

Written by ASMLOVER in 1812 AD. All rights are kept.  
Donations please send to x.y.z @ list.ru

## Глава 30

### Использование внешних библиотек и создание дополнительных команд языка Blitz3d

Blitz3d написан на другом языке высокого уровня – C++. Сам же компилятор C++ пишется с использованием ассемблера и (или) других языков. Поэтому разработчики, как правило, дают возможность программе-транслятору нового языка использовать в той или иной степени возможности того программного средства, на котором велась разработка. Также может обеспечиваться и доступ к системным библиотекам Windows (которая сама написана на C++, C и ассемблере).

Для чего может понадобиться внешняя библиотека? Вообще-то, в нашей программе можно обойтись и без нее. Но некоторые задачи реализовать средствами самого Blitz3d будет нелегко. Например, если мы захотим зашифровать свой файл или внедрить различные ограничения на его использование. Если у нас есть обширная библиотека функций, разработанная на других языках программирования и мы не хотим тратить время на их портирование на платформу Blitz3d. Если наша программа использует тысячи тригонометрических вычислений в цикле, наверняка придется разрабатывать таблицу синусов и быструю функцию для обращения к ней. Если мы хотим отключить часть клавиатуры. Наконец, если мы хотим реализовать поддержку функций DirectX в нашем проекте. Но если идти дальше, то тогда все у нас будет сделано средствами других языков, а для Blitz3d останется почетная функция вызова одной внешней процедуры. Спрашивается тогда, зачем мы столько времени на нем программировали? Поэтому поступим разумно и для начала продемонстрируем возможность использования простой функции, созданной на языке C++, в программе, написанной на Blitz3d. Функцию поместим во внешнюю программу – библиотеку динамической связи (DLL). Для создания внешней библиотеки воспользуемся стандартным компилятором фирмы Microsoft – Visual C++, но можно использовать практически любой другой компилятор.

Тестовую задачу сформулируем следующим образом. Нужно возвести в квадрат четыре целых числа. Произошло ли переполнение результата (это будет в случае, если исходное число содержит более 16 двоичных разрядов), нас не интересует. Исходные данные не должны быть изменены. Вызывающая программа должна быть уведомлена о том, что данные обработаны.

Таким образом, наша внешняя функция из внешней библиотеки, вызванная из Blitz3d, должна прочитать где-то в памяти четыре числа по четыре байта, произвести возведение их в квадрат и где-нибудь в памяти их записать. После успешной работы она должна вернуть в вызывающую программу ненулевое значение.

Сначала займемся вызывающей программой. Начнем с создания буферной зоны обмена информацией. Для этого воспользуемся новой командой

**CreateBank ([size])**

**СоздатьБуфер([размер]),**

которая выделяет область памяти запрошенного размера (в байтах) и возвращает ее идентификатор. (Слова банк и буфер – синонимы, можно использовать и то, и другое значение). Так как мы должны сохранить исходные данные, то нам потребуется два буфера – один для чтения информации, а второй для записи результатов работы. В нашем конкретном случае

создаем два одинаковых по размеру буфера:

```
DataToDLL=CreateBank(16)  
DataFromDll=CreateBank(16).
```

Первый предназначен для исходных данных, второй – для результата, каждый по 16 байт. Иницилируем исходные данные - четыре значения в буфере DataToDLL :

```
For i=0 To 12 Step 4  
    PokeInt DataToDLL,i,i  
Next
```

Новая команда

**PokeInt bank,offset,value**  
**ЗаписатьЦелое буфер, смещение,значение**

Записывает в указанный буфер по заданному смещению от начала буфера целое 4-х байтное число. В нашем случае в начало буфера записывается 0, со смещением 4 записывается 4, со смещением 8 – 8 и со смещением 12 -12. Теперь проверим, что содержат буфера DataToDLL и DataFromDll:

```
For i=0 To 12 Step 4  
    Print PeekInt (DataToDLL,i)  
Next
```

```
For i=0 To 12 Step 4  
    Print PeekInt (DataFromDLL,i)  
Next
```

Команда

**PeekInt (bank,offset)**  
**ПрочитатьЦелое (буфер, смещение)**

возвращает считанное 4-х байтное целое число из указанного буфера по заданному смещению. Еще одна новая команда:

**Print [string\$]**  
**Напечатать [строка\$]**

выводит на экран (путем записи во FrontBuffer) указанную строку. Если она не указана, просто производит запись символов перевод каретки и возврат строки, то есть последующий вывод текстовой информации на экран будет начат с новой строки (как правило, в ранних версиях Бейсиков с команды Print начиналось описание языка, а мы добрались до нее только в самом конце книги).

