# Введение

Целью данной лабораторной работы по дисциплине "Системное программное обеспечение" является изучение форматов PE- и COFF-файлов, а также процессов компоновки и загрузки исполняемых модулей в системах Win32.

В ходе выполнения данной лабораторной работы студенту необходимо последовательно пройти этапы компиляции исходного текста задания, получения исполняемого модуля (компоновки) и загрузки полученной программы.

Данное справочное руководство разбито на три части: теоретическая часть, описание интерфейса лабораторной установки и описание порядка выполнения лабораторной работы. В теоретической части содержится подробное описание форматов объектных и исполняемых файлов системы Win32. Кроме собственно форматов также рассматриваются основные принципы работы программ под управлением системы Win32.

По всем вопросам возникающим при выполнении лабораторной работы, а также при обнаружений ошибок или неточностей просьба обращаться непосредственно к авторам программы. Кроме этого приветствуется конструктивная критика или пожелания.

С уважением, студенты группы ВМ-41 А.Ю. Плотников и Л.С. Ласкин.

2002

# Форматы PE- и COFF-файлов

Формат исполняемого файла операционной системы в значительной степени отражает встроенные в операционную систему предположения и режимы поведения.

Хотя освоение всех деталей формата исполняемого файла и не является одной из главных задач обучения программированию, тем не менее, из этого можно почерпнуть немало ценной информации об операционной системе. Динамическая компоновка, поведение загрузчика и управление памятью - это три примера специфических свойств операционной системы, которые можно понять по мере изучения формата исполняемых файлов.

Переносимый исполняемый (PE - Portable Executable) формат файла, фирма Microsoft разработала для использования во всех ее операционных системах Win32

(Windows NT, Windows 9x, Win32s). Следует отметить, что внутри самой Windows используются те же ключевые структуры данных, что и в файлах формата. Так, например, Windows отображает заголовок PE-файла в память и использует его для представления загружаемого

модуля. Для того чтобы понять, как работает ядро Windows, необходимо разобраться с PE-форматом.

Вместе с новыми форматами исполняемых файлов Microsoft также ввела новые форматы объектных модулей и библиотек, создаваемые ее собственными

компиляторами и ассемблерами. Новый файловый формат LIB, по существу, представляет собой просто связку объектных файлов, упорядоченных с помощью индекса. Эти новые объектные и LIB-файловые форматы имеют немало общих концепций с форматом PE. Общеизвестно, что Windows NT (первая из операционных систем Win32) унаследовала многое от VAX VMS и UNIX. Многие ведущие разработчики NT перед своим приходом в Microsoft программировали и работали именно над этими системами. Вполне естественно, что, когда им пришлось создавать NT, чтобы сохранить свое время и силы, они использовали ранее написанные и опробованные средства. Исполняемый формат и формат объектного модуля, который эти средства создавали и с которым они работали, называется COFF (Common Object File Format - стандартный формат обьектного файла).

Относительно устаревшую (по компьютерным меркам) сущность COFF можно усмотреть в том,

что некоторые поля файла имеют восьмеричный формат.

COFF-формат был сам по себе неплохой отправной точкой, но нуждался в расширении, чтобы удовлетворить потребностям новых операционных систем, таких как Windows NT или Windows 98. Результатом такого усовершенствования явился PE-формат (не забывайте: PE означает Portable Executable - переносимый исполняемый). Этот формат называется переносимым, так как все реализации Windows NT в различных системах (Intel 386, MIPS, Alpha, Power PC и т.д.) используют один и тот же исполняемый формат. Конечно, имеются различия, например, связанные с двоичной кодировкой команд процессора. Нельзя запустить на Intel исполняемый PE-файл, откомпилированный в MIPS. Тем не менее, существенно, что нет нужды полностью переписывать загрузчик операционной системы и программные средства для каждого нового процессора.

Microsoft стремилась усовершенствовать Windows NT, и это хорошо

иллюстрируется тем, что Microsoft отказалась от своих существующих 32-разрядных средств и файловых форматов. Драйверы виртуальных устройств, написанные для Windows 3.x, использовали другой 32-разрядный формат файла (LE-формат) задолго до появления NT на

свет. Следуя принципу "Если не поломано, не надо и чинить", заложенному в Windows, Windows 98 использует как PE", так и LE-формат. Это позволило Microsoft широко использовать существующие программы под Windows 3.x.

Вполне естественно ожидать совершенно другого исполняемого формата для совершенно новой операционной системы (какой является Windows NT), но другой вопрос - форматы объектных модулей (.OBJ и LIB). До появления 32-разрядной версии Visual C++ все компиляторы Microsoft пользовались спецификацией Intel OMF (Object Module Format - формат объектного модуля). Компиляторы Microsoft для реализации Win32 создают объектные файлы в формате COFF. Некоторые конкуренты Microsoft, например Borland, отказались от формата COFF объектных файлов и продолжали придерживаться формата OMF Intel. В результате компании, производящие объектные и LiB-файлы, рассчитанные на использование с несколькими компиляторами, будут вынуждены возвратиться к системе поставок различных версий и продуктов для различных компиляторов (если они не сделали этого до сих пор).

Те пользователи, которые любят усматривать во всех действиях Microsoft скрытность, могут увидеть в смене объектных форматов стремление Microsoft воспрепятствовать своим конкурентам. Чтобы гарантировать "совместимость"

Microsoft вплоть до уровня объектных файлов, другие фирмы будут вынуждены конвертировать все свои 32-разрядние средства в форматы COFF OBJ и LIB. Подводя итог, можно сказать, что объектные и LIB-файловые форматы являются еще одним примером отказа Microsoft от существующих стандартов при выборе приоритетов развития этой фирмы.

Вместе с некоторыми определениями структур для объектных файлов формат COFF

PE-формат задокументирован (в самом размытом смысле этого слова) в файле заголовка WINNT.H. Примерно посредине WINNT.H находится секция, озаглавленная "Image Format". Эта секция начинается с небольших фрагментов из старых добрых заголовков форматов DOS MZ и NE перед переходом к новой информации, связанной с PE. WINNT.H дает определения структур исходных данных, используемых PE-файлами, однако содержит всего лишь прозрачный намек на полезные комментарии, объясняющие назначение структур и флагов. Автор заголовочного файла для PE-формата (некий Michael J. O'Leary) определенно питает склонность к длинным, описательным именам, а также к глубоко вложенным структурам и макросам.

Программируя с использованием WINNT.H, нередко можно встретить, например, такое выражение:

pNTHeader.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DEBUG].VirtualAddress;

# Заголовок PE-файла

Как и всякий другой исполняемый формат файла Microsoft, PE-файл имеет набор полей, расположенных в легко доступном (по крайней мере, легко находимом) месте файла; эти поля определяют, как будет выглядеть остальная часть файла. Заголовок PE-файла содержит такую важную информацию, как расположение и размер областей кода программ и данных, указание на то, с какой операционной системой предполагается использовать данный файл, а также начальный размер стека.

Как и в других исполняемых форматах от Microsoft, заголовок не находится в самом начале файла. Вместо этого несколько сотен первых байтов типичного PE-файла заняты под *заглушку DOS.* Эта заглушка представляет собой минимальную DOS-программу, которая выводит что-либо вроде: "Эта программа не может быть запущена под DOS". Все это предусматривает случай, когда пользователь запускает программу Win32 в среде, которая не поддерживает Win32, получая при этом приведенное выше сообщение об ошибке. После того как загрузчик Win32 отобразил в память PE-файл, первый байт отображения файла соответствует первому байту заглушки DOS. И это не так уж плохо. С каждой запускаемой Win32 программой пользователь получает дополнительную DOS-программу, загруженную просто так! (В Win 16 заглушка DOS не загружается в память.)

Как и в других исполняемых форматах Microsoft, настоящий заголовок можно обнаружить, найдя его стартовое смещение, которое хранится в заголовке DOS. Файл WINNT.H содержит определение структуры

для заголовка заглушки DOS, что делает очень простым нахождение начала заголовка PE-файла. Поле **e_lfanew** собственно и содержит относительное смещение (RVA) настоящего заголовка PE-файла. Чтобы установить указатель в памяти на заголовок PE-файла, достаточно просто сложить значение в поле с базовым адресом отображения:

```
//Пренебрегаем для ясности преобразованием типов и указателей...

pNTHeader = dosHeader + dosHeader->e_lfranew;
```

Основной заголовок PE-файла представляет структуру типа IMAGE_NT_HEADERS, определенную в файле WINNT.H. Структура IMAGE_NT_HEADERS в памяти - это то, что Windows использует в качестве своей базы данных модуля в памяти. Каждый загруженный EXE-файл или DLL представлены в Windows структурой IMAGE_NT_HEADERS. Эта структура состоит из двойного слова и двух подструктур, как показано ниже:

```
DWORD Signature;

IMAGE_FILE_HEADER FileHeader;

IMAGE_OPTIONAL_HEADER OptionaI Header;
```

Поле Signature (сигнатура - подпись), представленное как ASCII код, - это PE00 (два нулевых байта после PE). Если поле **e_lfanew** в заголовке DOS указало вместо обозначения PE обозначение NE в этом месте, значит, вы работаете с файлом Win16 NE. Аналогично, если указано обозначение LE в поле Signature, то это файл VxD (VirtualDeviceDriver- драйвер виртуального устройства).

Обозначение LX указывает на файл старой соперницы Windows - OS/2.

За двойным словом - сигнатурой PE, в заголовке PE-файла следует структура типа IMAGE_FILE_HEADER. Поля этой структуры содержат только самую общую информацию о файле. Структура не изменилась по сравнению с исходными COFF - реализациями. Эта структура является частью заголовка PE-файла, кроме того, появляется в самом начале объектных COFF - файлов, создаваемых компиляторами Microsoft Win32. Далее приводятся поля IMAGE_FILE_HEADER.

*WORD Machine*

Это центральный процессор, для которого предназначен файл. Определены следующие идентификаторы процессоров:

| Процессор | Значение |
|---|---|
| Intel I386 | 0x14C |
| Intel I860 | 0x14D |
| MIPSR3000 | 0x162 |
| MIPS R4000 | 0x166 |
| DEC Alpha AXP | 0x184 |
| Power PC | 0x1FO (little endian) |
| Motorola 68000 | 0x268 |
| PA RISC | 0x290 (Precision Architecture) |

*WORD NumberOfSections*

Количество секций в EXE или OBJ - файле.

*DWORD TimeDateStamp*

Время, когда файл был создан компоновщиком (или компилятором, если это OBJ - файл). В этом поле указано количество секунд, истекших с 16:00 31 декабря 1969 года.

*DWORD PointerToSymbolTable*

Файловое смещение COFF-таблицы символов. Это поле используется только в OBJ и PE-файлах с информацией COFF-отладчика. PE-файлы поддерживают разнообразные отладочные форматы, так что отладчики должны ссылаться на вход IMAGE_DIRECTORY_ENTRY_DEBUG в каталоге данных.

*DWORD NumberOfSymbols*

Количество символов в COFF-таблице символов.

*WORD SizeOfOptionalHeader*

Размер необязательного заголовка, который может следовать за этой структурой. В исполняемых файлах - это размер структуры IMAGE_OPTIONAL_HEADER, которая следует за этой структурой. В объектных файлах, по утверждению Microsoft, это поле всегда содержит нуль. Однако при просмотре библиотеки вывода KERNEL32.LIB можно обнаружить объектный файл с ненулевым значением в этом поле, так что относитесь к высказыванию Microsoft с некоторым скептицизмом.

*WORD Characteristics*

Флаги, содержащие информацию о файле. Здесь описываются некоторые важные поля (другие поля определены в WINNT.H).

0x0001 Файл не содержит перемещений.

0x0002 Файл представляет исполняемое отображение (т.е. это не OBJ- или LIB-файл).

0x2000 Файл является библиотекой динамической компоновки (DLL), а не программой.

Третьим компонентом заголовка PE-файла является структура типа IMAGE_OPTIONAL_HEADER. Для PE-файлов эта часть является обязательной. Формат COFF разрешает индивидуальные реализации для определения структуры дополнительной информации, помимо стандартного IMAGE_FILE_HEADER. В полях в IMAGE_OPTIONAL_HEADER разработчики PE-формата поместили то, что они посчитали важным дополнением к общей информации в IMAGE_FILE_HEADER,

Для пользователя не является критическим знание всех полей в IMAGE_OPTIONAL_HEADER. Наиболее важными полями являются поля ImageBase и Subsystem.

*WORD Magic*

Слово-сигнатура, определяющее состояние отображенного файла. Определены следующие значения;

0x0107 Отображение ПЗУ

## 0x010B Нормальное исполняемое отображение (Значение для большей части файлов)

*BYTE MajorLinkerVersion*

*BYTE MinorLinkerVersion*

Версия компоновщика, который создал данный файл. Числа должны быть представлены в десятичном виде, а не в шестнадцатеричном. Типичная версия компоновщика 2.23.

*DWORD SizeOfCode*

Суммарный размер программных секций, округленный к верхней границе. Обычно большинство файлов имеют только одну программную секцию, так что это поле обычно соответствует размеру секции .text.

*DWORD SizeOfInitalizedData*

Предполагается, что это общий размер всех секций, состоящих из инициализированных данных (не включая сегменты программного кода.) Однако не похоже, чтобы это совпадало с размером секций инициализированных данных в файле.

*DWORD SizeOfUnInitalizedData*

Размер секций, под которые загрузчик выделяет место в виртуальном адресном пространстве, но которые не занимают никакого места в дисковом файле. В начале работы программы эти секции не обязаны иметь каких-либо определенных значений - отсюда название *неинициализированные данные (Uninitialized Data).* Неинициализированные данные обычно находятся в секции под названием .bss.

*DWORD AddressOfEntryPoint*

Адрес, с которого отображение начинает выполнение. Это RVA, который можно найти в секции .text. Это поле применимо как для EXE-файла, так и для DLL.

*DWORD BaseOfCode*

RVA, с которого начинаются программные секции файла. Программные секции кода обычно идут в памяти перед секциями данных и после заголовка PE-файла. Этот RVA обычно равен 0x1000 для EXE-файлов, созданных компоновщиками Microsoft. Для TLINK32 (Borland) значение этого поля равно 0x10000, так как по умолчанию этот компоновщик выравнивает объекты на границу в 64 Кбайт в отличие от 4 Кбайт в случае компоновщика Microsoft.

*DWORD BaseOfData*

RVA, с которого начинаются секции данных файла. Секции данных обычно идут последними в памяти, после заголовка PE-файла и программных секций.

*DWORD ImageBase*

Когда компоновщик создаст исполняемый файл, он предполагает, что файл будет отображен в определенное место в памяти. Вот именно этот адрес и хранится в этом поле. Знание адреса загрузки позволяет компоновщику провести оптимизацию. Если загрузчик действительно отобразил файл в память по этому адресу, то программа перед запуском не нуждается ни в какой настройке. В исполняемых файлах NT 3.1 адрес отображения по умолчанию равен 0x10000. В случае DLL этот адрес по умолчанию равен 0x400000. В Windows 9x адрес 0x10000 нельзя использовать для

загрузки 32-разрядных файлов EXE, так как он лежит в пределах линейной области адресного пространства, общего для всех процессов. Поэтому для Windows NT 3.5 Microsoft изменила для исполняемых файлов Win32 базовый адрес по умолчанию, сделав его равным 0x400000. Более старые программы, которые были скомпонованы в предположении, что базовый адрес равен 0x10000, загружаются Windows 9x дольше, потому что загрузчик должен применить базовые поправки.

*DWORD SectionAlignment*

После отображения в память каждая секция будет обязательно начинаться с виртуального адреса, кратного данной величине. С учетом подкачки страниц минимальная величина этого поля 0x1000 используется компоновщиком Microsoft по умолчанию. TLINK в Borland C++ использует по умолчанию 0x10000 (64 Кбайт).

*DWORD FileAlignment*

В случае PE-файла исходные данные, которые входят в состав каждой секции, будут обязательно начинаться с адреса, кратного данной величине. Значение, устанавливаемое по умолчанию, равно 0x200 байт и, вероятно, выбрано так для того, чтобы начало секции всегда совпадало с началом дискового сектора (0x200 байт- это как раз размер дискового сектора). Это поле эквивалентно размеру выравнивания сегмента/ресурса в NE-файлах. В отличие от NE-файлов, PE-файлы не состоят из сотен секций, так что память, теряемая при выравнивании секций файла, обычно очень незначительна.

*WORD MajorOperatingSystemVersion*

*WORD MinorOperatingSystemVersion*

Самая старая версия операционной системы, которая может использовать данный исполняемый файл. Назначение этого поля не совсем ясно, так как поля подсистемы (приведены ниже), похоже, имеют такое же предназначение. В большей части файлов Win32 в этом поле содержится значение, соответствующее версии 1.0 .

*WORD MajorImageVersion*

*WORD MinorImageVersion*

Определяемое пользователем поле. Это поле позволяет иметь различные версии EXE-файлов и DLL. Эти поля устанавливаются с помощью ключа компоновщика /VERSION, например: LINK/VERSION: 2,0 myobj.obj

*WORD MajorSubsystemVersion*

*WORD MinorSubsystemVersion*

Это поле содержит самую старую версию подсистемы, позволяющую запускать данный исполняемый файл. Типичное значение в этом поле 4.0 (обозначает Windows 4.0, что равносильно Windows 95).

*DWORD Reserved 1*

Это поле, по-видимому, всегда равно нулю.

*DWORD SizeOfImage*

Представляет общий размер всех частей отображения, находящихся под контролем загрузчика. Эта величина равна размеру области памяти, начиная с базового

адреса отображения и заканчивая адресом конца последней секции. Адрес конца секции выровнен на ближайшую верхнюю границу секции.

*DWORD SizeOfHeaders*

Размер заголовка PE-файла и таблицы секции (объекта). Исходные данные для секций начинаются сразу после всех составляющих частей заголовка.

