**Разработка DLL в Borland Delphi**

Если ваш компьютер работает под управлением операционной системы Windows, то вы не можете не знать о существовании динамических подсоединяемых библиотек (dynamic link libraries - DLL). Достаточно взглянуть на список файлов, расположенных в системном каталоге Windows . Порой количество используемых операционной системой динамических библиотек достигает нескольких сотен. Таким образом, мне кажется, не стоит отрицать тот факт, что DLL являются неотъемлемой частью функционирования операционных систем семейства Microsoft Windows. Однако для вас может быть неочевидна необходимость использования динамических библиотек при разработке приложений. В рамках данной статьи мы поговорим о принципах функционирования DLL и их использования в процессе создания ваших собственных программ.

Для начала давайте выясним, что собой представляет динамическая подключаемая библиотека. Итак, DLL - это один или несколько логически законченных фрагментов кода, сохраненных в файле с расширением .dll. Этот код может быть запущен на выполнение в процессе функционирования какой-либо другой программы (такие приложения называются вызывающими по отношению к библиотеке), но сама DLL не является исполняемым файлом.

Существует два типа динамических библиотек - исполняемые и библиотеки ресурсов. Однако это не означает, что в одном файле не может находиться и код некоторой функции и какие-либо ресурсы. Просто иногда бывает удобно разнести реализацию исполняемых процедур и используемые приложением ресурсы в разные файлы.

Итак, процедуры и функции, содержащиеся в динамической библиотеке, можно разделить на два типа: те, которые могут быть вызваны из других приложений. Рассмотрим следующий пример:

Screen.Cursors[myCursor] := LoadCursor(HInstance, MYCURSOR');

LoadCursor - функция Windows API, которая вызывается приложением из динамической библиотеки User 32.dll. Кстати, примером хранимых в динамической библиотеке ресурсов могут являться такие стандартные диалоги Windows , как диалог открытия файла, диалог печати или настройки принтера. Эти диалоги находятся в файле Comctl32.dll. Однако многие прикладные разработчики используют функции вызова форм этих диалогов, совершенно не задумываясь, где хранится их описание.

Второй тип процедур - это те, которые используются только внутри самого файла библиотеки.

**Аргументы в пользу использования DLL**

Итак, прежде чем перейти к обсуждению структуры динамических библиотек, необходимо поговорить о тех преимуществах, которые предоставляет их использование разработчику. Во-первых, это повторное использование кода. Думаю, нет необходимости пояснять удобство использования один раз разработанных процедур и функций при создании нескольких приложений? Кроме того, в дальнейшем вы сможете продать некоторые из своих библиотек, не раскрывая исходных кодов. А чем тогда это лучше компонентов, спросите вы? А тем, что функции, хранящиеся в библиотеке, могут быть вызваны на выполнение из приложений, разработанных не на Object Pascal, а, например, с использованием C++Builder, Visual Basic, Visual C++ и т.д. Такой подход накладывает некоторые ограничения на принцип разработки библиотеки, но это возможно. Звучит заманчиво? Мне кажется, даже очень. Но это еще не все.

Во-вторых, использование DLL предоставляет возможность использования один загруженного в оперативную память кода несколькими приложениями. К примеру, если вы разрабатываете программное обеспечение для большого предприятия, то вполне возможно, что в различных созданных вами приложениях будут использоваться одни и те же функции, процедуры, формы и другие ресурсы. Естественно, что при выделении общих для нескольких приложений данных в DLL может привести к экономии как дискового пространства, так и оперативной памяти, иногда очень даже существенному.

В-третьих, следует поговорить вот о чем. Всего несколько лет назад при разработке программного обеспечения вы могли совершенно не волноваться относительно распространения ваших продуктов где-либо, кроме вашей страны. Я хочу сказать, что проблема перевода на другие языки текста на элементах управления (пункты меню, кнопки, выпадающие списки, подсказки), сообщений об ошибках и т.д. не стояла так остро, как сейчас. Однако, с повсеместным внедрением Интернета у вас появилась возможность бытрой передачи готовых программных продуктов практически в любую точку мира. И что будут делать с вашей программой где-нибудь в Объединенных Арабских Эмиратах, если кроме как по-русски, она с пользователем общаться не умеет? Вы сами можете оценить этот эффект, если хоть раз на экране вашего компьютера вместо с детства знакомого русского языка появляется «арабская вязь» (например, из-за «сбоя» шрифтов). Итак, уже сейчас вы должны планировать возможность распространения ваших приложений в других странах (если, конечно, у вас есть желание получить как можно больше прибыли). Соответственно, встает вопрос быстрого перевода интерфейса вашей программы на другие языки. Одним из путей может являться создание ресурсов интерфейсов внутри DLL . К примеру, можно создать одно приложение, которое в зависимости от версии динамической библиотеки будет выводить сообщения на различных языках.

