем, что должно быть написано на кнопочке? А для этих целей существуют три

массива, которые следуют непосредственно за структурой типа DLGITEMTEMPLATE. Каждый из этих массивов состоит из одного или нескольких шестнадцатиразрядных элементов.

Первый массив (его называют массив класса) определяет тип элемента управления. Если первый элемент массива класса равен 0xffff, то в этом случае элемент управления является типовым, то есть уже определён в Windows. В таком случае массив класса содержит ещё один элемент, определяющий этот тип. Возможные типы и их значения я привожу в таблице ниже.

Значение	Тип элемента управления
0x0080	Кнопка (button)
0x0081	Поле ввода текста (edit)
0x0082	Статический элемент, надпись (static text)
0x0083	Окно списка (list box)
0x0084	Полоса прокрутки (scroll bar)
0x0085	Окно списка с полем ввода текста (combo box)

Но если первый элемент этого массива не равен 0xffff, то тогда за первым элементом следуют несколько символов Unicode, которые составляют имя зарегистрированного приложением типа элемента.

Второй массив, который называют массивом заголовка, следует сразу же за массивом класса. В нём записан заголовок элемента управления. Тут сразу же возникает проблема – а как быть, скажем, если элемент в принципе не может иметь заголовка? Иконка, например, или, скажем, курсор? Проблема решается точно так же, как и в случае первого массива. Если первый элемент массива заголовка равен 0xffff, то тогда массив заголовка содержит ещё один элемент, который содержит идентификатор ресурса, курсора, например, или той же иконки. В противном случае за первым элементом следуют несколько символов Unicode, составляющих заголовок элемента.

И, наконец, третий массив. Он не имеет какого-то определённого формата и содержит данные, специфические для данного приложения. Эти данные Windows передаёт элементу при посредстве функции CreateWindowEx(). Приложение должно понимать содержание и формат этих данных.

Итак, с диалоговыми окнами мы закончили. На очереди –

Ресурс типа «меню»

Наверное, мой уважаемый читатель уже привык к тому, что описание любой структуры начинается с описания. Не являются исключением и структуры, связанные с меню. Описание меню начинается со структуры типа MENUITEMTEMPLATEHEADER, которая в файл winuser.h описывается следующим образом:

```
typedef struct {
   WORD versionNumber;
   WORD offset;
} MENUITEMTEMPLATEHEADER, *PMENUITEMTEMPLATEHEADER;
```

Поле versionNumber определяет номер версии. Но версии чего? Я перерыл горы материалов и пытался найти ответ на этот вопрос. Но нигде не указно, версия ЧЕГО определяется в структуре. Что ж, придётся просто принять к сведению, что это номер версии чего-то. Правда, в различных описаниях, и, в частности, в файле помощи по Win32 API, который поставляется с Borland C++, указывается, что это поле должно всегда быть равным нулю. Тогда мне вовсе непонятно, о какой версии идёт речь. В таком случае я возьму на себя смелость утверждать, что в настоящий момент это поле зарезервировано для будущего использования, что его значение должно быть равно нулю и что в будущем, вероятно, оно будет содержать номер версии чего-то. Как, ничего утверждение? ☺

Второе поле, offset, содержит смещение в байтах списка описаний элементов данного меню, отсчитанное от конца заголовка. Обычно, как пишется в разных описаниях, это поле равно нулю, потому что список описаний элементов меню следует непосредственно за структурой типа MENUITEMTEMPLATEHEADER. Этот список описаний элементов меню состоит из структур типа MENUITEMTEMPLATE. Каждая структура описывает один элемент меню. Эта структура в файле winuser.h описана следующим образом:

```
typedef struct {      // version 0
   WORD mtOption;
   WORD mtID;
   WCHAR mtString[1];
} MENUITEMTEMPLATE, *PMENUITEMTEMPLATE;
```

Эта структура описывает один элемент меню. Первый элемент этой структуры – это логическая шкала, которая описывает элемент меню. Она занимает одно слово. Возможные значения битов этой логической шкалы я привожу в следующей таблице:

