**Динамические библиотеки для начинающих**

(статья была опубликована в журнале "Программист")

<http://citforum.ru/book/cook/dll0.shtml>

В наше время Windows-разработчик шагу не может ступить без **динамических библиотек** (**D**ynamic **L**ink **L**ibrary - **DLL**); а перед начинающими программистами, желающими разобраться в предмете, встает масса вопросов:

* 1. как эффективно использовать чужие DLL?
  2. как создать свою собственную?
  3. какие способы загрузки DLL существуют, и чем они отличаются?
  4. как загружать ресурсы из DLL?

Обо всем этом (и многом другом) рассказывает настоящая глава. Материал рассчитан на пользователей **Microsoft Visual C++**, а поклонникам других языков и компиляторов придется разбираться с ключами компиляции приведенных примеров самостоятельно.

**Создание собственной DLL**

С точки зрения программиста - DLL представляет собой библиотеку функций (ресурсов), которыми может пользоваться любой процесс, загрузивший эту библиотеку. Сама загрузка, кстати, отнимает время и увеличивает расход потребляемой приложением памяти; поэтому бездумное дробление одного приложения на множество DLL ничего хорошего не принесет.

Другое дело - если какие-то функции используются несколькими приложениями. Тогда, поместив их в одну DLL, мы избавимся от дублирования кода и сократим общий объем приложений - и на диске, и в оперативной памяти. Можно выносить в DLL и редко используемые функции отдельного приложения; например, немногие пользователи текстового редактора используют в документах формулы и диаграммы - так зачем же соответствующим функциям впустую "отъедать" память?

Загрузившему DLL процессу доступны не все ее функции, а лишь явно предоставляемые самой DLL для "внешнего мира" - т. н. ***экспортируемые***. Функции, предназначенные сугубо для "внутреннего" пользования, экспортировать бессмысленно (хотя и не запрещено). Чем больше функций экспортирует DLL - тем медленнее она загружается; поэтому к проектированию ***интерфейса*** (способа взаимодействия DLL с вызывающим кодом) следует отнестись повнимательнее. Хороший интерфейс интуитивно понятен программисту, немногословен и элегантен: как говорится, ни добавить, ни отнять. Строгих рекомендаций на этот счет дать невозможно - умение приходит с опытом

Для экспортирования функции из DLL - перед ее описанием следует указать ключевое слово ***\_\_declspec(dllexport)***, как показано в следующем примере:

// myfirstdll.c

#include <stdio.h>

// Ключевое слово \_\_declspec(dllexport)

// делает функцию экспортируемой

**\_\_declspec(dllexport)** void Demo(char \*str)

{

// Выводим на экран переданную функции Demo строку

printf(str);

}

**Листинг 10 Демонстрация экспорта функции из DLL**

Для компиляции этого примера в режиме командной строки можно запустить компилятор Microsoft Visual Studio: "*cl.exe myfirstdll.c /LD*". Ключ "/LD" указывает линкеру, что требуется получить именно DLL.

Для сборки DLL из интегрированной оболочки Microsoft Visual Studio - при создании нового проекта нужно выбрать пункт "*Win32 Dynamics Link Library*", затем "*An Empty DLL project*"; потом перейти к закладке "*File View*" окна "*Workspace*" - и, выбрав правой клавишей мыши папку "*Source Files*", добавить в проект новый файл ("*Add Files to Folder*"). Компиляция осуществляется как обычно ("Build" ( "Build").

Если все прошло успешно - в текущей директории (или в директории Release\Debug при компиляции из оболочки) появится новый файл - "MyFirstDLL.dll". Давайте заглянем в него через "микроскоп" - утилиту **dumpbin**, входящую в штатную поставку SDK и Microsoft Visual Studio: "*dumpbin /EXPORTS MyFirstDLL.dll*". Ответ программы в несколько сокращенно виде должен выглядеть так:

Section contains the following exports for myfirst.dll

0 characteristics

0.00 version

1 ordinal base

1 number of functions

1 number of names

**1 0 00001000 Demo**

Получилось! Созданная нами DLL действительно экспортирует функцию "Demo" - остается только разобраться, как ее вызывать

**Вызов функций из DLL**

Существует два способа загрузки DLL: с ***явной*** и ***неявной компоновкой***.