Перед проверкой работоспособности добавим строчку в конец программы:

## WaitKey

### ОжидатьНажатиеКлавиши,

которая просто останавливает программу до тех пор, пока любая из клавиш клавиатуры не будет нажата.

```
DataToDLL=CreateBank(16)
DataFromDll=CreateBank(16)

For i=0 To 12 Step 4
    PokeInt DataToDLL,i,i
Next

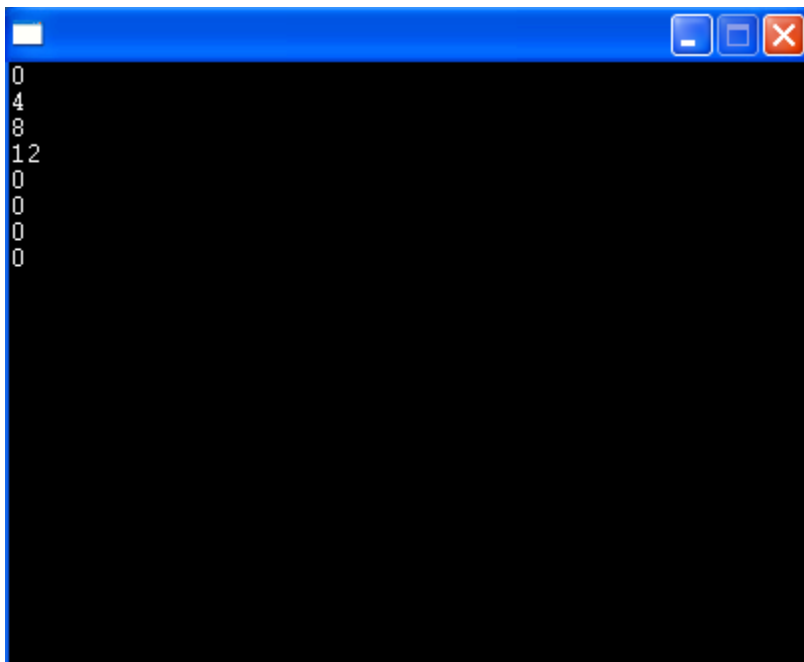
For i=0 To 12 Step 4
    Print PeekInt (DataToDLL,i)
Next

For i=0 To 12 Step 4
    Print PeekInt (DataFromDLL,i)
Next

WaitKey
```

Программа 30-1.

Запускаем программу и проверяем ее работу. До нажатия любой клавиши будет высвечиваться окно с результатом :



Результат работы программы 30-1.

Видно, что результирующий буфер имеет нулевые значения.

Для работы с выделенными для обмена информации буферами памяти можно использовать команды, оперирующие и другими форматами данных:

**PokeByte bank,offset,value**

**PokeShort bank,offset,value**  
**PokeFloat bank,offset,value**

**PeekByte (bank,offset)**  
**PeekShort (bank,offset)**  
**PeekFloat (bank,offset).**

Первые три команды заносят в память значения целого числа размером в байт, два байта и число с плавающей точкой, занимающее 4 байта. Следующие три – считывают данные этого типа из памяти. Ответственность за интерпретацию и правильное использование форматов данных полностью ложится на программиста (хотя на него и так все ложится).

Внешняя DLL вызывается специальной командой Blitz3d:

**CallDLL( dll\_name\$, proc\_name\$, in\_bank, out\_bank) )**  
**ВызватьВнешнююБиблиотеку (имя\_библиотеки\$, имя\_процедуры\$ [, буфер\_данных, буфер\_результата]).**

Если файл библиотеки не находится в текущем каталоге, (откуда была вызвана программа 30-1), то необходимо указать полный путь до него или, что не совсем корректно, разместить его вместе с системными DLL. Расширение dll можно не указывать. Также безразличен регистр набора, то есть, если файл DLL называется test.dll, то правильно будет и Test.dll, и test.dll и просто TEST. Но будьте внимательны - для вызываемой процедуры прописные и строчные буквы имеют значение, поэтому, если процедуру Square попробовать вызвать как square, то Вас постигнет неудача. Если буфера данных и результата не используются, их можно не указывать. (В общем случае процедура может не использовать входных данных и не возвращать никакого значения. Напомню, что в классическом понимании функция – это частный вид процедуры, возвращающий значение). В случае, если применяется одна область данных для обмена информацией, то имя буфера данных будет совпадать с именем буфера результата. Сейчас в программе можно поставить любые значения для имени библиотеки и процедуры.

Завершим нашу тестовую программу:

```

DataToDLL=CreateBank(16)
DataFromDll=CreateBank(16)

For i=0 To 12 Step 4
    PokeInt DataToDLL,i,i
Next

For i=0 To 12 Step 4
    Print PeekInt (DataToDLL,i)
Next

For i=0 To 12 Step 4
    Print PeekInt (DataFromDLL,i)
Next

result=CallDLL("C:\temp\test\debug\test","Square", DataToDLL,DataFromDll)
If result Print "DLL works!" Else Print "DLL failed."
Print "Result : "+result

For i=0 To 12 Step 4
    Print PeekInt (DataFromDLL,i)
Next

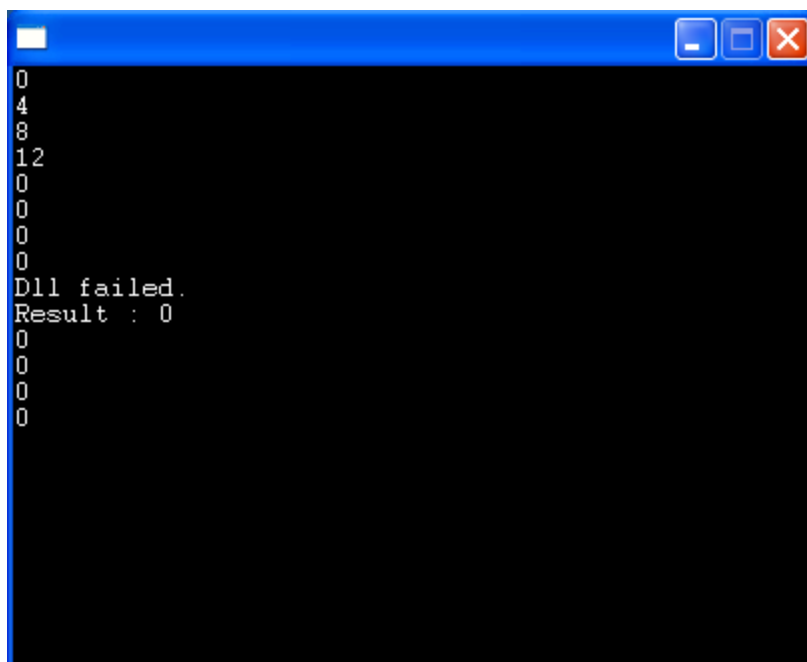
WaitKey