*DWORD Checksum*

Предположительно отвечает контрольной сумме контроля циклическим избыточным кодом (CRC-контроль) для данного файла. Как и для других исполняемых форматов Microsoft, это поле обычно игнорируется и устанавливается в нуль. Однако для всех DLL драйверов, DLL, загруженных во время загрузки ОС, и серверных DLL эта контрольная сумма должна иметь правильное значение. Алгоритм для контрольной суммы можно найти в IMAGEIILP.DLL. Исходники IMAGEHLP.DLL поставляются в WIN32 SDK.

*WORD Subsystem*

Тип подсистемы, которую данный исполняемый файл использует для своего пользовательского интерфейса. WINNT.H определяет следующие значения:

NATIVE Подсистема не требуется (например, для драйвера устройства)

WINDOWS _GUI Запускается в подсистеме Windows GUI

WINDOWS_CUI Запускается в подсистеме Windows character

OS2_CUI Запускается в подсистеме OS/2 (только приложения OS/2 1.x)

POSIX_CUI Запускается в подсистеме Posix

*WORD DllCharacteristics (обозначен как вышедший из употребления в NT 3.5)*

Набор флагов, показывающий, при каких обстоятельствах будет вызываться функция инициализации DLL (например, DllMain()). Эта величина, по-видимому, всегда должна устанавливаться в нуль, однако операционная система вызывает функцию инициализации DLL для всех четырех случаев.

Определены следующие значения:

1 - Вызов, когда DLL впервые загружена в адресное пространство процесса

2 - Вызов, когда цепочка заканчивает работу

4 - Вызов, когда цепочка начинает работу

8 - Вызов при выходе из DLL

*DWORD SizeOfStackReserve*

Объем виртуальной памяти, резервируемой под начальный стек цепочки. Однако не вся эта память выделяется (см. следующее поле). По умолчанию это поле устанавливается в 0x100000 (1 Мбайт). Если пользователь указывает 0 в качестве размера стека в CreateThread(), получившаяся цепочка будет иметь стек того же размера.

*DWORD SizeOfStackCommit*

Количество памяти, изначально выделяемой под исходный стек цепочки. Это поле по умолчанию равно 0x1000 байт (1 страница) для компоновщиков Microsoft, тогда как TLINK32 делает его равным 0x2000 (2 страницы).

*DWORD SizeOfHeapReserve*

Объем виртуальной памяти, резервируемой под изначальную кучу программы. Этот дескриптор кучи можно получить, вызвав GetProcessHeap(). Однако не вся эта память выделяется (см. следующее поле).

*DWORD SizeOfHeapCommit*

Объем виртуальной памяти, изначально выделяемой под кучу процесса. По умолчанию компоновщик делает это поле равным 0x1000 байт.

*DWORD LoaderFlags (обозначен как вышедший из употребления в NT 3.5)*

Как следует из WINNT.H, эти поля, по-видимому, связаны с поддержкой отладчика. Определены следующие значения:

1 - Запускать ли команду прерывания перед запуском процесса?

2 - Запускать ли отладчик программы после процесса?

*DWORD NumberOfRvaAndSizes*

Количество входов в массиве DataDirectory (см. описание следующего поля). Современные программные средства всегда делают это значение равным 16.

IMAGE_DATA_DIRECTORY DataDirectory [IMAGE_NUMBEROF_DIRECTORY_ENTRIES]

Массив структур типа IMAGE_DATA_DIRECTORY. Начальные элементы массива содержат стартовый RVA и размеры важных частей исполняемого файла. В настоящее время некоторые элементы в конце массива не используются. Первый элемент массива - это всегда адрес и размер экспортированной таблицы функций (если она присутствует). Второй элемент массива- адрес и размер импортированной таблицы функций и т.д. Для того чтобы увидеть полный перечень определений элементов массива, см. IMAGE_DIRECTORY_ENTRY_xxx директивы #define в WINNT.H.

Этот массив предназначен для того, чтобы загрузчик мог быстро найти определенную секцию отображения (например, импортированную таблицу функций) без последовательного перебора всех секций отображения, чтобы каждый раз не сравнивать имена. Большая часть элементов массива описывает все данные в секции. Тем не менее элемент IMAGE_DIRECTORY_ENTRY_DEBUG охватывает только небольшую часть байтов в секции .rdata.

# Основные сведения о форматах Win32 и PE

Перед изучением формата PE следует рассмотреть несколько новых идей, позволивших создать такой формат. Одним из основных понятий PE формата и Win32 является понятие модуля. Под словом "модуль" понимается текст программ, данные и ресурсы исполняемого файла или DLL, которые были загружены в память. Помимо текста программы и данных, которые использует непосредственно программа, модуль также включает вспомогательные данные, используемые Windows для того, чтобы определить, где расположены в памяти текст программы и данные. В Win16 вспомогательные структуры данных находятся в базе данных модуля (сегмент, на который ссылается HMODULE). В Win32 эта информация содержится в заголовке PE-файла (структура IMAGE_NT_HEADERS).

Самое важное из того, что следует знать о PE-файлах - это то, что исполняемый файл на диске и модуль, получаемый после загрузки, очень похожи. Причиной этого является то, что загрузчик Windows должен создать из дискового файла исполняемый процесс без больших усилий. Точнее говоря, загрузчик попросту использует отображенные в память файлы Win32, чтобы загрузить соответствующие части PE-файла в адресное пространство программы. Здесь уместна аналогия со строительством сборных домиков. У вас есть относительно немного элементов, расставляя их по своим местам и скрепляя стандартными соединениями, вы достаточно быстро собираете целый дом, - вся работа состоит из простого защелкивания таких стандартных соединений. И такой же простой задачей, как подключение электричества и водопровода к

мирному домику, является соединение PE-файла с внешним миром (т.е. подключение к нему DLL и т.д.).

Так же просто загружается и DLL. После того как EXE или .DLL модуль загружены. Windows обращается с ними так, как и с другими отображенными в память файлами. Совершенно иная ситуация в 16-разрядной Windows. 16-разрядный NE-загрузчик файла считывает файл в память порциями и создаст отдельные структуры данных для представления модуля в памяти. Когда необходимо загрузить сегмент программы или данных, загрузчик должен выделить новый сегмент из общей кучи, обнаружить, где хранятся исходные данные в исполняемом файле, отыскать это место, считать исходные данные и применить любой подходящий крепеж. Кроме того, каждый 16-разрядный модуль обязан запоминать все используемые в данный момент селекторы, независимо от того, выгружен ли сегмент.

В Win32, напротив, память, используемая под программы, данные, ресурсы, таблицы ввода, таблицы вывода и другие элементы, представляет собой один сплошной линейный массив адресного пространства. Все, что достаточно знать в этом случае, - это адрес, в который загрузчик отобразил в памяти исполняемый файл. Тогда для того чтобы найти любой элемент модуля, достаточно следовать указателям, которые хранятся как часть отображения.

Другим важным понятием является RVA (Relative Virtual Address - относительный виртуальный адрес). Многие поля в PE-файлах задаются именно с помощью их RVA. RVA - это просто смещение данного элемента по отношению к адресу, с которого начинается отображение файла в памяти. Пусть, к примеру, загрузчик Windows отобразил PE-файл в память,

начиная с адреса 0x400000 в виртуальном адресном пространстве. Если некая таблица в отображении начинается с адреса 0x401464, то RVA данной таблицы 0x1464:

(виртуальный адрес 0x401464) - (базовый адрес 0x400000) = RVA 0x1464

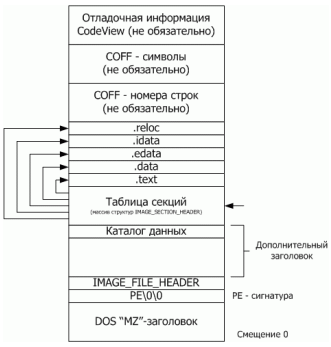
Чтобы перевести RVA в указатель памяти, необходимо просто прибавить RVA к базовому адресу, начиная с которого был загружен модуль. Термин *базовый адрес* представляет еще одно важное понятие. Базовый адрес - это адрес, с которого начинается отображенный в память EXE-файл или DLL. Для удобства Windows NT и Windows 9x используют базовый адрес модуля в качестве дескриптора образца модуля (HINSTANCE - instance handle). То, что в Win32 базовый адрес называется HINSTANCE, может вызвать недоразумения, так как термин дескриптор образца происходит из 16-разрядной Windows. Каждая копия приложения в Win16 получает свой собственный сегмент данных (и связанный с ним глобальный дескриптор), который отличает эту копию приложения от других; отсюда и название дескриптор образца.

В Win32 нет нужды различать отдельные копии приложений, так как у них нет общего адресного пространства. Однако, термин HINSTANCE сохранен, чтобы отразить преемственность Win32 по отношению к Win16. В случае Win32 существенно то, что можно вызвать GetModuleHandle для любой DLL, которая используется в процессе, и получить указатель, который можно использовать для доступа к компонентам модуля. Под компонентами модуля подразумеваются импортируемые и экспортируемые им

функции, его перемещения, его программные секции и секции данных и т.п.

Еще одним понятием, с которым следует ознакомиться для того, чтобы исследовать PE- и COFF OBJ-файлы, является секция. Секция в файлах PE или COFF примерно эквивалентна сегменту или ресурсам в 16-разрядном NE-файле. Секции содержат либо код программ, либо данные. Некоторые секции содержат код и данные, непосредственно объявляемые и используемые программами, тогда как другие секции данных создаются компоновщиками и библиотекарями специально для пользователя и содержат информацию, необходимую для работы операционной системы. В некоторых описаниях формата PE фирмы Microsoft секции также называются объектами. Однако этот последний термин имеет так много (возможно, противоречащих друг другу) значений, что лучше придерживаться термина "секция" для обозначения областей программного кода и данных.

Общий формат PE-файла описывается следующим рисунком:




Общий формат PE-файла

# Таблица секций

Между заголовком PE-файла и исходными данными для секций отображения находится таблица секций. Эта таблица содержит информацию о каждой секции отображения. Секции в отображении упорядочены по их стартовому адресу, а не в алфавитном порядке.

Для изучения таблицы секций необходимо четко разъяснить, что же такое секция. В NE-файле программный код и данные хранятся в различных сегментах в файле. Часть заголовка NE-файла представляет собой массив структур - по одной для каждого сегмента, используемого программой. Каждая структура массива содержит информацию об одном сегменте. Хранимая информация включает тип сегмента (программа или данные), его размер и его расположение, где бы он ни находился в файле. В PE-файле таблица секций аналогична таблице сегментов в NE-файле.

Однако, в отличие от таблицы сегментов NE-файла, таблица секций PE-файла не хранит значение селектора для каждого куска программного кода или данных. Вместо этого каждый элемент таблицы секций хранит адрес, по которому исходные данные файла были отображены в память. Несмотря на то, что секции аналогичны 32-разрядным сегментам, они на самом деле не являются индивидуальными сегментами. Вместо этого секция просто отвечает диапазону памяти виртуального адресного пространства процесса.

Другим отличием PE-файлов от NE-файлов является то, как они управляют вспомогательными данными, которые используются не программой, а операционной системой. В качестве двух примеров можно привести перечень DLL, используемых исполняемыми файлами, и местонахождение таблицы привязки (fixup). В NE-файлах ресурсы не считаются сегментами. И хотя они имеют приписанные им

селекторы, информация о ресурсах не хранится в таблице сегментов заголовка NE-файла. Вместо этого ресурсы сведены в отдельную таблицу в конце заголовка NE-фаЙла. Информации об импортированных и экспортированных функциях тоже не гарантируется выделение своего собственного сегмента, она накапливается в пределах заголовка NE-файла.

Другая ситуация в случае PE-файла. Все, что считается важным программным кодом или данными, хранится в полнокровной секции. Таким образом, информация об импортированных функциях хранится в своей собственной секции так же, как и таблица экспортируемых модулем функций. То же самое справедливо и для данных настройки. Любая программа или данные, которые могут понадобиться программе или операционной системе, получают свою собственную секцию.

Сразу после заголовка PE-файла в памяти следует массив из IMAGE_SECTION_HEADER. Количество элементов этого массива задается в заголовке PE-файла (поле IMAGE_NT_HEADER.FileHeader.NumberOfSections).

Каждый IMAGE_SECTION_HEADER представляет собой полную базу данных об одной секции файла EXE или OBJ имеет следующий формат.

*BYTE Name [IMAGE_SIZEOF_SHORT_NAME]*

Это 8-байтовое имя в стандарте ANSI (не Unicode), которое именует секцию. Большинство имен секций начинается с точки (например, .text), но это не обязательно, вопреки тому, в чем пытаются уверить отдельные документы по PE-файлам. Пользователь может давать имена своим собственным секциям с помощью либо сегментной директивы в ассемблере, либо с помощью директив #pragma data_seg и #pragma code_seg компилятора Microsoft C/C++. Пользователи Borland C++ должны использовать #pragma codeseg. Необходимо отметить, что

если имя секции занимает 8 полных байтов, отсутствует завершающий байт NULL.

---

Union {

DWORD PhysicalAdiIress

DWORD VirtualSIze

} Misc;

---

Это поле имеет различные назначения в зависимости от того, встречается ли оно в EXE- или OBJ-файле. В EXE-файле оно содержит виртуальный размер секции программного кода или данных. Это размер до округления на ближайшую верхнюю границу файла. Поле SizeOfRawData дальше в этой структуре содержит это округленное значение. Интересно, что Borland TLINK32 меняет местами значение этого поля и поля SizeOfRawData и, тем не менее, остается правильным компоновщиком. В случае OBJ-файлов это поле указывает физический адрес секции. Первая секция начинается с адреса 0. Чтобы получить физический адрес следующей секции, надо прибавить значение в SizeOfRawData к физическому адресу данной секции.

*DWORD VirtualAddress*

В случае EXE-файлов это поле содержит RVA, куда загрузчик должен отобразить секцию. Чтобы вычислить реальный начальный адрес данной секции в памяти, необходимо к виртуальному адресу секции, содержащемуся в этом поле, прибавить базовый адрес отображения, Средства Microsoft устанавливают по умолчанию RVA первой секции равным 0x1000. Для объектных файлов это поле не несет никакого смысла и устанавливается в 0.

*DWORD SizeOfRawData*

В EXE-файлах это поле содержит размер секции, выровненный на ближайшую верхнюю границу размера файла. Например, допустим, что размер выравнивания файла 0x200. Если поле VirtualSize указывает, что длина секции Ox35A байт то в данном поле будет указано, что размер секции 0x400 байт. Для OBJ-файлов это поле содержит точный размер секции, сгенерированной компилятором или ассемблером. Другими словами, для OBJ-файлов оно эквивалентно полю VirtualSize в EXE-файлах.

*DWORD PointerToRawData*

Это файловое смещение участка, где находятся исходные данные для секции. Если пользователь сам отображает в память PE- или COFF-файл (вместо того, чтобы доверить загрузку операционной системе), это поле важнее, чем поле VirtualAddress. Причиной является то, что в этом случае получится абсолютно линейное отображение всего файла, так что данные для секций будут находиться по этому смещению, а не по RVA, указанному в поле VirtualAddress.

*DWORD PointerToRelocations*

В объектных файлах это файловое смещение информации о поправках для данной секции. Информация о поправках в любой секции объектного файла следует за исходными данными для этой секции, В EXE-файлах это (и следующее) поле не несет смысловой нагрузки и устанавливается в нуль. Когда компоновщик создаст EXE-файл, он разрешает большинство привязок, а во время загрузки остается разрешить только базовые адресные поправки и импортированные функции. Информация о базовых поправках и импортированных функциях хранится в секциях базовых поправок и импортированных функций, так что нет необходимости в EXE-файле помещать данные поправок для каждой секции после исходных данных секции.

*DWORD PointerToLinenumhers*

Файловое смещение таблицы номеров строк. Таблица номеров строк ставит в соответствие номера строк исходного файла адресам, по которым можно найти код, сгенерированный для данной строки. В современных отладочных форматах, таких как формат CodeView, информация о номерах строк хранится как часть информации отладчика. В отладочном формате COFF, однако, информация о номерах строк концептуально отлична от информации о символьных именах/типах. Обычно только секции с программным кодом (например, .text или CODE) имеют номера строк. В EXE-файлах номера строк собраны в конце файла после исходных данных для секций. В объектных файлах таблица номеров строк для секции следует за исходными данными секции и таблицей перемещений для этой секции.

### *WORD NumberOfRelocations*

Количество перемещений в таблице поправок для данной секции (поле PointerToRelocations приведено выше). Это поле используется, по-видимому, только в объектных файлах.

### *WORD NumberOfLinenumbers*

Количество номеров строк в таблице номеров строк для данной секции (поле PointerToLinenumbers приведено выше).

### *DWORD Characteristics*

То, что большая часть программистов называет *флагами* (flags), формат COFF/PE называет *характеристиками* (characteristics). Это поле представляет собой набор флагов, которые указывают на атрибуты секции (программа/данные, предназначен для чтения, предназначен для записи и т.п.). Полный перечень всех возможных атрибутов секции находится в файле заголовка WINNT.H. Некоторые из самых важных флагов приведены ниже.