Описание	Значение	Назначение
MF_ENABLED	0x00000000L	Обычное состояние элемента меню – разрешённое
MF_STRING	0x00000000L	Элемент меню содержит только текстовую строку
MF_GRAYED	0x00000001L	При выборе элемента меню никаких действий не производится, текст элемента меню прорисован серым цветом
MF_DISABLED	0x00000002L	При выборе элементов меню никаких действий не производится
MF_BITMAP	0x00000004L	В качестве элемента меню используется картинка
MF_CHECKED	0x00000008L	Показывает, что у элемента меню есть «галочка», показывающая, что элемент «установлен»
MF_POPUP	0x00000010L	При выборе элемента возникает новое меню
MF_MENUBARBREAK	0x00000020L	Новый элемент меню помещается в новый столбец (для всплывающих меню)
MF_MENUBREAK	0x00000040L	Новый элемент меню помещается в новую строку или новый столбец
MF_OWNERDRAW	0x00000100L	Программа сама прорисовывает элемент меню
MF_SEPARATOR	0x00000800L	Элементы меню разделяются горизонтальной линией

После того, как была написана Windows'95, некоторые приведённые выше флаги были переименованы. В очередной таблице приводятся сведения о том, какие флаги были переименованы.

Старое название	Значение	Новое название
MF_STRING	0x00000000L	MFT_STRING
MF_BITMAP	0x00000004L	MFT_BITMAP
MF_MENUBARBREAK	0x00000020L	MFT_MENUBARBREAK
MF_MENUBREAK	0x00000040L	MFT_MENUBREAK
MF_OWNERDRAW	0x00000100L	MFT_OWNERDRAW
	0x00000200L	MFT_RADIOCHECK
MF_SEPARATOR	0x00000800L	MFT_SEPARATOR
	0x00002000L	MFT_RIGHTORDER
MF_RUGHTJUSTIFY	0x00004000L	MFT_RIGHTJUSTIFY
	0x00000003L	MFS_GRAYED
	0x00000003L	MFS_DISABLED
MF_CHECKED	0x00000008L	MFS_CHECKED
MF_HILITE	0x00000080L	MFS_HILITE
MF_ENABLED	0x00000000L	MFS_ENABLED
MF_UNCHECKED	0x00000000L	MFS_UNCHECKED
MF_UNHILITE	0x00000000L	MFS_UNHILITE
MF_DEFAULT	0x00001000L	MFS_DEFAULT

При декомпиляции флагов меню в исполняемых файлах я думаю, лучше применять переименованные флаги, чем их устаревшие значения.

Следующее поле тоже занимает одно слово и хранит идентификатор элемента меню, причём если элемент меню является подменю или выпадающим меню, то это поле отсутствует. В этом случае сразу за полем mtOption следует поле mtString, в котором хранится та строка, которая отображается в элементе меню. Тем самым мы закончили рассмотрение структуры меню, применяемой в Windows.

Ресурс типа «таблица строк»

Таблица базовых поправок в исполняемом файле.

Локальная память потока

Процессы и связанные с ними потоки

Для того, чтобы проанализировать, какая информация хранится в памяти, нам необходимо знать по меньшей мере две вещи. Первая - это адрес, по которому находится в памяти интересующая нас информация. Вторая - это формат этой информации. Этот раздел будет посвящен нахождению адреса, по которому находятся в памяти загруженные модули.

Для того, чтобы определить, какие модули система уже загрузила в память, а также где они находятся, нам необходимо будет обратиться к некоторым вспомогательным функциям, при помощи которых Microsoft «милостиво» позволяет получить некоторые частички данных о том, что происходит в системе.

В Windows'95 структуры данных, определяющие размещение в памяти информации о процессах, потоках и тому подобному, не документированы. Получить подобную информацию напрямую из системы, используя при этом только функции API, невозможно. Однако, Windows'95 позволяет сделать это только опосредствованно, при помощи совокупности функций, получивших общее название «toolhelp». Это название является данью недавнему прошлому. Дело в том, что в Windows 3.1 функции, осуществляющие доступ к внутренним недокументированным структурам данных, были объединены в библиотеку, которой было дано имя «toolhelp.dll». Сейчас эти функции размещаются в kernel32.dll, но название сохранилось.