Естественно, выше приведены лишь некоторые из аргументов в пользу использования динамически подключаемых библиотек при разработке приложений. Итак, надеюсь, вы все еще заинтересованы в том, чтобы узнать, как собственно, DLL создаются. Если так, то вперед.

**Основы разработки DLL**

Разработка динамических библиотек не представляет собой некий сверхсложный процесс, доступный лишь избранным. Если вы достаточно хорошо знакомы с разработкой приложений на Object Pascal , то вам не составит особого труда научиться работать с механизмом DLL . Итак, рассмотрим те особенности создания DLL , которые вам необходимо знать, а в завершении статьи разработаем свою собственную библиотеку.

Как и любой другой модуль, модуль динамической библиотеки имеет фиксированный формат. Взгляните на листинг, представленный ниже.

library MyFirstDLL;

uses

SysUtils,

Classes,

Forms,

Windows;

procedure HelloWorld(AForm : TForm);

begin

MessageBox(AForm.Handle, Hello world!',

DLL Message Box', MB\_OK or MB\_ICONEXCLAMATION);

end;

exports

HelloWorld;

begin

end.

Первое, на что следует обратить внимание, это ключевое слово library, находящееся вверху страницы. Library определяет этот модуль как модуль библиотеки DLL . Далее идет название библиотеки. В нашем примере мы имеем дело с динамической библиотекой, содержащей единственную процедуру: HelloWorld. Причем обратите внимание, что данная процедура по структуре ничем не отличается от тех, которые вы помещаете в модули своих приложений. Ключевое слово exports сигнализирует компилятору о том, что перечисленные ниже функции и/или процедуры должны быть доступны из вызывающих приложений (т.е. они как бы «экспортируются» из библиотеки). Подробнее о механизме экспорта мы поговорим чуть позже.

И, наконец, в конце модуля можно увидеть ключевые слова begin и end. Внутри данного блока вы можете поместить код, который должен выполняться в процессе загрузки библиотеки. Достаточно часто этот блок остается пустым.

Как уже говорилось выше, все процедуры и функции, помещаемые в DLL , могут быть разделены на две группы: экспортируемые (вызываемые из других приложений) и локальные. Естественно, внутри библиотеки также могут быть описаны классы, которые в свою очередь содержат методы, но в рамках данной статьи я не буду на этом останавливаться.

Описание и реализация процедур и функций, вызываемых в пределах текущей DLL , ничем не отличаются от их аналогов в обычных проектах-приложениях. Их специфика заключается лишь в том, что вызывающая программа не будет иметь к ним доступа. Она просто не будет ничего знать об их существования, так же, как одни классы ничего не знают о тех методах, которые описаны в секции private других классов.

В дополнение к процедурам и функциям, DLL может содержать глобальные данные, доступ к которым разрешен для всех процедур и функций в библиотеке. Для 16-битных приложений эти данные существовали в единственном экземпляре независимо от количества загруженных в оперативную память программ, которые используют текущую библиотеку. Другими словами, если одна программа изменяет значение глобальной переменной a на 100, то для всех остальных приложений a будет значение 100. Для 32-битных приложений это не так. Теперь для каждого приложения создается отдельная копия глобальной области данных.

**Экспорт функций из DLL**

Как уже говорилось выше, для экспорта процедур и функций из DLL , необходимо использовать ключевое слово export. Еще раз обратите внимание на представленный выше листинг библиотеки MiFirstDll. Поскольку процедура HelloWorld определена как экспортируемая, то она может быть вызвана на выполнение из других библиотек или приложений. Существуют следующие способы экспорта процедур и функций: экспорт по имени и экспорт по порядковому номеру.

