Работа с библиотеками динамической компоновки (DLL)

Оглавление

С самого рождения (или чуть позже) операционная система Windows использовала библиотеки динамической компоновки DLL (Dynamic Link Library), в которых содержались реализации наиболее часто применяемых функций. Наследники Windows - NT и Windows 95, а также OS/2 - тоже зависят от библиотек DLL в плане обеспечения значительной части их функциональных возможностей.

Рассмотрим ряд аспектов создания и использования библиотек DLL:

- как статически подключать библиотеки DLL;
- как динамически загружать библиотеки DLL;
- как создавать библиотеки DLL;
- как создавать расширения MFC библиотек DLL.

Использование DLL

Практически невозможно создать приложение Windows, в котором не использовались бы библиотеки DLL. В DLL содержатся все функции Win32 API и несчетное количество других функций операционных систем Win32.

Вообще говоря, DLL - это просто наборы функций, собранные в библиотеки. Однако, в отличие от своих статических родственников (файлов . lib), библиотеки DLL не присоединены непосредственно к выполняемым файлам с помощью редактора связей. В выполняемый файл занесена только информация об их местонахождении. В момент выполнения программы загружается вся библиотека целиком. Благодаря этому разные процессы могут пользоваться совместно одними и теми же библиотеками, находящимися в памяти. Такой подход позволяет сократить объем памяти, необходимый для нескольких приложений, использующих много общих библиотек, а также контролировать размеры EXE-файлов.

Однако, если библиотека используется только одним приложением, лучше сделать ее обычной, статической. Конечно, если входящие в ее состав функции будут использоваться только в одной программе, можно просто вставить в нее соответствующий файл с исходным текстом.

Чаще всего проект подключается к DLL статически, или неявно, на этапе компоновки. Загрузкой DLL при выполнении программы управляет операционная система. Однако, DLL можно загрузить и явно, или динамически, в ходе работы приложения. **Библиотеки импортирования**

При статическом подключении DLL имя .lib-файла определяется среди прочих параметров редактора связей в командной строке или на вкладке "Link" диалогового окна "Project Settings" среды Developer Studio. Однако .lib-файл, используемый **при неявном подключении DLL,** - это не обычная статическая библиотека. Такие .lib-файлы называются **библиотеками импортирования** (import libraries). В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла DLL, в котором все и хранится. В результате библиотеки импортирования, как правило, имеют меньший размер, чем DLL-файлы. К способам их создания вернемся позднее. А сейчас рассмотрим другие вопросы, касающиеся неявного подключения динамических библиотек. **Согласование интерфейсов**

При использовании собственных библиотек или библиотек независимых разработчиков придется обратить внимание на согласование вызова функции с ее прототипом.

Если бы мир был совершенен, то программистам не пришлось бы беспокоиться о согласовании интерфейсов функций при подключении библиотек - все они были бы одинаковыми. Однако мир далек от совершенства, и многие большие программы написаны с помощью различных библиотек без C++.

По умолчанию в Visual C++ интерфейсы функций согласуются по правилам C++. Это значит, что параметры заносятся в стек справа налево, вызывающая программа отвечает за их удаление из стека при выходе из функции и расширении ее имени. Расширение имен (name mangling) позволяет редактору связей различать перегруженные функции, т.е. функции с одинаковыми именами, но

разными списками аргументов. Однако в старой библиотеке C функции с расширенными именами отсутствуют.

Хотя все остальные правила вызова функции в C идентичны правилам вызова функции в C++, в библиотеках C имена функций не расширяются. К ним только добавляется впереди символ подчеркивания (_).

Если необходимо подключить библиотеку на C к приложению на C++, все функции из этой библиотеки придется объявить как внешние в формате C:

```
extern "C" int MyOldCFunction(int myParam);
```

Объявления функций библиотеки обычно помещаются в файле заголовка этой библиотеки, хотя заголовки большинства библиотек C не рассчитаны на применение в проектах на C++. В этом случае необходимо создать копию файла заголовка и включить в нее модификатор extern "C" к объявлению всех используемых функций библиотеки. Модификатор extern "C" можно применить и к целому блоку, к которому с помощью директивы #tinclude подключен файл старого заголовка C. Таким образом, вместо модификации каждой функции в отдельности можно обойтись всего тремя строками:

```
extern "C"
{
#include "MyCLib.h"
}
```

В программах для старых версий Windows использовались также соглашения о вызове функций языка PASCAL для функций Windows API. В новых программах следует использовать модификатор winapi, преобразуемый в _stdcall. Хотя это и не стандартный интерфейс функций C или C++, но именно он используется для обращений к функциям Windows API. Однако обычно все это уже учтено в стандартных заголовках Windows.