При***неявной компоновке*** функции загружаемой DLL добавляются в секцию импорта вызывающего файла. При запуске такого файла загрузчик операционной системы анализирует секцию импорта и подключает все указанные библиотеки. Ввиду своей простоты этот способ пользуется большой популярностью; но простота - простотой, а неявной компоновке присущи определенные недостатки и ограничения:

1. все подключенные DLL загружаются *всегда*, даже если в течение всего сеанса работы программа ни разу не обратится ни к одной из них;
2. если хотя бы одна из требуемых DLL отсутствует (или DLL не экспортирует хотя бы одной требуемой функции) - загрузка исполняемого файла прерывается сообщением "*Dynamic link library could not be found*" (или что-то в этом роде) - даже если отсутствие этой DLL некритично для исполнения программы. Например, текстовой редактор мог бы вполне работать и в минимальной комплектации - без модуля печати, вывода таблиц, графиков, формул и прочих второстепенных компонентов, но если эти DLL загружаются неявной компоновкой - хочешь не хочешь, придется "тянуть" их за собой.
3. поиск DLL происходит в следующем порядке: в каталоге, содержащем вызывающий файл; в текущем каталоге процесса; в системном каталоге %Windows%System%; в основном каталоге %Windows%; в каталогах, указанных в переменной PATH. Задать другой путь поиска невозможно (вернее - возможно, но для этого потребуется вносить изменения в системный реестр, и эти изменения окажут влияние на все процессы, исполняющиеся в системе - что не есть хорошо).

***Явная компоновка*** устраняет все эти недостатки - ценой некоторого усложнения кода. Программисту самому придется позаботиться о загрузке DLL и подключении экспортируемых функций (не забывая при этом о контроле над ошибками, иначе в один прекрасный момент дело кончится зависанием системы). Зато явная компоновка позволяет подгружать DLL по мере необходимости и дает программисту возможность самостоятельно обрабатывать ситуации с отсутствием DLL. Можно пойти и дальше - не задавать имя DLL в программе явно, а сканировать такой-то каталог на предмет наличия динамических библиотек и подключать все найденные к приложению. Именно так работает механизм поддержки plug-in’ов в популярном файл-менеджере FAR (да и не только в нем).

Таким образом, *неявной компоновкой целесообразно пользоваться лишь для подключения загружаемых в каждом сеансе, жизненно необходимых для работы приложения динамических библиотек; во всех остальных случаях - предпочтительнее явная компоновка*.

**Загрузка DLL с неявной компоновкой**

Чтобы вызвать функцию из DLL, ее необходимо объявить в вызывающем коде - либо как ***external*** (т. е. как обычную внешнюю функцию), либо предварить ключевым словом ***\_\_declspec(dllimport)***. Первый способ более популярен, но второй все же предпочтительнее - в этом случае компилятор, поняв, что функция вызывается именно из DLL, сможет соответствующим образом оптимизировать код. Например, функция "Demo" из созданной нами библиотеки - "*MyFirstDll*" вызывается так:

// ImplictDll.c

// Объявляем внешнюю функцию Demo

**\_\_declspec(dllimport**) void Demo(char \*str);

main()

{

// Вызываем функцию Demo из DLL

Demo("Hello, World!\n");

}

**Листинг 11 Демонстрация вызова функции из DLL неявной компоновкой**

Из командной строки данный пример компилируется так: "*cl.exe ImplictDll.c myfirstdll.lib*", где "*myfirstdll.lib*" - имя библиотеки, автоматически сформированной компоновщиком при создании нашей DLL.

Разумеется, "чужие" DLL не всегда поставляются вместе с сопутствующими библиотеками, но их можно легко изготовить самостоятельно! На этот случай предусмотрена специальная утилита ***implib***, поставляемая вместе с компилятором, и вызываемая так: "*implib.exe Имя\_файла \_создаваемой\_библиотеки Имя\_DLL*".

