Лабораторная работа №2

Динамически подключаемые библиотеки (DLL)

Цель: Получить навыки разработки программного обеспечения с применением библиотек динамической компоновки.

1. Теоретическая база

Библиотеки динамической компоновки DLL (Dynamic Link Libraries) являются стержневой компонентой операционной системы Windows и многих ее приложений. Без преувеличения можно сказать, что вся операционная система Windows, все ее драйверы, а также другие расширения есть ни что иное, как набор библиотек динамической компоновки. Редкое крупное Windows-приложение не имеет собственных библиотек динамической компоновки, и ни одно приложение не может обойтись без вызова функций, расположенных в таких библиотеках. В частности, все функции программного интерфейса Windows находятся именно в библиотеках динамической компоновки DLL.

1.1. Статическая и динамическая компоновка

Прежде чем приступить к изучению особенностей использования библиотек динамической компоновки в операционной системе Microsoft Windows, напомним кратко, чем отличаются друг от друга статическая и динамическая компоновка.

При использовании статической компоновки вы готовили исходный текст приложения, затем транслировали его для получения объектного модуля. После этого редактор связей компоновал объектные модули, полученные в результате трансляции исходных текстов, и модули из библиотек объектных модулей в один исполнимый exe-файл. В процессе запуска файл программы загружался полностью в оперативную память и ему передавалось управление. Таким образом, при использовании статической компоновки редактор связей записывает в файл программы все модули, необходимые для работы. В любой момент времени в оперативной памяти компьютера находится весь код, необходимый для работы запущенной программы.

В среде мультизадачной операционной системы статическая компоновка неэффективна, так как приводит к неэкономному использованию очень дефицитного ресурса - оперативной памяти. Представьте себе, что в системе одновременно работают 5 приложений и все они вызывают такие функции, как sprintf, memcpy, strcmp и так далее. Если приложения были собраны с использованием статической компоновки, в памяти будут находиться одновременно 5 копий функции sprintf, 5 копий функции memcpy и т . д. Очевидно, использование оперативной памяти было бы намного эффективнее, если бы в памяти находилось только по одной копии функций, а все работающие параллельно программы могли их вызывать.

Практически в любой многозадачной операционной системе для любого компьютера используется именно такой способ обращения к функциям, нужным одновременно большому количеству работающих параллельно программ. При использовании динамической компоновки загрузочный код нескольких (или нескольких десятков) функций объединяется в отдельные файлы, загружаемые в оперативную память в единственном экземпляре. Программы, работающие параллельно, вызывают функции, загруженные в память из файлов библиотек динамической компоновки, а не из файлов программ. Таким образом, используя механизм динамической компоновки, в загрузочном файле программы можно расположить только те функции, которые являются специфическими для данной программы. Те же функции, которые нужны многим программам, работающим параллельно, можно вынести в отдельные файлы - библиотеки динамической компоновки и хранить в

памяти в единственном экземпляре. Эти файлы можно загружать в память только при необходимости, например, когда какая-нибудь программа захочет вызвать функцию, код которой расположен в библиотеке.

В операционной системе Windows файлы библиотек динамической компоновки имеют расширение имени dll, хотя можно использовать любое другое, например exe. В первых версиях Windows DLL-библиотеки располагались в файлах с расширением имени exe. Возможно, поэтому файлы kernal.exe, gdi.exe и user.exe имели расширение имени exe, а не dll, несмотря на то что перечисленные выше файлы, составляющие ядро операционной системы Windows, есть не что иное, как DLL. Наиболее важные компоненты операционной системы Microsoft Windows расположены в библиотеках с именами kernel.dll (ядро операционной системы), user.dll (функции пользовательского интерфейса), gdi.dll (функции для рисования изображений и текста).

Механизм динамической компоновки был изобретен задолго до появления операционных систем Windows и OS/2 (которая также активно использует механизм динамической компоновки). Например, в мультизадачных многопользовательских операционных системах VS1, VS2, MVS, VM, созданных для компьютеров IBM-370 и аналогичных, код функций, нужных параллельно работающим программам, располагается в отдельных библиотеках и может загружаться при необходимости в специально выделенную общую область памяти.