```

Программа 30-2.

Переменной result будет присваиваться значение, которая возвращает функция CallDLL вызывающей программе. В случае ненулевого значения (то есть когда DLL отработала нормально), на экран выводится строка “DLL works!” (“DLL работает!”), в противном случае фраза будет другая – “DLL failed” (“DLL не отработала”). После чего на экран будет выведено значение переменной result и состояние буфера результата.

Запускаем программу.



```

0
4
8
12
0
0
0
0
0
Dll failed.
Result : 0
0
0
0
0
0

```

Результат работы программы 30-2 без DLL.

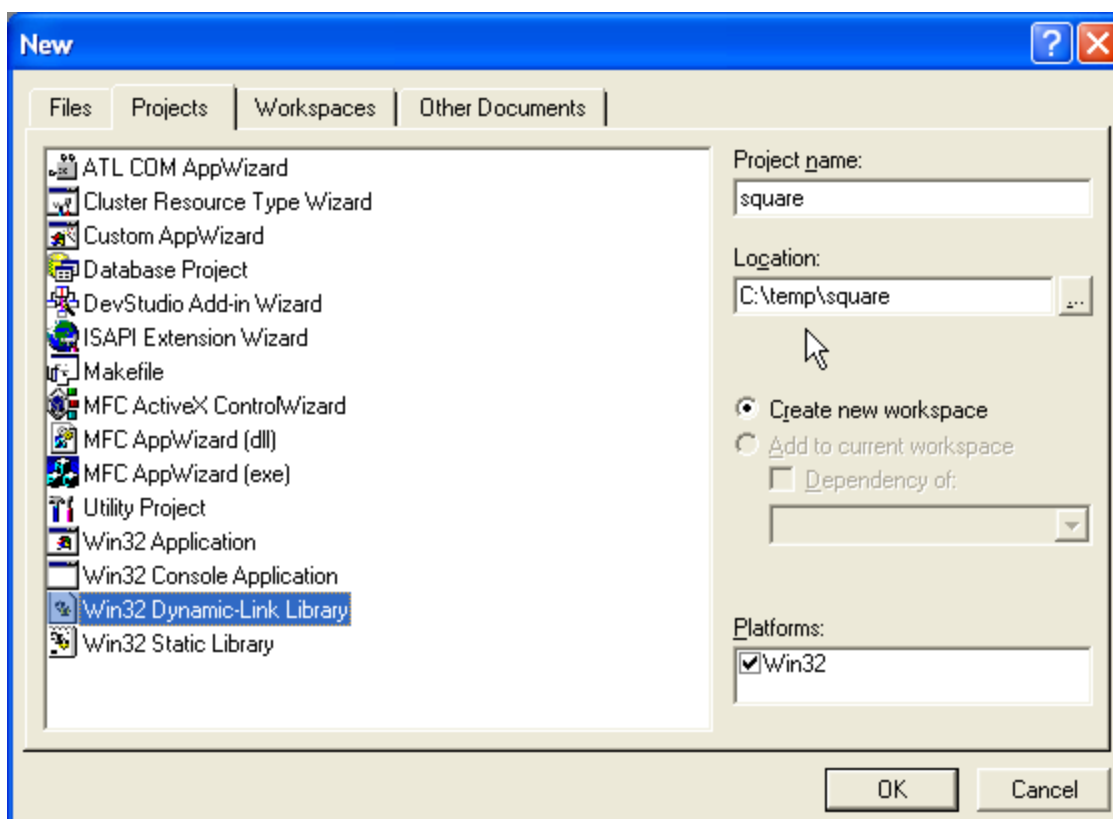
Как и следовало ожидать, никто наше обращение CallDLL не обработал. Все приходится делать своими руками.

Еще несколько слов, о том для чего вообще нужны DLL в Windows. Говоря упрощенно, DLL – это обыкновенные файлы с произвольным набором функций. Если эти функции используются только в одной программе – то смысла в использования DLL нет никакого – проще было бы ввести все функции в тело программы (исключение – именно наш случай, то есть когда для разработки программы использовались различные инструментальные средства и языки программирования). Другое дело, когда функции (например, глобальные функции Windows) используются многими процессами одновременно – наличие в памяти только одной копии функции, позволяющее работать всем программам, которые к ней обращаются, существенно экономит ресурсы компьютера и повышает быстродействие системы в целом. Также это положительно отражается и на размерах программ.