В нашем случае - не будь у нас файла "*MyFirstDLL.lib*", его пришлось бы получить так: "*implib.exe MyFirstDLL.lib MyFirstDLL.dll*". Со всеми стандартными DLL, входящими в состав Windows, эту операцию проделывать не нужно, т.к. необходимые библиотеки распространяются вместе с самим компилятором.

Для подключения библиотеки в интегрированной среде Microsoft Visual Studio - в меню "*Project*" выберите пункт "*Project Settings*", в открывшемся диалоговом окне перейдите к закладке "*Link*" и допишите имя библиотеки в конец строки "*Object/Library Modules*", отделив ее от остальных символом пробела.

Если все прошло успешно, появится новый файл "*ImplictDll.exe*", который, будучи запущенным, горделиво выведет на экран "*Hello, Word!*". Это означает, что наша DLL подключена и успешно работает.

Заглянем внутрь: как это происходит? Запустим "*dumpbin /IMPORTS ImplictDll.exe"* ипосмотрим, что нам сообщит программа:

File Type: EXECUTABLE IMAGE

Section contains the following imports:

**myfirstdll.dll**

404090 Import Address Table

4044C8 Import Name Table

0 time date stamp

0 Index of first forwarder reference

**0 Demo**

KERNEL32.dll

404000 Import Address Table

404438 Import Name Table

0 time date stamp

0 Index of first forwarder reference

19B HeapCreate

2BF VirtualFree

CA GetCommandLineA

174 GetVersion

7D ExitProcess

29E TerminateProcess

F7 GetCurrentProcess

Вот она - "*Myfirstdll.dll*" (в тексте выделена жирным шрифтом), и вот функция "*Demo*", а кроме нее - обнаруживает свое присутствие библиотека *KERNEL32.DLL*– она необходима RTL-коду (***R****un****T****ime****L****ibrary - библиотека времени исполнения*), насильно помещенному компилятором в наше приложение. RTL-код обеспечивает работу с динамической памятью (heap), считывает аргументы командной строки, проверяет версию Windows и многое-многое другое! Отсюда и появляются в таблице импорта функции *HeapCreate*, *GetCommandLine*, *GetVersion* и т.д. Так что - не удивляйтесь, увидев "левый" импорт в своем приложении!

Проследить, как именно происходит загрузка DLL, можно с помощью отладчика. Общепризнанный лидер - это, конечно, SoftIce от NuMega, но для наших экспериментов вполне сойдет и штатный отладчик Microsoft Visual Studio. Откомпилировав нашу вызывающую программу, нажмем <F10> для пошагового прогона приложения

Оппаньки! Не успело еще выполниться ни строчки кода, как в окне "output" отладчика появились следующие строки, свидетельствующие о загрузке внешних DLL: NTDLL.DLL, MyFirstDll.dll и Kernel32.dll. Так и должно быть - при неявной компоновке динамические библиотеки подключаются сразу же при загрузке файла, задолго до выполнения функции main!

Loaded 'C:\WINNT\System32\ntdll.dll', no matching symbolic information found.

Loaded 'F:\ARTICLE\PRG\DLL.files\myfirstdll.dll', no matching symbolic information found.

Loaded 'C:\WINNT\system32\kernel32.dll', no matching symbolic information found.

**Загрузка DLL с явной компоновкой**

Явную загрузку динамических библиотек осуществляет функция HINSTANCE **LoadLibrary**(LPCTSTR lpLibFileName) или ее расширенный аналог HINSTANCE **LoadLibraryEx**(LPCTSTR lpLibFileName, HANDLE hFile, DWORD dwFlags).