1.2. DLL в операционной системе Windows

В операционной системе Microsoft Windows после загрузки DLL становится как бы частью операционной системы. DLL-библиотека является модулем и находится в памяти в единственном экземпляре, содержит сегменты кода и ресурсы, а также один сегмент данных. Можно сказать, что для DLL-библиотеки создается одна копия (instance), состоящая только из сегмента данных, и один модуль, состоящий из кода и ресурсов.

DLL, в отличие от приложения, не имеет стека и очереди сообщения. Функции, расположенные в модуле DLL, выполняются в контексте вызвавшей их задачи. При этом они пользуются стеком копии приложения, так как собственного стека в DLL не предусмотрено.

В операционной системе Microsoft Windows каждое приложение работает в рамках отдельного адресного пространства. Поэтому для того чтобы приложение могло вызывать функции из DLL, эта библиотека должна находиться в адресном пространстве приложения. Поэтому DLL загружается в страницы виртуальной памяти, которые отображаются в адресные пространства всех "заинтересованных" приложений, которым нужны функции из этой библиотеки. Что же касается данных DLL, то для каждого приложения в глобальном пуле создается отдельная область. Таким образом, различные приложения не могут в этом случае передавать друг другу данные через общую область данных DLL. Однако принципиальная возможность создания глобальных областей памяти DLL, доступных разным процессам, существует. Для этого необходимо при редактировании описать область данных DLL как SHARED. Заметим, одна и та же функция DLL может отображаться на разные адреса в различные адресные пространства приложений. Это же относится к глобальным и статическим переменным DLL - они будут отображаться на разные адреса для различных приложений.

Когда первое приложение загрузит DLL в память (явно или неявно), эта библиотека (точнее говоря, страницы памяти, в которые она загружена) будет отображена в адресное пространство этого приложения. Если теперь другое приложение попытается загрузить ту же самую библиотеку еще раз, то для него будет создано новое отображение тех же самых страниц. На этот раз страницы могут быть отображены на другие адреса. Кроме того, для каждой DLL система ведет счетчик использования (usage count). Содержимое этого счетчика увеличивается при очередной загрузке библиотеки в память и уменьшается при

освобождении библиотеки. Когда содержимое счетчика использования DLL станет равным нулю, библиотека будет выгружена из памяти.

В среде операционной системы из семейства Microsoft Windows NT DLL не имеют собственных областей данных, а отображаются в адресные пространства приложений, загружающих эти библиотеки. Как результат приложения не могут получать адреса статических и глобальных переменных и использовать эти переменные для обмена данными, ведь адрес, верный в контексте одного приложения, не будет иметь никакого смысла для другого приложения.

Приложение также может сделать попытку изменить содержимое статической или глобальной переменной, с тем чтобы другие приложения могли прочитать новое значение. Однако этот способ передачи данных между приложениями не будет работать. Попытка изменения данных будет зафиксирована операционной системой, которая создаст для этого приложения копию страницы памяти, в которой находятся изменившиеся данные, с использованием механизма копирования при записи (copy-on-write). Если по какой-либо причине вы не можете использовать для обмена данными между приложениями файлы, отображаемые на память, можно создать DLL с данными, имеющими атрибут SHARED. При обращении к глобальным переменным, расположенным в секции с атрибутом SHARED, процессы должны выполнять взаимную синхронизацию с использованием таких средств, как критические секции, объекты-события, семафоры.

1.3. Как работает DLL

При обращении к библиотеке в операционной системе Microsoft Windows используется функция DLLMain, которая выполняет все необходимые задачи по инициализации библиотеки и при необходимости освобождает заказанные ранее ресурсы (имя функции инициализации может быть любым). DLLMain вызывается всякий раз, когда выполняется инициализация процесса или задачи, обращающихся к функциям библиотеки, а также при явной загрузке и выгрузке библиотеки функциями LoadLibrary и FreeLibrary.