DLL может подключаться различными способами. Помимо неявного динамического подключения, которое мы рассмотрим сначала, в Blitz3D существует возможность их явного и неявного статического подключения.

Поскольку DLL может создаваться различными трансляторами и использовать нестандартные или неподдерживаемые вызывающей программой способы передачи и возвращения параметров, не всякую библиотеку можно подключить к программе на Blitz3d (без дополнительной программной обработки). Поэтому сначала сделаем свою DLL.

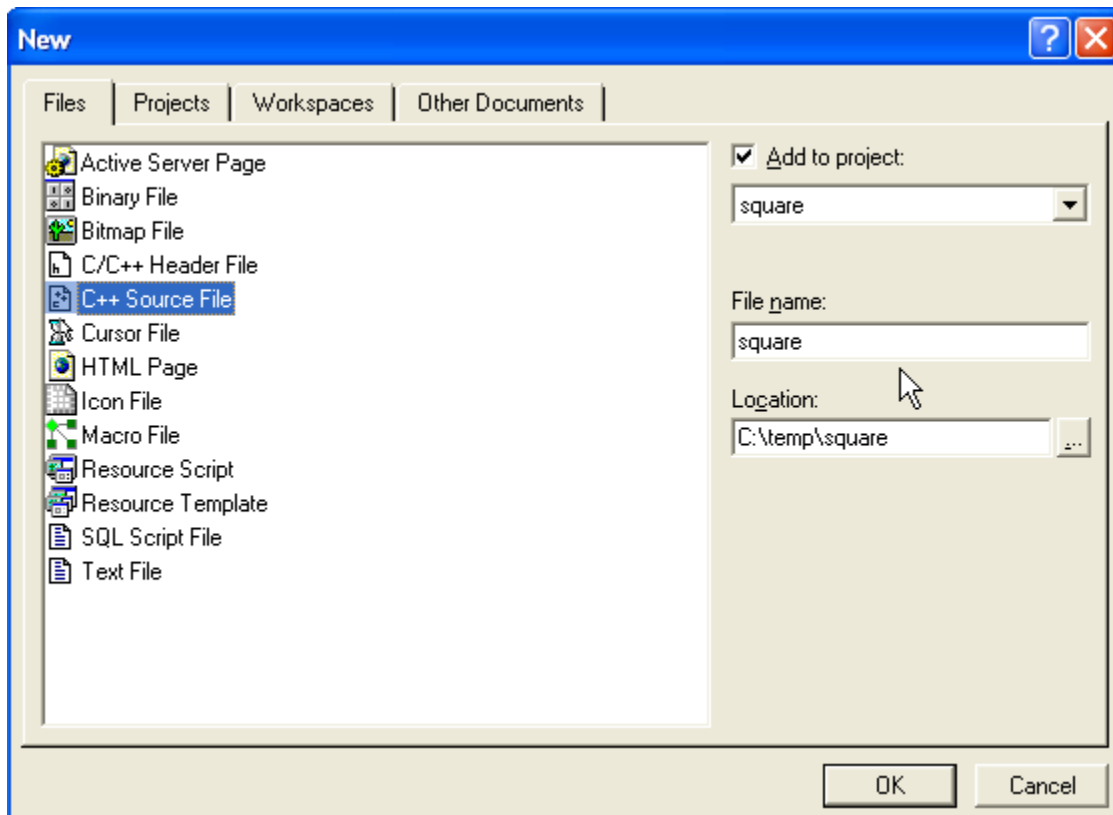
Запускаем VisualC++ (на иллюстрациях – версия 6.0 ) и, выбрав в меню File опцию New, получаем вот такой экран :



Начало нового проекта square.

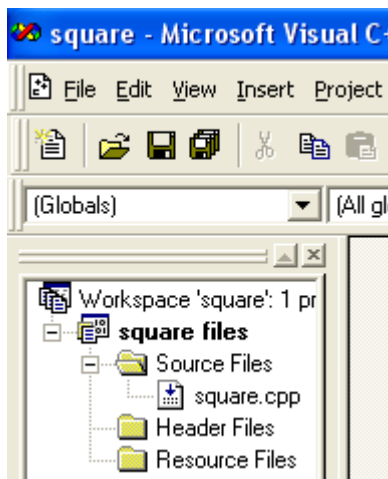
Выбираем проект для создания Win32 Dynamic-Link Library, в поле Project name вводим имя нашего проекта – square, выбираем в поле Location место, где будут размещаться файлы проекта. Нажимаем клавишу OK. В следующих двух окнах нажимаем на Finish и OK и попадаем в главное окно проекта. Теперь нужно создать файл с кодом программы square.cpp, для чего опять идем путем File\New, в появившемся окне выбираем C++ Source File, указываем

имя и нажимаем ОК.



Создание

нового файла проекта “square.cpp”.



Файл “square.cpp” появился на левой панели при выборе закладки FileView.