Получение снимка (snapshot) системы

Я не зря сказал, что функции toolhelp'а не позволяют получить информацию из системы напрямую. Для этих целей используется снимок (snapshot) системы, который содержит информацию о состоянии системы в момент производства снимка. Мне это кажется вполне оправданным. Почему? Приведу пример. Представим себе, что поток, осуществляющий перебор списка потоков, на какое-то время передал управление другому потоку. Вполне вероятно, что за время работы другого потока состояние списка потоков может измениться. Наверное, здесь дело в том, что во время работы программы состояние системы может измениться. Вполне вероятно, что какие-то процессы завершат свою работу, а другие, наоборот, будут созданы. После того, как снимок получен, который можно получить при помощи функции CreateToolHelp32Snapshot, которая описана в файле tlhelp32.h так:

```
HANDLE WINAPI CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID);
```

Эта функция возвращает хэндл созданного снимка. Первым аргументом этой функции является набор битовых флагов, определяющих, какая информация будет записана в снимок, а также будет ли хэндл снимка наследоваться дочерними процессами. Эти флаги определены в файле tlhelp32.h и приведены в таблице ниже.

Флаг	Значение	Описание
TH32CS_SNAPHEAPLIST	0x00000001	В снимок включается список куч, принадлежащих указанному процессу
TH32CS_SNAPPROCESS	0x00000002	В снимок включается список процессов, имеющихся в системе
TH32CS_SNAPTHREAD	0x00000004	В снимок включается список потоков

TH32CS_SNAPMODULE	0x00000008	В снимок включается список модулей, принадлежащих указанному процессу
TH32CS_SNAPALL	TH32CS_SNAPHEAPLIST | TH32CS_SNAPPROCESS | TH32CS_SNAPTHREAD | TH32CS_SNAPMODULE	
TH32CS_INHERIT	0x80000000	Создаваемый список может наследоваться дочерними процессами

Что касается второго аргумента, то он определяет, о каком процессе нам необходима информация. Если этот аргумент равен нулю, то это означает, что нам необходима информация о текущем процессе. Этот аргумент используется только в тех случаях, когда нам необходимо получить список куч и модулей. В остальных случаях этот параметр игнорируется. Из этого факта есть одно крайне неприятное следствие. Представим себе, что мы хотим получить список загруженных на данный момент модулей. Для этого нам необходимо будет перебрать список процессов и для каждого процесса сделать новый снимок. Но будет ли полученная информация идентична той, которая была на момент создания первого списка? То же самое можно сказать и о тех случаях, когда нам необходимо получить список куч. Как выходить из этого положения, используя только документированные возможности API, Microsoft не объясняет. Я все же уверен в том, что помимо списка процессов и потоков, существует также и список модулей. Знай мы указатель на этот список, проблема была бы большей частью решена. То же можно сказать и о кучах. Но, к сожалению, мне не удалось обойтись без дисассемблирования и без отладочной версии Windows'95. Но об этом позже.

Итак, мы пришли к выводу о том, что при помощи документированных функций возможно получить только список процессов и потоков. Что ж, и то хлеб, как говорится.