Ниже мы привели прототип функции DLLMain:

```
BOOL WINAPI  DLLMain(
HINSTANCE hinstDLL,      //идентификатор модуля DLL-библиотеки DWORD
fdwReason,               //  код причины вызова функции
LPVOID     lpvReserved); //  зарезервировано
```

Через параметр hinstDLL функции DLLMain передается идентификатор модуля DLL, который можно использовать при обращении к ресурсам, расположенным в файле этой библиотеки. Что же касается параметра fdwReason, то он зависит от причины, по которой произошел вызов функции DLLMain. Этот параметр может принимать следующие значения:

Значение	*Описание*
DLL_PROCESS_ATTACH	Библиотека отображается в адресное пространство процесса в результате запуска процесса или вызова функции LoadLibrary
DLL_THREAD_ATTACH	Текущий процесс создал новую задачу, после чего система вызывает функции DLLMain всех DLL, подключенных к процессу
DLL_THREAD_DETACH	Этот код причины передается функции DLLEntryPoint, когда задача завершает свою работу нормальным (не аварийным) способом
DLL_PROCESS_DETACH	Отображение DLL в адресное пространство отменяется в результате нормального завершения процесса или вызова функции FreeLibrary

Параметр lpvReserved зарезервирован. В SDK тем не менее сказано, что значение параметра lpvReserved равно NULL во всех случаях, кроме двух следующих:

- когда параметр fdwReason равен DLL_PROCESS_ATTACH и используется статическая загрузка DLL
- когда параметр fdwReason равен DLL_PROCESS_DETACH и функция DLLEntryPoint вызвана в результате завершения процесса, а не вызова функции FreeLibrary.

В процессе инициализации функция DLLMain может отменить загрузку DLL. Если код причины вызова равен DLL_PROCESS_ATTACH, функция DLLMain отменяет загрузку библиотеки, возвращая значение FALSE. Если же инициализация выполнена успешно, функция должна возвратить значение TRUE.

В том случае, когда приложение пыталось загрузить DLL функцией LoadLibrary, а функция DLLMain отменила загрузку, функция LoadLibrary возвратит значение NULL. Если же приложение выполняет инициализацию DLL неявно, при отмене загрузки библиотеки приложение также не будет загружено для выполнения.

1.4.Экспортирование функций и глобальных переменных

Хотя существуют DLL без исполнимого кода и предназначенные для хранения ресурсов, подавляющее большинство DLL экспортируют функции для их совместного использования несколькими приложениями. Кроме функции DLLEntryPoint в библиотеках операционной системы Microsoft Windows могут быть определены экспортируемые и неэкспортируемые функции.

Экспортируемые функции доступны для вызова приложениям Windows. Неэкспортируемые функции являются локальными для DLL, они доступны только для функций библиотеки. При необходимости вы можете экспортировать из DLL не только функции, но и глобальные переменные.

Самый простой способ сделать функцию экспортируемой - перечислить все экспортируемые функции в файле определения модуля при помощи оператора EXPORTS:

```
EXPORTS
ИмяТочкиВхода [=ВнутрИмя] [@Номер] [NONAME] [CONSTANT].
```

Здесь ИмяТочкиВхода задает имя, под которым экспортируемая из DLL функция будет доступна для вызова. Внутри DLL эта функция может иметь другое имя. В этом случае необходимо указать ее внутреннее имя ВнутрИмя. С помощью параметра @Номер вы можете задать порядковый номер экспортируемой функции. Если вы не укажете порядковые номера экспортируемых функций, при компоновке загрузочного файла DLL редактор связи создаст свою собственную нумерацию, которая может изменяться при внесении изменений в исходные тексты функций и последующей повторной компоновке.

Заметим, что ссылка на экспортируемую функцию может выполняться двумя различными способами - по имени функции и по ее порядковому номеру. Если функция вызывается по имени, ее порядковый номер не имеет значения. Однако вызов функции по порядковому номеру выполняется быстрее, поэтому использование порядковых номеров предпочтительнее.

Если при помощи файла определения модуля DLL вы задаете фиксированное распределение порядковых номеров экспортируемых функций, при внесении изменений в исходные тексты DLL-библиотеки это распределение не изменится. В этом случае все приложения, ссылающиеся на функции из этой библиотеки по их порядковым номерам, будут работать правильно. Если же вы не определили порядковые номера функций, у приложений могут возникнуть проблемы с правильной адресацией функции из-за возможного изменения этих номеров.