Загрузка неявно подключаемой DLL

При запуске приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности.

- Каталог, в котором находится EXE-файл.
- Текущий каталог процесса.
- Системный каталог Windows.

Если библиотека DLL не обнаружена, приложение выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Затем процесс отключается.

Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

Динамическая загрузка и выгрузка DLL

Вместо того, чтобы Windows выполняла динамическое связывание с DLL при первой загрузке приложения в оперативную память, можно связать программу с модулем библиотеки во время выполнения программы (при таком способе в процессе создания приложения не нужно использовать библиотеку импорта). В частности, можно определить, какая из библиотек DLL доступна пользователю, или разрешить пользователю выбрать, какая из библиотек будет загружаться. Таким образом можно использовать разные DLL, в которых реализованы одни и те же функции, выполняющие различные действия. Например, приложение, предназначенное для независимой

передачи данных, сможет в ходе выполнения принять решение, загружать ли DLL для протокола TCP/IP или для другого протокола.

Загрузка обычной DLL

Первое, что необходимо сделать при динамической загрузке DLL, - это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции **::LoadLibrary**, имеющей единственный аргумент - имя загружаемого модуля. Соответствующий фрагмент программы должен выглядеть так:

```
HINSTANCE hMyDll;
::
if((hMyDll=::LoadLibrary("MyDLL"))==NULL) { /* не удалось загрузить DLL */ }
else { /* приложение имеет право пользоваться функциями DLL через hMyDll */ }
```

Стандартным расширением файла библиотеки Windows считает .dll, если не указать другое расширение. Если в имени файла указан и путь, то только он будет использоваться для поиска файла. В противном случае Windows будет искать файл по той же схеме, что и в случае неявно подключенных DLL, начиная с каталога, из которого загружается exe-файл, и продолжая в соответствии со значением PATH.

Когда Windows обнаружит файл, его полный путь будет сравнен с путем библиотек DLL, уже загруженных данным процессом. Если обнаружится тождество, вместо загрузки копии приложения возвращется дескриптор уже подключенной библиотеки.

Если файл обнаружен и библиотека успешно загрузилась, функция **::LoadLibrary** возвращает ее дескриптор, который используется для доступа к функциям библиотеки.

Перед тем, как использовать функции библиотеки, необходимо получить их адрес. Для этого сначала следует воспользоваться директивой **typedef** для определения типа указателя на функцию и определить переменую этого нового типа, например:

```
// тип PFN_MyFunction будет объявлять указатель на функцию,
// принимающую указатель на символьный буфер и выдающую значение типа int
typedef int (WINAPI *PFN_MyFunction)(char *);
::
PFN_MyFunction pfnMyFunction;
```

Затем следует получить дескриптор библиотеки, при помощи которого и определить адреса функций, например адрес фунции с именем MyFunction:

```
hMyDll=::LoadLibrary("MyDLL");
pfnMyFunction=(PFN_MyFunction)::GetProcAddress(hMyDll,"MyFunction");
::
int iCode=(*pfnMyFunction)("Hello");
```

Адрес функции определяется при помощи функции **::GetProcAddress**, ей следует передать имя библиотеки и имя функции. Последнее должно передаваться в том виде, в котором эксаортируется из DLL.

Можно также сослаться на функцию по порядковому номеру, по которому она экспортируется (при этом для создания библиотеки должен использоваться def-файл, об этом будет рассказано далее):

```
pfnMyFunction=(PFN_MyFunction)::GetProcAddress(hMyDll,
MAKEINTRESOURCE(1));
```

После завершения работы с библиотекой динамической компоновки, ее можно выгрузить из памяти процесса с помощью функции ::**FreeLibrary**:

```
::FreeLibrary(hMyDll);
```

Загрузка MFC-расширений динамических библиотек

При загрузке MFC-расширений для DLL (подробно о которых рассказывается далее) вместо функций **LoadLibrary**и **FreeLibrary** используются функции **AfxLoadLibrary** и **AfxFreeLibrary**. Последние почти идентичны функциям Win32 API. Они лишь гарантируют дополнительно, что структуры MFC, инициализированные расширением DLL, не были запорчены другими потоками.