Флаги COFF-секций

| Флаг | Использование |
|---|---|
| 0x00000020 | Эта секция содержит программный код. Как правило, устанавливается вместе с флагом (0x80000000) |
| 0x00000040 | Данная секция содержит инициализированные данные. Почти для всех секций, кроме исполняемых и .bss секций, этот флаг установлен |
| 0x00000080 | Данная секция содержит неинициализированные данные (например, .bss секции) |
| 0x00000200 | Данная секция содержит комментарии или какой-нибудь другой вид информации. Типичное использование такой секции - это секция .drectve, создаваемая компилятором и содержащая команды для компоновщика |
| 0x00000800 | Содержимое данной секции не должно быть помещено в конечный EXE-файл. Такая секция используется компилятором/ассемблером для передачи информации компоновщику |
| 0x02000000 | Данную секцию можно отбросить, так как она не используется программой, после того как последняя загружена. Чаще всего встречается отбрасываемая секция - это секция базовых поправок (.reloc) |
| 0x10000000 | Данная секция является совместно используемой. При использовании с DLL данные в такой секции используются совместно всеми процессами, использующими эту DLL. По умолчанию секции данных не являются совместно используемыми, т.е. каждый процесс, использующий DLL, имеет свою собственную отдельную копию такой секции данных. Говоря более техническим языком, совместно используемая секция дает указание менеджеру памяти устанавливать отображение страниц для этой секции так, что все процессы, |

| | |
|---|---|
| | используующие DLL, ссылаются на одну и ту же физическую страницу в памяти. Чтобы сделать секцию совместно используемой, следует установить атрибут SHARED во время компоновки. Например: LINK/SECT!ON:MYDATA,RWS указывает компоновщику, что секция с названием MYDATA должна быть доступной для чтения, записи и совместно используемой. По умолчанию сегменты данных DLL Borland C++ имеют атрибуты совместного использования |
| 0x20000000 | Данная секция является исполняемой. Этот флаг обычно устанавливается каждый раз, когда устанавливается флаг "Программа" (Contains Code) (0x00000020) |
| 0x40000000 | Данная секция предназначена для чтения. Этот флаг почти всегда установлен для секций EXE-файлов |
| 0x80000000 | Данная секция предназначена для записи. Если этот флаг не установлен в секции EXE-файла, загрузчик должен отметить отображенные в память страницы как предназначенные только для чтения или только для исполнения. Типичные секции с этим атрибутом - это .data и .bss |

Интересно отметить, чего не хватает в информации, хранящейся в каждой секции. Во-первых, следует обратить внимание на отсутствие любых атрибутов PRELOAD. Файловый формат NE позволяет пользователю определять атрибуты PRELOAD для сегментов, которые должны быть загружены сразу во время загрузки модуля. Файловый формат OS/2 2.0 LX имеет нечто похожее, что позволяет пользователю давать указание о том, что предварительно должно быть загружено до восьми страниц. PE-формат напротив, не имеет ничего подобного. Исходя из этого,

приходится заключить, что Microsoft абсолютно уверена в исполнимости требований загрузки страниц для своих реализации Win32.

В PE-формате также отсутствует таблица поиска промежуточных страниц. Эквивалент IMAGE_SECTION_HEADER в файловом формате OS/2 LX не указывает, непосредственно, где находятся в файле данные и программный код секции. Вместо этого файл формата OS/2 LX содержит таблицу поиска страниц, определяющую атрибуты и расположение в файле определенных диапазонов страниц внутри секции. PE-формат обходится без всего этого и гарантирует, что данные из секции будут храниться непрерывно в файле. Сравнивая два формата, можно сказать, что LX-метод более гибок, тогда как сталь PE намного проще в работе.

Другим благоприятным отличием PE-формата от более старого NE-формата является то, что расположения элементов хранятся в виде простых смещений типа DWORD. В NE-формате расположение практически любого элемента хранилось в виде величины сектора. Чтобы посчитать действительное файловое смещение, нужно было сначала найти размер выравнивания в заголовке NE-файла и перевести его в размер сектора (обычно 16 или 512 байт). Затем нужно было умножить размер сектора на указанное смещение сектора, чтобы получить действительное файловое смещение. Если что-нибудь по случайности не хранится в виде секторного смешения в NE-файле, оно, вероятно, хранится как смещение относительно заголовка NE-файла. Ввиду того, что заголовок NE-файла не находится в начале файла, пользователю приходится привлекать в свою программу файловое смещение заголовка NE-файла. В противоположность этому PE-файлы определяют положение различных элементов, используя простые смещения относительно того положения, в которое файл был отображен в памяти. В общем, с PE значительно проще

работать, чем с форматами NE, LX или LE (при условии, что можно использовать отображаемые в память файлы).

## Часто встречающиеся секции

Секции представлены в порядке важности и в соответствии с вероятной частотой их появления.

### Секция .text

В этой секции собран весь программный код общего назначения, генерируемый компилятором или ассемблером. Поскольку PE-файлы работают в 32-разрядном режиме и не привязаны к 16-разрядным сегментам, нет необходимости разбивать программный код из разных файлов-источников по разным секциям. Вместо этого компоновщик объединяет все секции .text из различных объектных файлов в одну большую секцию .text в EXE-файле. Компилятор Borland C++ помещает весь программный код в сегмент с названием CODE. Таким образом, PE-файлы, созданные с помощью Borland C++, имеют секцию с названием CODE вместо секции .text.

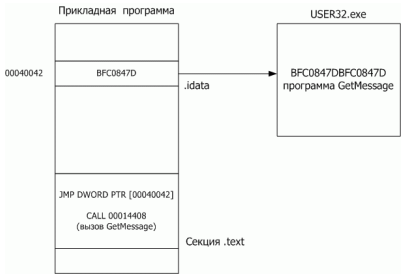
Кроме кода вашей программы в файле будет присутствовать дополнительный программный код в секции .text, помимо того, который создется компилятором или используется из библиотек поддержки выполнения программы. В PE-файле, в случае вызова функции из другого модуля (например, GetMessage() из USER32.DLL), инструкция CALL, сгенерированная компилятором, не передает управление непосредственно данной функции в DLL. Вместо этого инструкция CALL передает управление команде JMP DWORD PTR[XXXXXXXX], также находящейся в секции .text. Команда JMP перескакивает к адресу, хранящемуся в двойном слове в секции .idata. Это двойное слово в секции .idata содержит настоящий адрес точки входа функции операционной системы, как показано на рисунке ниже.

Организовав таким образом вызовы DLL в PE-файлах и стянув все вызовы данной функции DLL в одно место, загрузчик не будет "латать" каждую инструкцию, вызывающую DLL. Все, что остается загрузчику, это поместить правильный адрес целевой функции в двойное слово в секции .idata. Не нужно "латать" никаких инструкций CALL. Это представляет большое отличие от ситуации с NE-файлами, в которых каждый сегмент содержит перечень привязок, которые должны применяться к сегменту. Если какой-нибудь сегмент вызывает данную функцию из DLL 20 раз,

загрузчику нужно будет 20 раз копировать адрес функции в этот сегмент. Недостатком PE-метода является то, что пользователь не может инициализировать переменную истинным адресом функции DLL. Например, пользователь может полагать, что нечто вроде:

FARPROC pfnGetMessage = GetMessage;

поместит адрес GetMessage в переменную pfnGetMessage. В Win16 это сработает, а в Win32 - нет. В Win32 переменная pfnGetMessage в итоге будет содержать адрес переходника JMP DWORD PTR [XXXXXXXX] в секции .text, о котором было упомянуто раньше. Если бы пользователь захотел сделать вызов с помощью указателя функции, то все произошло бы так, как пользователь и ожидал. Однако, прочитать байты в начале GetMessage(), не удастся (если не проделать дополнительной работы, проследовав за "указателем" .idata самостоятельно).




Вызов импортируемых функций из PE-файла

Изменения произошли после выпуска Visual C++ 2.0. Эта версия содержала новинку в вызове импортированных функций. Если заглянуть в системные файлы заголовков из Visual C++ 2.0 (например, WINBASE.H), можно обнаружить отличие от заголовков Visual C++ 1.0. В Visual C++ 2.0 прототипы функций операционной системы в системных DLL включают _declspec(dllimport) как часть их определения. _declspec(dllimport) имеет полезное свойство при вызове импортированных функций. Когда пользователь вызывает импортированную функцию, прототипированную с помощью _declspec(dllimport), компилятор не создает вызов инструкции JMP DWORD PTR[XXXXXXXX] где-нибудь еще в модуле. Вместо этого компилятор генерирует вызов функции в виде CALL DWORD PTR|XXXXXXXX], Адрес [XXXXXXXX] находится в секции .idata. Это тот же самый адрес, который использовала, если бы использовался наш старый знакомый JMP DWORD PTR[XXXXXXXX]. Все версии Borland C++, вплоть до 4.5, не имеют этого свойства.

## Секции Borland CODE и .icode

Компилятор и компоновщик Borland C++ не работают с COFF-форматом объектных файлов. Вместо этого Borland предпочел придерживаться 32-разрядной версии формата Intel OMF. Хотя Borland мог бы заставить компилятор генерировать сегменты с именем .text, эта фирма предпочла CODE в качестве имени сегмента по умолчанию. Чтобы определить имя секции в PE-файле, компоновщик Borland (TLINK32.EXE) извлекает имя сегмента из объектного файла и обрезает его до 8 символов (в случае необходимости). Из-за этого PE-файлы Borland C++ имеют секцию CODE, а не секцию .text.

Разница в именах секций, конечно, не является главной, существует более важное отличие в том, как Borland PE-файлы компонуются с другими модулями. Как было сказано раньше, при обсуждении секции .text, все вызовы объектных файлов идут через переходник JMP DWORD PTR [XXXXXXXX]. В системе Microsoft этот переходник приходит в EXE-файл из секции .text импортируемой библиотеки. Менеджер библиотек создает импортируемую библиотеку (и переходник), когда пользователь присоединяет внешнюю DLL. В результате компоновщик не обязан "знать", как создавать эти переходники самому. Библиотека импорта - это в действительности только некоторое количество программного кода и данных для компоновки в PE-файл.

Система оперирования с импортированными функциями Borland иная и представляет просто обобщение действий, которые проводились для 16-разрядных NE-файлов. Библиотеки импорта, используемые компоновщиком Borland, в действительности представляют перечень имен функций и DLL, в которых они находятся. Таким образом, TLINK32 отвечает за определение того, какие привязки предназначены для внешних DLL, и за генерацию соответствующего переходника JMP DWORD PTR[XXXXXXXX]. В Borland C++ 4.0 TLINK32 хранит переходники, которые он создает, в секции с именем .icode. В Borland C++ 4.02 TLINK32 изменен, чтобы собрать все переходники JMP DWORD PTR[XXXXXXXX] в секции CODE.

## Секция .data

Как по умолчанию программный код попадает в секцию .text, так и инициализированные данные попадают в секцию .data.

Инициализированные данные состоят из тех глобальных и статических переменных, которые были проинициализированы во время компиляции. Они также включают строковые литералы (например, строку "Hello World"). Компоновщик объединяет все секции .data из разных объектных и LIB-файлов в одну секцию .data в EXE-файле. Локальные переменные расположены в стеке цепочки и не занимают места в секциях .data и .bss.

### Секция DATA

Borland C++ использует по умолчанию имя DATA для секции данных. Она эквивалентна секции .data в компилятора Microsoft (см. предыдущий пункт "Секция .data").

### Секция .bss

В секции .bss хранятся неинициализированные статические и глобальные переменные. Компоновщик объединяет все секции .bss из разных объектных и LIB-файлов в одну секцию .bss в EXE-файле. В таблице секций ноле RawDataOffset для секции .bss устанавливается в 0, показывая, что эта секция не занимает никакого места в файле. TLINK32 нс создает секцию .bss. Вместо этого он расширяет виртуальный размер секции DATA так, чтобы вместить неинициализированные данные.

### Секция .CRT

Еще одна секция для инициализированных данных, используемая библиотеками поддержки выполнения программы Microsoft C/C++ (отсюда и название .CRT - C/C++ runtime libraries). Данные из этой секции используются для таких целей, как вызов конструкторов статических классов C++ перед вызовом main или WinMain.

### Секция .rsrc

Секция .rsrc содержит ресурсы модуля. На ранних стадиях развития NT выходной .RES-файл 16-разрядного RC.EXE имел формат, который не воспринимался компоновщиком Microsoft. Программа CVTRES переводила эти .RES-файлы в объектные файлы COFF-формата, помещая данные ресурсов в секцию .rsrc внутри объектного файла. После этого компоновщик мог рассматривать объектный файл ресурсов как еще один объектный файл для компоновки, что позволяет компоновщику не вникать во что-либо

особенное о ресурсах. Более современные компоновщики Microsoft оказались способными обрабатывать файлы .RES непосредственно.

## Секция .idata

Секция .idata содержит информацию о функциях (и данных), которые модуль импортирует из других DLL. Эта секция эквивалентна справочной таблице модуля в NE-файле. Коренное отличие состоит в том, что каждая функция, импортируемая PE-файлом, перечислена в этой секции. Чтобы отыскать эквивалентную информацию в NE-файле, пользователю пришлось бы рыться в поправках в конце исходных данных для каждого из сегментов.

## Секция .edata

Секция .edata представляет перечень функций и данных, которые PE-файл экспортирует для использования другими модулями. Ее эквивалент для NE-файла - это комбинация таблицы входа, таблицы резидентных имен и таблицы нерезидентных имен. В отличие от Win16, здесь редко возникает необходимость экспортировать что-либо из EXE-файлов, так что обычно секцию .edata можно увидеть только в DLL. Исключением являются EXE-файлы, созданные Borland C++, которые, по-видимому, всегда экспортируют функцию (_GetExceptDLLinfo) для внутреннего использования библиотекой поддержки исполнения программы.

При использовании средств Microsoft данные из секции .edata попадают в PE-файл через файл .EXP. Другими словами, компоновщик не создает эту информацию сам. Вместо этого он полагается на менеджера библиотек (LIB32), сканирующего OBJ-файлы и создающего файл .EXP, который компоновщик добавляет в свой перечень модулей для компоновки. Эти файлы .EXP- в действительности всего лишь OBJ-файлы с другим расширением. Используя программу PEDUMP с ключом /S (показать таблицу символов), можно увидеть функции, экспортируемые через файлы .EXP.

## Секция .reloc

Секция .reloc содержит таблицу *базовых поправок* (base relocation). Базовая поправка - это настройка по отношению к инструкции или значению инициализированной переменной; EXE-файлы или DLL

нуждаются в такой поправке, если загрузчик не может загрузить файл по адресу, который предполагался компоновщиком. Если загрузчику удастся загрузить отображение по указанному компоновщиком базовому адресу, загрузчик игнорирует поправочную информацию в этой секции.

В случае если вы уверены, что загрузчик всегда сможет загрузить отображение по указанному компоновщиком базовому адресу, используйте ключ /FIXED, чтобы компоновщик удалил эту информацию. Хотя это и сохраняет место в исполняемом файле, однако это же может сделать данный файл неработающим на других платформах Win32. Пусть, например, вы создали EXE-файл для NT и расположили его по адресу 0x10000. Если вы дали указание компоновщику удалить поправки, данный EXE-файл не будет работать в Windows 9x, так как там адрес 0x10000 недоступен (наименьший адрес загрузки в Windows 9x - 0x400000, т.е. 4 Мбайт).

Необходимо отметить, что инструкции JMP и CALL, генерируемые компилятором, используют смещения относительно этих инструкций, а не действительные смещения в 32-разрядном сегменте. Если отображение необходимо загрузить по базовому адресу, отличному от указанного компоновщиком, не нужно изменять эти инструкции, поскольку они используют относительную адресацию. В результате поправок не так много, как могло бы показаться. Поправки обычно требуются только для инструкций, использующих 32-разрядные смещения для данных. Допустим, имеются следующие объявления глобальных переменных:

```
int i;

int *ptr = &i;
```

Если компоновщик указал в качестве базового адреса отображения 0x10000, адрес переменной i будет в итоге содержать что-нибудь наподобие 0x12004. В памяти, используемой под указатель ptr, компоновщик поместит значение 0x12004, поскольку это - адрес переменной i. Если загрузчик решит (по каким-либо причинам) загрузить файл по базовому адресу 0x70000, адресом переменной i будет в таком случае 0x72004. Однако значение переменной prt перед инициализацией

теперь будет неправильным, так как i сейчас находится на 0x60000 байт выше в памяти.

Вот здесь-то поправочная информация и вступает в игру. Секция .reloc - это перечень перемещений, т.е. мест в отображении, а которых необходимо принимать в учет различие между принятым компоновщиком адресом загрузки и реальным адресом загрузки.

**Секция .tls**

Когда используется директива компилятора "__declspec(thread)", определяемые данные не попадают ни в секцию .data, ни в секцию .bss. Вместо этого их копия в итоге оказывается в секции .tis. Имя секции .tis происходит от *thread local storage* (локальная память цепочки). Эта секция связана с семейством функций TlsAlloc().

Локальную память цепочки можно представить как отдельный набор глобальных переменных для каждой цепочки. Это означает, что каждая цепочка может иметь свой набор величин статических данных, однако программный код, использует эти данные, безотносительно к тому, какая цепочка исполняется. Рассмотрим программу, имеющую несколько цепочек, которые работают над одной и той же задачей, т.е. исполняют один и тот же программный код. Если пользователь объявил переменную локального хранения цепочки, например:

__declspec (thread) int i = 0; //Это объявление глобальной переменной.

то каждая цепочка будет иметь свою собственную копию переменной i.

Также возможен явный запрос на использование локальной памяти цепочки во время исполнения программы с помощью функций TlsAlloc, TlsSetValue и TlsGetValue. В большинстве случаев гораздо проще объявлять данные в программе с помощью _declspec (thread), чем распределять память для цепочки и запоминать указатель на нее в слоте, выделенном функцией TlsAlloc().

Необходимо упомянуть об одной отрицательной стороне секции .tis и переменных _declspec (thread). В NT и Windows 9x механизм локального хранения цепочки не будет работать для DLL, если эта DLL загружена динамически с помощью LoadLibrary(). Для EXE или

неявно загруженной DLL все будет работать прекрасно. Если нет возможности явно скомпоновать DLL, а для цепочки необходима ее локальная намять, то придется ее распределить динамически с помощью TlsAlloc() и TlsGetValue(). Необходимо отметить, что на самом деле блоки памяти для цепочки не хранятся в секции .tls во время исполнения программы. Другими словами, переключая цепочки, менеджер памяти не изменяет физическую страницу памяти, отображенную в секцию .tis модуля. Вместо этого секция .tls представляет просто данные, используемые для инициализации настоящих блоков данных для цепочек. Инициализация областей данных для цепочек производится совместными усилиями операционной системы и библиотек поддержки выполнения программы. Это требует дополнительных данных - каталога TLS, хранящегося в секции .rdata.