Указав флаг NONAME и порядковый номер, вы сделаете имя экспортируемой функции невидимым. При этом экспортируемую функцию можно будет вызвать только по порядковому номеру, так как имя такой функции не попадет в таблицу экспортируемых имен DLL.

Флаг CONSTANT позволяет экспортировать из DLL не только функции, но и данные. При этом параметр ИмяТочкиВхода задает имя экспортируемой глобальной переменной, определенной в DLL.

Пример экспортирования функций и глобальных переменных из DLL:

```
EXPORTS
DrawBitmap=MyDraw @ 4
ShowAll
HideAll
MyPoolPtr   @5 CONSTANT
GetMyPool   @8 NONAME
FreeMyPool  @9 NONAME
```

В приведенном выше примере в разделе EXPORTS перечислены имена нескольких экспортируемых функций - DrawBitmap, ShowAll, HideAll, GetMyPool, FreeMyPool - и глобальной переменной MyPoolPtr. Функция MyDraw, определенная в DLL, экспортируется под именем DrawBitmap. Она также доступна под номером 4. Функции ShowAll и HideAll экспортируются под своими "настоящими" именами, с которыми они определены в DLL. Для них не заданы порядковые номера. Функции GetMyPool и FreeMyPool экспортируются с флагом NONAME, поэтому к ним можно обращаться только по их порядковым номерам, которые равны соответственно 8 и 9. Имя MyPoolPtr экспортируется с флагом CONSTANT, поэтому оно является именем глобальной переменной, определенной в DLL и доступной для приложений, загружающих эту библиотеку.

1.5. Импортирование функций

Когда вы используете статическую компоновку, то включаете в файл проекта приложения соответствующий lib-файл, содержащий нужную вам библиотеку объектных модулей. Такая библиотека содержит исполняемый код модулей, который на этапе статической компоновки включается в exe-файл загрузочного модуля. Если используется динамическая компоновка, то в загрузочный exe-файл приложения записывается не исполнимый код функций, а ссылка на соответствующую DLL-библиотеку и функцию внутри нее. Как мы уже говорили, эта ссылка может быть организована с использованием либо имени функции, либо ее порядкового номера в DLL. Откуда при компоновке приложения редактор связей узнает имя DLL, имя или порядковый номер экспортируемой функции?

Для динамической компоновки функции из DLL-библиотеки можно использовать различные способы.

Библиотека импорта

Для того чтобы редактор связей мог создать ссылку, в файл проекта приложения вы должны включить так называемую *библиотеку импорта* (import library). Эта библиотека создается автоматически системой разработки. Следует заметить, что стандартные библиотеки систем разработки приложений Windows содержат как обычные объектные модули, предназначенные для статической компоновки, так и ссылки на различные стандарт-

ные DLL, экспортирующие функции программного интерфейса операционной системы Windows.

Динамический импорт функций во время выполнения приложения

В некоторых случаях невозможно выполнить динамическую компоновку на этапе, редактирования. Вы можете, например, создать приложение, которое состоит из основного модуля и дополнительных, реализованных в виде DLL. Состав этих дополнительных модулей и имена файлов, содержащих DLL, может изменяться, при этом в приложение могут добавляться новые возможности. Если вы, например, разрабатываете систему распознавания речи, то можете сделать ее в виде основного приложения и набора DLL, по одной библиотеке для каждого национального языка. В продажу система может поступить в комплекте с одной или двумя библиотеками, но в дальнейшем пользователь сможет купить дополнительные библиотеки и, просто переписав новые библиотеки на диск, получить возможность работы с другими языками. При этом основной модуль приложения не может "знать" заранее имена файлов дополнительных DLL, поэтому статическая компоновка с использованием библиотеки импорта невозможна. Однако приложение может в любой момент загрузить любую DLL, вызвав специально предназначенную для этого функцию программного интерфейса Windows с именем LoadLibrary. Приведем ее прототип:

```
HINSTANCE  WINAPI  LoadLibrary(LPCSTR lpszLibFileName);
```

Параметр функции является указателем на текстовую строку, закрытую двоичным нулем. В эту строку перед вызовом функции следует записать путь к файлу DLL или имя этого файла. Если путь к файлу не указан, при поиске выполняется последовательный просмотр следующих каталогов:

- каталог, из которого запущено приложение;
- текущий каталог;
- системный каталог Microsoft Windows;
- каталог, в котором находится операционная система Windows;
- каталоги, перечисленные в переменной описания среды PATH.