Ресурсы DLL

Динамическая загрузка применима и к ресурсам DLL, используемым MFC для загрузки стандартных ресурсов приложения. Для этого сначала необходимо вызвать функцию **LoadLibrary** и разместить DLL в памяти. Затем с помощью функции **AfxSetResourceHandle** нужно подготовить окно программы к приему ресурсов из вновь загруженной библиотеки. В противном случае ресурсы будут загружаться из файлов, подключенных к выполняемому файлу процесса. Такой подход удобен, если нужно использовать различные наборы ресурсов, например для разных языков.

Замечание. С помощью функции **LoadLibrary** можно также загружать в память исполняемые файлы (не запускать их на выполнение!). Дескриптор выполняемого модуля может затем использоваться при обращении к функциям **FindResource** и **LoadResource** для поиска и загрузки ресурсов приложения. Выгружают модули из памяти также при помощи функции **FreeLibrary**.

Пример обычной DLL и способов загрузки

Приведем исходный код динамически подключаемой библиотеки, которая называется MyDLL и содержит одну функцию MyFunction, которая просто выводит сообщение.

Сначала в заголовочном файле определяется макроконтстанта **EXPORT**. Использование этого ключевого слова при определении некоторой функции динамически подключаемой билиотеке позволяет сообщить компоновщику, что эта функция доступна для использования другими программами, в результате чего он заносит ее в библилтеку импорта. Кроме этого, такая функция, точно так же, как и оконная процедура, должна определяться с помощью константы **CALLBACK**:

```
MyDLL.h
#define EXPORT extern "C" __declspec (dllexport)
EXPORT int CALLBACK MyFunction(char *str);
```

Файл библиотеки также несколько отличается от обычных файлов на языке C для Windows. В нем вместо функции **WinMain**имеется функция **DllMain**. Эта функция используется для выполнения инициализации, о чем будет рассказано позже. Для того, чтобы библиотека осталась после ее загрузки в памяти, и можно было вызывать ее функции, необходимо, чтобы ее возвращаемым значением было TRUE:

MyDLL.c

```
#include <windows.h>
#include "MyDLL.h"

int WINAPI DllMain(HINSTANCE hInstance, DWORD fdReason, PVOID pvReserved)
{
return TRUE;
}
EXPORT int CALLBACK MyFunction(char *str)
{
MessageBox(NULL,str,"Function from DLL",MB_OK);
}
```

После трансляции и компоновки этих файлов появятся два файла - MyDLL.dll (сама динамически подключаемая библиотека) и MyDLL.lib (ее библиотека импорта).

Пример неявного поключения DLL приложением

Приведем теперь исходный код простого приложения, которое использует функцию MyFunction из библиотеки MyDLL.dll:

```
#include <windows.h>
#include "MyDLL.h"

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
int iCode=MyFunction("Hello");
return 0;
}
```

Эта программа выглядит как обычная программ для Windows, чем она в сущности и является. Тем не менее, следует обратить внимание, что в исходный ее текст помимо вызова функции MyFunction из DLL-библиотеки включен и заголовочный файл этой библиотеки MyDLL.h. Также необходимо на этапе компоновки приложения подключить к нему библиотеку импорта MyDLL.lib (процесс неявного подключения DLL к исполняемому модулю).

Чрезвычайно важно понимать, что сам код функции MyFunction не включается в файл MyApp.exe. Вместо этого там просто имеется ссылка на файл MyDLL.dll и ссылка на функцию MyFunction, которая находится в этом файле. Файл MyApp.exe требует запуска файла MyDLL.dll.

Заголовочный файл MyDLL.h включен в файл с исходным текстом программы MyApp.c точно так же, как туда включен файл windows.h. Включение библиотеки импорта MyDLL.lib для компоновки аналогично включению туда всех библиотек импорта Windows. Когда програма MyApp.exe работает, она подключается к библиотеке MyDLL.dll точно так же, как ко всем стандартным динамически подключаемым библиотекам Windows.