Набираем текст файла:

```

#define EXPORT extern "C" __declspec (dllexport)

EXPORT int _cdecl Square(int *toDLL,int in_size,int *fromDLL,int out_size)
{
    for( int i=0;i<out_size/4;++i )
    {
        fromDLL[i]=toDLL[i]*toDLL[i];
    }

    return 1;
}

```

Листинг файла “square.cpp”

Чтобы вызывающая программа могла обращаться к функциям динамической библиотеки, наша функция должна попасть в таблицу экспортируемых функций DLL (внутри самой DLL). Есть два способа занести ее в эту таблицу на этапе компиляции, с применением метода `__declspec(dllexport)` и путем создания определений в def-файле. Мы пойдем первым путем и, для начала, определим макрос `EXPORT`. Макрос в данном случае – это просто короткое имя, присвоенное часто используемому и более длинному фрагменту кода (фрагмент может начинаться и заканчиваться где угодно, на любом символе, хоть в середине слова). В нашем случае слово `EXPORT`, примененное после его определения в первой строке, при компиляции будет просто заменено на тот текст, что стоит правее него при его определении – то есть на `extern "C" __declspec (dllexport)`. Это сделает листинг программы более читабельным и позволит набирать код быстрее и без ошибок, что особенно полезно, если функций будет много. В этом тексте `extern "C"` сообщает компилятору чтобы он не использовал расширение имени функции (mangling). Зачем компилятор делает это расширение? Функции C++ (в отличие от функций языка C и других) для реализации полиморфизма могут использовать одно и то же имя для работы с различным числом и типом аргументов. Компилятор такие функции различает по числу и типу передаваемых параметров и добавляет к имени некоторый довесок, который бы не позволил найти нашу функцию по имени `Square` при вызове из `Blitz3d`. Мы, таким образом, сохраняем наше имя в неприкосновенности (но, тем не менее, C++ попытается еще раз попробовать его изменить).

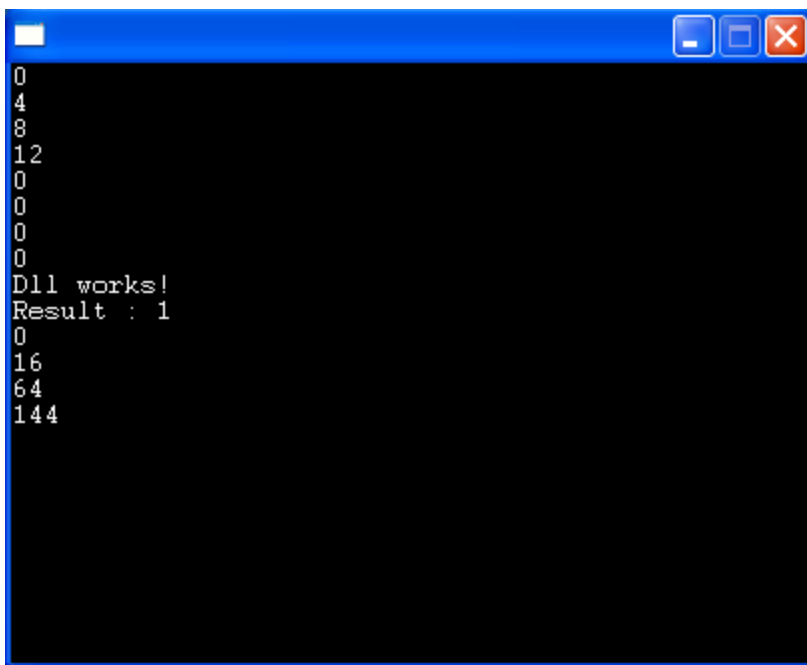
Далее определяется сама функция `Square`, `int` говорит о том, что функция возвращает целое значение, а `_cdecl` предотвращает дополнительное декорирование (decoration) имени функции (вот оно – второе нападение), иначе будет считаться, что применен стандартный вызов языка C `_stdcall` и в DLL наша функция насильно превратится из `Square` в `_Square@16`, где 16 – это число байтов, использованных под параметры функции.

В качестве параметров используются указатель на входной буфер `*toDLL`, размер входного буфера `in_size`, указатель на выходной буфер `*fromDLL` и его длину `out_size`. Затем, в цикле `for...next`, написанном по правилам языка C++, каждому значению выходного буфера присваивается соответствующее значение входного буфера, возведенное в квадрат и, после отработки цикла функции, ей присваивается возвращающее значение единица.

В среде редактора C++ нажимаем клавишу F7 и, если не было ошибок, в папке `c:\temp\square\debug` будет создана программа “square.dll”. Еще раз запустим вызывающую программу из `Blitz3d`.

Вот как будет выглядеть экран во время выполнения программы:





```
0
4
8
12
0
0
0
0
0
Dll works!
Result : 1
0
16
64
144
```

Результат работы программы 30-2 с работающей DLL.

На этот раз внешняя библиотека была подключена и функция, написанная на C++, отработала корректно.

Вы можете заметить, что параметр `in_size` в программе не используется. Может, он лишний в строке параметров функции и его следует оттуда убрать и сэкономить место? (Внимательный читатель задаст и другой вопрос - где, когда и каким значением была инициализирована другая, используемая переменная, `out_size`?). Если убрать `in_size` из строки параметров то в этом случае после компиляции DLL программа работать не будет, (или будет, но не так, как должна – скорее всего Вы получите сообщение о нарушении доступа к памяти или будут еще более негативные последствия). Дело в том, что вызов DLL из Blitz3d командой `CallDLL` предполагает наличие определенного прототипа (шаблона) для функции (который можно было описать в файле “test.h”, как обычно это делается, и включить в проект) и независимо от нас DLL автоматически получает от Blitz3d эти параметры в указанном порядке (если буфера обмена были указаны в команде `CallDLL`). Мы могли бы, в нашем конкретном случае, переписать цикл `for...next` с использованием `in_size` и убрать переменную `out_size`. Тогда программа будет работать корректно, так как все работающие переменные получают свои значения. В случае же удаления переменной `in_size` из середины списка параметров мы нарушаем порядок использования именованных областей памяти, что приводит к непредсказуемым результатам, вплоть до зависания компьютера. Так что будьте очень аккуратны и при отладке своих DLL контролируйте завершение не только программ на экране, но и создаваемых ими процессов в памяти.