Получение списка процессов

Для получения информации о процессах, протекающих в системе, возможно использование двух функций - Process32First() и Process32Next(). В файле tlhelp32.h эти функции описаны следующим образом:

```
BOOL WINAPI Process32First(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
BOOL WINAPI Process32Next(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
```

Аргументы этих функций одинаковы. Первый аргумент - хэндл созданного ранее снимка системы, второй аргумент - указатель на структуру типа PROCESSENTRY32, которая после возврата функции заполняется информацией о процессе. В том же файле tlhelp32.h эта структура описана следующим образом:

```
typedef struct tagPROCESSENTRY32
{
  DWORD  dwSize;
```

```
  DWORD   cntUsage;
  DWORD   th32ProcessID;          // this process
  DWORD   th32DefaultHeapID;
  DWORD   th32ModuleID;           // associated exe
  DWORD   cntThreads;
  DWORD   th32ParentProcessID;    // this process's parent process
  LONG    pcPriClassBase;         // Base priority of process's threads
  DWORD   dwFlags;
  char    szExeFile[MAX_PATH];    // Path
} PROCESSENTRY32;
typedef PROCESSENTRY32 *  PPROCESSENTRY32;
typedef PROCESSENTRY32 *  LPPROCESSENTRY32;
```

Первое поле этой структуры - dwSize - должно перед вызовом функции содержать размер этой структуры в байтах,то есть перед вызовом функции

```
DWORD dwSize = sizeof(PROCESSENTRY32);
```

Второе поле - cntUsage - содержит число ссылок на процесс, то есть число потоков, которые в настоящий момент используют какие-либо данные процесса. Как только число ссылок становится равным нулю, процесс прекращает свое существование.

Поле третье - th32ProcessID - является идентификатором процесса.

На очереди очередное, при этом очень странное поле th32DefaultHeapID. Дело в том, что Microsoft описывает его как идентификатор по умолчанию кучи процесса, однако тут же добавляет, что этот идентификатор имеет смысл только в контексте toolhelp-функций. Вполне очевидно, что с этим идентификатором производятся какие-то манипуляции для того, чтобы скрыть от посторонних глаз возможность найти кучу процесса.

Аналогично описано и поле th32ModuleID. Вроде бы оно является идентификатором процесса, но опять-таки в контексте toolhelp-функций.

Более или менее информативным представляется мне следующее поле, th32ParentProcessID. Оно определяет число потоков, принадлежащих процессу.

Поле th32ParentProcessID является идентификатором родительского по отношению к текущему процесса.

Поле pcPriClassBase содержит приоритет, который присваивается вновь создаваемым потокам.

Поле dwFlags зарезервировано и не используется.

И, наконец, последнее поле - szExeFile[MAX_PATH] - содержит полное имя файла, создавшего процесс. Здесь, наверное, добавлять нечего, за исключением того, что макрос MAX_PATH определен в файле windef.h и равен 260.

Техника работы с функциями Process32First и Process32Next напоминает работу со списком файлов в незабвенной MS DOS. Для того, чтобы получить информацию о первом процессе в снимке, необходимо вызвать функцию Process32First(). В случае успешного завершения функция возвращает TRUE. Для того, чтобы перебрать все оставшиеся процессы, нуж-

но вызывать функцию Process32Next() до тех пор, пока она не вернет FALSE.

Получение списка потоков

Итак, в предыдущем разделе мы узнали о том, как пройти по списку процессов, имеющихся в системе. Теперь нам необходимо узнать о том, как мы можем пройти по списку потоков.

Техника перебора потоков совершенно идентична технике перебора процессов. Для перебора потоков используются функции Thread32First() и Thread32Next(), которые описаны в файле tlhelp32.h следующим образом:

```
BOOL WINAPI Thread32First(HANDLE hSnapshot, LPTHREADENTRY32 lpte);
BOOL WINAPI Thread32Next(HANDLE hSnapshot, LPTHREADENTRY32 lpte);
```

Первый аргумент этих функций - уже известный нам хэндл снимка системы. Второй аргумент - указатель на структуру типа THREADENTRY32. Эта структура так описана в файле tlhelp32.h:

```
typedef struct tagTHREADENTRY32
{
  DWORD  dwSize;
  DWORD  cntUsage;
  DWORD  th32ThreadID;      // this thread
  DWORD  th32OwnerProcessID; // Process this thread is associated with
  LONG   tpBasePri;
  LONG   tpDeltaPri;
  DWORD  dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *  PTHREADENTRY32;
typedef THREADENTRY32 *  LPTHREADENTRY32;
```

Некоторые поля этой структуры напоминают аналогичные поля структуры типа PROCESSENTRY32. Точно так же, перед обращениям к функциям

```
DWORD dwSize = sizeof(THREADENTRY32);
```

Поле cntUsage содержит число ссылок на поток.