Пример динамической загрузки DLL приложением

Приведем теперь полностью исходный код простого приложения, которое использует функцию MyFunction из библиотеки MyDLL.dll, используя динамическую загрузку библиотеки:

```
#include <windows.h>
typedef int (WINAPI *PFN_MyFunction)(char *);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
HINSTANCE hMyDll;
PFN_MyFunction pfnMyFunction;
pfnMyFunction=(PFN_MyFunction)GetProcAddress(hMyDll,"MyFunction");
int iCode=(*pfnMyFunction)("Hello");

FreeLibrary(hMyDll);
return 0;
}
```

Создание DLL

Теперь, познакомившись с принципами работы библиотек DLL в приложениях, рассмотрим способы их создания. При разработке приложении функции, к которым обращается несколько процессов, желательно размещать в DLL. Это позволяет более рационально использовать память в Windows.

Проще всего создать новый проект DLL с помощью мастера AppWizard, который автоматически выполняет многие операции. Для простых DLL, таких как рассмотренные в этой главе, необходимо выбрать тип проекта Win32 Dynamic-Link Library. Новому проекту будут присвоены все необходимые параметры для создания библиотеки DLL. Файлы исходных текстов придется добавлять к проекту вручную.

Если же планируется в полной мере использовать функциональные возможности MFC, такие как документы и представления, или намерены создать сервер автоматизации OLE, лучше выбрать тип проекта MFC AppWizard (dll). В этом случае, помимо присвоения проекту параметров для подключения динамических библиотек, мастер проделает некоторую дополнительную работу. В проект будут добавлены необходимые ссылки на библиотеки MFC и файлы исходных текстов, содержащие описание и реализацию в библиотеке DLL объекта класса приложения, производного от **CWinApp**.

Иногда удобно сначала создать проект типа MFC AppWizard (dll) в качестве тестового приложения, а затем - библиотеку DLL в виде его составной части. В результате DLL в случае необходимости будет создаваться автоматически.

Функция DllMain

Большинство библиотек DLL - просто коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Кроме функций, предназначенных для экспортирования, в каждой библиотеке DLL есть функция **DllMain**. Эта функция предназначена для инициализации и очистки DLL. Она пришла на смену функциям **LibMain** и **WEP**, применявшимся в предыдущих версиях Windows. Структура простейшей функции **DllMain** может выглядеть, например, так:

```
BOOL WINAPI DllMain (HANDLE hInst,DWORD dwReason, LPVOID IpReserved)
{
```

```
BOOL bAllWentWell=TRUE;
switch (dwReason)
{
case DLL_PROCESS_ATTACH: // Инициализация процесса.
break;
case DLL_THREAD_ATTACH: // Инициализация потока.
break;
case DLL_THREAD_DETACH: // Очистка структур потока.
break;
case DLL_PROCESS_DETACH: // Очистка структур процесса.
break;
}
if(bAllWentWell) return TRUE;
else return FALSE; }
```

Функция **DllMain** вызывается в нескольких случаях. Причина ее вызова определяется параметром dwReason, который может принимать одно из следующих значений.

При первой загрузке библиотеки DLL процессом вызывается функция **DllMain** с dwReason, равным DLL_PROCESS_ATTACH. Каждый раз при создании процессом нового потока DllMainO вызывается с dwReason, равным DLL_THREAD_ATTACH (кроме первого потока, потому что в этом случае dwReason равен DLL_PROCESS_ATTACH).

По окончании работы процесса с DLL функция **DllMain** вызывается с параметром dwReason, равным DLL_PROCESS_DETACH. При уничтожении потока (кроме первого) dwReason будет равен DLL_THREAD_DETACH.

Все операции по инициализации и очистке для процессов и потоков, в которых нуждается DLL, необходимо выполнять на основании значения dwReason, как было показано в предыдущем примере. Инициализация процессов обычно ограничивается выделением ресурсов, совместно используемых потоками, в частности загрузкой разделяемых файлов и инициализацией библиотек. Инициализация потоков применяется для настройки режимов, свойственных только данному потоку, например для инициализации локальной памяти.