Если файл DLL найден, функция LoadLibrary возвращает идентификатор модуля библиотеки. В противном случае возвращается значение NULL. При этом код ошибки можно получить при помощи функции GetLastError.

В Microsoft Windows при многократном вызове функции LoadLibrary различными процессами функция инициализации DLL получает несколько раз управление с кодом причины вызова, равным значению DLL_PROCESS_ATTACH.

После использования DLL освобождается при помощи функции FreeLibrary, прототип который показан ниже:

```
WINAPI  FreeLibrary(HINSTANCE hLibrary);
```

В качестве параметра этой функции следует передать идентификатор освобождаемой библиотеки.

При освобождении DLL ее счетчик использования уменьшается. Если этот счетчик становится равным нулю (что происходит, когда все приложения, работавшие с библиотекой, освободили ее или завершили свою работу), DLL выгружается из памяти. Каждый раз при освобождении DLL вызывается функция DLLEntryPoint с параметрами DLL_PROCESS_DETACH или DLL_THREAD_DETACH, выполняющая все необходимые завершающие действия.

1.6. Функция GetProcAddress

Для того чтобы вызвать функцию из библиотеки, зная ее идентификатор, необходимо получить значение дальнего указателя на эту функцию, вызвав функцию GetProcAddress:

```
WINAPI GetProcAddress(HINSTANCE hLibrary, LPCSTR lpszProcName);
```

Через параметр hLibrary вы должны передать функции идентификатор DLL, полученный ранее от функции LoadLibrary. Параметр lpszProcName является дальним указателем на строку, содержащую имя функции или ее порядковый номер, преобразованный макро-командой MAKEINTRESOURCE.

1.7. Пример вызова экспортированной функции

Например, динамически подключаемая библиотека myLib.dll должна экспортировать функцию myProc. Для этого экспортируемую функцию необходимо описать в файле определения модуля с расширением .def следующим образом:

```
LIBRARY myLib
        EXPORTS
                myProc @1
```

Такое описание позволит получить ее адрес с помощью GetProcAddress двумя способами: по имени функции или по ее номеру.

Пусть в исходном коде динамически подключаемой библиотеки функция myProc определена следующим образом:

```
void myProc(HWND hWnd, int Param)
{
        //Какой то код
}
```

Тогда для использования экспортируемой функции в коде вызывающей программы необходимо провести следующие операции:

```
typedef void (*Proc)(HWND, int);                //Определить тип экспортируемой функции
Proc ExportProc;  //Определить переменную, имеющую тип экспортируемой функции
HMODULE hLib; //Определить переменную под дескриптор загружаемой библиотеки
hLib=LoadLibrary("myLib.dll"); //Спроецировать dll в адресное пространство процесса
if (hLib!=0) {
  ExportProc=(Proc)GetProcAddress(hLib,"myProc"); //Инициализировать переменную
                        //ExportProc адресом функции myProc из подключенной библиотеки
  ExportProc(hMainWindow, 23);      //Вызвать экспортированную функцию с
                        //фактическими аргументами hMainWindow  и  значением 23
FreeLibrary(hLib);      //Выгрузить dll из адресного пространства процесса
}
```

1.8. Создание пользовательских диалоговых окон

Microsoft Visual Studio позволяет с помощью конструктора создавать диалоговые окна. Для этого необходимо в рабочем проекте приложения или динамически подключаемой библиотеки добавить новый ресурс - Диалог. В появившемся конструкторе отобразиться простейший диалог с кнопками: Ok и Cancel. Рабочую область диалогового окна можно регулировать с помощью мыши или окна свойств диалога. Microsoft Visual Studio имеет панель элементов, которые можно внедрить в диалоговое окно и использовать как средства ввода и отображения данных (рис. 1).