Поле th32ThreadID является идентификатором потока. Опять та же ссылка Microsoft на то, что это поле имеет смысл только в контексте toolhelp-функций.

Очередное поле - th32OwnerProcessID - является идентификатором родительского процесса.

Поле tpBasePri содержит текущий приоритет потока.

Следующее поле - tpDeltaPri - содержит разность между текущим уровнем приоритета потока и базовым, то есть тем, который присваивается при создании потока. Возможные значения этого параметра приведены в таблице ниже.

Уровень приоритета	Значение
THREAD_PRIORITY_LOWEST	THREAD_BASE_PRIORITY_MIN
THREAD_PRIORITY_BELOW_NORMAL	THREAD_PRIORITY_LOWEST+1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_ABOVE_NORMAL	THREAD_PRIORITY_HIGHEST-1
THREAD_PRIORITY_ERROR_RETURN	MAXLONG
THREAD_PRIORITY_TIME_CRITICAL	THREAD_BASE_PRIORITY_ LOWRT
THREAD_PRIORITY_IDLE	THREAD_BASE_PRIORITY_IDLE

Поле dwFlags, как и в предыдущем случае, не используется.

К настоящему моменту мы готовы к тому, чтобы начать писать программу. Полностью эта программа описана в приложении. А сейчас давайте попробуем сделать заготовку этой программы. Сейчас будет приведена та часть программы, которая формирует список процессов и потоков.

Для работы программы необходим файл ресурсов:

```
#include "helper.h"

ProcessesDialog DIALOG 0, 0, 240, 221
EXSTYLE WS_EX_DLGMODALFRAME
STYLE DS_MODALFRAME | DS_3DLOOK | DS_CENTER | WS_POPUP |
      WS_VISIBLE | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
CAPTION "Current Processes and Threads"
FONT 8, "MS Sans Serif"
{
 CONTROL "ProcessesTree", IDC_ProcessesTree, "SysTreeView32",
          TVS_HASLINES | TVS_LINESATROOT | TVS_HASBUTTONS |
          WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
          5, 5, 230, 204
 CONTROL "StatusWindow1", IDC_StatusBar, "msctls_statusbar32",
          3 | WS_CHILD | WS_VISIBLE, 0, 209, 480, 12
 CONTROL "ModulesTree", IDC_ModulesTree, "SysTreeView32",
          TVS_HASLINES | TVS_LINESATROOT | TVS_HASBUTTONS |
          WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
          0, 0, 0, 0
}
```

Ниже я привожу файл заголовков:

```
#define IDC_ProcessesTree 101
#define IDC_ModulesTree 102
#define IDC_StatusBar 103
#define IDM_FILL 1001

#define CX_ICON 16
#define CY_ICON 16

typedef HANDLE (WINAPI* PCREATESNAPSHOT)(DWORD, DWORD);
typedef HANDLE (WINAPI* PPROCESSWALK)
                (HANDLE, LPPROCESSENTRY32);
typedef HANDLE (WINAPI* PTHREADWALK)
                (HANDLE, LPTHREADENTRY32);
typedef HANDLE (WINAPI* PMODULEWALK)
                (HANDLE, LPMODULEENTRY32);
```

```
BOOL CALLBACK ProcessesTreeProc(HWND, UINT, WPARAM, LPARAM);
void Cls_OnClose(HWND);
BOOL Cls_OnInitDialog(HWND, HWND, LPARAM);
BOOL InitToolHelp();
BOOL DesignLists(HWND);
HTREEITEM InsertItemInTree(HWND, HTREEITEM, HTREEITEM, UINT,
                           LPSTR, int, int, LPARAM);

void Cls_OnCommand(HWND, int, HWND, UINT);
void FillTree(HTREEITEM, HWND);
void AddProcessInTree(LPPROCESSENTRY32, HTREEITEM, HWND);
```