В состав DLL могут входить ресурсы, не принадлежащие вызывающему эту библиотеку приложению. Если функции DLL работают с ресурсами DLL, было бы, очевидно, полезно сохранить где-нибудь в укромном месте дескриптор hInst и использовать его при загрузке ресурсов из DLL. Указатель IpReserved зарезервирован для внутреннего использования Windows. Следовательно, приложение не должно претендовать на него. Можно лишь проверить его значение. Если библиотека DLL была загружена динамически, оно будет равно NULL. При статической загрузке этот указатель будет ненулевым.

В случае успешного завершения функция **DllMain** должна возвращать TRUE. В случае возникновения ошибки возвращается FALSE, и дальнейшие действия прекращаются.

Замечание. Если не написать собственной функции DllMain(), компилятор подключит стандартную версию, которая просто возвращает TRUE.

Экспортирование функций из DLL

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице экспортируемых функций DLL. Есть два способа занести функцию в эту таблицу на этапе компиляции.

Метод __declspec (dllexport)

Можно экспортировать функцию из DLL, поставив в начале ее описания модификатор __declspec (dllexport) . Кроме того, в состав MFC входит несколько макросов, определяющих __declspec (dllexport), в том числе AFX_CLASS_EXPORT, AFX_DATA_EXPORT и AFX_API_EXPORT.

Метод __declspec применяется не так часто, как второй метод, работающий с файлами определения модуля (.def), и позволяет лучше управлять процессом экспортирования.

Файлы определения модуля

Синтаксис файлов с расширением .def в Visual C++ достаточно прямолинеен, главным образом потому, что сложные параметры, использовавшиеся в ранних версиях Windows, в Win32 более не применяются. Как станет ясно из следующего простого примера, .def-файл содержит имя и описание библиотеки, а также список экспортируемых функций:

```
MyDLL.def
LIBRARY "MyDLL"
DESCRIPTION 'MyDLL - пример DLL-библиотеки'

EXPORTS
MyFunction @1
```

В строке экспорта функции можно указать ее порядковый номер, поставив перед ним символ @. Этот номер будет затем использоваться при обращении к **GetProcAddress** (). На самом деле компилятор присваивает порядковые номера всем экспортируемым объектам. Однако способ, которым он это делает, отчасти непредсказуем, если не присвоить эти номера явно.

В строке экспорта можно использовать параметр NONAME. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL:

```
MyFunction @1 NONAME
```

Иногда это позволяет сэкономить много места в файле DLL. Приложения, использующие библитеку импортирования для неявного подключения DLL, не "заметят" разницы, поскоьку при неявном подключении порядковые номера используются автоматически. Приложениям, загружающим библиотеки DLL динамически, потребуется передавать в **GetProcAddress** порядковый номер, а не имя функции.

При использовании вышеприведенного def-файл описания экспортируемых функций DLL-библиотеки может быть,например, не таким:

```
#define EXPORT extern "C" __declspec (dllexport)
EXPORT int CALLBACK MyFunction(char *str);
```

а таким:

```
extern "C" int CALLBACK MyFunction(char *str);
```

Экспортирование классов

Создание .def-файла для экспортирования даже простых классов из динамической библиотеки может оказаться довольно сложным делом. Понадобится явно экспортировать каждую функцию, которая может быть использована внешним приложением.

Если взглянуть на реализованный в классе файл распределения памяти, в нем можно заметить некоторые весьма необычные функции. Оказывается, здесь есть неявные конструкторы и деструкторы, функции, объявленные в макросах MFC, в частности _DECLARE_MESSAGE_MAP, а также функции, которые написанные программистом.

Хотя можно экспортировать каждую из этих функций в отдельности, есть более простой способ. Если в объявлении класса воспользоваться макромодификатором AFX_CLASS_EXPORT, компилятор сам позаботится об экспортировании необходимых функций, позволяющих приложению использовать класс, содержащийся в DLL.

Память DLL

В отличие от статических библиотек, которые, по существу, становятся частью кода приложения, библиотеки динамической компоновки в 16-разрядных версиях Windows работали с памятью несколько иначе. Под управлением Win 16 память DLL размещалась вне адресного пространства задачи. Размещение динамических библиотек в глобальной памяти обеспечивало возможность совместного использования их различными задачами.