Чтобы убедиться в том, что параметры `in_size` и `out_size` инициализированы, вместо возвращаемого значения для оператора `return`, равного единице, подставьте значение `in_size` или `out_size`, перекомпилируйте “square.dll” и запустите вызывающую программу. Теперь DLL возвращает значение, равное размеру буфера в байтах, т.е. 16, хотя мы никоим образом не причастны к инициализации переменных `in_size` и `out_size` – все это обеспечил системный вызов `CallDLL`.

Следует учитывать еще одну особенность применения динамически подключаемых библиотек DLL – факт вызова отсутствующей DLL система игнорирует и спокойно продолжает работу, что, на самом деле, абсолютно некорректно (вернее, некорректно оставить без внимания

разработчику программы, использующую DLL). Поэтому на этапе разработки программы необходимо тщательно проверять правильную работу подключаемых библиотек, расставив точки контроля для случая, если DLL себя не проявила в нужном месте. Например, вот так:

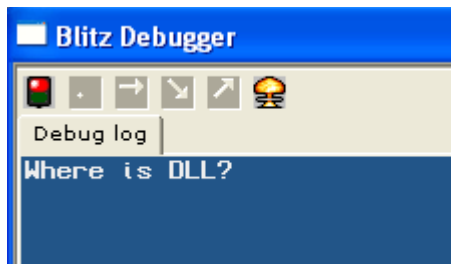
```
DataToDLL=CreateBank(16)
DataFromDll=CreateBank(16)

result=CallDLL("C:\temp\test\debug\test","Square", DataToDLL,DataFromDll)
If Not result
    DebugLog ("Where is DLL?")
EndIf

WaitKey
```

Программа 30-3

Теперь, если убрать файл библиотеки из указанной папки или указать неверный путь к DLL, можно будет увидеть, что log-файл содержит строку “Where is DLL?” (“Где DLL?”) и надо будет разбираться, почему внешняя функция не сработала. При этом ошибки выполнения не происходит.



Окно отладчика перед завершением программы 30-3

Если же произвести статическое связывание библиотеки, то, в случае отсутствия DLL, мы сразу получим ошибку выполнения. Вот это – правильно, поскольку DLL может содержать жизненно важные функции или основной код программы.

А чем же еще отличается статическое связывание от динамического?

Статическое связывание подключает нужные функции постоянно на все время действия использующей DLL программы. Поэтому можно добавлять в программу дополнительные функции, которые будут выглядеть, как обычные команды Blitz3d (они также будут подсвечиваться голубым цветом в редакторе и, при желании, в систему может быть добавлена справочная информация по их использованию). К созданию самой DLL при этом немного другие требования, она должна создаваться с параметром `_stdcall`. В качестве передаваемых в DLL параметров могут выступать указатели (идентификаторы) на объект или буфер памяти. Информация о размере этих данных автоматически, как в предыдущем случае, не передается. Объекты и буфера памяти не могут являться возвращаемой величиной, также при работе с такими функциями не поддерживаются массивы. Правда, при определенных навыках, эти ограничения можно обойти.

Фактически, мы можем произвести подключение любых DLL, которые удовлетворяют вышеперечисленным условиям и пользоваться их функциями, если нам доступны их описания (это касается и глобальных функций Windows!).

Сначала попробуем создать новую функцию – команду Blitz3d `Square`, которая будет возводить в квадрат передаваемый ей аргумент типа `float`.

Удалим папку `C:\temp\square` со всеми файлами и DLL, так как они нам больше не понадобятся, и в VisualC++ создаем новый DLL-проект `square` и новый файл “`square.cpp`”:

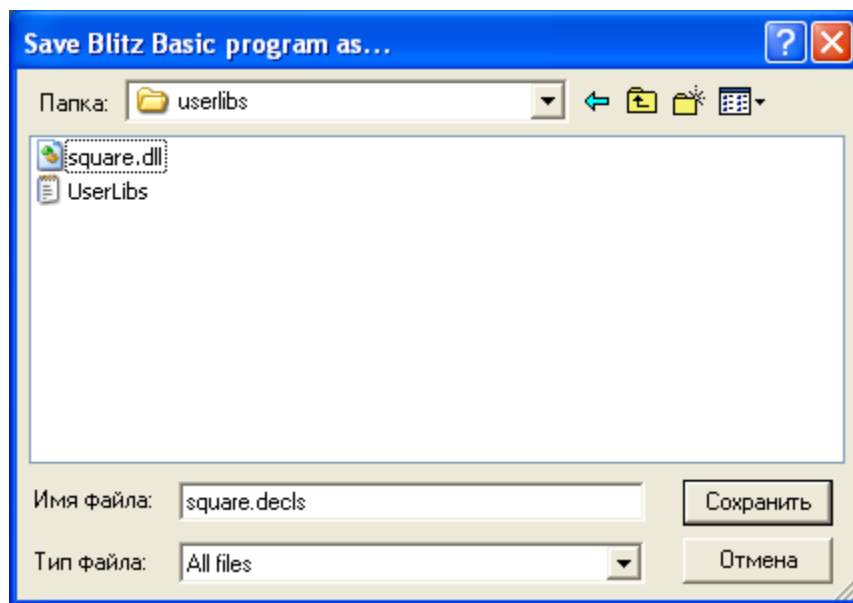
```
#define EXPORT extern "C" __declspec (dllexport)
```

```
EXPORT float _stdcall Square(float number)  
{ return number*number; }
```