Наиболее распространенный способ экспорта - по имени. Взглянем на приведенный ниже текст:

exports

SayHello,

DoSomething,

DoSomethingReallyCool;

Следует обратить внимание на то, что Delphi автоматически назначает порядковый номер каждой экспортируемой функции (процедуре) независимо от того, определяете вы его явно или нет. Явное определение индекса позволяет вам лично управлять порядковым номером экспортируемой функции или процедуры.

Для того, чтобы определить выполняется ли ваш кодек в DLL или в вызывающем приложении, можно воспользоваться глобальной переменной IsLibrary. Она принимает значение ИСТИНА в том случае, если код вызывается из библиотеки и ЛОЖЬ в случае выполнения процедуры или функции из вызывающего приложения.

Кроме этого, в поставку Delphi входит весьма полезная утилита tdump , которая предоставляет данные о том, какая информация экспортируется из указанной DLL.

**Использование DLLProc**

Выше я уже говорил о том, что код инициализации динамической библиотеки может быть помещен в блок begin...end. Однако кроме этого зачастую необходимо предусмотреть некоторые действия, выполняемые в процессе выгрузки DLL из оперативной памяти. В отличии от других типов модулей, модуль DLL не имеет ни секции initialization, ни секции finalization. К примеру, вы можете динамически выделить память в главном блоке, однако не понятно, где эта память должна быть освобождена. Для решения этой проблемы существует DLLProc - специальная процедура, вызываемая в определенные моменты функционирования DLL.

Для начала следует сказать о самой причине существования DLLProc . Динамическая библиотека получает сообщения от EN-US'>Windows в моменты своей загрузки и выгрузки из оперативной памяти, а также в тех случаях, когда какой-нибудь очередной процесс, использующий функции и/или ресурсы, хранящиеся в библиотеке, загружается в память. Такая ситуация возможно в том случае, когда библиотека необходима для функционирования нескольких style="mso-spacerun: yes">љ приложений. А для того, чтобы вы имели возможность указывать, что именно должно происходить в такие моменты, необходимо описать специальную процедуру, которая и будет ответственна за такие действия. К примеру, она может выглядеть следующим образом:

procedure MyFirstDLLProc(Reason: Integer);

begin

if Reason = DLL\_PROCESS\_DETACH then

{DLL is unloading. Cleanup code here.}

end;

Однако системе совершенно не очевидно, что именно процедура MyFirstDllProc ответственна за обработку рассмотренных выше ситуаций. Поэтому вы должны поставить в соответствие адрес нашей процедуры глобальной переменной DLLProc. Это необходимо сделать в блоке begin...end примерно так:

begin

DLLProc := @MyDLLProc;

{ Что-нибудь еще, что должно выполняться в

процессе инициализации библиотеки }

end.

Ниже представлен код, демонстрирующий один из возможных вариантов применения DLLProc.

library MyFirstDLL;

uses

SysUtils,

Classes,

Forms,

Windows;

var

SomeBuffer : Pointer;

procedure MyFirstDLLProc(Reason: Integer);

begin

if Reason = DLL\_PROCESS\_DETACH then

{DLL is выгружается из памяти.

Освобождаем память, выделенную под буфер.}

FreeMem(SomeBuffer);

end;

procedure HelloWorld(AForm : TForm);

begin

MessageBox(AForm.Handle, Hello world!',

DLL Message Box', MB\_OK or MB\_ICONEXCLAMATION);

end;

{Какой-нибудь код, в котором используется SomeBuffer.}

exports

HelloWorld;

begin

{Ставим в соответствие переменной

DLLProc адрес нашей процедуры.}

DLLProc := @MyFirstDLLProc;

SomeBuffer := AllocMem(1024);

end.

Как можно увидеть, в качестве признака того или иного события, в результате которого вызывается процедура MyFirstDll, является значение переменной Reason. Ниже приведены возможные значения этой переменной.

DLL\_PROCESS\_DETACH - библиотека выгружается из памяти; используется один раз;

DLL\_THREAD\_ATTACH - в оперативную память загружается новый процесс, использующий ресурсы и/или код из данной библиотеки;

DLL\_THREAD\_DETACH - один из процессов, использующих библиотеку, 'выгружается' из памяти.

**Загрузка DLL**

Прежде чем начать использование какой-либо процедуры или функции, находящейся в динамической библиотеке, вам необходимо загрузить DLL в оперативную память. Загрузка библиотеки может быть осуществлена одним из двух способов: статическая загрузка и динамическая загрузка. Оба метода имеют как преимущества, так и недостатки.