## Секция .rdata

Секция .rdata имеет, как минимум, четыре предназначения. Во-первых, в EXE-файлах, созданных с помощью компоновщика Microsoft Link, секция .rdata содержит каталог отладки (в объектных файлах такого каталога нет). В EXE-файлах, созданных с помощью компоновщика TLINK32, каталог отладки находится в секции .debug. Каталог отладки представляет массив структур IMAGE_DEBUG_DIRECTORY. Эти структуры содержат информацию о типе, размере и местонахождении различных видов отладочной информации, содержащейся в файле. Есть три главных вида отладочной информации:

CodeView, COFF и FPO. Ниже показан вывод типичного каталога отладки с помощью программы PEDUMP.

| Type | Size | Address | FilePtr | Charactr | TimeData | Version |
|---|---|---|---|---|---|---|
| COFF | 000065C5 | 00000000 | 00009200 | 00000000 | 2CF8CF3D | 0.00 |
| (unknown) | 00000114 | 00000000 | 0000F7C8 | 00000000 | 2CF8CF3D | 0.00 |
| FPO | 000004BO | 00000000 | 0000F8DC | 00000000 | 2CF8CF3D | 0.00 |
| CODEVIEW | 0000B0B4 | 00000000 | 0000FD8C | 00000000 | 2CF8CF3D | 0.00 |

Типичный каталог отладки

Каталог отладки не обязательно должен находиться в начале секции .rdata. Для того чтобы обнаружить начало каталога отладки, следует использовать RVA, содержащийся в седьмой

строке (IMAGE_DIRECTORY_ENTRY_DEBUG) каталога данных. (Каталог данных находится в конце заголовка PE-файла.) Чтобы определить количество входов в каталоге отладки для компоновщика Microsoft, нужно разделить размер этого каталога (находится в поле размера в указанной выше строке каталога данных) на размер структуры IMAGE_DEBUG_DIRECTORY. В случае же компоновщика TLINK32 соответствующее поле размера уже содержит количество строк каталога отладки, а не общую длину в байтах. Программа PEDUMP обрабатывает обе ситуации.

Другой важной частью секции .rdata является строка описания. Если пользователь определяет элемент DESCRIPTION в файле .DEF в своей программе, то в секции .rdata появляется строка описания. В NE-формате строка описания всегда является первой строкой нерезидентной таблицы имен. Строка описания предназначена для хранения полезного текста, описывающего файл.

Кроме того, секция .rdata используется для GUID при OLE-программировании. Библиотека импорта UUID.LIB содержит набор 16-разрядных GUID, используемых в случаях ID-интерфейсов. Эти GUID в итоге оказываются в секции .rdata EXE-файла или DLL.

Последнее применение секции .rdata - это место для хранения каталога TLS (Thread Local Storage - локальная память цепочки). Каталог TLS - это специальная структура данных, используемая библиотекой поддержки выполнения программы для явного обеспечения локальной памяти цепочки для переменных, объявленных в программе. Формат каталога TLS можно найти на CD-ROM MSDN (Microsoft Developer Network) в спецификации Portable Executable and Common Object File Format. Наибольший интерес в каталоге TLS представляют указатели начала и конца копии данных, используемых для инициализации каждого блока локальной памяти цепочки. RVA каталога TLS находится в элементе IMAGE_DIRECTORY_ENTRY_TLS в каталоге данных заголовка PE-файла.

### Секции .debug$S и .debug$T

Секции .debug$S и .debug$T есть только в COFF-объектных файлах, и они содержат информацию о символах CodeView и их типах. Названия этих секций произошли от названий сегментов, используемых для этой цели предыдущими компиляторами Microsoft ($$SYMBOLS и $$TYPES). Единственное назначение секции

.debug$T - хранить путь к файлу .PDB, содержащему информацию CodeView о типах для всех объектных файлов проекта. Информацию CodeView для создаваемого EXE-файла компоновщик помещает в файл .PDB.

### Секция .drective

Эта секция есть только в объектных файлах. Она содержит текст команд для компоновщика. Например, следующие строки появлялись в секции .drective в любом объектном файле, который компилируется с помощью компилятора Microsoft Visual C++;

-defaultlib:LIBC -defaultlib.OLDNAMES.

При использовании в программе _declspec(export) компилятор просто вырабатывает эквивалент командной строки в секции .drectve (например, export: MyFunction).

### Секции, содержащие символ $ (для LIB и объектных файлов)

В объектных файлах секции, содержащие $ (например, .idata$2) обрабатываются компоновщиком по-особому. Компоновщик объединяет все секции, имеющие одинаковые символы в имени перед символом $. Именем получившейся секции считается то, что находится перед символом $. Таким образом, если компоновщик встречает секции с именами ,idataS2 и .idata$6, он объединяет их в одну секцию с именем .idata.

Упорядочение объединяемых секций происходит в соответствии с символами после $. Компоновщик соблюдает лексический порядок, так что секция .idata$2 будет идти перед секцией .idata$6, а секция .data$A - перед секцией ,data$B.

Чаще всего символ $ используется библиотеками импорта, которые в секциях .idata$x хранят различные порции суммарной секции .idata. Это достаточно интересно. Компоновщик не должен создавать секцию .idata с нуля. Вместо этого итоговая секция .idata создается из секций объектных или LIB-файлов, которые компоновщик рассматривает как любую другую секцию, подлежащую компоновке.

### Разнообразные секции

Существуют и другие секции. Например, в Windows 9x GDI32.DLL содержит секцию данных под названием _GPFIX, назначение которой предположительно связано с обработкой ошибок GP.

Отсюда можно извлечь двойной урок. Не обязательно придерживаться использования только стандартных секций, производимых компилятором или ассемблером. Если необходима отдельная секция, не бойтесь использовать ее. При работе с компилятором Microsoft C/C++ можно пользоваться #pragma codeseg и #pragma dataseg. Пользователи компилятора Borland могут использовать #pragma codeseg и #pragma dataseg. В ассемблере можно просто создать 32-разрядный сегмент с именем, отличным от имен стандартных секций. Компоновщик TLINK32 объединяет сегменты программного кода одного класса, так что следует либо присваивать каждому сегменту программного кода свое уникальное имя класса, либо отключить упаковку сегментов программного кода. Другой урок: необычные имена секций часто позволяют глубже взглянуть на назначение и реализацию конкретного PE-файла.

# Импортирование в PE-файлах

Итак, вызовы функций из внешних DLL не обращаются к этим DLL непосредственно. Вместо этого инструкция CALL передаёт управление инструкции JMP DWORD PTR[XXXXXXXX] где-то в секции .text исполняемого файла (или в секции .icode, если используется Borland C++ 4.0). Если используется _declspec(dllimport) в Visual C++, вызов функции принимает вид CALL DWORD PTR[XXXXXXXX]. В обоих случаях адрес, который ищет инструкция JMP или CALL, хранится в секции .idata. Инструкция JMP или CALL передает управление по этому адресу, являющемуся предполагаемым адресом цели.

Перед загрузкой в память информация, хранящаяся в секции .idata PE-файла, содержит информацию, необходимую для того, чтобы загрузчик мог определить адреса целевых функций и пристыковать их к отображению исполняемого файла. После загрузки секция .idata содержит указатели функций, импортируемых EXE-файлом или DLL. Все массивы и структуры, обсуждаемые в этом разделе, содержатся в секции .idata.

Секция .idata (таблица импорта) начинается с массива, состоящего из IMAGE_IMPORT_DESCRIPTOR. Каждый элемент (IMAGE_IMPORT_DESCRIPTOR) соответствует одной из DLL, с которой неявно связан данный PE-файл. Количество элементов в массиве нигде не учитывается. Вместо этого последняя структура массива IMAGE_IMPORT_DESCRIPTOR имеет поля, содержащие NULL. Структура IMAGE_IMPORT_DESCRIPTOR имеет следующий формат.

*DWORD Characteristics/OriginalFitrstThunk*

В этом поле содержится смещение (RVA) массива двойных слов. Каждое из этих двойных слов в действительности является объединением IMAGE_THUNK_DATA. Каждое двойное слово IMAGE_THUNK_DATA соответствует одной функции, импортируемой данным EXE - файлом или DLL. Если используется утилита BIND, то этот массив двойных слов не изменяется, а модифицируется массив двойных слов FirstThunk.

*DWORD TimeDateStamp*

Отметка о времени и дате, указывающая, когда был создан данный файл. Обычно это поле содержит 0. Тем не менее утилита Microsoft BIND обновляет это поле датой и временем из DLL, на которую указывает данный IMAGE_IMPORT_DESCRIPTOR.

*DWORD ForwarderChain*

Это поле имеет отношение к передаче, когда одна DLL передает ссылку на какую-то свою функцию другой DLL. Например, в Windows NT KERNEL32.DLL посылает несколько своих экспортируемых функций NTDLL.DLL. Приложение может посчитать это вызовом функции в KERNEL32.DLL, но в итоге это будет вызов в NTDLL.DLL. Это поле содержит указатель в массив FirstThunk. Функция, указанная этим полем, будет послана в другую DLL. К сожалению, формат посылки функции лишь вкратце описан в документации Microsoft.

*DWORD Name*

Это RVA строки символов ASCII, оканчивающейся нулем и содержащей имена импортируемых DLL (например, KERNEL32.DLL или USER32.DLL).

*PIMAGE_THUNK_DATA FirstThunk*

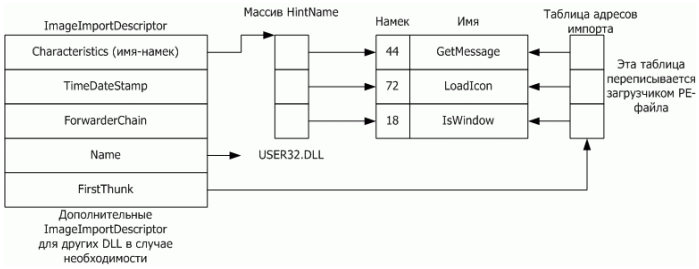
RVA-смещение массива двойных слов IMAGE_THUNK_DATA. В большинстве случаев двойное слово рассматривается как указатель на структуру IMAGE_IMPORT_BY_NAME. Однако можно импортировать функцию также и по порядковому номеру.

Важными частями IMAGE_IMPORT_DESCRIPTOR являются имя импортируемой- DLL и два массива элементов IMAGE_THUNK_DATA DWORD. Каждое двойное слово IMAGE_THUNK_DATA соответствует одной импортируемой функции. В EXE-файлах оба эти массива (на них указывают поля Characteristics и FirstThunk) идут параллельно друг другу и оканчиваются элементом-указателем NULL в конце каждого массива.

Зачем нужны два параллельных массива указателей на структуры IMAGE_THUNK_DATA? Первый массив (на него указывает поле Characteristics) оставляется неизменным. Иногда его называют *таблицей имен-намеков* (hint-name table). Второй массив, на который указывает поле FirstThunk в IMAGE_IMPORT_DESCRIPTOR, переписывается PE - загрузчиком. Загрузчик последовательно перебирает IMAGE_THUNK_DATA и находит адрес функции, на которую ссылается последний. Затем загрузчик записывает в двойное слово IMAGE_THUNK_DATA адрес импортируемой функции.

Ранее было описано, что вызовы функций DLL проходят через переходник "JMP DWORD PTR[XXXXXXXX]". [XXXXXXXX] в переходнике ссылается на один из элементов массива FirstThunk. Поскольку массив, состоящий из IMAGE_THUNK_DATA, переписывается загрузчиком и в итоге содержит адреса всех

импортируемых функций, он называется "Таблица адресов импорта". Рисунок показывает оба этих массива.




Как PE-файл импортирует функции Для пользователей Borland есть некоторая дополнительная тонкость в этом описании. В PE-файле, созданном TLINK32, отсутствует один из массивов. В таких файлах поле Characteristics в IMAGE_IMPORT_DESCRIPTOR (т.е. в массиве имен-намеков) равно нулю (очевидно, загрузчики Win32 не нуждаются в этом массиве). Таким образом, во всех PE-файлах вообще обязан быть только массив, на который указывает поле FirstThunk (таблица адресов импорта).

В постоянной погоне за оптимизацией Microsoft "оптимизировала" массивы IMAGE_THUNK_DATA в системных DLL под Windows NT (например, KERNEL32.DLL). После этой оптимизации IMAGE_THUNK_DATA не содержит информации, необходимой для нахождения импортируемой функции. Вместо этого двойные слова IMAGE_TIIUNK_DATA уже содержат адреса импортируемых функций. Другими словами, для загрузчика нет необходимости выискивать адреса функций и переписывать массив переходников с адресами импортируемых функций, Массив уже содержит адреса импортируемых функций еще до загрузки. (Утилита BIND из Win32 SDK осуществляет такую оптимизацию.) К сожалению, это вызывает трудности при работе программ просмотра,

предполагающих, что массив содержит смещения RVA для элементов IMAGE_THUNK_DATA. Вы можете подумать: "А почему не использовать таблицу имен-намеков?" Это было бы идеальным решением, если бы таблица имен-намеков существовала в файлах Borland.

Поскольку таблица адресов импорта находится обычно в доступной для записи секции, то не представляет труда перехватить вызовы, сделанные EXE или DLL другой DLL. Для этого нужно просто "залатать" соответствующий элемент таблицы адресов импорта так, чтобы он указывал на желаемую функцию-перехватчик. Не нужно модифицировать код ни в вызывающей, ни в вызываемой функции. Эта возможность представляется очень полезной.

Интересно заметить, что в PE-файлах Microsoft таблица импорта не полностью синтезируется компоновщиком. Вместо этого все элементы, необходимые для вызова функций из других DLL, располагаются в библиотеке импорта. При компоновке DLL менеджер библиотеки (LIB.EXE) сканирует компонуемые объектные файлы и создаст библиотеку импорта. Эта библиотека отличается от библиотек импорта, используемых 16-разрядными компоновщиками NE-файлов. Библиотека импорта, создаваемая 32-разрядными LIB-файлами, имеет секцию .text и несколько секций .idata$. Секция .text в библиотеке содержит переходник JMP DWORD PTR[XXXXXXXX], о котором я упоминал раньше. Имя этого переходника хранится в таблице символов объектного файла. Имя символа идентично имени экспортируемой DLL функции (например _DispatchMessage@4).

Одна из секций .idata$ в библиотеке импорте содержит двойное слово переходник. Другая секция .idata$

резервирует место для "номера намека", за которым следует имя импортируемой функции. Этих два поля составляют структуру IMAGE_IMPORT_BY_NAME. При компоновке PE-файла, использующего библиотеку импорта, секции библиотеки импорта добавляются к перечню секций объектного файла, подлежащих обработке компоновщиком. Ввиду того, что переходник в библиотеке импорта имеет такое же имя, как и импортируемая функция, компоновщик воспринимает переходник как импортируемую функцию и настраивает вызовы импортируемой функции так, чтобы они указывали на переходник. Поэтому переходник в библиотеке импорта трактуется как импортируемая функция.

Помимо обеспечения порций кода для переходника импортируемой функции библиотека импорта поставляет части секции .idata (или таблицы импорта) PE-файла. Эти части поступают из разных секций .idata$, помещаемых библиотекарем в библиотеку импорта. Короче говоря, компоновщик не различает импортированные функции и функции из другого объектного файла. Компоновщик просто следует своим предписаниям при создании и объединении секций, и все происходит вполне естественно.

## IMAGE_THUNK_DATA DWORD

Каждое двойное слово IMAGE_THUNK_DATA соответствует импортируемой функции. Интерпретация двойного слова зависит от того, был ли файл уже загружен в память и была ли функция импортирована по имени или по номеру (импортирование по имени встречается чаще).

При импортировании функции по номеру (что бывает редко) старший бит (0x80000000) двойного слова IMAGE_THUNK_DATA данного EXE-файла устанавливается в 1. Например, рассмотрим IMAGE_THUNK_DATA со значением 0x80000112 в массиве GDI32.DLL. Этот IMAGE_THUNK_DATA импортирует 112-ю экспортируемую функцию из GDI32.DLL. Проблема при импортировании по номеру состоит в том, что фирма Microsoft не позаботилась, чтобы поддержать соответствие номеров экспорта функций Win32 API для Windows NT, Windows 9x5 и Win32s.

Если функция импортируется по имени, то ее двойное слово IMAGE_THUNK_DATA содержит RVA структуры IMAGE_IMPORT_BY_NAME. Это простая структура выглядит следующим образом.

*WORD Hint*

Наилучшая догадка о том, какой номер экспорта у импортируемой функции. В отличие от NE-файлов эта величина не обязана быть верной. Загрузчик использует ее в качестве начального предполагаемого значения для бинарного поиска экспортируемой функции.

*BYTE[?]*

Строка ASCIIZ с именем импортируемой функции. Окончательная интерпретация двойного слова IMAGE_THUNK_DATA происходит после того, как PE-файл загружен загрузчиком Win32. Загрузчик Win32 использует исходную информацию из двойного слова IMAGE_THUNK_DATA для поиска адреса импортируемой функции (либо по имени, либо по номеру). Затем загрузчик записывает в двойное слово IMAGE_THUNK_DATA адрес импортируемой функции.