В Win32 библиотека DLL располагается в области памяти загружающего ее процесса. Каждому процессу предоставляется отдельная копия "глобальной" памяти DLL, которая реинициализируется каждый раз, когда ее загружает новый процесс. Это означает, что динамическая библиотека не может использоваться совместно, в общей памяти, как это было в Winl6.

И все же, выполнив ряд замысловатых манипуляций над сегментом данных DLL, можно создать общую область памяти для всех процессов, использующих данную библиотеку.

Допустим, имеется массив целых чисел, который должен использоваться всеми процессами, загружающими данную DLL. Это можно запрограммировать следующим образом:

```
#pragma data_seg(".myseg")
int sharedInts[10];
// другие переменные общего пользования
#pragma data_seg()
#pragma comment(lib, "msvcrt" "-SECTION:.myseg,rws");
```

Все переменные, объявленные между директивами #pragma data_seg(), размещаются в сегменте .myseg. Директива #pragma comment () - не обычный комментарий. Она дает указание библиотеке выполняющей системы С пометить новый раздел как разрешенный для чтения, записи и совместного доступа.

Полная компиляция DLL

Если проект динамической библиотеки создан с помощью AppWizard и .def-файл модифицирован соответствующим образом - этого достаточно. Если же файлы проекта создаются вручную или другими способами без помощи AppWizard, в командную строку редактора связей следует включить параметр /DLL. В результате вместо автономного выполняемого файла будет создана библиотека DLL.

Если в .def-файле есть строка LIBRART, указывать явно параметр /DLL в командной строке редактора связей не нужно.

Для MFC предусмотрен ряд особых режимов, касающихся использования динамической библиотекой библиотек MFC. Этому вопросу посвящен следующий раздел.

DLL и MFC

Программист не обязан использовать MFC при создании динамических библиотек. Однако использование MFC открывает ряд очень важных возможностей.

Имеется два уровня использования структуры MFC в DLL. Первый из них - это обычная динамическая библиотека на основе MFC, **MFC DLL** (regular MFC DLL). Она может использовать MFC, но не может передавать указатели на объекты MFC между DLL и приложениями. Второй уровень реализован в **динамических расширениях MFC** (MFC extensions DLL). Использование этого вида динамических библиотек требует некоторых дополнительных усилий по настройке, но позволяет свободно обмениваться указателями на объекты MFC между DLL и приложением.

Обычные MFC DLL

Обычные MFC DLL позволяют применять MFC в динамических библиотеках. При этом приложения, обращающиеся к таким библиотекам, не обязательно должны быть построены на основе MFC. В обычных DLL можно использовать MFC любым способом, в том числе создавая в DLL новые классы на базе классов MFC и экспортируя их в приложения.

Однако обычные DLL не могут обмениваться с приложениями указателями на классы, производные от MFC.

Если приложению необходимо обмениваться с DLL указателями на объекты классов MFC или их производных, нужно использовать расширение DLL, описанное в следующем разделе.

Архитектура обычных DLL рассчитана на использование другими средами программирования, такими как Visual Basic и PowerBuilder.

При создании обычной библиотеки MFC DLL с помощью AppWizard выбирается новый проект типа **MFC AppWizard (dll)**. В первом диалоговом окне мастера приложений необходимо выбрать один из режимов для обычных динамических библиотек: "Regular DLL with MFC statistically linked" или "Regular DLL using shared MFC DLL". Первый предусматривает статическое, а второй - динамическое подключение библиотек MFC. Впоследствии режим подключения MFC к DLL можно будет изменить с помощью комбинированного списка на вкладке "General" диалогового окна "Project settings".

Управление информацией о состоянии MFC

В каждом модуле процесса MFC содержится информация о его состоянии. Таким образом, информация о состоянии DLL отлична от информации о состоянии вызвавшего ее приложения. Поэтому любые экспортируемые из библиотеки функции, обращение к которым исходит непосредственно из приложений, должны сообщать MFC, какую информацию состояния использовать. В обычной MFC DLL, использующей динамические библиотеки MFC, перед вызовом любой подпрограммы MFC в начале экспортируемой функции нужно поместить следующую строку:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

Данный оператор определяет использование соответствующей информации о состоянии во время выполнения функции, обратившейся к данной подпрограмме.