Статическая загрузка означает, что динамическая библиотека загружается автоматически при запуске на выполнение использующего ее приложения. Для того чтобы использовать такой способ загрузки, вам необходимо воспользоваться ключевым словом external при описании экспортируемой из динамической библиотеки функции или процедуры. DLL автоматически загружается при старте программы, и Вы сможете использовать любые экспортируемые из нее подпрограммы точно так же, как если бы они были описаны внутри модулей приложения. Это наиболее легкий способ использования кода, помещенного в DLL . Недостаток метода заключается в том, что если файл библиотеки, на который имеется ссылка в приложении, отсутствует, программа откажется загружаться.

Смысл динамического метода заключается в том, что вы загружаете библиотеку не при старте приложения, а в тот момент, когда вам это действительно необходимо. Сами посудите, ведь если функция, описанная в динамической библиотеке, используется только при 10% запусков программы, то совершенно нет смысла использовать статический метод загрузки. Выгрузка библиотеки из памяти в данном случае также осуществляется под вашим контролем. Еще одно преимущества такого способа загрузки DLL - это уменьшение (по понятным причинам) времени старта вашего приложения. А какие же у этого способа имеются недостатки? Основной, как мне кажется, - это то, что использование данного метода является более хлопотным, чем рассмотренная выше статическая загрузка. Сначала вам необходимо воспользоваться функцией Windows API LoadLibrary . Для получения указателя на экспортируемой процедуры или функции должна использоваться функция GetProcAddress. После завершения использования библиотеки DLL должна быть выгружена с применением FreeLibrary.

**Вызов процедур и функций, загруженных из DLL.**

Способ вызова процедур и функций зависит от того, каким образом вы загрузили динамическую библиотеку, в которой эти подпрограммы находятся.

Вызов функций и процедур из статически загруженных DLL достаточно прост. Первоначально в приложении должно содержаться описание экспортируемой функции (процедуры). После этого вы можете их использовать точно так же, как если бы они были описаны в одном из модулей вашего приложения. Для импорта функции или процедуры, содержащейся в DLL , необходимо использовать модификатор external в их объявлении. К примеру, для рассмотренной нами выше процедуры HelloWorld в вызывающем приложении должна быть помещена следующая строка:

procedure SayHello(AForm : TForm); external myfirstdll.dll';

Ключевое слово external сообщает компилятору, что данная процедура может быть найдена в динамической библиотеке (в нашем случае - myfirstdll.dll). Далее вызов этой процедуры выглядит следующим образом:

...

HelloWorld(self);

...

При импорте функции и процедур будьте особенно внимательны при написании их имен и интерфейсов! Дело в том, что в процессе компиляции приложения не производится проверки на правильность имен объектов, экспортируемых из DLL, осуществляться не будет, и если вы неправильно описали какую-нибудь функцию, то исключение будет сгенерировано только на этапе выполнения приложения.

Импорт из DLL может проводиться по имени процедуры (функции), порядковому номеру или с присвоением другого имени.

В первом случае вы просто объявляете имя процедуры и библиотеку, из которой ее импортируете (мы это рассмотрели чуть выше). Импорт по порядковому номеру требует от вас указание этого самого номера:

procedure HelloWorld(AForm : TForm);

external myfirstdll.dll' index 15;

В этом случае имя, которое вы даете процедуре при импорте не обязательно должно совпадать с тем, которое было указано для нее в самой DLL. Т.е. приведенная выше запись означает, что вы импортируете из динамической библиотеки myfirstdll.dll процедуру, которая в ней экспортировалась пятнадцатой, и при этом в рамках вашего приложения этой процедуре дается имя SayHello.

Если вы по каким-то причинам не применяете описанный выше способ импорта, но тем не менее хотите изменить имя импортируемой функции (процедуры), то можно воспользоваться третьим методом:

procedure CoolProcedure;

external myfirstdll.dll' name DoSomethingReallyCool';

Здесь импортируемой процедуре CoolProcedure дается имя DoSomethingReallyCool. Вызов процедур и функций, импортируемых из динамически загружаемых библиотек несколько более сложен, чем рассмотренный нами выше способ. В данном случае требуется объявить указатель на функцию или процедуру, которую вы собираетесь использовать. Помните процедуру HelloWorld? Давайте посмотрим, что необходимо сделать для того, чтобы вызвать ее на выполнение в случае динамической загрузки DLL. Во-первых, вам необходимо объявить тип, который описывал бы эту процедуру:

type

THelloWorld = procedure(AForm : TForm);