## Сравнение IMAGE_IMPORT_DESCRIPTOR и IMAGE_THUNK_DATA

Теперь, после обзора структур IMAGE_IMPORT_DESCRIPTOR и IMAGE_THUNK_DATA, будет просто составить отчет обо всех импортируемых функциях, которые использует EXE-файл или DLL. Для этого нужно просто перебрать последовательно все элементы массива IMAGE_IMPORT_DESCRIPTOR (каждый из которых соответствует одной импортируемой DLL). Для каждого элемента IMAGE_IMPORT_DESCRIPTOR найдите массив двойных слов IMAGE_THUNK_DATA и интерпретируйте его соответствующим образом. Ниже показан вывод программы PEDUMP для этой операции. (Функции без имен импортируются по номеру.)

---

Imports Table:

USER32.dll

Hint/Name Table: 0001F50C

TimeDateStamp: 2EB9CE9B

ForwarderChai: FFFFFFFF

First thunk RVA: 0001FC24

Ordn Name

268 GetScroll Info

133 DispatchMessageA

333 IsRectEmpty

431 SendMessageCallbackA

255 GstMessagePos

// Остальная часть таблицы опущена.

GDI32.dll

Hint/Name Table: 0001F178

TimeDateStamp: 2EB9CE9B

ForwarderChai: FFFFFFFF

First thunk RVA: 0001F890

Ordn Name

31 CreateCompatibleDC

309 SetTextColor

276 SetBkColor

99 ExtTextOutA

9 BitBIt

// Остальная часть таблицы опущена.

---

Типичная таблица импорта в EXE-файле

# Экспорт в PE-файлах

Противоположностью импорту функций является их экспорт для использования EXE-файлами или другими DLL, Информация об экспортируемых функциях хранится в секции .edata PE-файла. Как правило, EXE-файлы, созданные Microsoft LINK, ничего не экспортируют и поэтому не имеют секции .edata. EXE-файлы, созданные с помощью TLINK32, напротив, обычно экспортируют один символ и имеют секцию .edata. Большинство DLL экспортируют функции и имеют секцию .edata. Главными компонентами секции .edata (или, другими словами, таблицы экспорта) являются таблицы имен функций, адреса точек входа и номера экспорта. В NE-файле эквивалентами таблицы экспорта являются таблица элементов, таблица резидентных имен и таблица нерезидентных имен. В NE-файлах эти таблицы хранятся как часть заголовка NE-файла, а не в сегментах или ресурсах.

В начале секции .edata расположена структура IMAGE_EXPORT_DIRECTORY. После этой структуры сразу идут данные, на которые указывают поля структуры IMAGE_EXPORT_DIRECTORY. IMAGE_EXPORT_DIRECTORY выглядит следующим образом.

*DWORD Characteristics*

Это поле, по-видимому, никогда не используется и всегда устанавливается в 0.

*DWORD TimeDateStamp*

Отметка о времени и дате, указывающая время создания файла.

*WORD MajorVersion*

*WORD MinorVersion*

Эти ноля, по-видимому, никогда не используются и всегда устанавливаются в 0.

*DWORD Name*

RVA строки ASCIIZ с именем этой DLL (например, MYDLL.DLL).

*DWORD Base*

Начальный номер экспорта для функций, экспортируемых данным модулем. Например, если номера экспортируемых функций 10, 11 и 12, то это поле будет содержать 10.

*DWORD NumberOf Functions*

Количество элементов в массиве AddressOfFunctions. Это также число экспортируемых данным модулем функций. Обычно это значение такое же, как и в поле NumberOfNames (см. следующее описание), хотя они могут быть и различными.

*DWORD NumberOfNames*

Количество элементов в массиве AddressOfNames. Это значение соответствует количеству функций, экспортируемых по имени, которое обычно (хотя и не всегда) равно общему количеству экспортируемых функций.

*PDWORD * AddressOfFunctions*

Это поле является RVA и указывает на массив адресов функций. Адреса функций - это RVA точек входа для каждой экспортируемой модулем функции.

*PDWORD *AddressOfNames*

Это поле является RVA и указывает на массив указателей строки. Строки содержат имена функций, экспортируемых по имени из данного модуля.

*PWORD * AddressOfNameOrdinals*

Это поле является RVA и указывает на массив слов. По существу, в этих словах хранятся номера экспорта всех экспортируемых из данного модуля по имени функций. Однако не забудьте прибавить начальный номер экспорта, указанный в поле Base (описано выше).

Формат таблицы экспорта несколько странен. Обязательными при экспортировании функции являются адрес и номер экспорта. Если вы решили экспортировать функцию по имени, здесь будет имя функции. Можно было бы подумать, что разработчики PE-формата могли поместить все эти три компонента в одну структуру и после этого оперировать массивом таких структур. Вместо этого приходится выискивать различные части в трех различных массивах.

Наиболее важным из всех массивов, на которые указывает IMAGE_EXPORT_DIRECTORY, является массив, на который указывает поле AddressOfFunctions. Это массив двойных слов, в котором каждое двойное слово содержит RVA одной из экспортируемых функций. Номер экспорта каждой экспортируемой функции соответствует ее положению в массиве. Так, например, принимая, что номера экспорта начинаются с 1, адрес, по которому хранится адрес функции с номером

экспорта, равным 1, содержится в первом элементе массива. Адрес, но которому хранится адрес функции с номером экспорта, равным 2, содержится во втором элементе массива и т.д.

Необходимо помнить о двух моментах относительно массива AddressOfFunctions. Во-первых, нельзя забывать о том, что отсчет номеров экспорта начинается с числа, содержащегося в поле Base структуры IMAGE_EXPORT_DIRECTORY. Так, если поле Base содержит 10, то первое двойное слово в массиве AddressOfFunctions соответствует номеру экспорта 10, второе - 11 и т.д. Во-вторых, следует иметь в виду, что номера экспертов могут иметь пропуски. Допустим, что явно экспортируются две функции с номерами 1 и 3. Не смотря на то, что экспортированы только две функции, массив AddressOfFunctions обязан содержать три элемента. Любые элементы массива, не отвечающие экспортируемым функциям, содержат 0.

Когда загрузчик Win32 связывает вызов функции, экспортируемой по номеру, он выполняет совсем незначительный объем работ. Он просто использует номер функции как индекс в массиве AddressOfFunctions модуля цели. Конечно, загрузчик должен учесть, что наименьший номер экспорта может быть не равен 1, и должен поправить индексацию соответствующим образом.

Чаще всего EXE-файлы и DLL под Win32 импортируют функции по имени, а не по номеру. Здесь выходят на сцену два других массива, на которые указывает структура IMAGE_EXPORT_DIRECTORY. Массивы AddressOfNames и AdressOfNameOrdinals существуют для того, чтобы загрузчик мог быстро найти номер экспорта по заданному имени функции. Массивы AddressOfNames

и AddressOfNameOrdinals содержат одинаковое количество элементов (заданное в поле NumberOfNames структуры IMAGE_EXPORT_DIRECTORY). Массив AddressOfNames - это массив указателей на имена функций, а массив AddressOfNameOrdinals - массив индексов для массива AddressOfFunctions.

Итак, как же загрузчик Win32 обрабатывает вызов функции, импортируемой по имени? Сначала загрузчик будет искать строки, на которые указывает массив AddressOfNames. Допустим, он находит искомую строку в третьем элементе. Затем загрузчик использует найденный индекс для поиска соответствующего элемента в массиве AdressOfNameOrdinals (в данном случае это третий элемент). Последний массив - это просто набор слов, где каждое слово играет роль индекса в массиве AddressOfFunctions. Наконец последний шаг - взять значение в массиве AddressOfNameOrdinals и использовать его в качестве индекса для массива AddressOfFunctions.

В программе на C++ нахождение адреса функции, импортируемой по имени, будет выглядеть примерно следующим образом:

---

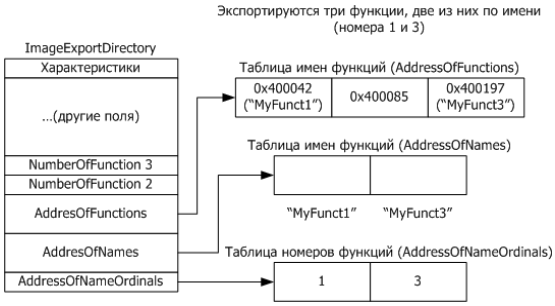
```
WORD nameIndex = FindIndexOfString(AddressOfNames, "GetMessageA");

WORD functionIndex = AddressOfNameOrdinals[nameIndex];

DWORD functionAddress = AddressOfFunctions[functionIndex - OrdinalBase];
```

---

Рисунок демонстрирует формат секции экспорта и три ее массива.




Типичная таблица экспорта из EXE-фаила:

Ниже показан вывод программы PEDUMP секции экспорта в KERNEL32.DLL.

---

Name: KERNEL32.dll

Characteristics: 00000000

TimeDateStamp: 2C4857D3

Version: 0.00

Ordinal base: 00000001

# of functions: 0000021F

# of Names: 0000021F

Entry Point Ordn Name

00005090 1 AddAtomA

00005100 2 AddAtamM

00025540 3 AddConsoleAliasA

00025500 4 AddConsoleAliasW

00026ACO 5 AllocConsole

00001000 6 BackupRead

00001E90 7 BackupSeek

00002100 8 BackupWrite

0002520C 9 BaseAttachCompleteThunk

00024C50 10 BasepOebugDump

//Остальная часть таблицы опущена

---

Распечатка секции экспорта для библиотеки KERNEL32.DLL с помощью программы PEDUMP

Если вы просматриваете экспорт в системных DLL (например, KERNEL32.DLL или USER32.DLL), вы можете случайно обнаружить, что часто две функции отличаются только одним символом в конце имени, например CreateWindowExA и CreateWindowExW. Вот так "явно" осуществлена поддержка уникода (unicode). Функции, оканчивающиеся на A, являются ASCII-совместимыми (или ANSI-) функциями. Функции, оканчивающиеся на W, - это Unicode-версии этих функций.

Программируя, пользователь не указывает явно, какую функцию надо вызывать. Вместо этого соответствующая функция выбирается в WINDOWS.H с помощью директивы препроцессора #ifdefs. Это

иллюстрируется следующим отрывком из NT WINDOWS.H:

---

```
#ifdef UNICODE

#define DefWindowProc DefWindowProcW

#else

#define DefWindowProc DefWindowProcA

#end if // ! UNICODE
```

---

**Передача экспорта**

Иногда в DLL полезно экспортировать функцию, а ее программный код иметь в другой DLL. При таком сценарии одна DLL может передавать функцию другой DLL. Если загрузчик Win32 встречает вызов передаваемой функции, он разрешает ссылку на функцию в ту DLL, которая содержит настоящий программный код.

Проиллюстрируем сказанное примером. Рассмотрим следующий отрывок из вывода программой PEDUMP Windows NT 3.5 KERNEL32.DLL:

---

```
00043FC3 335 HeapAlloc (forwarder -> NTDLL.RtlAllocateHeap)

00044005 339 HeapFree (forwarder -> NTDLL.RtlFreeHeap)

0004402C 341 HeapReAlloc (forwarder -> NTDLL.RtlReAllocateHeap)
```

0004404D 342 HeapSize (forwarder -> NTDLL.RtlSizeHeap)

0004466F 442 RtlFillMemory (forwarder -> NTDLL.RtlFillMemory)

00044691 443 RtlMoveMemory (forwarder -> NTDLL.RtlMoveMemory)

000446AF 444 RtlUnwind (forwarder -> NTDLL.,RtlUnwind)

000446CD 445 RtlZeroMemory (forwarder -> NTDLL.RtlZeroMemory)

---

Каждая функция в этом выводе передается функции в NTDLL. Таким образом, программа, вызывающая функцию HeapAlloc, в действительности вызывает функцию RtlAllocateHeap из NTDLL.DLL. Аналогично, вызов HeapFree является в действительности вызовом функции RtlHeapFree из NTDLL.

Каким образом можно узнать, что функция передается? Единственным указанием на то, что функция передается, является наличие ее адреса в таблице экспорта (секция .edata). В этом случае так называемый адрес функции в действительности является RVA строки, содержащей передаваемую DLL и имя функции. Например, в предыдущем выводе RVA для HeapAlloc равно Ox43FC3. Смещение Ox43FC3 в KERNEL32.DLL попадает в секцию .edata. Это смещение имеет строка NTDLL.RtlAllocateHeap.

Хотя передача экспорта кажется очень приятным свойством, Microsoft не даст описания того, как использовать передачу в пользовательских DLL. Даже несмотря на то, что встречаются DLL с передачами

экспорта под Windows 9x, загрузчик Windows 9x5, тем не менее, поддерживает это свойство.

# Базовые поправки PE-файла

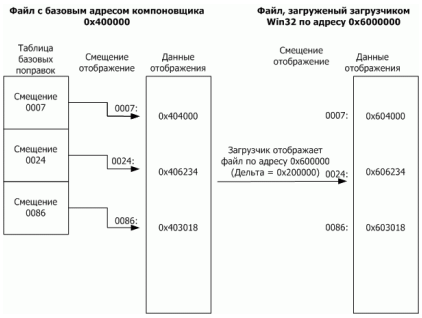
Компоновщик, создавая EXE-файл, предполагает, где в памяти будет отображен файл, и затем помещает предполагаемые адреса элементов программного кода и данных в исполняемый файл. Если, однако, исполняемый файл загружается куда-нибудь в другое место в виртуальном адресном пространстве, адреса, проставленные компоновщиком, оказываются неверными. Информация, хранящаяся в секци .reloc, позволяет загрузчику PE-файла исправить эти адреса в загружаемом модуле. Когда загрузчику удастся загрузить файл по базовому адресу, указанному компоновщиком, информация в секции .reloc игнорируется за ненадобностью. Элементы секции .reloc называются базовыми поправками, потому что их использование зависит от значения базового адреса загружаемого отображения.

В отличие от поправок в NE-файлах, базовые поправки PE-файлов чрезвычайно просты. Они не ссылаются на внешние DLL или на другие секции модуля. Вместо этого базовые поправки сводятся к перечню тех мест в отображении, где нужно прибавить некоторую величину.

Вот пример того, как работают базовые поправки. Предположим, что EXE-файл скомпонован в допущении, что базовый адрес равен 0x400000. Пусть указатель, содержащий адрес какой-либо строки, имеет смещение 0x2134 в отображении. Строка начинается с физического адреса 0x404002, так что указатель содержит это значение. В момент загрузки загрузчик решает, что модуль нужно отобразить в память, начиная с физического адреса 0x600000. Разность

между предполагаемым компоновщиком базовым адресом и реальным адресом загрузки называется *дельта.* В нашем случае дельта равна 0x200000 (0x600000-0x400000). Поскольку все отображение оказывается на 0x200000 байт выше в памяти, адрес строки теперь 0x604002. Указатель на строку теперь содержит неверное значение. Чтобы исправить его, к нему необходимо прибавить дельту (в нашем случае 0x200000).

Чтобы загрузчик Windows сделал это исправление, исполняемый файл содержит базовую поправку для того места в памяти, в котором находится указатель (его смещение в отображении равно 0x2134). Чтобы разрешить базовую поправку, загрузчик добавляет дельту к исходному значению, находящемуся по адресу, указанному в базовой настройке. В нашем случае загрузчик должен прибавить 0x200000 к исходному значению указателя (0x404002) и поместить это значение (0x604002) обратно в указатель. Раз строка действительно находится по адресу 0x604002, все снова становится правильным. Рисунок показывает весь этот процесс.




Базовые поправки PE-файла

Формирование данных базовых поправок выглядит несколько странно. Поправки упаковываются сериями смежных кусков различной длины. Каждый кусок

описывает поправки для одной четырехкилобайтовой страницы отображения и начинается со структуры IMAGE_BASE_RELOCATION, которая выглядит следующим образом.

*DWORD VirtualAddress*

Это поле содержит стартовый RVA для данного куска поправок. Смещение каждой поправки, которая следует дальше, добавляется к этой величине для получения истинного RVA, к которому должна быть применена данная поправка.

*DWORD SizeOfBlock*

Размер данной структуры плюс все последующие поправки типа WORD. Чтобы определить количество поправок в данном блоке, нужно из значения этого поля вычесть размер IMAGE_BASE_RELOCATION (8 байт) и затем разделить на 2 (размер типа WORD). Например, если это поле содержит значение 44, то в блоке имеется 18 поправок: (44 - sizeof (IMAGE_BASE_RELOCATION)) / sizeof (WORD) = 18

*WORD TypeOffset*

На самом деле это не отдельное слово, а массив слов, количество элементов в котором вычисляется но формуле, приведенной в описании предыдущего двойного слова. Младших 12 разрядов каждого из этих слов представляют поправочное смещение, которое должно быть прибавлено к значению в поле VirtualAddress из заголовка данного блока поправок. Старших 4 разряда каждого слова являются типом поправки. Для PE-файлов, исполняемых на процессорах серии Intel, существуют только два типа поправок:

• 0 (IMAGE_REL_BASED_ABSOLUTE). Эта поправка не имеет смысла и используется как заполнитель для выравнивания на границу следующего двойного слова.

• 3 (IMAGE_REL_BASED_HIGHLOW). Поправка подразумевает прибавление как старших, так и младших 16 разрядов дельты к двойному слову, на которое указывает вычисленный RVA.

Есть также другие поправки, определенные в WINNT.H, большая часть которых рассчитана на архитектуры процессоров, отличных от i386.

Ниже представлены некоторые базовые поправки, выведенные программой PEDUMP. Заметьте, что показанные значения RVA, были уже заранее определены полем VirtualAddress структуры IMAGE_BASE_RELOCATION.

---

Virtual Address: 00001000 size: 0000012C

00001032 HIGHLOW

0000106D HIGHLOW

000010AF HIGHLOW

000010C5 HIGHLOW

//Остальная часть опущена...

Virtual Address: 00002000 size: 0000009C

000020A6 HIGHLOW

00002110 HiGHLOW

00002136 HIGHLOW

00002156 HIGHLOW

//Остальная часть опущена...