Новый файл “square.cpp”

Нажимаем F7 и будет создана новая DLL “square.dll”. Теперь нужно переместить этот файл в специальное место – папку Userlibs основного каталога Blitz3d C:\Program Files\Blitz3D\userlibs (если Вы поменяли путь по умолчанию при инсталляции Blitz3d, то он может быть и другим). Демо-версия такой каталог не создает, в этом случае надо просто создать его самим.

Туда же нужно поместить новый текстовый файл “square.decls”, содержащий описание библиотеки и экспортируемые функции. Текстовые файлы можно создавать в редакторе Blitz3d (текст любой программы с расширением .bb представляет собой обычный текстовый файл), только при сохранении на диск необходимо применять способ Save As.., в появившемся окне вместо установленного по умолчанию типа файла .bb поставить опцию All files и назвать файл нужным именем с любым расширением после точки. Создадим новый файл в редакторе Blitz3d и сохраним его как “square.decls”:



Сохранение текстового файла “square.decls”

Теперь наполним его содержанием – укажем, какие функции и в каких библиотеках будут новыми командами языка (учтите, что при объявлении в decls-файле аргументам функций нельзя присваивать значения):

```
.lib "square.dll"  
Square (number#): "_Square@4"
```

Файл “square.decls”

Так как при компиляции DLL использовалась опция \_stdcall, внутри самой DLL имя Square изменилось на \_Square@4. Для того, чтобы новая команда выглядела без этих излишеств, мы указываем желаемое имя и, после двоеточия, реальное имя функции в DLL. Чтобы она

появилась и была готова к использованию, необходимо перезапустить программу Blitz3d. Достаточно теперь снова открыть в редакторе Blitz3d файл “square.decls”, чтобы убедиться в этом - Square подсвечивается голубым цветом. У нас появилась новая команда! Проверьте, правильно ли она считает. Теперь очень легко нашу команду переименовать, изменив всего лишь одну строчку в decls-файле, например:

```
.lib "square.dll"  
Mul2#( number#): "_Square@4"
```

Изменение имени команды на Mul2.

Теперь, после перезапуска Blitz3d, в системе больше нет Square, но появилась команда Mul2. Таким образом можно переопределить (случайно или намеренно) любую встроенную функцию Blitz3d, будьте внимательны. Вернем назад определение Square и займемся созданием новой команды с использованием функций операционной системы.

Если известны названия функции, тип и количество ее аргументов, то командами языка можно сделать глобальные функции Windows. Для примера, введем в язык новую команду Msg, выводящую на экран окно с сообщением, которая в действительности будет являться вызовом функции MessageBox из системной библиотеки Windows - user32.dll. Прототип функции следующий:

int MessageBox (HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType),

где

HWND hWnd – определяет дескриптор окна, являющееся собственником этого окна, если – NULL, то создаваемое окно не имеет собственника;

LPCSTR lpText – указатель на текстовую строку с сообщением или само сообщение, заключенное в кавычки;

LPCSTR lpCaption – указатель на текстовую строку с заголовком окна или сам текст заголовка, заключенный в кавычки;

UINT uType – комбинация флагов, определяющая содержимое и поведение окна. Префикс перед параметром функции определяет тип данных. Разумеется, C++ превосходит Blitz3d в разнообразии типов данных, но ничего страшного в том, что в Blitz3d таких типов не имеется, нет, в самом худшем случае можно прибегнуть к преобразованию.

Функция возвращает 0, если открытие окна не удалось и, в зависимости от того, какая кнопка окна была нажата ( а можно сделать так, чтобы появлялись кнопки Abort, Cancel, Ignore, No, Ok, Retry, Yes и Help) возвращает соответствующее значение, которое определяется операционной системой.

Команда добавляется очень просто – записывается две дополнительные строки в файл “square.decls” и, после перезагрузки Blitz3d, ее уже будет видно как команду языка и можно будет использовать:

```
.lib "square.dll"  
Square#( number#): "_Square@4"  
  
.lib "user32.dll"  
Msg( hWnd, lpText$, lpCaption$, uType): "MessageBox"
```

Добавление функции из библиотеки Windows user32.dll в файл “square.decls”

Поскольку DLL системная, то нам не нужно заботиться о ее местонахождении, система сама найдет ее (она обычно находится в папке C:\Windows\System32).