Теперь вы должны загрузить динамическую библиотеку, с помощью GetProcAddress получить указатель на процедуру, вызвать эту процедуру на выполнение, и, наконец, выгрузить DLL из памяти. Ниже приведен код, демонстрирующий, как это можно сделать:

var

DLLInstance : THandle;

HelloWorld : THelloWorld;

begin

{ загружаем DLL }

DLLInstance := LoadLibrary(myfirstdll.dll');

{ получаем указатель }

@HelloWorld := GetProcAddress(DLLInstance, HelloWorld');

{ вызываем процедуру на выполнение }

HelloWorld(Self);

{ выгружаем DLL из оперативной памяти }

FreeLibrary(DLLInstance);

end;

Как уже говорилось выше, одним из недостатков статической загрузки DLL является невозможность продолжения работы приложения при отсутствии одной или нескольких библиотек. В случае с динамической загрузкой у вас появляется возможность программно обрабатывать такие ситуации и не допускать, чтобы программа «вываливалась» самостоятельно. По возвращаемому функциями LoadLibrary и GetProcAddress значениям можно определить, успешно ли прошла загрузка библиотеки и найдена ли в ней необходимая приложению процедура. Приведенный ниже код демонстрирует это.

procedure TForm1.DynamicLoadBtnClick(Sender: TObject);

type

THelloWorld = procedure(AForm : TForm);

var

DLLInstance : THandle;

HelloWorld : THelloWorld;

begin

DLLInstance := LoadLibrary(myfirstdll.dll');

if DLLInstance = 0 then begin

MessageDlg(Невозможно загрузить DLL', mtError, [mbOK], 0);

Exit;

end;

@HelloWorld := GetProcAddress(DLLInstance, HelloWorld');

if @HelloWorld <> nil then

HelloWorld (Self)

else

MessageDlg(Не найдена искомая процедура!.', mtError, [mbOK], 0);

FreeLibrary(DLLInstance);

end;

В DLL можно хранить не только код, но и формы. Причем создание и помещение форм в динамическую библиотеку не слишком сильно отличается от работы с формами в обычном проекте. Сначала мы рассмотрим, каким образом можно написать библиотеку, содержащую формы, а затем мы поговорим об использовании технологии MDI в DLL.

Разработку DLL, содержащую форму, я продемонстрирую на примере.

Итак, во-первых, создадим новый проект динамической библиотеки. Для этого выберем пункт меню File|New, а затем дважды щелкнем на иконку DLL . После этого вы увидите примерно следующий код:

library Project2;

{здесь были комментарии}

uses

SysUtils,

Classes;

{$R \*.RES}

begin

end.

Сохраните полученный проект. Назовем его DllForms.dpr.

Теперь следует создать новую форму. Это можно сделать по-разному. Например, выбрав пункт меню File|New Form. Добавьте на форму какие-нибудь компоненты. Назовем форму DllForm и сохраним получившийся модуль под именем DllFormUnit.pas .

Вернемся к главному модулю проекта и поместим в него функцию ShowForm, в задачу которой будет входить создание формы и ее вывод на экран. Используйте для этого приведенный ниже код.

function ShowForm : Integer; stdcall;

var

Form : TDLLForm;

begin

Form := TDLLForm.Create(Application);

Result := Form.ShowModal;

Form.Free;

end;

Обращаю внимание, что для того, чтобы проект был скомпилирован без ошибок, необходимо добавить в секцию uses модуль Forms .

Экспортируем нашу функцию с использованием ключевого слова exports :

exports

ShowForm;

Компилируем проект и получаем файл dllforms.dll. Эти простые шаги - все, что необходимо сделать для создания динамической библиотеки, содержащей форму. Обратите внимание, что функция ShowForm объявлена с использованием ключевого слова stdcall . Оно сигнализирует компилятору использовать при экспорте функции соглашение по стандартному вызову (standard call calling convention). Экспорт функции таким образом создает возможность использования разработанной DLL не только в приложениях, созданных в Delphi.

Соглашение по вызову (Calling conventions) определяет, каким образом передаются аргументы при вызове функции. Существует пять основных соглашений: stdcall, cdecl, pascal, register и safecall. Подробнее об этом можно узнать, посмотрев раздел " Calling Conventions " в файле помощи Delphi.