Virtual Address: 00003000 size: 00000114

0000300A HIGHLOW

0000301E HIGHLOW

0000303B HIGHLOW

0000306A HIGHLOW

//Остальные поправки опущены...

---

Базовые поправки в EXE-файле

# Ресурсы PE-файла

Нахождение ресурсов в PE-файлах сложнее по сравнению с эквивалентными NE-файлами Формат индивидуальных ресурсов (например, меню) существенно не изменился, но в PE-файлах приходится рыскать по сложной иерархии, чтобы найти их

Пермещения по иерархии каталогов ресурсов похожи на перемещения по жесткому диску. Здесь есть главный каталог (корневой), имеющий свои подкаталоги. Подкаталоги имеют свои собственные подкаталоги. В этих подкаталогах находятся файлы. Файлы аналогичны исходным данным ресурсов, содержащим такие элементы, как диалоговые шаблоны. В PE-файлах как корневой каталог, так и его подкаталоги являются структурами типа IMAGE_RESOURCE_DIRECTORY. Структура IMAGE_RESOURCE_DIRECTORY имеет следующий формат:

*DWORD Characteristics*

Теоретически это поле может содержать флаги ресурсов, но, по-видимому, оно всегда равно 0.

*DWORD TimeDateStamp*

Отметка о времени создания ресурса.

*DWORD MajorVersion*

*DWORD MinorVersion*

Теоретически эти поля могла бы содержать номер версии ресурса. По-видимому, они всегда равны 0.

*DWORD NumberOfNamedEntries*

Количество элементов массива (описан ниже), использующих имена и следующих за этой структурой. За дополнительной информацией обращайтесь к описанию поля DirectoryEntries.

*DWORD NumberOfIdEntries*

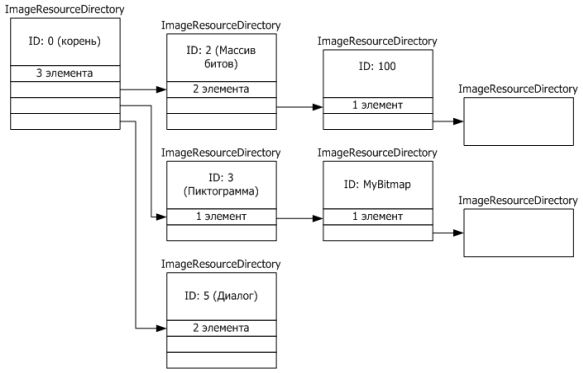
Количество элементов массива , использующих целые ID и следующих за этой структурой и всеми поименованными элементами. За дополнительной информацией обращайтесь к описанию поля DirectoryEntries.

*IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]*

Это поле формально не является частью структуры IMAGE_RESOURCE_DIRECTORY. За этим полем сразу следует массив структур IMAGE_RESOURCE_DIRECTORY_ENTRY. Количество элементов в массиве равно сумме полей NumberOfNamedEntries и NumberOfIDEntries. Элементы каталога, имеющие идентификаторы-имена (а не целые ID), находятся в начале массива.

Элемент каталога может указывать либо на подкаталог (т.е. на другую IMAGE_RESOURCE_DIRECTORY), либо на IMAGE_RESOURCE_DATA_ENTRY, которая описывает, где в файле находятся исходные данные ресурсов. Как правило, необходимо пройти как минимум три уровня каталогов, перед тем как попасть па IMAGE_RESOURCE_DATA_ENTRY для данного ресурса. Каталог верхнего уровня (только один) всегда находится в начале секции ресурсов (.rsrc). Подкаталоги каталога верхнего уровня соответствуют различным типам ресурсов, находящихся в файле. Например, если

PE-файл включает диалоги, таблицы строк и меню, этими тремя подкаталогами будут соответственно каталог диалогов, каталог таблицы строк и каталог меню. Каждый из этих "типов" подкаталогов будет в свою очередь иметь "ID"-подкаталоги. Для каждого образца заданного типа ресурса будет существовать один ID-подкаталог. Если в приведенном выше примере есть четыре диалоговых окна, каталог диалогов будет иметь четыре ID-подкаталога. Каждый ID-подкаталог будет иметь либо строковое имя (например, MyDialog), либо целый ID, используемый для идентификации ресурса в RC-файле. Ниже представлена иерархия каталогов ресурсов.




Иерархия ресурсов типичного PE-файла

Ниже показан вывод программы PEDUMP ресурсов файла CLOCK.EXE в Windows NT. На втором уровне отступов можно видеть пиктограммы, меню, диалоги, таблицы строк, пиктограммы групп и ресурсы версий. На третьем уровне - две пиктограммы (с ID 1 и 2), два меню (с именами CLOCK и GENERICMENU), два диалога (один с именем ABOUTBOX, а другой с целым ID, равным 0x64) и т.д. На четвертом уровне отступов - данные для значка 1 с RVA 0x9754 длиной 0x130 байт. Аналогично данные для меню CLOCK имеют смещение Ox952C и занимают OxEA байт.

Resources

ResDir (0) Named: 00 ID: 06 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (ICON) Named: 00 ID: 02 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (1) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 000001E0 Offset: 09754 Size: 00130 CodePage: 0

ResDir (2) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 000001F0 Offset: 09884 Size: 002E8 CodePage: 0

ResDir (MENU) Named: 02 ID: 00 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (CLOCK) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000200 Offset: 0952C Size: 000EA CodePage: 0

ResDir (GENERICMENU) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000210 Offset: 09618 Size: 0003A CodePage: 0

ResDir (DIALOG) Named: 01 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (ABOUTBOX) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000220 Offset: 09654 Size: 000FE CodePage: 0

ResDir (64) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000230 Offset: 092C0 Size: 0026A CodePage: 0

ResDir (STRING) Named: 00 ID: 02 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (1) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000240 Offset: 09EA8 Size: 000F2 CodePage: 0

ResDir (2) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000250 Offset: 09F9C Size: 00046 CodePage: 0

ResDir (GROUPICON) Named: 01 iD: 00 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (CCKK) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000260 Offset: 09B6C Size: 00022 CodePage: 0

ResDir (VERSION) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ResDir (1) Named: 00 ID: 01 TimeDate: 2E601E3C Vers: 0.00 Char: 0

ID: 00000409 DataEntryOffs: 00000270 Offset: 09B90 Size: 00318 CodePage: 0

---

Иерархия ресурсов для CLOCK.EXE

Каждый элемент каталога ресурсов - это структура типа IMAGE_RESOURCE_DIRECTORY_ENTRY. Каждая структура типа IMAGE_RESOURCE_DIRECTORY_ENTRY имеет следующий формат.

*DWORD Name*

Это поле содержит либо целый ID, либо указатель на структуру, содержащую строковое имя. Если старший разряд 0x80000000 равен 0, это поле интерпретируется как целый ID. Если старший разряд не равен 0, нижние разряды (только 31) являются смещением (по отношению к началу секции ресурсов) структуры IMAGE_RESOURCE_DIR_STRING_U. Эта структура содержит счетчик символов (счетчик типа WORD), за которым следует Unicode-строка с именем ресурса. Даже PE-файлы, не рассчитанные на Unicode-реализации Win32, используют здесь Unicode. Чтобы перевести Unicode-строку в стандарт ANSI, следует воспользоваться функцией WideCharToMultiByte.

*DWORD OffsetToData*

Это поле является либо смещением другого каталога ресурсов, либо указателем на информацию об особых образцах ресурсов. Если старший разряд (0x80000000) активизирован, этот элемент каталога ссылается на подкаталог. Младшие разряды (только 31) являются

смещением (по отношению к началу секции ресурсов) другой структуры IMAGE_RESOURCE_DIRECTORY. Если старший разряд не активизирован, младшие разряды (их 31) являются смещением по отношению к началу секции ресурсов структуры IMAGE_RESOURCE_DATA_ENTRY. Структура IMAGE_RESOURCE_DATA_ENTRY содержит местоположение исходных данных ресурса, их размер и кодовую страницу.

Описание индивидуальных форматов ресурсов различного вида находится в файле RESFMT.TXT Win32 SDK

# COFF-таблица символов

В любом объектном файле в COFF-стиле, созданном компилятором Microsoft, есть таблица символов. В отличие от информации CodeView, эта таблица символов не является дополнительным грузом, использующимся только при необходимости скомпоновать исполняемый файл с отладочной информацией. Напротив, эта таблица содержит информацию обо всех общеиспользуемых и внешних символах, на которые ссылается модуль. Информация о привязке, выдаваемая компилятором, относится к определенным элементам в этой таблице символов. Формат COFF-таблицы символов удивительно прост - по сравнению с очень запутанным форматом Microsoft/Intel OMF с его LNAME, PUBDEF и EXTDEF.

Если при компиляции отладочная информация не включается, то в таблице символов объектного файла будет находится лишь небольшое количество символов. Если же включить отладочную информацию (с помощью /Zi), то компилятор добавит дополнительную информацию о начале, конце и длине каждой функции модуля. Если затем провести компоновку либо с /DEBUGTYPE:COFF, либо с /DEBUGTYPE:BOTH, компоновщик поместит в получившийся EXE-фаил таблицу символов в COFF-стиле.

Зачем нужна COFF-информация, если есть намного более полная информация CodeView? Если используется системный отладчик NT (NTSD) или отладчик NT Kernel - KD (Kernel Debugger), то в игре участвует только COFF. К тому же если ваша PE-программа терпит катастрофу в Widows NT, DRWATSON32 может использовать эту информацию для "разбора полетов".

И в EXE-файлах, и в объектных файлах расположение и размер COFF-таблицы символов определены в структуре IMAGE_FILE_HEADER (см.- раздел "Заголовок PE-файла"). Таблица символов специально сделана простой и состоит из массива структур IMAGE_SYMBOL, Количество элементов в этом массиве задастся значением поля NumberOfSymbols структуры IMAGE_FILE_HEADER. Ниже показан пример вывода символов программой PEDUMP.

Каждая структура IMAGE_SYMBOL имеет следующий формат:

---

```
typedef struct _IMAGE_SYMBOL {

union {

BYTE ShortName[8];

struct {

DWORD Short;

DWORD Long;

} Name:

PBYTE LongName[2];

} N;

DWORD Value;

SHORT SectionNumber;

WORD Type;

BYTE StorageClass;
```

```
BYTE NumberOfAuxSymbols

} IMAGE_SYMBOL;

typedef IMAGE_SYMBOL UNALIGNED *PIMAGE_SYMBOL
```

---

Изучим каждое из этих полей детально.

*union N (Symbol name union)*

Символьное имя можно представить двумя способами, в зависимости от его длины. Если оно не длиннее 8 знаков, то член объединения ShortName содержит символьное имя в формате ASCIIZ. Следует быть осторожным в случае, когда символьное имя содержит в точности 8 знаков; при этом строка не оканчивается нулем. Если поле Name.Short не равно нулю, следует использовать член объединения ShortName. Другой способ представления символьного имени применяется, когда поле Name.Short равно 0. В этой ситуации поле Name.Long является байтовым смещением в таблице строк. Таблица строк - это не что иное, как массив ASCIIZ-строк, следующих одна за другой в памяти. Эта таблица начинается сразу за таблицей символов. Чтобы посчитать адрес начала таблицы строк, нужно просто умножить количество символов на размер структуры IMAGE_SYMBOL и прибавить результат к стартовому адресу таблицы символов. Длина таблицы строк в байтах находится в двойном слове, имеющем смещение 0 в таблице строк.

*DWORD Value*

Это поле содержит значение, связанное с символом. Для нормальных символов и символов данных (т.е. функции и глобальные переменные) поле Value содержит RVA элемента, на который ссылается данный символ. Это значение интерпретируется иначе для некоторых других символов. В таблице представлен краткий перечень некоторых назначений поля Value для специальных символов.

Специальные символы в COFF-таблицах символов

| Имя символа | Использование |
|---|---|
| file | Индекс символьной таблицы следующего символа .file. |
| data | Стартовый RVA области данных. Эта область определяется исходным файлом, заданным предыдущим символом .file. |
| .text | Стартовый RVA области программного кода. Эта область определяется исходным файлом, заданным предыдущим символом .file. |
| .lf | Количество элементов в таблице номеров строк для какой-либо функции. Функция задается предыдущим символом, определяющим функцию. |

### *SHORT SectionNumber*

Поле SectionNumber содержит номер секции, которой принадлежит символ. Например, символы для глобальных переменных будут, как правило, иметь в этом поле номер секции data. Помимо стандартных

секций PE-файла, определены три других специальных значения.

0 (IMAGE SYM_UNDEFINED). Символ не определен. Такой номер секции используется в объектных файлах для представления символов, находящихся вне модуля, например внешних функций и внешних глобальных переменных.

-1 (IMAGE_SYM_AB SOLUTE). Этот символ является абсолютной величиной и не связан ни с какой конкретной секцией. Примерами являются локальные и регистровые переменные.

-2 (IMAGE_SYM_DEBUG). Данный символ используется только отладчиком и не виден из программы. Символы .file, задающие имя исходного файла, - примеры такой символьной секции.

*WORD Type*

Тип символа. Файл WINNT.H определяет достаточно широкий спектр типов символов (int, struct, enum и т.д.). (См. полный перечень в директивах #defmes IMAGH_SYM_TYPE_xxx.) К сожалению, средства Microsoft, по-видимому, не генерируют символов всех возможных типов. Вместо этого все глобальные переменные и функции имеют тип NULL или тип функции, возвращающей NULL.

*BYTE StorageClass*

Класс памяти символа. Как и для типов символов файл WINNT.H определяет достаточно широкий спектр классов памяти: automatic, static, register, label и т.д. (См. полный перечень в директивах #define IMAGE_SYM_CLASS_xxx.) Опять-таки, как и в случае

типов, средства Microsoft создают только небольшое количество информации. Все глобальные переменные и функции имеют класс памяти внешний. По всей видимости, не существует способа создать символы для локальных переменных, регистровых переменных и т.д.

*BYTE NumberOfAuxSymbols*

Таблица символов не является в точности массивом структур IMAGE_SYMBOL. Если символ имеет ненулевое значение в записи NumberOfAuxSymbols, то за символом следует такое же число структур IMAGE_AUX_SYMBOL. Например, за символом .file следует столько структур IMAGE_AUX SYMBOL, сколько требуется, чтобы хранить полный путь к файлу-источнику.

К счастью, размер структуры IMAGE_AUX_SYMBOL такой же, как и у структуры IMAGE_SYMBOL, так что пользователь все же может рассматривать таблицу символов как массив структур IMAGE_SYMBOL. Следует помнить, что индекс символа должен рассматриваться как индекс массива, даже если некоторые элементы являются вспомогательными записями. Чтобы вычислить индекс следующего регулярного символа, нужно прибавить количество вспомогательных структур, используемых символом. Например, пусть символ имеет индекс 1. Если он использует три вспомогательных символа, то индекс следующего регулярного символа будет равен 4.

IMAGE_AUX_SYMBOL представляет собой запутанное объединение полей. Чтобы определить, какие члены объединения использовать, необходимо знать тип регулярного символа, связанного с данным вспомогательным символом. И хотя в документации Microsoft нет точного обьяснения, какие поля объединения должны быть использованы в каждом

случае, следует уяснил следующее: Символы, имеющие класс памяти IMAGE_SYM_CLASS_FILE, используют член объединения File в структуре IMAGE_AUX_SYMBOL.

Символы, имеющие класс памяти IMAGE_SYM_CLASS_STATIC, используют член объединения Section в структуре IMAGE_AUX_SYMBOL.

Изучая информацию внутри секции символов, можно заметить, что символы расположены не хаотически. Напротив, они сгруппированы по объектным модулям (или по исходным файлам), из которых они появились. Первой записью в COFF-таблице символов является запись .file. Значение записи -file - это индекс в таблице символов, указывающий на следующую запись .file. Следуя по этой цепочке записей .file, можно последовательно перебрать все объектные модули в EXE-фаЙле. Сразу за записью .file следуют другие записи, относящиеся к данному исходному файлу. Например, все общедоступные символы (глобальные переменные и функции), объявленные в исходном файле, идут сразу за записью .file, отвечающей данному исходному файлу. Для нормального исходного модуля "иерархия" записей выглядит следующим образом:

---

Source File record //Имя файла-источника.

Data Section record (e.g., ".data") //Данные, объявленные в файле.

GlobalVariable1 record // Информация о переменных.

GlobalVariable2 record

// Остальные записи глобальных переменных

Code Section record (e.g., ".text") // Программный код, объявленный в файле.

Function1 record // Информация о функции.

.BF record // Информация о начале функции.

.LF record // Информация о длине функции.

.EF record // Информация о конце функции.

Function2 record

.BF record

.LF record

.EF record // Остальные записи функции

---

# COFF-отладочная информация

Для среднего программиста термин *отладочная информация* включает как символьную информацию, так и информацию о номерах строк. В COFF-формате записи, относящиеся к символам и записи, относящиеся к номерам строк, находятся в разных областях файла. (В форматах фирмы Borland и в формате Code View для таблиц символов этих два вида информации поступают из одной и той же части файла.)

Вся COFF-таблица символов EXE-файла состоит из трех частей: заголовка, информации о номерах строк и таблицы символов. Они не обязательно расположены по соседству в памяти, но компоновщик Microsoft выстраивает их таким образом. Полная COFF-таблица символов выглядит так:

Структура IMAGE_COFF_SYMBOLS_HEADER

Таблицы номеров строк

Таблица символов (обсуждалась раньше)

Структура IMAGE_COFF_SYMBOLS_HEADER рассчитана на то, чтобы помочь отладчикам быстро найти необходимую им информацию. Эта структура содержит указатели на таблицы номеров строк и символов, а также на информацию, находящуюся где-либо в другом месте в файле.