И, наконец, основной файл программы:

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <stdio.h>
#include <tlhelp32.h>
#include "helper.h"

HWND hListsWnd[2], hStatusBar;
HIMAGELIST hiImageLists[2];
HTREEITEM hParent[2], hTestParent;
LPSTR pszStrings[2] = {"My computer", "Loaded modules"};
HANDLE hSnapShot;
HINSTANCE hInst;
PCREATESNAPSHOT pCreateSnapShot;
PPROCESSWALK pProcess32First, pProcess32Next;
PMODULEWALK pModule32First, pModule32Next;
PTHREADWALK pThread32First, pThread32Next;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR pszCmdLine, int nCmdShow)
 {
  hInst = hInstance;
  InitCommonControls();
  DialogBox(hInstance, "ProcessesDialog", NULL,
            (DLGPROC) ProcessesTreeProc);
  return 0;
 }

BOOL CALLBACK ProcessesTreeProc(HWND hDlg, UINT Message,
                                WPARAM wParam, LPARAM lParam)
 {
  switch(Message)
   {
    HANDLE_MSG(hDlg, WM_CLOSE, Cls_OnClose);
    HANDLE_MSG(hDlg, WM_INITDIALOG, Cls_OnInitDialog);
    HANDLE_MSG(hDlg, WM_COMMAND, Cls_OnCommand);
   }
  return FALSE;
 }
```

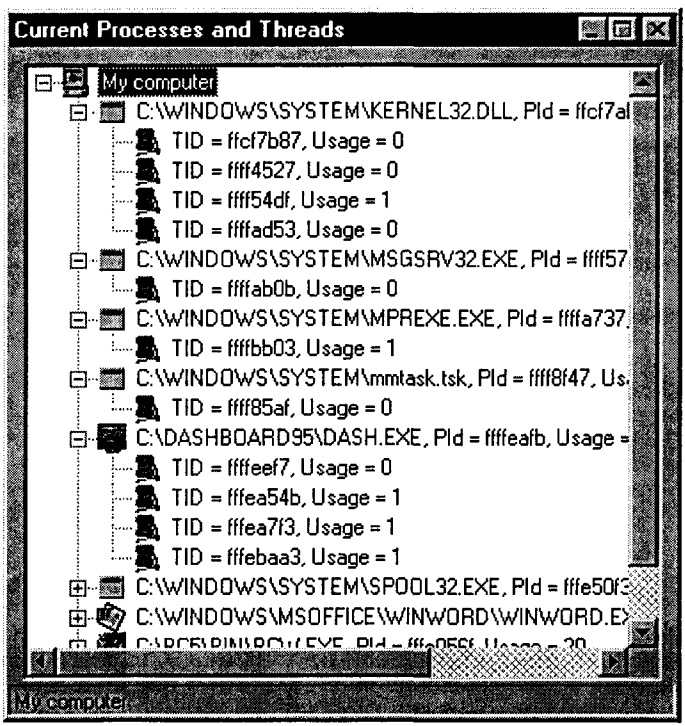
```
void Cls_OnClose(HWND hDlg)
 {
  EndDialog(hDlg, 0);
 }

BOOL Cls_OnInitDialog(HWND hDlg, HWND hWndFocus, LPARAM lParam)
 {
  BOOL bRet;
  int nEdges[2] = {360, -1};

  if(!InitToolHelp())
    bRet = FALSE;
   else
    if(!DesignLists(hDlg))
      bRet = FALSE;
     else
      bRet = TRUE;
  if(bRet)
    {
     hStatusBar = GetDlgItem(hDlg, IDC_StatusBar);
     SendMessage(hStatusBar, SB_SETPARTS, (WPARAM) 2,
                 (LPARAM) nEdges);
     for(int i = 0; i < 2; i++)
       SendMessage(hStatusBar, SB_SETTEXT, (WPARAM) i,
                   (LPARAM) pszStrings[i]);
     SendMessage(hDlg, WM_COMMAND, IDM_FILL, 0);
     TreeView_Expand(hListsWnd[0], hParent[0], TVE_EXPAND);
     SetFocus(hListsWnd[0]);
     TreeView_SelectItem(hListsWnd[0], hParent[0]);
    }
  return bRet;
 }