Обе они экспортируются из KERNEL32.DLL, следовательно, каждое приложение требует неявной компоновки по крайней мере этой библиотеки. В случае успешной загрузки DLL возвращается линейный адрес библиотеки в памяти. Передав его функции FARPROC **GetProcAddress**(HMODULE hModule, LPCSTR lpProcName) - мы получим указатель на функцию lpProcName, экспортируемую данной DLL. При возникновении ошибки обе функции возвращают NULL. После завершения работы с динамической библиотекой ее следует освободить вызовом функции BOOL **FreeLibrary**(HMODULE hLibModule). Для пояснения приведем код примера с подробными комментариями:

// DynCall.c

#include <stdio.h>

#include <windows.h>

main()

{

// Дескриптор загружаемой dll

HINSTANCE h;

// Объявление указателя на функцию, вызываемой из DLL

// Обратите внимание – имена объявляемой функции и

// функции, вызываемой из DLL, могут и не совпадать,

// т.к. за выбор вызываемой функции отвечает

// GetProcAddress

void (\*DllFunc) (char \*str);

// Загружаем MyFirstDLL

h=LoadLibrary("MyFirstDLL.dll");

// Контроль ошибок – если загрузка прошла успешно,

// функция вернет что-то отличное от нуля

if (!h)

{

printf("Ошибка - не могу найти MyFirstDLL.dll\n");

return;

}

// Вызовом GetProcAddress получаем адрес функции Demo

// и присваиваем его указателю DllFunc с явным

// приведением типов. Это необходимо т.к.

// GetProcAddress возвращает бестиповой far-указатель

DllFunc=(void (\*) (char \*str))

GetProcAddress(h,"Demo");

// Контроль ошибок – если вызов функции GetProcAddress

// завершился успешно, она вернет ненулевой указатель

if (!DllFunc)

{

printf("Ошибка! В MyFirstDLL "

"отсутствует ф-ция Demo\n");

return;

}

// Вызов функции Demo из DLL

DllFunc("Test");

// Выгрузка динамической библиотеки из памяти

FreeLibrary(h);

}

**Листинг 12 Демонстрация вызова функции из DLL явной компоновкой**

Компилировать так: "*cl DynCall.c*" - никаких дополнительных библиотек указывать не нужно (необходимые *kernel32.lib* и *LIBC.lib* компоновщик подключит самостоятельно).

В интегрированной среде Microsoft Visual Studio достаточно щелкнуть мышкой по иконке "*Build*" - и никаких дополнительных настроек!

Для изучения секции импорта только что полученного файла запустим утилиту *Dumpbin*– обратите внимание, что здесь отсутствует всякое упоминание о MyFirstDLL.dll, но обнаруживаются две функции: LoadLibrary и GetProcAddress - которые и загружают нашу библиотеку. Это очень важное обстоятельство - *изучение секции импорта исследуемого файла не всегда позволяет установить полный перечень функций и динамических библиотек, которые использует приложение*. Наличие LoadLibrary и GetProcAddress красноречиво свидетельствует о том, что приложение подгружает какие-то модули во время работы самостоятельно.

Чтобы выяснить какие - запустим его под отладчиком. Сразу же после загрузки исполняемого файла в окне Output появятся строки:

Loaded 'C:\WINNT\System32\ntdll.dll', no matching symbolic information found.

Loaded 'C:\WINNT\system32\kernel32.dll', no matching symbolic information found.

Это опять грузятся обязательные KERNEL32.DLL и NTDLL.DLL (последнее - только под Windows NT/2000). Никакого упоминания о MyFirstDLL.dll еще нет. Пошагово исполняя программу ("*Debug*" ( "*Step Over*"), дождемся выполнения функции LoadLibrary. Тут же в Output-окне появится следующая строка:

Loaded 'F:\ARTICLE\PRG\DLL.files\myfirstdll.dll', no matching symbolic information found.

Наша динамическая библиотека загрузилась; но не сразу после запуска файла (как это происходило при неявной компоновке), а только когда в ней возникла необходимость!

Если же по каким-то причинам DLL не найдется, или окажется, что в ней отсутствует функция Demo - операционная система не станет "убивать" приложение с "некрологом" критической ошибки, а предоставит программисту возможность действовать самостоятельно. В качестве эксперимента попробуйте удалить (переименовать) MyFirstDLL.dll и посмотрите, что из этого получится.