Динамические расширения MFC

MFC позволяет создавать такие библиотеки DLL, которые воспринимаются приложениями не как набор отдельных функций, а как расширения MFC. С помощью данного вида DLL можно создавать новые классы, производные от классов MFC, и использовать их в своих приложениях.

Чтобы обеспечить возможность свободного обмена указателями на объекты MFC между приложением и DLL, нужно создать динамическое расширение MFC. DLL этого типа подключаются к динамическим библиотекам MFC так же, как и любые приложения, использующие динамическое расширение MFC.

Чтобы создать новое динамическое расширение MFC, проще всего, воспользовавшись мастером приложении, присвоить проекту тип **MFC AppWizard (dll)** и на шаге 1 включить режим "MFC Extension DLL". В результате новому проекту будут присвоены все необходимые атрибуты динамического расширения MFC. Кроме того, будет создана функция **DllMain** для DLL, выполняющая ряд специфических операций по инициализации расширения DLL. Следует обратить внимание, что динамические библиотеки данного типа не содержат и не должны содержать объектов, производных от **CWinApp**.

Инициализация динамических расширений

Чтобы "вписаться" в структуру MFC, динамические расширения MFC требуют дополнительной начальной настройки. Соответствующие операции выполняются функцией **DllMain**. Рассмотрим пример этой функции, созданный мастером AppWizard.

```
static AFX_EXTENSION_MODULE MyExtDLL = { NULL, NULL } ;
extern "C" int APIENTRY
DllMain(HINSTANCE hinstance, DWORD dwReason, LPVOID IpReserved)
{
if (dwReason == DLL_PROCESS_ATTACH)
{
TRACED("MYEXT.DLL Initializing! ") ;
// Insert this DLL into the resource chain
new CDynLinkLibrary(MyExtDLL);
}
else if (dwReason == DLL_PROCESS_DETACH)
{
TRACED("MYEXT.DLL Terminating! ");
}
return 1; // ok
}
```

Самой важной частью этой функции является вызов **AfxInitExtensionModule**. Это инициализация динамической библиотеки, позволяющая ей корректно работать в составе структуры MFC. Аргументами данной функции являются передаваемый в DllMain дескриптор библиотеки DLL и структура AFX_EXTENSION_MODULE, содержащая информацию о подключаемой к MFC динамической библиотеке.

Нет необходимости инициализировать структуру AFX_EXTENSION_MODULE явно. Однако объявить ее нужно обязательно. Инициализацией же займется конструктор **CDynLinkLibrary**. В DLL необходимо создать класс **CDynLinkLibrary**. Его конструктор не только будет инициализировать структуру AFX_EXTENSION_MODULE, но и добавит новую библиотеку в список DLL, с которыми может работать MFC.

Загрузка динамических расширений MFC

Начиная с версии 4.0 MFC позволяет динамически загружать и выгружать DLL, в том числе и расширения. Для корректного выполнения этих операций над создаваемой DLL в ее функцию **DllMain** в момент отключения от процесса необходимо добавить вызов **AfxTermExtensionModule**. Последней функции в качестве параметра передается уже использовавшаяся выше структура AFX_EXTENSION_MODULE. Для этого в текст **DllMain** нужно добавить следующие строки.

```
if(dwReason == DLL_PROCESS_DETACH)
{
AfxTermExtensionModule(MyExtDLL);
}
```

Кроме того, следует помнить, что новая библиотека DLL является динамическим расширением и должна загружаться и выгружаться динамически, с помощью функций **AfxLoadLibrary** и **AfxFreeLibrary**,а не **LoadLibrary** и **FreeLibrary**.

Экспортирование функций из динамических расширений

Рассмотрим теперь, как осуществляется экспортирование в приложение функций и классов из динамического расширения. Хотя добавить в DEF-файл все расширенные имена можно и вручную, лучше использовать модификаторы для объявлений экспортируемых классов и функций, такие как AFX_EXT_CLASS и AFX_EXT_API,например:

```
class AFX_EXT_CLASS CMyClass : public CObject
(
// Your class declaration
}
void AFX_EXT_API MyFunc() ;
```