void Cls_OnCommand(HWND hDlg, int id, HWND hwndCtl, UINT codeNotify)
 {

  UINT wParam = MAKEWPARAM( (UINT) id, codeNotify);
  if(wParam == IDM_FILL)
    {
     FillTree(hParent[0], hDlg);
    }
 }

BOOL InitToolHelp()
 {

  HMODULE hKernel;

  if( (hKernel = GetModuleHandle("kernel32.dll")) )
    {
     pCreateSnapShot = (PCREATESNAPSHOT) GetProcAddress(hKernel,
                                                        "CreateToolhelp32Snapshot");
```

```
        pProcess32First = (PPROCESSWALK) GetProcAddress(hKernel,
                                                        "Process32First");
        pProcess32Next = (PPROCESSWALK) GetProcAddress(hKernel,
                                                        "Process32Next");
        pModule32First = (PMODULEWALK) GetProcAddress(hKernel,
                                                        "Module32First");
        pModule32Next = (PMODULEWALK) GetProcAddress(hKernel,
                                                        "Module32Next");
        pThread32First = (PTHREADWALK)
        GetProcAddress(hKernel, "Thread32First");
        pThread32Next = (PTHREADWALK) GetProcAddress(hKernel,
                                                        "Thread32Next");
        return TRUE;
      }
     else
      return FALSE;
 }

BOOL DesignLists(HWND hDlg)
 {
  BOOL bRet = TRUE;
  LPSTR pszIcons[3] = {"Mysystem.ico", "software.ico", "iathread.ico"};

  for(int i = 0; (i < 2) && (bRet == TRUE); i++)
   {
    hListsWnd[i] = GetDlgItem(hDlg, i + 101);
    if( (hilImageLists[i] = ImageList_Create(CX_ICON, CY_ICON,
                                             ILC_MASK, 2, 2)) )
      {
       ImageList_AddIcon(hilImageLists[i], LoadImage(hInst, pszIcons[i],
                         IMAGE_ICON, 0, 0, LR_LOADFROMFILE));
       if(i == 0)
          ImageList_AddIcon(hilImageLists[i], LoadImage(hInst, pszIcons[2],
                            IMAGE_ICON, 0, 0, LR_LOADFROMFILE));
       TreeView_SetImageList(hListsWnd[i], hilImageLists[i],
                             TVSIL_NORMAL);
       if( NULL == (hParent[i] = InsertItemInTree(hListsWnd[i], TVI_ROOT,
                                                  TVI_LAST, TVIF_TEXT |
                                                  TVIF_IMAGE |
                                                  TVIF_SELECTEDIMAGE,
                                                  pszStrings[i],
                                                  0, 0, 0)) )
          bRet = FALSE;
      }
     else
      bRet = FALSE;
   }
  return bRet;
 }

HTREEITEM InsertItemInTree(HWND hWnd, HTREEITEM hParent,
                           HTREEITEM hInsertAfter, UINT mask,
                           LPSTR pszText, int iImage, int iSelectedImage,
```

```
                                    LPARAM lParam)
{
  TV_INSERTSTRUCT InsertStruct;