Чтобы отыскать структуру IMAGE_COFF_SYMBOLS_HEADER, нужно заглянуть в массив структур IMAGE_DEBUG_DIRECTORY в секции .rdata в файле. Структура IMAGE_DEBUG_DIRECTORY содержащая в поле Type значение 1 (IMAGE_DEBUG_TYPE_COFF), содержит указатель на

COFF-таблицу символов. Итак: каталог данных (в конце заголовка PE-файла) содержит RVA массива структур IMAGE_DEBUG_DIRECTORY. Каждому типу отладочной информации, находящейся в файле, соответствует одна структура IMAGE_DEBUG_DIRECTORY. Если одна из этих структур IMAGE_DEBUG_DIRECTORY ссылается на отладочную информацию в COFF-стиле, она содержит RVA структуры IMAGE_COFF_SYMBOLS_HEADER. Структура IMAGE_COFF_SYMBOLS_HEADER в свою очередь содержит указатели на COFF-таблицу символов и информацию о номерах строк. Структура IMAGE_COFF_SYMBOLS_HEADER имеет следующий формат:

---

```
typedef struct IMAGE_COFF_SYMBOLS_HEADER

{

DWORD NumberOfSymbols;

DWORD LvaToFirstSymbol;

DWORD NumberOfLineNumbers;

DWORD LvaToFirstLineNumber;

DWORD RvaToFirstByteOfCode;

DWORD RvaToLastByteOfCode;

DWORD RvaToFirstByteOfData;

DWORD RvaToLastByteOfData:

} IMAGE_COFF_SYMBOLS_HEADER,
*PIMAGE_COFF_SYMBOLS_HEADER;
```

---

Рассмотрим подробнее поля структуры IMAGE_COFF_SYMBOLS_HEADER.

*DWORD NumberOfSymbols*

Количество символов в COFF-таблице символов. Данное поле содержит такое же значение, как и поле IMAGE_FILE_HEADER.NumberOfSymbols.

*DWORD LvaToFirstSymbol*

Байтовое смещение COFF-таблицы символов по отношению к началу рассматриваемой структуры. Прибавление этой величины к RVA структуры IMAGE_COFF_SYMBOLS_HEADER даст результат, совпадающий со значением поля IMAGE_FILE_HEADER.PointerToSymbolTable.

*DWORD NumberOfLinenumbers*

Количество элементов в таблице номеров строк.

---

LineNumbers

SymIndex: C ( _DuimpDebugDirectory)

Addr: 016A9 Line: 0008

Addr: 016B5 Line: 0009

Addr: 016BF Line: 000A

Addr: 016C4 Line: 000E

//Остальные номера строк для функции опущены...

SyьIndex: 13 ( _GetResourceTypeName)

Addr: 0184A Line: 0001

Addr: 01854 Line: 0002

Addr: 0186F Line: 0003

Addr; 01874 Line: 0004

// Остальные номера строк для функции опущены...

SymIndex: 1A (_GetResourceNameFromId)

Addr: 01897 Line: 0004

Addr: 018A1 Line: 0006

Addr: 018B6 Line: 0007

Addr: 018BB Line: 000A

// Остальные номера строк опущены...

---

Типичный пример информации из таблицы номеров строк в EXE-файле

*DWORD LvaToFirstLinenumber*

Байтовое смещение COFF-таблицы номеров строк по отношению к началу рассматриваемой структуры.

*DWORD RvaToFirstByteOfCode*

RVA первого байта исполняемого программного кода в отображении. Это поле обычно содержит значение равное RVA секции .text. Это значение также можно найти, просматривая таблицу секций исполняемого файла.

*DWORD RvaToLastByteOfCode*

RVA последнего байта исполняемого программного кода в отображении. Если есть только одна программная секция (.text), то данное поле будет равно RVA этой секции плюс размер ее исходных данных. Это значение также можно найти, просматривая таблицу секций исполняемого файла.

*DWORD RvaToFirstByteOfData*

RVA первого байта данных в отображении. Значение этого поля обычно равно RVA секции .bss.

*DWORD RvaToLastByteOfData*

RVA последнего байта доступных программе данных в отображении. Область, охватываемая полями FirstByteOtData и LastByteOfData, может перекрывать несколько секций (например, .bss, .rdata и .data).

# COFF-таблица номеров строк

COFF-таблица номеров строк, на которую указывает структура IMAGE_COFF_SYMBOLS_HEADER, является очень простой - это просто массив структур IMAGE_LINENUMBER. Каждая структура ставит в соответствие одной строке программного кода источника ее RVA в исполняемом отображении. Ниже показан образец таблицы номеров строк, выведенной программой PEDUMP. Структура IMAGE_LINENUMBER имеет два поля - объединение и слово.

---

*union*

*{*

*DWORD SymbolTableIndex*

*DWORD VirtualAddress*

*} Type*

---

Если поле Linenumber (см. ниже) ненулевое, то его следует трактовать как RVA строки программного кода. Если поле Linenumber равно нулю, то данное поле содержит индекс в таблице символов, Символьная запись, на которую ссылается этот индекс, обозначает функцию. Вес записи номеров строк для этой функции следуют вслед за этой специальной записью. Из рассмотрения вывода программы PEDUMP видно, что таблица номеров строк состоит из записи индекса таблицы символов, за которой идут обычные записи номеров строк, а после них - другая запись индекса таблицы символов и т. д.

*WORD Linenumber*

Содержит номер строки относительно начала функции. Это поле не является номером строки в файле. Чтобы перевести его в удобный для использования помер строки в файле, следует найти в таблице символов номер начальной строки соответствующей функции. Соответствующая функция - это функция, имеющая 0 в данном поле в самой последней записи номера строки.

Если необходим доступ только к номерам строк данной программной секции, то следует искать лишь соответствующий диапазон элементов, относящихся к номерам строк, в таблице секций. Структура IMAGE_SECTION_HEADER данной секции содержит файловое смещение и счетчик номеров своих строк внутри таблицы. Объектные файлы COFF-формата тоже содержат информацию о номерах строк в формате, который был только что описан. В объектных файлах отсутствует структура IMAGE_COFF_SYMBOLS_HEADER, поэтому пользователю придется искать записи номеров строк с помощью структур IMAGE_SECTION_HEADER.

# Различия между PE-файлами и обьектными COFF-файлами

Схожесть двух файловых форматов не случайна. Ее цель - максимально упростить работу компоновщика. Теоретически создание EXE-файла из одного объектного файла должно сводиться к вставке нескольких таблиц и изменению парочки файловых смещений в отображении. Имея это в виду можно представлять себе объектный COFF-файл как зародыш PE-файла. Отсутствуют или отличаются лишь несколько деталей, вот они:

Объектные COFF-файлы начинаются сразу с IMAGE_FILE_HEADER. Перед заголовкам нет части кода DOS, и нет сигнатуры PE перед IMAGE_FILE_HEADER.

В объектных файлах отсутствует IMAGE_OPTIONAL_HEADER. В PE-файлах эта структура следует сразу за IMAGE_FILE_HEADER. Интересно отметить, что некоторые объектные файлы внутри файлов COFF LIB все-таки содержат IMAGE_OPTIONAL_HEADER.

В объектных файлах нет базовых поправок. Вместо этого они имеют привязки, основанные на таблице символов. Информация о поправках

COFF-файлов, весьма запутанна, и поэтому не описана. PointerToRelocations и NumberOfRelocations в строках таблицы секций указывают на поправки для каждой секции. Поправки представляют собой массив структур IMAGE_RELOCATION, определенный в файле WINNT.H.

Информация CodeView в объектном файле хранится в двух секциях - .debug$S и .debug$T.

Компоновщик, обрабатывая объектные файлы, не помещает эти секции в PE-файл.

Вместо этого он собирает все эти секции и создает единую таблицу символов, которая хранится в конце файла. Формально таблица символов не является секцией (т.е. в таблице секций PE-файла нет элемента, соответствующего ей).

# Описание меню

Подменю "Файл" содержит следующие команды:

**"Новый проект"** создать новый проект на основе существующего COFF-файла;

**"Открыть проект"** открыть существующий проект;

**"Сохранить проект"** сохранить текущий проект;

**"Сохранить проект как"** сохранить текущий проект под другим именем;

**"Выход"** завершить работу с программой

Подменю "Вид" содержит следующие команды:

**"Панель управления"** показать/спрятать боковую панель управления;

**"Протокол ошибок"** показать/спрятать протокол ошибок.

Подменю "Компоновщик" содержит следующие команды:

**"Компоновать PE-файл"** проверить проект на наличие ошибок и создать PE-файл;

**"Выполнить PE-файл"** проверить проект на наличие ошибок, создать PE-файл и выполнить его.

Подменю "Окно" содержит команды управления рабочими окнами проекта.

Подменю "Справка" содержит следующие команды:

**"Вызов справки"** вызвать справочную систему;

**"О программе"** J

# Описание интерфейса

Все основные действия при выполнении лабораторной работы выполняются с помощью панели управления, которая становиться доступной только после создания нового проекта или открытия существующего. Панель управления состоит из четырех групп. Содержимое первых двух фиксировано, содержание последних двух может меняться в процессе работы. Структура панели управления представлена ниже.

**"COFF-файл"**

Содержит информацию об объектном файле: структуру и дампы всех его секций. Вся информация в этой группе доступна только для чтения.

**"Заголовок PE-файла"**

В этой группе находятся команды позволяющие изменять содержимое конкретных частей заголовка PE-файла.

**"Таблица секций"**

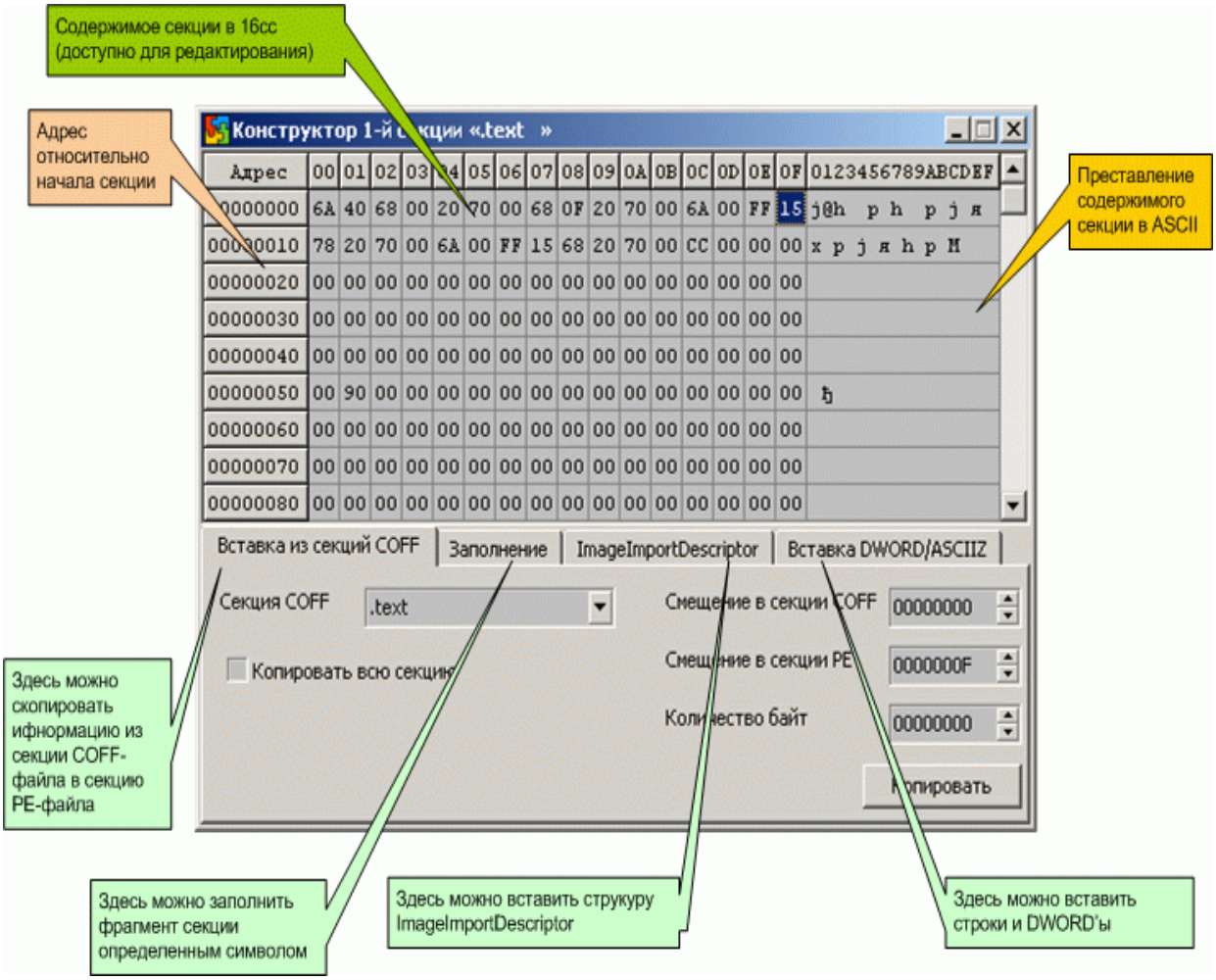
Здесь расположены команды для изменения заголовков отдельных секций PE-файла.

**"Секции"**

В этой группе собраны команды для редактирования содержимого секции.

При выборе любой команды в панели управления открывается окно с редактором соответствующего типа.

Работы с редакторами осуществляется стандартными способами, принятыми в среде Windows.




Окно редактора секций

## Замечания

1. **Все числа в лабораторной установке представлены в 16 с.с.(Hex).**

2. Для создания секций нужно установить их количество в поле *NumberOfSections* файлового заголовка.

3. Если в окне есть кнопка "Применить", то для сохранения изменений необходимо нажать эту кнопку. В противном случае изменения сохраняются автоматически.

4. Программа не обнаруживает ошибки связанные с неправильным заполнением секций PE.

P.S. Авторы заранее приносят свои извинения за возможные недочеты и ошибки.

## Пример выполнения программы:

| DOS программа | Windows программа |
|---|---|
| | #define STD_OUTPUT_HANDLE -11UL |
| | #define hello "hello, world" |
| | __declspec(dllimport) unsigned long __stdcall GetStdHandle(unsigned long |
| #include <stdio.h> | __declspec(dllimport) unsigned long __stdcall WriteConsoleA(unsig hConsoleOutput, |
| int main(void) | const void *buffer, unsigned long chrs, unsigned long *written, unsigned l |
| { | ); |
| puts(hello,world); | static unsigned long written; |
| return 0; | void startup(void) |
| } | { |
| | WriteConsoleA(GetStdHandle(STD_OUTPUT_HANDLE),hello,sizeof(hello)-1,& return; |
| | } |

А вот как выглядит объектный модуль на ассемблере: startup:

; параметры для WriteConsole()

6A 00 push 0x00000000

68 ?? ?? ?? ?? push offset _written

6A 0D push 0x0000000d

68 ?? ?? ?? ?? push offset hello

; параметры для GetStdHandle()

6A F5 push 0xfffffff5

2E FF 15 ?? ?? ?? ?? call dword ptr cs:__imp__GetStdHandle@4

; результат - последний параметр для WriteConsole()

50 push eax

2E FF 15 ?? ?? ?? ?? call dword ptr cs:__imp__WriteConsoleA@20

C3 ret

hello:

68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A "hello, world"

_written:

00 00 00 00

Функции WriteConsoleA() и GetStdHandle() находятся в библиотеке kernel32.dll.

Итак, приступим к созданию исполняемого файла. Знаки (?) будут рассмотрены дальше.

Первым идет DOS-заголовок (заглушка), начинающийся со смещения 0x0 и занимающий 0x40 байт:

00 | 4d 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00

10 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

20 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

30 | 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00

Не трудно заметить, что это не настоящая DOS программа. Это всего лишь заголовок с сигнатурой MZ и полем e_lfanew установленным на конец заголовка. Заголовок не содержит кода и не может быть запущен в DOS.

За DOS-заголовком идет сигнатура, начинающаяся со смещения 0x40 и занимающая 0x4 байта

50 45 00 00

За сигнатурой находится файловый заголовок.

(Смещение : 0x44; Размер : 0x14)

| Поле | Значение | Комментарии |
|---|---|---|
| Machine | 4c 01 | i386 |
| NumberOfSections | 02 00 | Код и данные |
| TimeDateStamp | 00 00 00 00 | Кому это нужно? |
| PointerToSymbolTable | 00 00 00 00 | Не используется |
| NumberOfSymbols | 00 00 00 00 | Не используется |
| SizeOfOptionalHeader | e0 00 | Константа |
| Characteristics | 02 01 | Выполняется на 32-битной машине |

После файлового заголовка идет дополнительный заголовок.

(Смещение : 0x58; Размер : 0x60)

| Поле | Значение | Комментарии |
|---|---|---|
| Magic | 0b 01 | Константа |
| MajorLinkerVersion | 00 | Версия 0.0 |
| MinorLinkerVersion | 00 | |
| SizeOfCode | 20 00 00 00 | 32 байта кода |
| SizeOfInitializedData | ?? ?? ?? ?? | Предстоит выяснить |
| SizeOfUninitializedData | 00 00 00 00 | У нас нет BSS |
| AddressOfEntryPoint | ?? ?? ?? ?? | Предстоит выяснить |
| BaseOfCode | ?? ?? ?? ?? | Предстоит выяснить |
| BaseOfData | ?? ?? ?? ?? | Предстоит выяснить |
| ImageBase | 00 00 10 00 | 1 MB, выбирается произвольно |
| SectionAlignment | 20 00 00 00 | 32-байтовое выравнивание |
| FileAlignment | 20 00 00 00 | 32-байтовое выравнивание |
| MajorOperatingSystemVersion | 04 00 | NT 4.0 |
| MinorOperatingSystemVersion | 00 00 | |
| MajorImageVersion | 00 00 | Версия 0.0 |
| MinorImageVersion | 00 00 | |
| MajorSubsystemVersion | 04 00 | Win32 4.0 |
| MinorSubsystemVersion | 00 00 | |
| Win32VersionValue | 00 00 00 00 | Не используется |
| SizeOfImage | ?? ?? ?? ?? | Предстоит выяснить |
| SizeOfHeaders | ?? ?? ?? ?? | Предстоит выяснить |
| CheckSum | 00 00 00 00 | Только для драйверов устройств |
| Subsystem | 03 00 | Консольное приложение |
| DllCharacteristics | 00 00 | Не используется (устарело) |
| SizeOfStackReserve | 00 00 10 00 | 1 Мб стек |
| SizeOfStackCommit | 00 10 00 00 | Выделять по 4 Кб |
| SizeOfHeapReserve | 00 00 10 00 | 1 Мб куча (heap) |
| SizeOfHeapCommit | 00 10 00 00 | Выделять по 4 Кб |
| LoaderFlags | 00 00 00 00 | Не используется (устарело) |
| NumberOfRvaAndSizes | 10 00 00 00 | Константа |

Известно, что в файле будут 2 секции. Кодовая секция и секция для всего остального (данные, константы, импорт). Файл не будет содержать базовых поправок и ресурсов.