Как эту новую команду можно применить? Она может эффективно использоваться при создании интерфейса программы или обработки собственных исключительных ситуаций. Для демонстрации различных параметров окна, создаваемого командой Msg, наберите код, приведенный ниже.

```
;MessageBox Game Using user32.dll

;Buttons
Const MB_OK=0
Const MB_OKCANCEL=1
Const MB_ABORTRETRYIGNORE=2
Const MB_YESNOCANCEL=3
Const MB_YESNO=4
Const MB_RETRYCANCEL=5
Const MB_ABORTRETRYCONTINUE=6
;Icon
Const MB_STOP=16
Const MB_QUESTION=32
Const MB_EXCLAMATION=48
Const MB_INFORMATION=64
;

Message$="Pause for cup of tea"
Caption$="Info"

While True

Select Int(Msg(0,"Game with windows","Deno",MB_OKCANCEL+MB_INFORMATION))
  Case 1
    Select Int(Msg(0,"Continue?","Question",MB_ABORTRETRYCONTINUE+MB_QUESTION))
      Case 2
        Select Int(Msg(0,"Really Exit?","About to Finish",MB_YESNO+MB_STOP))
          Case 6
            End
          End Select
        End Select
      Case 10
        Msg(0, "Impossible. Try next time", "Joke",MB_OK)
      Case 11
        Msg(0, "Nice choise!", "Congratulations!",MB_OK+MB_EXCLAMATION)
      End Select
    Case 2
      Select Int(Msg(0, "Really Exit?", "About to Finish",MB_YESNO+MB_STOP))
        Case 6
          End
        End Select
      End Select
    End Select
  End Select

Msg (0,Message,Caption,MB_OK+MB_EXCLAMATION)

Wend
```

#### Программа 30-4

Два блока констант, Buttons (Кнопки) и Icon (Пиктограмма), определяют набор флагов иType (в примере определены не все). Итоговое значение флага определяется путем суммирования значения флага кнопок и флага пиктограммы ( имеет смысл взятие только одного из каждого блока). Какие кнопки появятся, написано в идентификаторе, например, использование значения MB\_YESNO приведет к появлению двух кнопок –YES и NO (в локализованной версии Windows появятся надписи на языке локализации). Префикс MB означает то, что эти константы принадлежат системной функции типа MessageBox. Соответственно, флаги второй группы определяют графическую пиктограмму, появляющуюся в рабочей области окна.

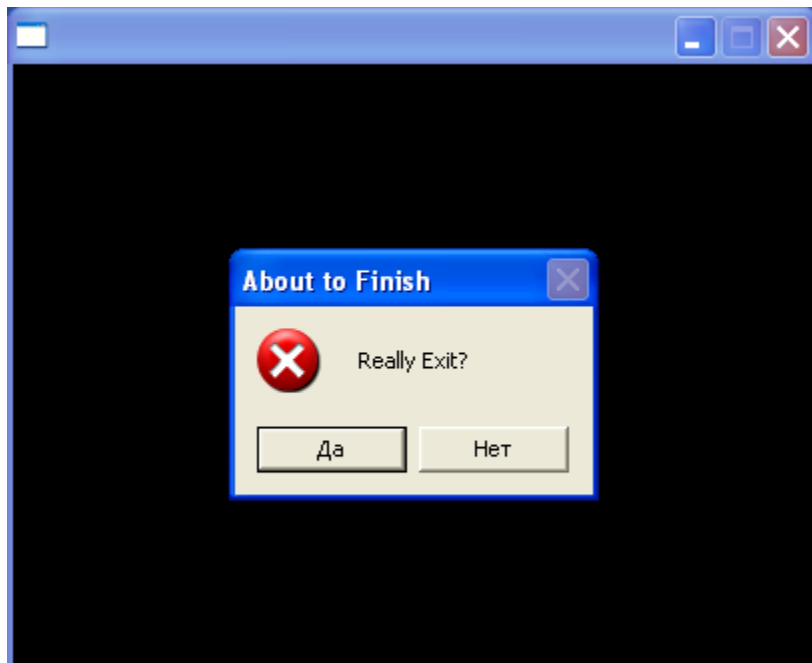
Если нам точно неизвестно, какого типа переменная будет возвращена функцией, необходимо

привести его к какому-то определенному типу, в нашем случае мы приводим к целочисленному значению типа `int`.

Программа в цикле будет по очереди выводить на экран различные варианты оформления окон, пока явно не будет выбран выход из программы. Обратите внимание на строку

```
Msg (0, Message, Caption, MB_OK+MB_EXCLAMATION)
```

В ней, для иллюстрации, вместо непосредственных текстовых строк для текста и заголовка окна использованы ссылки на локальные переменные `Blitz3d`, которые становятся аргументами глобальной функции `Windows MessageBoxA`.



Вид окон программы 3D-4 перед ее завершением.

Есть еще несколько интересных флагов, например, `MB_HELP=16384` добавляет кнопку `Help`, `MB_TOPMOST=262144` заставляет окно всегда находиться поверх остальных, `MB_RIGHT=524288` выравнивает текст и заголовок окна по правому краю. Для описания всех флагов и других функций системных библиотек необходимо обратиться к справочной литературе или, естественно, Интернету. Учите английский язык!

В случае использования команды `Msg` Вы должны закончить работу с окном сообщения, прежде чем продолжится выполнение программы на `Blitz3d`. До тех пор никакие комбинации клавиш, определенные в программе на языке `Blitz3d`, действовать не будут!