**Выгрузка динамических библиотек из памяти**

Когда загруженная динамическая библиотека больше не нужна - ее можно освободить, вызвав функцию *BOOL* ***FreeLibrary****(HMODULE hLibModule) и* передав ей дескриптор библиотеки, ранее возвращенный функцией LoadLibrary. Обратите внимание - DLL можно именно ***освободить***, но ***не выгрузить***! Выгрузка DLL из памяти ***не гарантируется***, даже если работу с ней завершили все ранее загрузившие ее процессы.

Задержка выгрузки предусмотрена специально - на тот случай, если эта же DLL через некоторое время вновь понадобится какому-то процессу. Такой трюк оптимизирует работу часто используемых динамических библиотек, но плохо подходит для редко используемых DLL, загружаемых лишь однажды на короткое время. Никаких документированных способов насильно выгрузить динамическую библиотеку из памяти нет; а те, что есть - работают с ядром на низком уровне и не могут похвастаться переносимостью. Поэтому здесь мы их рассматривать не будем. К тому же - тактика освобождения и выгрузки DLL по-разному реализована в каждой версии Windows: Microsoft, стремясь подобрать наилучшую стратегию, непрерывно изменяет этот алгоритм; а потому и отказывается его документировать.

Нельзя не обратить внимания на одно очень важное обстоятельство: *динамическая библиотека не владеет никакими ресурсами - ими владеет, независимо от способа компоновки, загрузивший ее процесс*. Динамическая библиотека может открывать файлы, выделять память и т. д., но память не будет автоматически освобождена после вызова FreeLibrary, а файлы не окажутся сами собой закрыты - все это произойдет лишь после завершения процесса, но не раньше! Естественно, если программист сам не освободит все ненужные ресурсы вручную, с помощью функций CloseHandle, FreeMemory и подобных им.

Если функция FreeLibrary пропущена, DLL освобождается (но не факт, что выгружается!) только после завершения вызвавшего процесса. Могут возникнуть сомнения: раз FreeLibrary немедленно не выгружает динамическую библиотеку из памяти, так зачем она вообще нужна? Не лучше ли тогда все пустить на самотек - все равно ведь загруженные DLL будут гарантированно освобождены после завершения процесса? Что ж, доля правды тут есть, и автор сам порой так и поступает; но при недостатке памяти операционная система может беспрепятственно использовать место, занятое освобожденными динамическими библиотеками под что-то полезное - а если DLL еще не освобождены, их придется "скидывать" в файл подкачки, теряя драгоценное время. Поэтому лучше освобождайте DLL сразу же после их использования!

**ООП и DLL**

Динамические библиотеки ничего не знают ни о каком ООП! Они не имеют ни малейшего представления о существовании классов! Все, что умеют DLL - экспортировать одно или несколько имен функций (ресурсов, глобальных переменных), а уж как его использовать - решать программисту или компилятору.

Что ж; испытаем компилятор на "сообразительность", включив в описание класса ключевое слово ***\_\_declspec(dllexport)****–* и посмотрим, что из этого выйдет:

// DLLclass.cpp

#include <stdio.h>

class ***\_\_declspec(dllexport)*** MyDllClass{

public:

Demo(char \*str);

};

MyDllClass::Demo(char \*str)

{

printf(str);

}

**Листинг 13 Демонстрация экспорта класса из DLL**

Откомпилируем этот код как обычную DLL и заглянем в таблицу импорта утилитой dumpbin:

dumpbin /EXPORTS DLLclass.dll

File Type: DLL

Section contains the following exports for DLLclass.dll

0 characteristics

3B1B98E6 time date stamp Mon Jun 04 18:19:18 2001

0.00 version

1 ordinal base

2 number of functions

2 number of names

ordinal hint RVA name

**1 0 00001000 ??4MyDllClass@@QAEAAV0@ABV0@@Z**

**2 1 00001020 ?Demo@MyDllClass@@QAEHPAD@Z**

Таблица импорта явно не пуста - но выглядит странно. Компилятор искалечил имена, чтобы втиснуть в них информацию о классах и аргументах функций без нарушений жестких ограничений, налагаемых стандартом на символы, допустимые в экспортируемых именах (например, запрещается использовать знак двоеточие, скобка и т. д.).