Настраиваем директории данных.

(Смещение : 0xB8; Размер : 0x80)

| Значение | Директория |
|---|---|
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_EXPORT (0) |
| ?? ?? ?? ?? ?? ?? ?? ?? | IMAGE_DIRECTORY_ENTRY_IMPORT (1) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_RESOURCE (2) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_EXCEPTION (3) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_SECURITY (4) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_BASERELOC (5) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_DEBUG (6) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_TLS (9) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11) |
| 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_IAT (12) |
| 00 00 00 00 00 00 00 00 | (13) |
| 00 00 00 00 00 00 00 00 | (14) |
| 00 00 00 00 00 00 00 00 | (15) |

В данном примере используется только директория импорта.

Далее находится таблица секций. Сначала создается кодовая секция, которая будет содержать код программы. Размер секции 32 байта.

(Смещение : 0x138; Размер : 0x28)

| Поле | Значение | Комментарии |
|---|---|---|
| Name | 2e 63 6f 64 65 00 00 00 | «.code» |
| VirtualSize | 00 00 00 00 | Не используется |
| VirtualAddress | ?? ?? ?? ?? | Предстоит выяснить |
| SizeOfRawData | 20 00 00 00 | Размер кода |
| PointerToRawData | ?? ?? ?? ?? | Предстоит выяснить |
| PointerToRelocations | 00 00 00 00 | Не используется |
| PointerToLinenumbers | 00 00 00 00 | Не используется |
| NumberOfRelocations | 00 00 | Не используется |
| NumberOfLinenumbers | 00 00 | Не используется |
| Characteristics | 20 00 00 60 | Содержит код; Выполнимая: Для чтения |

Вторая секция содержит данные.

(Смещение : 0x160; Размер : 0x28)

| Поле | Значение | Комментарии |
|---|---|---|
| Name | 2e 64 61 74 61 00 00 00 | «.data» |
| VirtualSize | 00 00 00 00 | Не используется |
| VirtualAddress | ?? ?? ?? ?? | Предстоит выяснить |
| SizeOfRawData | 20 00 00 00 | Размер кода |
| PointerToRawData | ?? ?? ?? ?? | Предстоит выяснить |
| PointerToRelocations | 00 00 00 00 | Не используется |
| PointerToLinenumbers | 00 00 00 00 | Не используется |
| NumberOfRelocations | 00 00 | Не используется |
| NumberOfLinenumbers | 00 00 | Не используется |
| Characteristics | 40 00 00 c0 | Иниц.; Для чтения; Для записи; |

Секция должна быть выровнена на 32 байта (поле в дополнительном заголовке) , т.е. необходимо заполнить нулями пространство до 0x1a0.

00 00 00 00 00 00 ; Выравнивание

00 00 00 00 00 00

00 00 00 00 00 00

00 00 00 00 00 00

Теперь сама секция «.code».

(Смещение : 0x1a0; Размер : 0x20)

6A 00 ; push 0x00000000

68 ?? ?? ?? ?? ; push offset _written

6A 0D ; push 0x0000000d

68 ?? ?? ?? ?? ; push offset hello_string

6A F5 ; push 0xfffffff5

2E FF 15 ?? ?? ?? ?? ; call dword ptr cs:__imp__GetStdHandle@4

50 ; push eax

2E FF 15 ?? ?? ?? ?? ; call dword ptr cs:__imp__WriteConsoleA@20

C3 ; ret

Поскольку размер предыдущей секции кратен 32, её выравнивать не нужно. Сразу за секцией «.code» идет секция «.data».

(Смещение : 0x1c0)

68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A ; "hello, world"

00 00 00 ; Выравнивание для _written

00 00 00 00 ; _written

Теперь необходимо настроить директорию импорта. Она импортирует 2 функции из "kernel32.dll", и непосредственно следует за «.data»

Для начала выравниваем секцию на 32 байта:

00 00 00 00 00 00 00 00 00 00 00 00 ; Выравнивание

IMAGE_IMPORT_DESCRIPTOR.

(Смещение : 0x1e0)

| Поле | Значение | Комментарии |
|---|---|---|
| OriginalFirstThunk | ?? ?? ?? ?? | Предстоит выяснить |
| TimeDateStamp | 00 00 00 00 | Не используется |
| ForwarderChain | FF FF FF FF | no forwarders |
| Name | ?? ?? ?? ?? | Предстоит выяснить |
| FirstThunk | ?? ?? ?? ?? | Предстоит выяснить |

Необходимо завершить директорию импорта нулевым дескриптором (Смещение :0x1f4):

| Поле | Значение | Комментарии |
|---|---|---|
| OriginalFirstThunk | 00 00 00 00 | Завершить |
| TimeDateStamp | 00 00 00 00 | |

| | |
|---|---|
| ForwarderChain | 00 00 00 00 |
| Name | 00 00 00 00 |
| FirstThunk | 00 00 00 00 |

Далее следует заполнить секцию импорта. Для заполнения секции импорта необходимо указать: имена используемых DLL, массивы IMAGE_THUNK_DATA (OriginalFirstThunk и FirstThunk) и имена функций (массивы IMAGE_IMPORT_BY_NAME).

Имя DLL, завершающий 0.

(Смещение : 0x208)

6b 65 72 6e 65 6c 33 32 2e 64 6c 6c 00 ; "kernel32.dll"

00 00 00 ; выравнивание

Массив элементов IMAGE_THUNK_DATA (OriginalFirstThunk): (Смещение : 0x218)

AddressOfData ?? ?? ?? ?? ; RVA имени функции "WriteConsoleA"

AddressOfData ?? ?? ?? ?? ; RVA имени функции "GetStdHandle"

00 00 00 00 ; выравниваем !

Массив элементов IMAGE_THUNK_DATA (FirstThunk): (Смещение : 0x224)

(__imp__WriteConsoleA@20, смещение 0x224)

AddressOfData ?? ?? ?? ?? ; RVA имени функции "WriteConsoleA"

(__imp__GetStdHandle@4, смещение 0x228)

AddressOfData ?? ?? ?? ?? ; RVA имени функции "GetStdHandle"

00 00 00 00 ; выравниваем и завершаем

Указываем имена функций (задаем массивы IMAGE_IMPORT_BY_NAME).

(Смещение : 0x230)

01 00 ; Подсказка (Hint)

57 72 69 74 65 43 6f 6e 73 6f 6c 65 41 00 ; "WriteConsoleA"

02 00 ; Подсказка (Hint)

47 65 74 53 74 64 48 61 6e 64 6c 65 00 ; "GetStdHandle"

Вот и все! Остаётся только выровнять секцию до смещения 0x260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; выравнивание

Так как все смещения известны, то можно заполнить все поля. Ниже представлена структура готового к выполнению PE-файла.

DOS-заголовок.

(Смещение 0x0)

000 | 4d 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00

010 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

020 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

030 | 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00

Сигнатура.

(Смещение 0x40)

040 | 50 45 00 00

Файловый заголовок.

(Смещение 0x44)

| Смещение | Поле | Значение | Комментарии |
|---|---|---|---|
| 044 | Machine | 4c 01 | i386 |
| 046 | NumberOfSections | 02 00 | Код и данные |
| 048 | TimeDateStamp | 00 00 00 00 | Кому это нужно? |
| 04C | PointerToSymbolTable | 00 00 00 00 | Не используется |
| 050 | NumberOfSymbols | 00 00 00 00 | Не используется |
| 054 | SizeOfOptionalHeader | e0 00 | Константа |
| 056 | Characteristics | 02 01 | Выполняется на 32-битной машине |

Дополнительный заголовок.

(Смещение 0x58)

| Смещение | Поле | Значение | Комментарии |
|---|---|---|---|
| 058 | Magic | 0b 01 | Константа |
| 05A | MajorLinkerVersion | 00 | Версия 0.0 |
| 05B | MinorLinkerVersion | 00 | |
| 05C | SizeOfCode | 20 00 00 00 | 32 байта кода |
| 060 | SizeOfInitializedData | a0 00 00 00 | Размер секции данных 160 байт |
| 064 | SizeOfUninitializedData | 00 00 00 00 | У нас нет BSS |
| 068 | AddressOfEntryPoint | a0 01 00 00 | Начало кодовой секции |
| 06C | BaseOfCode | a0 01 00 00 | RVA кодовой секции |
| 070 | BaseOfData | c0 01 00 00 | RVA секции данных |
| 074 | ImageBase | 00 00 10 00 | 1 MB, выбирается произвольно |
| 078 | SectionAlignment | 20 00 00 00 | 32-байтовое выравнивание |
| 07C | FileAlignment | 20 00 00 00 | 32-байтовое выравнивание |
| 080 | MajorOperatingSystemVersion | 04 00 | NT 4.0 |
| 082 | MinorOperatingSystemVersion | 00 00 | |
| 084 | MajorImageVersion | 00 00 | Версия 0.0 |
| 086 | MinorImageVersion | 00 00 | |
| 088 | MajorSubsystemVersion | 04 00 | Win32 4.0 |
| 08A | MinorSubsystemVersion | 00 00 | |
| 08C | Win32VersionValue | 00 00 00 00 | Не используется |
| 090 | SizeOfImage | c0 00 00 00 | Суммарный размер всех секций |
| 094 | SizeOfHeaders | a0 01 00 00 | Смещение первой секции |
| 098 | CheckSum | 00 00 00 00 | Только для драйверов устройств |
| 09C | Subsystem | 03 00 | Консольное приложение |
| 09E | DllCharacteristics | 00 00 | Не используется (устарело) |
| 0A0 | SizeOfStackReserve | 00 00 10 00 | 1 Мб стек |
| 0A4 | SizeOfStackCommit | 00 10 00 | Выделять по 4 |

| | | | |
|---|---|---|---|
| | | 00 | Кб |
| 0A8 | SizeOfHeapReserve | 00 00 10 00 | 1 Мб куча (heap) |
| 0AC | SizeOfHeapCommit | 00 10 00 00 | Выделять по 4 Кб |
| 0B0 | LoaderFlags | 00 00 00 00 | Не используется (устарело) |
| 0B4 | NumberOfRvaAndSizes | 10 00 00 00 | Константа |

Директория данных.

(Смещение 0xB8)

| Смещение | Значение | Директория |
|---|---|---|
| 0B8 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_EXPORT (0) |
| 0C0 | e0 01 00 00 6f 00 00 00 | IMAGE_DIRECTORY_ENTRY_IMPORT (1) |
| 0C8 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_RESOURCE (2) |
| 0D0 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_EXCEPTION (3) |
| 0D8 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_SECURITY (4) |
| 0E0 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_BASERELOC (5) |
| 0E8 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_DEBUG (6) |
| 0F0 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7) |
| 0F8 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8) |
| 100 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_TLS (9) |
| 108 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10) |

| | | |
|---|---|---|
| 110 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11) |
| 118 | 00 00 00 00 00 00 00 00 | IMAGE_DIRECTORY_ENTRY_IAT (12) |
| 120 | 00 00 00 00 00 00 00 00 | (13) |
| 128 | 00 00 00 00 00 00 00 00 | (14) |
| 130 | 00 00 00 00 00 00 00 00 | (15) |

Таблица секций (секция кода).

(Смещение 0x138)

| Смещение | Поле | Значение | Комментарии |
|---|---|---|---|
| 138 | Name | 2e 63 6f 64 65 00 00 00 | «.code» |
| 140 | VirtualSize | 00 00 00 00 | Не используется |
| 144 | VirtualAddress | a0 01 00 00 | RVA кодовой секции |
| 148 | SizeOfRawData | 20 00 00 00 | Размер кода |
| 14C | PointerToRawData | a0 01 00 00 | Файловое смещение кодовой секции |
| 150 | PointerToRelocations | 00 00 00 00 | Не используется |
| 154 | PointerToLinenumbers | 00 00 00 00 | Не используется |
| 158 | NumberOfRelocations | 00 00 | Не используется |
| 15A | NumberOfLinenumbers | 00 00 | Не используется |
| 15C | Characteristics | 20 00 00 60 | Содержит код; Выполнимая: Для чтения |

Таблица секций (секция данных).

(Смещение 0x160)

| Смещение | Поле | Значение | Коментарии |
|---|---|---|---|
| 160 | Name | 2e 64 61 74 | «.data» |

| | | | |
|---|---|---|---|
| | | 61 00 00 00 | |
| 168 | VirtualSize | 00 00 00 00 | Не используется |
| 16C | VirtualAddress | c0 01 00 00 | RVA секции данных |
| 170 | SizeOfRawData | 20 00 00 00 | Размер кода |
| 174 | PointerToRawData | c0 01 00 00 | Файловое смещение секции данных |
| 178 | PointerToRelocations | 00 00 00 00 | Не используется |
| 17C | PointerToLinenumbers | 00 00 00 00 | Не используется |
| 180 | NumberOfRelocations | 00 00 | Не используется |
| 182 | NumberOfLinenumbers | 00 00 | Не используется |
| 184 | Characteristics | 40 00 00 c0 | Иниц.; Для чтения; Для записи; |

Выравнивание

188 | 00 00 00 00 00 00

18E | 00 00 00 00 00 00

194 | 00 00 00 00 00 00

19A | 00 00 00 00 00 00

Секция кода.

(Смещение 0x1a0)

1A0 | 6A 00 ; push 0x00000000

1A2 | 68 d0 01 10 00 ; push offset _written

1A7 | 6A 0D ; push 0x0000000d

1A9 | 68 c0 01 10 00 ; push offset hello_string

1AE | 6A F5 ; push 0xfffffff5

1B0 | 2E FF 15 28 02 10 00 ; call dword ptr cs:__imp__GetStdHandle@4

1B7 | 50 ; push eax

1B8 | 2E FF 15 24 02 10 00 ; call dword ptr cs:__imp__WriteConsoleA@20

1BF | C3 ; ret

Секция данных.

(Смещение 0x1c0)

1C0 | 68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A ; "hello, world"

1CD | 00 00 00 ; выравнивание

1D0 | 00 00 00 00 ; _written

Выравнивание

1D4 | 00 00 00 00 00 00 00 00 00 00 00 00 ; выравнивание

IMAGE_IMPORT_DESCRIPTOR.

(Смещение 0x1e0)

| Смещение | Поле | Значение | Комментарии |
|---|---|---|---|
| 1E0 | OriginalFirstThunk | 18 02 00 00 | RVA массива OriginalFirstThunk |
| 1E4 | TimeDateStamp | 00 00 00 00 | Не используется |
| 1E8 | ForwarderChain | FF FF FF FF | no forwarders |
| 1EC | Name | 08 02 00 00 | RVA имени DLL |
| 1F0 | FirstThunk | 24 02 00 00 | RVA массива FirstThunk |

Нулевой IMAGE_IMPORT_DESCRIPTOR.

(Смещение 0x1f4)

| Поле | Значение | Комментарии |
|---|---|---|
| OriginalFirstThunk | 00 00 00 00 | Завершить |
| TimeDateStamp | 00 00 00 00 | |
| ForwarderChain | 00 00 00 00 | |
| Name | 00 00 00 00 | |
| FirstThunk | 00 00 00 00 | |

Имя DLL.

(Смещение 0x208)

208 | 6b 65 72 6e 65 6c 33 32 2e 64 6c 6c 00 ; "kernel32.dll"

215 | 00 00 00 ; выравнивание на 32-х битовую границу

Массив OriginalFirstThunk.

(Смещение 0x218)

218 | AddressOfData 30 02 00 00 ; RVA to function name "WriteConsoleA"

21C | AddressOfData 40 02 00 00 ; RVA to function name "GetStdHandle"

220 | 00 00 00 00 ; завершить

Массив FirstThunk.

(Смещение 0x224)

224 | AddressOfData 30 02 00 00 ; RVA to function name "WriteConsoleA"

228 | AddressOfData 40 02 00 00 ; RVA to function name "GetStdHandle"

22C | 00 00 00 00 ; завершить

Массив IMAGE_IMPORT_BY_NAME.

(Смещение 0x230)

230 | 01 00 ; Подсказка (Hint)

232 | 57 72 69 74 65 43 6f 6e 73 6f 6c 65 41 00 ; "WriteConsoleA"

Массив IMAGE_IMPORT_BY_NAME.

(Смещение 0x240)

240 | 02 00 ; Подсказка (Hint)

242 | 47 65 74 53 74 64 48 61 6e 64 6c 65 00 ; "GetStdHandle"

Выравнивание

24F | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

25F | 00

Первый не используемый байт: 0x260

Примечание:

Созданное приложение работает в Windows NT, но не работает в Windows 95. Для запуска приложений в Windows 95 необходимо установить выравнивание секции 4 Кб, и файловое выравнивание 512 байт. Для запуска в Windows 95 необходимо увеличит выравнивание и изменить RVA.