Также обратите внимание, что значение, возвращаемое функцией ShowForm , соответствует значению ShowModal. Таким образом вы можете передавать некоторую информацию о состоянии формы вызывающему приложению.

Ниже представлено два листинга, первый из которых содержит полный код файла проекта DLL (модуль с формой здесь не приводится), а второй - модуль вызывающего приложения, в котором используется только что разработанная нами библиотека.

library DllForms;

uses

SysUtils,

Classes,

Forms,

DllFormUnit in 'DllFormUnit.pas' {DllForm};

{$R \*.RES}

function ShowForm : Integer; stdcall;

var

Form : TDLLForm;

begin

Form := TDLLForm.Create(Application);

Result := Form.ShowModal;

Form.Free;

end;

begin

end.

unit TestAppUnit;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics,

Controls, Forms, Dialogs, StdCtrls;

type

TForm1 = class(TForm)

Button1: TButton;

procedure Button1Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

function ShowForm : Integer; stdcall;

external dllforms.dll';

implementation

{$R \*.DFM}

procedure TForm1.Button1Click(Sender: TObject);

begin

ShowForm;

end;

end.

Прошу заметить, что при экспорте функции также было использовано ключевое слово stdcall.

Следует обратить особое внимание на работу с дочерними формами в DLL. Если, к примеру, в вызывающем приложении главная форма имеет значение свойства FormStyle, равным MDIForm, то при попытке вызова из DLL MDIChild-формы, на экране появится сообщение об ошибке, в котором будет говориться, что нет ни одной активной MDI-формы.

В тот момент, когда вы пытаетесь показать ваше дочернее окно, VCL проверяет корректность свойства FormStyle главной формы приложения. Однако в нашем случае все вроде бы верно. Так в чем же дело? Проблема в том, что при проведении такой проверки, рассматривается объект Application, принадлежащий не вызывающему приложению, а собственно динамической библиотеке. Ну, и естественно, поскольку в DLL нет главной формы, проверка выдает ошибку. Для того чтобы избежать такой ситуации, надо назначить объекту Application динамической библиотеки объект Application вызывающего приложения. Естественно, это заработает только в том случае, когда вызывающая программа - VCL-приложение. Кроме того, перед выгрузкой библиотеки из памяти необходимо вернуть значение объекта Application библиотеки в первоначальное состояние. Это позволит менеджеру памяти очистить оперативную память, занимаемую библиотекой. Следовательно, вам нужно сохранить указатель на «родной» для библиотеки объект Application в глобальной переменной, которая может быть использована при восстановлении его значения.

Итак, вернемся немного назад и перечислим шаги, необходимые нам для работы с помещенным в DLL MDIChild-формами.

В динамической библиотеке создаем глобальную переменную типа TApplication.

Сохраняем указатель на объект Application DLL в глобальной переменной.

Объекту Application динамической библиотеки ставим в соответствие указатель на Application вызывающего приложения.

Создаем MDIChild-форму и работаем с ней.

Возвращаем в первоначальное состояние значение объекта Application динамической библиотеки и выгружаем DLL из памяти.

Первый шаг прост. Просто помещаем следующий код в верхней части модуля DLL:

var

DllApp : TApplication;

Затем создаем процедуру, которая будет изменять значение объекта Application и создавать дочернюю форму. Процедура может выглядеть примерно так:

procedure ShowMDIChild(MainApp : TApplication);

var

Child : TMDIChild;

begin

if not Assigned(DllApp) then begin

DllApp := Application;

Application := MainApp;

end;

Child := TMDIChild.Create(Application.MainForm);

Child.Show;

end;

Все, что нам теперь необходимо сделать, - это предусмотреть возвращение значения объекта Application в исходное состояние. Делаем это с помощью процедуры MyDllProc:

procedure MyDLLProc(Reason: Integer);

begin

if Reason = DLL\_PROCESS\_DETACH then

{ DLL is выгружается. Восстанавливаем значение указателя Application}

if Assigned(DllApp) then

Application := DllApp;

end;

Вместо заключения.

Использование динамически подключаемых библиотек не так сложно, как это может показаться на первый взгляд. DLL предоставляют широчайшие возможности для оптимизации работы приложений, а также работы самих программистов. Используйте DLL и, возможно, ваша жизнь станет легче!