Как же со всем этим работать? Попробуй-ка угадай - во что превратится то или иное имя после компиляции! Впрочем, при неявной компоновке ни о чем гадать не придется, т. к. обо всем позаботится сам компилятор, а от программиста потребуется лишь описать класс, предварив его ключевым словом ***\_\_declspec(dllimport)***:

// DLLClassCall.cpp

#include <stdio.h>

class **\_\_declspec(dllimport)** MyDllClass{

public:

Demo(char \*str);

};

main()

{

MyDllClass zzz;

zzz.Demo("Привет, ООП! А мы тут!");

}

**Листинг 14 Демонстрация импорта класса из DLL неявной компоновкой**

Откомпилируйте пример как обычную программу с неявной компоновкой ("*cl DLLClassCall.cpp DLLClass.lib*") и попробуйте запустить полученный файл. Работает? Никакой разницы с "классическим" Си нет, не правда ли? Вот только как подключить DLL с явной компоновкой? Неужели нельзя запретить компилятору "калечить" имена функций?! Конечно же, можно

**Мангл и как его побороть или импорт классов из DLL явной компоновкой**

Искажение функций Cи ++ компилятором называется по-английски "***mangle***", и среди русскоязычных программистов широко распространена его калька - "манглить". В принципе "манглеж" функций не препятствует их явному вызову посредством GetProcAddress - достаточно лишь знать, как точно называется та или иная функция, что нетрудно выяснить тем же dumpbin. Однако засорение программы подобными "заученными" именами выглядит не очень-то красиво; к тому же - всякий компилятор "манглит" имена по-своему, и непосредственное использование имен приводит к непереносимости программы.

Существует способ обойти искажение имен - для этого необходимо подключить к линкеру специальный DEF-файл, перечисляющий имена, которые не должны измениться. В нашем случае он должен выглядеть так

// DllClass.def:

**EXPORTS**

Demo

**Листинг 15 Отказ от "замангления" имен**

Сперва идет ключевое слово "EXPORTS", за которым следуют одно или несколько "неприкасаемых" имен. Каждое имя начинается с новой строки, и в его конце **не** указывается точка с запятой.

Для подключения DEF-файла при компиляции из командной строки - используйте опцию "*/link /DEF:имя\_файла.def*", например так: "*cl DLLclass.cpp /LD /link /DEF:DLLClass.def*".

Для подключения DEF-файла в интегрированной среде Microsoft Visual Studio - перейдите к закладке "*File View*" окна "*Workspace*" и, щелкнув правой клавишей по папке "*Source Files*", выберите в контекстом меню пункт "*Add Files to Folder*", а затем укажите путь к DEF-файлу. Откомпилируйте проект как обычно: "Build" ( "Build".

Заглянув в таблицу импорта полученного DLL-файла, мы, среди прочей информации, увидим следующее:

1 0 00001000 ??4MyDllClass@@QAEAAV0@ABV0@@Z

**2 1 00001020 Demo**

Теперь имя функции Demo выглядит "как положено". А абракадабра, расположенная строчкой выше - это конструктор класса MyDllClass, который, хоть и не был специально объявлен, все равно экспортируется из динамической библиотеки.

Однако, избавившись от одной проблемы, мы получаем другую - имя функции Demo потеряло всякое представление о классе, которому оно принадлежало; и теперь придется загружать его вручную, повторяя эту операцию для каждого элемента класса. Фактически - придется в вызываемой программе собирать "скелет" класса из "косточек" заново. Но иного способа явной загрузки класса из DLL не существует.

Следующий пример демонстрирует вызов функции *MyDllClass* из динамической библиотеки с явной компоновкой. Обработка ошибок для упрощения опущена. Обратите внимание, как объявляется функция Demo - описания класса в DLL и в вызывающей программе существенно отличаются, поэтому с идеей поместить описания класса в общий для всех include-файл придется расстаться.