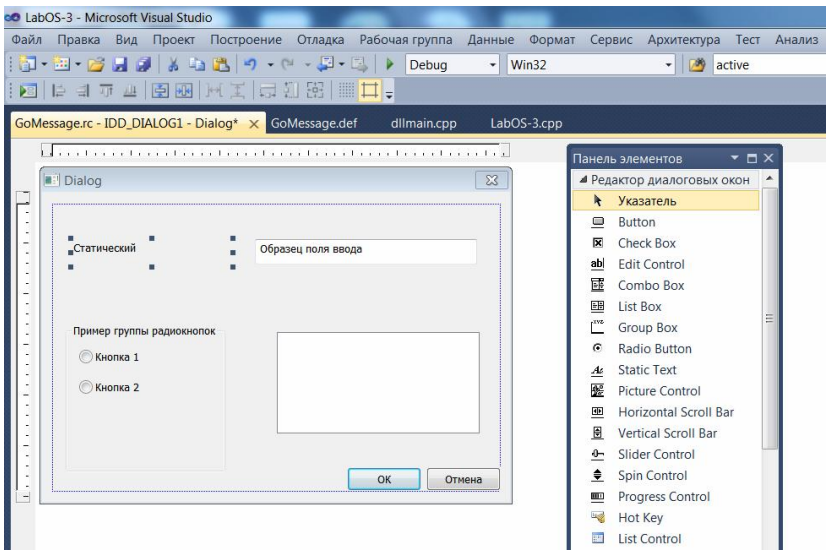

Рис. 1. Конструктор диалогового окна

Конструктор диалоговых окон позволяет компоновать управляющие элементы на рабочей области и строить некий шаблон. Каждый элемент управления, помещенный в рабочую область диалогового окна получает уникальный идентификатор, используя который можно узнать текущее состояние элемента (прочитать хранимые им данные), либо установить требуемые значения. Например, диалоговое окно содержит управляющий элемент Edit (редактируемая текстовая строка), который имеет идентификатор ID_CONTROLEDIT. Тогда с помощью вызова функции

```
GetDlgItemText(hDlg, ID_CONTROLEDIT, szStr, 50);
```

можно получить текст, который был введен пользователем программы в поле этого элемента управления и разместить его в программной переменной szStr. Первый аргумент этой функции - дескриптор диалогового окна, последний - максимальное количество символов, которое можно скопировать в переменную szStr.

Конструктор диалоговых окон задает только их форму, а функциональные возможности должна обеспечивать процедура, которая в Windows может иметь произвольное название, но строго заданный синтаксис, похожий на синтаксис функции главного окна приложения

```
BOOL CALLBACK DlgProc(HWND hwndDlg, UINT message, WPARAM wParam,  LPARAM lParam)
```

Данная функция является callback-функцией, т.е. ее вызывает операционная система, когда в очереди сообщений диалогового окна появляются новые сообщения. При этом операционная система передает callback-функции четыре аргумента: hwndDlg - дескриптор диалогового окна, которому принадлежит элемент управления, сгенерировавший сообщение; message - тип сообщения = WM_COMMAND; wParam содержит идентификатор элемента управления; lParam пока оставим без внимания. Таким образом, все функциональные возможности диалоговое окно приобретает благодаря callback-функции. Поэтому вызов процедур, подобной GetDlgItemText, должен осуществляться в ответ на какое то сообщение и реализовываться внутри callback-функции.

Например, при закрытии диалогового окна кнопкой Ok нужно считать данные поля редактирования в программную переменную. Для этого callback-функция программируется следующим образом

```
BOOL CALLBACK DlgProc(HWND hwndDlg, UINT message, WPARAM wParam,  LPARAM lParam)
{
        switch (message) {
                case WM_COMMAND:
```

```
                switch(wParam) {
                case IDOK:              //Нажатие кнопки Ok
                        GetDlgItemText(hwndDlg, ID_CONTROLEDIT, szStr, 50);
                }
        }
        EndDialog(hwndDlg);
        return true;
}
```

Функция обработки сообщений диалогового окна должна заканчиваться вызовом процедуры EndDialog.