  InsertStruct.hParent = hParent;
  InsertStruct.hInsertAfter = hInsertAfter;
  InsertStruct.item.mask = mask;
  InsertStruct.item.pszText = pszText;
  InsertStruct.item.iImage = iImage;
  InsertStruct.item.iSelectedImage = iSelectedImage;
  InsertStruct.item.lParam = lParam;
  return TreeView_InsertItem(hWnd, &InsertStruct);
}

void FillTree(HTREEITEM hParent, HWND hDlg)
{

  PROCESSENTRY32 ProcessEntry32;

  hSnapShot = pCreateSnapShot(TH32CS_SNAPALL, 0);
  ProcessEntry32.dwSize = sizeof(PROCESSENTRY32);
  pProcess32First(hSnapShot, &ProcessEntry32);
  AddProcessInTree(&ProcessEntry32, hParent, hDlg);
  ProcessEntry32.dwSize = sizeof(PROCESSENTRY32);
  while(pProcess32Next(hSnapShot, &ProcessEntry32))
    AddProcessInTree(&ProcessEntry32, hParent, hDlg);
  CloseHandle(hSnapShot);
}

void AddProcessInTree(LPPROCESSENTRY32 pProcessEntry32,
                      HTREEITEM hParent, HWND hDlg)
{

  char cString[80];
  THREADENTRY32 ThreadEntry32;

  sprintf(cString, "%s, PId = %x, Usage = %d",
                   pProcessEntry32->szExeFile, pProcessEntry32->th32ProcessID,
                   pProcessEntry32->cntUsage);
  HICON hIcon = ExtractIcon(hInst, pProcessEntry32->szExeFile, 0);
  if( (hIcon == 0) || (hIcon == (void*) 1) )
     hIcon = LoadIcon(NULL, IDI_APPLICATION);
  int i = ImageList_AddIcon(hilImageLists[0], hIcon);
  HANDLE hLocalParent = InsertItemInTree(hListsWnd[0], hParent, TVI_LAST,
                                         TVIF_TEXT | TVIF_PARAM |
                                         TVIF_IMAGE |
                                         TVIF_SELECTEDIMAGE,
                                         cString, i, i, (LPARAM) 0);
  i = 0;
  ThreadEntry32.dwSize = sizeof(THREADENTRY32);
  pThread32First(hSnapShot, &ThreadEntry32);
  if(ThreadEntry32.th32OwnerProcessID == pProcessEntry32->th32ProcessID)
     {
```

```
        sprintf(cString,"TID = %x, Usage = %x",
                ThreadEntry32.th32ThreadID, ThreadEntry32.cntUsage);
        InsertItemInTree(hListsWnd[0], (HTREEITEM) hLocalParent, TVI_LAST,
                         TVIF_TEXT | TVIF_PARAM | TVIF_IMAGE |
                         TVIF_SELECTEDIMAGE, cString, 1, 1, (LPARAM) 0);
        i++;
      }
    while(i < pProcessEntry32->cntThreads)
     {
      pThread32Next(hSnapShot, &ThreadEntry32);
      if(ThreadEntry32.th32OwnerProcessID == pProcessEntry32->th32ProcessID)
        {
         sprintf(cString,"TID = %x, Usage = %x",
                 ThreadEntry32.th32ThreadID, ThreadEntry32.cntUsage);
         InsertItemInTree(hListsWnd[0], (HTREEITEM) hLocalParent, TVI_LAST,
                          TVIF_TEXT | TVIF_PARAM | TVIF_IMAGE |
                          TVIF_SELECTEDIMAGE, cString, 1, 1, (LPARAM) 0);
         i++;
        }
     }
  }
```

Теперь пришло время увидеть, что же получается в результате работы этой программы:



Заключение

Что ж, уважаемый читатель, закрыта последняя страница моей второй книги. Повторит ли она успех первой книги или нет? Конечно, хотелось бы, чтобы её судьба была столь же удачна, как и судьба "Азбуки программирования для Win32 API". Но, как бы то ни было, я пытался рассказать о том, что знаю и умею, и надеюсь, что читатель найдёт в этой книге для себя что-то полезное. Завтра книга поедет в издательство. Как и в прошлый раз, я буду волноваться и укорять себя за то, что что-то написал не так, как хотелось бы. Я очень надеюсь, что Вы, уважаемый читатель, не будете судить меня строго. Я сделал всё, что мог. И опять, как и в прошлый раз, я не хочу прощаться с Вами, уважаемый читатель. Я говорю «До свидания, мой читатель!»

Оглавление