// DeMangle.cpp

#include <windows.h>

class MyDllClass{

public:

void (\*Demo) (char \*str);

};

main()

{

HINSTANCE h=LoadLibrary("DllClass.dll");

MyDllClass zzz;

// Внимание! Выполнение конструктора / деструктора

// класса при явной загрузке не происходит

// автоматически и при необходимости эту операцию

// следует выполнить вручную

zzz.Demo=(void (\*) (char \*str))

GetProcAddress(h,"Demo");

zzz.Demo("Test");

}

**Листинг 16 Демонстрация вызова функции MyDllCLass::Demo явной компоновкой**

**Загрузка ресурсов из DLL**

Помимо функций, динамические библиотеки могут содержать и ресурсы - строки, иконки, рисунки, диалоги и т. д. Хранение ресурсов в DLL очень удобно; в частности - при создании приложений с многоязычным интерфейсом: заменив одну DLL на другую, мы заменяем все надписи в программе, скажем, с английского на русский - и, заметьте, без всяких "хирургических вмешательств" в код приложения! Аналогично можно менять иконки, внешний вид диалогов и т. д.

Создание DLL, содержащей ресурсы, ничем не отличается от создания исполняемого приложения с ресурсами; сначала необходимо создать сам файл ресурсов - например, так:

// MyResDll.rc

#pragma code\_page(1251)

STRINGTABLE DISCARDABLE

BEGIN

1 "Hello, World!"

END

**Листинг 17 Создание DLL, содержащей одни лишь ресурсы**

Файл ресурсов надо скомпилировать - "*rc MyResDll.rc*" - и преобразовать линкером в DLL, обязательно указав флаг "*/NOENTRY*", т. к. эта динамическая библиотека содержит исключительно одни ресурсы и ни строки кода: "*link MyRedDll.res /DLL /NOENTRY*".

В Visual Studio это сделать еще проще - достаточно кликнуть по папке "*Resourses*" окна "*File View*" и добавить новый файл ресурса, который затем можно будет модифицировать визуальным редактором по своему усмотрению.

Для загрузки ресурса из DLL - в принципе, можно воспользоваться уже знакомой нам функцией LoadLibray, и передавать возращенный ею дескриптор LoadString или другой функции, работающей с ресурсами. Однако загрузку динамической библиотеки можно значительно ускорить, если "объяснить" системе, что эта DLL не содержит ничего, кроме ресурсов, и нам достаточно лишь спроецировать ее на адресное пространство процесса, а обо всем остальном мы сумеем позаботиться и самостоятельно.

Вот тут-то и пригодится функция LoadLibraryEx: ее первый аргумент, как и у коллеги LoadLibrary, задает имя динамической библиотеки для загрузки, второй - зарезервирован и должен быть равен нулю, а третий, будучи равным ***LOAD\_LIBRARY\_AS\_DATAFILE,*** заставляет функцию делать именно то, что нам нужно - загружать DLL как базу данных (если динамическая библиотека содержит помимо ресурсов еще и код, то загрузка с этим ключом проходит все равно успешно, но функции загруженной DLL не будут доступны - только ресурсы):

// DllLoadRes.c

#include <stdio.h>

#include <windows.h>

main()

{

HINSTANCE h;

char buf[100];

h=LoadLibraryEx("MyResDll.dll",0,

LOAD\_LIBRARY\_AS\_DATAFILE);

LoadString(h,1,&buf[0],99);

printf("%s\n",&buf[0]);

}

**Листинг 18 Демонстрация оптимизированной загрузки DLL, не содержащей ничего кроме ресурсов**

Эта программа компилируются точно так же, как и предыдущие примеры явной компоновки - и после запуска победно выводит на экране "*Hello, Word!*", подтверждая, что ресурс "строка" из динамической библиотеки был успешно загружен! Аналогичным способом можно загружать ресурсы из исполняемых файлов; с этой точки зрения они ничем не отличаются от динамических библиотек.