И, наконец, когда создан шаблон диалогового окна, разработана функция обработки сообщений, только тогда появляется возможность его использования в приложении. Для в коде приложения нудно вызвать функцию (как правило, в теле обработчика сообщений приложения)

DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate, HWND hWndParent, DLGPROC lpDialogFunc);

в которой первый аргумент - дескриптор вызывающего приложения; второй содержит идентификатор ресурса диалога; третий - назначает родительское окно; четвертый - определяет функцию обработки сообщений.

Например, в исходном коде приложения его дескриптор храниться в глобальной переменной hInst, тогда вызов диалогового окна можно осуществить при поступлении сообщения о выборе пункта меню главного окна

```
…..
case WM_COMMAND:
  switch(wParam) {
        case IDM_SHOWDIALOG:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_MYDIALOG), hWnd, DlgProc);
        break;
….
```

2. Демонстрационная программа

В папке LabOS-2(Demo) находится приложение LabOS-2.exe, демонстрирующее результат выполнение студентом лабораторной работы. Приложение использует динамически подключаемую библиотеку для ввода сообщения через диалоговое окно. После нажатия кнопки Ok введенное сообщение отображается в рабочей области окна. Пока dll не загружена функция ввода нового сообщения не доступна.

3. Задание к лабораторной работе

Для выполнения лабораторной работы используйте решение Microsoft Visual Studio LabOS-2. Оно содержит заготовки двух проектов: LabOS-2 и GoMessage. Первый проект предназначен для построения exe-файла, а второй – библиотеки динамической компоновки.

1. Создайте для проекта LabOS-2 ресурс меню, подобный ресурсу демонстрационной программы.

2. Модифицируйте функцию обработки сообщений главного окна так, чтобы она позволяла подключать и отключать библиотеку GoMessage.dll и исполнять экспортируемую функцию inMessage, а также отображать с помощью MessageBox сведения о разработчике.
3. В проекте GoMessage постройте шаблон диалогового окна с использованием элемента управления Edit - поле редактирования. В рабочей области диалогового окна должен содержаться статический текст с указанием разработчика.
4. Создайте в проекте GoMessage функцию, которая будет обрабатывать сообщения Вашего диалогового окна. Она должна реализовывать считывание текстовой строки из поля ввода и копировать его в массив типа char при закрытии диалогового окна кнопкой Ok.
5. Создайте в проекте GoMessage экспортируемую функцию inMessage, которая будет вызывать DialogBox при обращении к ней. По завершении своей работы функция inMessage должна передавать копию введенного сообщения приложению LabOS-2.
6. В файле определения модуля с расширением .def опишите раздел экспорта с указанием единственной экспортируемой функции inMessage. Сделайте так, чтобы функцию можно было вызвать не только по имени, но и по номеру.
7. Постройте файлы LabOS-2.exe и GoMessage.dll. Запустите и протестируйте работу приложения. Результат проделанной Вами работы обязательно покажите преподавателю.

4. Контрольные вопросы

1. В чем заключается эффективность динамической компоновки кода программы?
2. Может ли с помощью DLL одно приложение получить доступ к коду или данным другого приложения?
3. Каким образом DLL может быть загружена в оперативную память?
4. Когда DLL выгружается из оперативной памяти? Кто отвечает за выгрузку DLL?
5. Как динамически подключаемая библиотека предоставляет доступ к своим программным ресурсам?
6. Может ли DLL использовать программные ресурсы других библиотек?
7. Какие функции Windows необходимо задействовать для подключения и отключения DLL?
8. Каким образом приложение может вызвать функцию, экспортируемую DLL?
9. Каким должен быть программный код приложения для вызова импортируемой функции по ее номеру в библиотеке динамической компоновки?
10. Назовите этапы построения диалогового окна.
11. Какие сообщения генерируются операционной системой при воздействии на элементы управления диалогового окна?
12. Каким образом организована обработка сообщений диалоговых окон?